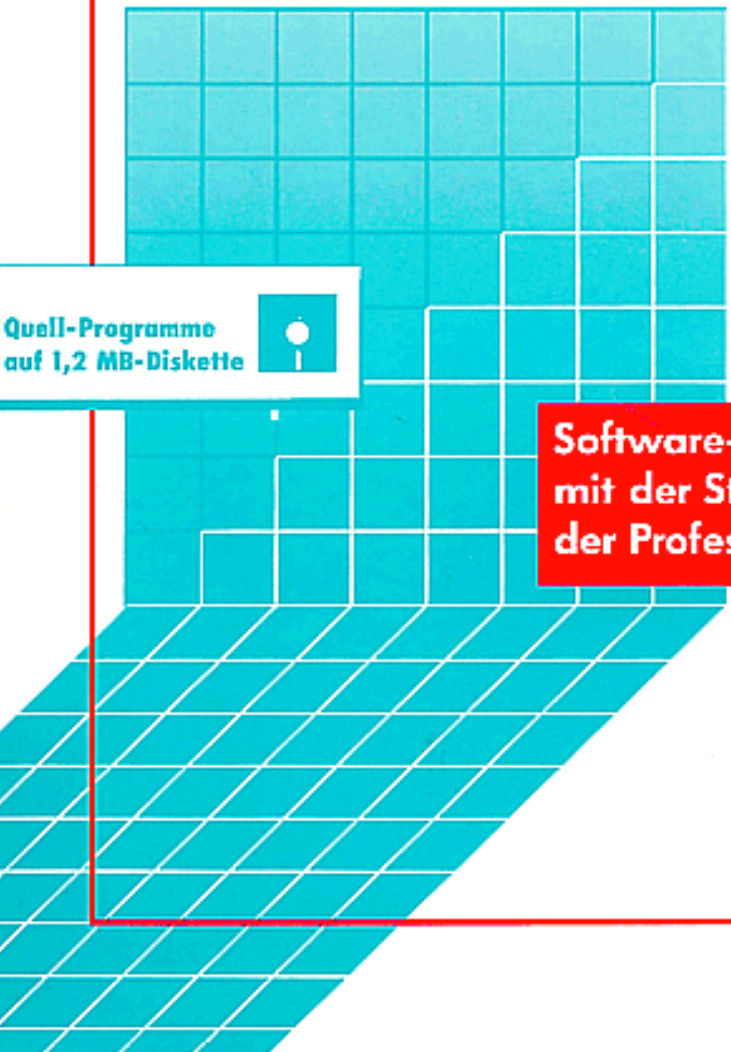
FREDERIK RAMM

PROGRAMMIEREN MIT VISUAL BASIC FÜR DOS



Quell-Programme
auf 1,2 MB-Diskette

Software-Entwicklung
mit der Standard- und
der Professional-Edition

vieweg

Frederik Ramm

Programmieren mit Visual Basic für DOS

Software-Entwicklung mit der Standard- und der Professional-Edition

vieweg

Die Deutsche Bibliothek – CIP Einheitsaufnahme für die 2. Auflage

Ramm, Frederik:
Programmieren mit Visual Basic für DOS : Software-Entwicklung mit der Standard- und der Professional-Edition / Frederik Ramm. – Braunschweig; Wiesbaden: Vieweg, 1993
ISBN 3-528-05338-0

Die vorliegende PDF-Version erschien 2004. Sie ist mit Genehmigung des Autors durch Thomas Antoni (http://www.qbasic.de) erstellt und basiert auf der Papier-Ausgabe, die im Vieweg Verlag (http://www.vieweg.de) verlegt wurde.

Vorwort

Liebe Leserinnen,

bitte haben Sie Nachsicht mit mir, daß Ihr Geschlecht in diesem Buch weitgehend zu kurz kommt und immer nur die Rede von „dem Programmierer“, „dem Benutzer“ und „dem Anwender“ ist. In einem Buch, das Wert auf Prägnanz und Kürze legt, wäre es widersinnig, durch ganze Absätze hindurch die berüchtigten Doppelsubjekt-Querstrichkonstruktionen zu führen, und auch die moderne Schreibweise mit dem großen I im Wort stört den Lesefluß empfindlich. So benutze ich die althergebrachte Schreibweise und bitte um Ihr Verständnis.

„Totgesagte leben länger“ ist ein Spruch, der sich schon mehr als einmal in der Computerwelt bewahrheitet hat (man betrachte nur das Betriebssystem, das 95% von Ihnen höchstwahrscheinlich verwenden).

Auch BASIC ist nicht, wie viele prophezeit und einige gefürchtet haben, vom Markt verschwunden, sondern erfuhr stattdessen vor nun gut einem Jahr mit Visual BASIC für WINDOWS starken Auftrieb. BASIC wurde sozusagen zum Vehikel für ein neues Programmierkonzept, die „ereignisgesteuerte Programmierung“, und kam so (wieder einmal) zu Ehren.

Der Teil der Programmentwicklung, der schon immer überproportional viel Aufwand gekostet und sich durch die Einführung von WINDOWS nochmals stark aufgebläht hatte, nämlich die Aufstellung einer ansprechenden und funktionalen Benutzeroberfläche, wurde mit Visual BASIC fast zum Kinderspiel. Man entwirft seine Oberfläche wie mit einem Grafikprogramm unter Verwendung der Maus und füllt die entstehenden „Schablonen“ später mit dem adäquaten Code, ein Verfahren, das Visual BASIC weit über die Grenzen der BASIC-Anwendergemeinde hinaus zum begehrten Entwicklungswerkzeug machte.

Da nimmt es nicht wunder, daß diese Vorzüge mit Visual BASIC für DOS nun auch demjenigen zur Verfügung gestellt werden, der Programme nicht für

WINDOWS, sondern für den „normalen“ DOS-Betrieb schreibt. Um zu erkennen, daß DOS-Programme noch kein Schnee von gestern sind, braucht man nur einen Blick in die zahllosen Ausbildungseinrichtungen und Unternehmen zu werfen, die heute noch mit 286- oder gar 8086-Computern vor sich hinwerkeln.

Verglichen mit Visual BASIC für WINDOWS gibt es zwar bei der DOS-Version einige Einschränkungen, die vor allem daher rühren, daß Visual BASIC für DOS komplett im Textmodus abläuft; dafür können Sie allerdings mit Visual BASIC für DOS alle Programme unverändert kompilieren, die Sie für BASIC PDS 7.1 oder eine frühere Version geschrieben haben.

Dieses Buch richtet sich in der Hauptsache an BASIC-Programmierer, die schon einige Erfahrung haben und vermittelt eine Vielzahl professioneller Programmiertechniken und -tips; für BASIC-Anfänger oder Umsteiger von anderen Programmiersprachen fasse ich in einem Grundlagenkapitel sehr knapp und bündig das zusammen, was im Rest des Buches als „selbstverständlich“ vorausgesetzt wird.

Der Nachschlageteil am Ende des Buches enthält alle „ereignisgesteuerten“ Neuheiten und alle Spezialitäten der professionellen Version. In Anhang E finden Sie eine komplette Referenz der Standard-BASIC-Befehle. Diese ist auch auf der Diskette zum Buch enthalten. Mit der ebenfalls auf der Diskette beiliegenden Prozedur REFERENZ.BAT können Sie diese Befehlsreferenz an die Online-Hilfe von VBDOS anfügen, so daß Ihnen meine erweiterten Informationen bei der Arbeit mit VBDOS auf Knopfdruck zur Verfügung stehen.

Noch ein Wort zur Sprache: VBDOS kommt nicht nur mit deutschen Handbüchern daher, sondern auch der Editor und (fast) alle Hilfsprogramme sind übersetzt worden. Bei der Übersetzung wurde nach dem Motto „bloß kein englisches Wort übriglassen“ vorgegangen – ein Wahlspruch, den sich viele deutsche Informatiker auf die Fahnen geschrieben haben. Ich sehe das nicht ganz so streng und verwende weiter die englischen Begriffe, wo sie mir angebracht erscheinen: So heißt der Linker bei mir nicht „Binder“, und die DOS-Interrupts bleiben Interrupts, anstatt zu „Unterbrechungen“ zu mutieren. Weitgehend habe ich mich jedoch Microsofts Übersetzungen und deutschen Wortschöpfungen („Form“ mag ja im Englischen sinnvoll sein, aber beim deutschen Wort denke ich eher an Schillers „Glocke“ als an eine Bildschirmmaske) gebeugt, um Ihnen Verwirrungen mit dem Hilfesystem und den deutschen Handbüchern zu ersparen.

Abschließend möchte ich Gerd Hradetzky und Claudia Schwalbach sowie meinem Lektor Robert Schmitz beim Verlag für die konstruktive Kritik und Mitarbeit an diesem Buch danken.

Ich wünsche Ihnen viel Spaß und Erfolg mit Visual BASIC und diesem Buch.

Frederik Ramm

P.S.: Noch ein paar Worte zu den im Buch abgedruckten Listings: Die Ausdrucke stimmen nicht immer genau mit der Diskettenversion überein. Im Buch ist nur das abgedruckt, wovon ich glaube, daß ein Durchlesen sinnvoll ist. Häufig stehen hier mehrere Befehle in einer Zeile, um Platz zu sparen. Auf der Diskette sind alle Programme jedoch komplett enthalten.

Das Zeichen ↩ am Ende einer Zeile bedeutet, daß der Inhalt der Folgezeile noch angefügt werden muß.

Inhalt

Teil I: Der Start mit VBDOS

Teil II: Programmierumgebung & Programmerstellung

Teil VI: Zum Nachschlagen

Anhänge

Neuigkeiten 1

1.1 Ereignisgesteuerte Programmierung

Die ereignisgesteuerte Programmierung ist natürlich *die* Neuerung schlechthin. Im Kapitel 3 finden Sie eine lockere Einführung; Kapitel 7 schildert Details und sinnvolle Programmiertechniken. Das Kapitel 17 beschreibt, wie Sie mit der professionellen Ausgabe Steuerelemente selbst erstellen. Im Kapitel 21 finden Sie eine Beschreibung der gebrauchsfertigen Formen, die mit VBDOS und diesem Buch auf Diskette geliefert werden. Das Kapitel 25 schließlich ist der Objekt-Nachschlageteil, in dem alle Eigenschaften, Ereignisse und Methoden detailliert beschrieben werden.

1.2 VBDOS – Der neue Editor

Wenn Sie die Programmierumgebung VBDOS starten, wird Ihnen – egal, ob Sie vorher mit QBX oder QB gearbeitet haben – das veränderte Erscheinungsbild auffallen. Dem Quellcode steht ein viel kleineres Fenster zur Verfügung, und das bekannte „Immediate“-Fenster fehlt zunächst. Mit Hilfe von Strg + F10 und des Fenster-Menüs können Sie jedoch problemlos das Codefenster vergrößern wieder ein „Direkt“-Fenster öffnen. Im Gegensatz zu früher ist es jetzt möglich, auch eine größere Anzahl von Codefenstern nebeneinander anzuzeigen.

VBDOS kann flexibler als QBX mit dem Speicher umgehen. Es nutzt EMS und XMS und kann außerdem seinen Bedarf an konventionellem Speicher stark reduzieren, indem es Teile von sich auf die Festplatte auslagert. Mehr dazu finden Sie im Kapitel 4.

Eine letzte kleine Änderung betrifft die Eingabe von Anführungszeichen in String-Konstanten. Wollte man früher den Text

```
"Hallo," sagte sie.
```

auf dem Bildschirm ausgeben, mußte man dazu die Programmzeile

```
PRINT CHR$(34); "Hallo,"; CHR$(34); ", sagte sie."
```

verwenden. VBDOS wertet jedoch zwei direkt aufeinanderfolgende Anführungszeichen als ein eingeschlossenes, so daß jetzt auch

```
PRINT """Hallo,"" sagte sie."
```

möglich ist. Um ein einzelnes Anführungszeichen auszugeben, können Sie statt `PRINT CHR$(34)` jetzt auch `PRINT """"` schreiben. Viel Spaß...

1.3 Erweiterungen am Standard-BASIC

Wenn man von den neuen Features der ereignisgesteuerten Programmierung einmal absieht, bleiben fast keine neuen Standard-BASIC-Befehle: Die Funktionen zur Verarbeitung von Zeitcodes (NOW und Konsorten, siehe Kapitel 8), die im BASIC PDS als Add-On-Library enthalten waren, sind jetzt fester Bestandteil der Sprache; ebenso gibt es eine Funktion FORMAT$, die es ermöglicht, Zahlen (fast) so zu formatieren, wie PRINT USING sie ausgibt.

Schließlich ist eine längst überfällige Funktion namens ERROR$ implementiert worden, die zu einem aufgetretenen Fehler oder zu einer beliebigen Fehlernummer den Klartext zurückgibt.

Als letzte, ebenfalls sehr erfreuliche Änderung ist festzustellen, daß Funktionen jetzt mit AS *datentyp* anstelle des zuweilen lästigen Typbezeichners deklariert werden können. Wo es früher hieß

```
FUNCTION Fakultaet&(Zahl AS INTEGER)
```

kann man jetzt

```
FUNCTION Fakultaet (Zahl AS INTEGER) AS LONG
```

schreiben.

1.4 Standard- und professionelle Ausgabe

Schon früher gab es bei BASIC den „feinen Unterschied“ zwischen BASIC 6.0 und dem PDS auf der einen und QuickBASIC auf der anderen Seite. Nie aber mußten Sie sich so deutlich bekennen: Bin ich Profi oder eher „Standard“?

Die professionelle Ausgabe unterscheidet sich von ihrer kleineren Schwester vor allem durch

- die Möglichkeit, Programme mit Overlays zu erstellen (Kapitel 9),
- die eingebaute ISAM-Unterstützung (Kapitel 12);
- mehr Beispielprogramme und Toolboxen (Kapitel 13 und 21);
- die Möglichkeit, eigene Runtime-Module zu erstellen (Kapitel 14);
- die zusätzliche Alternate Math Library für bessere Fließkomma-Rechenleistung auf Rechnern ohne Coprozessor (Kapitel 15);
- die mitgelieferten Programme Profiler, CodeView und Custgen (Kapitel 16 bis 18);
- die Möglichkeit der Code-Optimierung für 286er und 386er Prozessoren (Kapitel 6) und
- einige Verzicht-Files (Kapitel 23).

Die beiden letztgenannten Leistungsmerkmale lassen sich jedoch auf Umwegen auch in der Standardversion nutzen – Details in den genannten Kapiteln.

1.5 Rückschritte gegenüber dem BASIC PDS

Wer vorher mit BASIC PDS gearbeitet hat, wird eventuell einiges vermissen:

- VBDOS bietet keine OS/2-Unterstützung an. Microsoft hat sich ja bekanntlich aus der Entwicklung von OS/2 zurückgezogen und bastelt stattdessen an einem WINDOWS NT, das, wenn es denn je erscheint, ein eigenständiges Betriebssystem sein und mit den Systemressourcen nicht so verschwenderisch wie sein kleiner Bruder umgehen soll (wer's glaubt...).
- In VBDOS kann nicht mehr zwischen Near und Far Strings gewählt werden; VBDOS verwendet grundsätzlich Far Strings.
- Die User Interface-Library des PDS wird, da die ereignisgesteuerte Programmierung viel leistungsfähiger ist, nicht mehr mitgeliefert. Falls sie allerdings einzelne Routinen weiterverwenden möchten – ich denke da z.B. an GetBackground und PutBackground – dürfte das problemlos möglich sein, solange Sie die Far String-Varianten wählen.
- Die ISAM-Unterstützung in Form eines speicherresidenten Programms (PROISAM.EXE oder PROISAMD.EXE) ist für VBDOS weiterhin erforderlich; während Sie beim PDS jedoch die Möglichkeit hatten, auch selbsterstellte Programme so zu gestalten, daß sie die speicherresident geladenen Routinen nutzen, ist das jetzt nicht mehr möglich. Stattdessen werden die ISAM-Routinen bei Kompilierung mit /O in das Programm eingebunden und befinden sich ansonsten im Runtime-Modul. Wenn Sie ein Runtime-Modul ohne ISAM-Unterstützung (NOISAM.OBJ) erzeugen und trotzdem ISAM-Routinen aufrufen, wird ein Fehler erzeugt, egal, ob Sie PROISAMD geladen haben oder nicht.

1.6 Leicht verbesserte Optimierung

Wer von QuickBASIC kommt, dem bietet VBDOS eine verbesserte Code-Erzeugung durch höhere Granularität der Libraries (vgl. Kapitel 23).

Eine Optimierung von Funktions- und Prozeduraufrufen, wie sie im BASIC PDS durch den Switch /Ot aktiviert werden konnte, findet jetzt standardmäßig statt.

Die professionelle Version bietet mit den Switches /G2 (gab's beim PDS schon) und /G3 (ist neu) eine Optimierung für 286er- oder 386er-Prozessoren an.

Grundlagen 2

In diesem Kapitel stelle ich kurz und bündig dar, was Sie beherrschen sollten, um strukturiert mit VBDOS zu programmieren und die weiteren Kapitel zu verstehen und einzusetzen.

2.1 Variablen und Datentypen

Zentrales Element in jedem Programm sind **Variablen**, veränderliche Größen. Je nachdem, welchen **Datentyp** eine Variable hat, kann sie verschiedene Werte annehmen und braucht unterschiedlich viel Speicher. BASIC kennt für Zahlen die numerischen Datentypen INTEGER, LONG, CURRENCY, SINGLE und DOUBLE. Für diese Typen gelten verschiedene erlaubte Zahlenbereiche (siehe Anhang A). Jede Variable hat einen **Variablennamen**, unter dem sie im Programm benutzt wird. Man verwendet entweder reine Variablennamen im Programm und sorgt durch einen DIM- oder einen DEF*xxx*-Befehl dafür, daß der Compiler weiß, welchen Datentyp eine Variable hat, oder man fügt an den Variablennamen einen **Typbezeichner** an, der diesen Typ festlegt (dann muß die Variable im ganzen Programm mit dem Typbezeichner benutzt werden). Die Typbezeichner sind einzelne Zeichen, und zwar für die genannten Datentypen (in gleicher Reihenfolge) %, &, @, ! und #. Neben den numerischen Datentypen existieren noch der Datentyp STRING (Typenbezeichner $), der beliebige Zeichen aufnehmen kann, sowie selbstdefinierte Datentypen. Die letztgenannten heißen in anderen Sprachen meist **Records** und werden im Abschnitt 3 in diesem Kapitel näher behandelt.

2.2 Operatoren

Mittels Operatoren kann man Variablen und Konstanten zu **Ausdrücken** verknüpfen. Es gibt Operatoren mit einem Argument, die meisten haben jedoch zwei Argumente.

Mathematische Operatoren

Mathematische Operatoren sind +, –, *, / für die vier Grundrechenarten, ^ für Potenz , \ für Divisionen mit Ganzzahlen und MOD für Modulo-Rechnung (nur Ganzzahlen). Die Klammern (und) können in Ausdrücken beliebig benutzt werden. BASIC arbeitet bei der Auswertung von Ausdrücken nach den üblichen Regeln der Mathematik (Punkt- vor Strichrechnung, Auswertung von links nach rechts). Im Zweifel sind immer Klammern zu setzen.

Als einziger von allen mathematischen Operatoren kann das Minuszeichen auch monadisch, also mit nur einem Argument eingesetzt werden (nämlich als Vorzeichen einer Zahl oder Variablen). Alle anderen brauchen zwei Argumente.

Bis auf den Operator + sind alle genannten Operatoren nur für numerische Variablen, also Zahlen, zulässig. Der Operator + kann auch für Strings benutzt werden und dient dann dazu, zwei Strings zu einem einzigen zu verbinden.

Vergleichsoperatoren

Vergleichende Operatoren sind =, >, <, <=, >= und <> (für ungleich). = und <> können auf alle Datentypen angewandt werden; die anderen sind nur für numerische Variablen und für Strings zulässig, wobei man sich für das Vergleichen zweier Strings vorstellen kann, daß jedes Zeichen des Strings in einen dreistelligen numerischen ASCII-Code umgewandelt wird und die so entstehenden (unter Umständen über 96.000 Stellen langen) Zahlen verglichen werden. Deshalb ist zum Beispiel der String „B“ „kleiner“ als der String „a“, weil der ASCII-Code von „B“ 65 ist, während „a“ den ASCII-Code 97 hat.

Welchen Wert der Ausdruck `5 + 7` hat, ist nicht schwer zu verstehen, aber welchen Wert ordnet BASIC dem Ausdruck `8 > 4 * 5` zu? Die Antwort ist, daß in BASIC Ausdrücke mit vergleichenden Operatoren (die ja so etwas wie „Behauptungen“ sind) den Wert –1 erhalten, wenn sie wahr sind, und 0, wenn sie falsch sind. Der eben genannte Ausdruck wäre also gleich 0. Aus diesem Grunde bezeichne ich den Wert –1 gelegentlich als TRUE und 0 als FALSE.

Logische und bitweise Operatoren

Logische bzw. bitweise Operatoren sind AND (logisches UND), OR (logisches ODER), XOR (exklusives ODER), EQV (Gleichheit) und IMP (Implikation). Diese Operatoren funktionieren nur mit Ganzzahlen und haben bei der Auswertung eines Ausdrucks die geringste Priorität (`5 * 3 AND 5` wäre also gleichbedeutend mit `(5 * 3) AND 5`).

Zuerst will ich auf die logische Funktion dieser Operatoren eingehen, obwohl sie sich eigentlich als Folge der bitweisen Funktion ergibt. Die logischen Funktionen sind nur dann problemlos anwendbar, wenn die Operatoren ausschließlich mit den Werten TRUE und FALSE, also 0 und –1, benutzt werden. Dann gelten die Wahrheitstabellen auf der folgenden Seite:

x AND y	x = T	x = F
y = T	T	F
y = F	F	F

x OR y	x = T	x = F
y = T	T	T
y = F	T	F

x EQV y	x = T	x = F
y = T	T	F
y = F	F	T

x XOR y	x = T	x = F
y = T	F	T
y = F	T	F

x IMP y	x = T	x = F
y = T	T	T
y = F	F	T

NOT x	x = T	x = F
	F	T

Insbesondere AND und OR werden oft in dieser logischen Funktion gebraucht. Bei dem Befehl `IF (x = 5 OR x > 10) AND NOT y < 7 THEN BEEP` würde nur gepiept, wenn der Gesamtausdruck TRUE ergäbe. Angenommen, x wäre 11 und y wäre 7, dann würde der Ausdruck so ausgewertet:

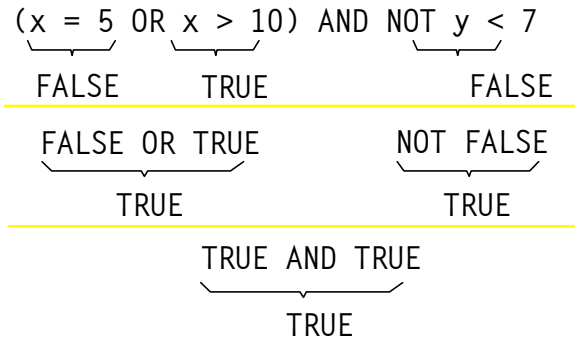


Abbildung 2–1: schrittweise Auswertung eines logischen Ausdrucks

Das Gesamtergebnis wäre also TRUE, und aus dem Lautsprecher ertönte ein (mehr oder weniger häßlicher) Pieps.

Eigentlich haben die genannten Operatoren jedoch eine bitweise Funktion, das heißt, daß die logische Funktion auf jedes Bit der Zahlen x und y angewandt wird (INTEGER-Zahlen bestehen aus 16 Bits, LONG-Zahlen aus 32 Bits; jedes Bit kann entweder 1 oder 0 sein). Weil bei der Zahl –1 alle Bits 1 und bei der Zahl 0 alle Bits 0 sind, klappt alles prima mit unseren TRUE- und FALSE-Werten. Was aber wäre zum Beispiel –131 AND 38?

Dazu müssen wir die bitweise Darstellung beider Zahlen heranziehen:

```
 131       1 1 1 1 1 1 1 1  0 1 1 1 1 1 0 1
  38       0 0 0 0 0 0 0 0  0 0 1 0 0 1 1 0
```

Das erste Bit (ganz rechts) hat den Wert 1, das zweite 2, das dritte 4 usw. Jedes Bit hat den doppelten Wert seines Vorgängers. Das sechzehnte Bit (ganz links) ist eine Ausnahme, es hat den Wert –32.768 statt 32.768. So ergibt sich diese Binärdarstellung der beiden Zahlen.

Verbinden Sie jetzt alle untereinanderstehenden Bits im Geiste mit AND. Sie erhalten:

```
           0 0 0 0 0 0 0 0  0 0 1 0 0 1 0 0
```

Und das ist (0*1+0*2+1*4+0*8+0*16+1*32+0*64...) genau 36. 38 AND –131 hat also den Wert 36.

2.3 Arrays und Records

Neben einfachen Variablen gibt es auch Verbünde von einfachen Variablen, und zwar in der Form von Records und in der Form von Arrays. Beide lassen sich kombinieren.

Arrays

Ein Array ist ein Verbund von lauter Elementen gleichen Typs. Dieser Verbund kann von den Dimensionen her beliebig organisiert werden. Zum Beispiel kann ich 64 INTEGER-Zahlen in einer eindimensionalen Liste von 64 Zahlen unterbringen:

```
DIM Liste(1 TO 64) AS INTEGER
```

Das wäre dann ein eindimensionales Array mit der Dimension 64. Aber ich kann auch eine Tabelle mit 16 Zeilen und vier Spalten erstellen, ein zweidimensionales Array also mit den Dimensionen 16 und 4:

```
DIM Tabelle%(1 TO 16, 1 TO 4)
```

(Natürlich wären auch 8 und 8 möglich, oder 2 und 32...) Auch ein Würfel könnte mein Array beherbergen; das wäre dann ein dreidimensionales Array mit den Dimensionen 4, 4 und 4:

```
DIM Wuerfel(1 TO 4, 1 TO 4, 1 TO 4) AS INTEGER
```

Und so geht es weiter und übersteigt jede Vorstellungskraft, denn ein Array kann bis zu 60 Dimensionen haben. Die Elemente eines Arrays werden behandelt wie gewöhnliche Variablen. Da ein Array aber nicht für jedes seiner (theoretisch bis zu 32.767^{60}) Elemente einen eigenen Namen haben kann, bekommt es nur einen einzigen Namen. Um auf ein bestimmtes Element

zuzugreifen, müssen in Klammern hinter dem Arraynamen die Indizes für jede Dimension angegeben werden. Wurde also ein dreidimensionales Array von 4x4x4 Elementen und dem Namen „Wuerfel“ mit dem oben genannten Befehl deklariert, dann kann man auf ein Element daraus zugreifen, indem man zum Beispiel `PRINT Wuerfel(3, 1, 2)` verwendet.

Der kleinste Index in jeder Dimension muß nicht unbedingt 1 sein. Man könnte beispielsweise auch ein zweidimensionales Array deklarieren, das nur die Zeilen 8 bis 15 und nur die Spalten 1980 bis 2000 enthält:

```
DIM Geschaeftsbericht(8 TO 15, 1980 TO 2000) AS CURRENCY
```

Records

Ein Record ist ein Verbund von Elementen verschiedenen Typs. Die Verwendung von Records setzt das Erstellen eines eigenen Datentyps voraus. Ein eigener Datentyp – er heißt hier im Buch selbstdefinierter Datentyp – ist eine Kombination aus beliebig vielen verschiedenen anderen Datentypen (außer Strings mit variabler Länge) und erhält einen eigenen Namen. Ein selbstdefinierter Datentyp namens „Geburtstagstyp“ könnte zum Beispiel drei INTEGER-Zahlen für Geburtstag, -monat und -jahr sowie einen STRING für den Namen des Geburtstagskindes enthalten. Jedes dieser vier Felder im Geburtstagstyp erhält dann wieder einen Namen. Die Deklaration eines neuen Datentyps sieht wie folgt aus:

```
TYPE Geburtstagstyp
   Name AS STRING * 50
   Tag AS INTEGER
   Monat AS INTEGER
   Jahr AS INTEGER
END TYPE
```

(Innerhalb selbstdefinierter Typen sind nur Strings mit fester Länge möglich, deshalb die Längenangabe * 50.)

Als nächstes ist eine Variable zu erzeugen, der der neue Datentyp zugeordnet wird. Da es für selbstdefinierte Typen keine Typenbezeichner gibt, die man einfach an den Namen anhängen kann, muß man den DIM-Befehl benutzen:

```
DIM Geburtstag AS Geburtstagstyp
```

Auf die einzelnen Elemente kann man jetzt zugreifen, indem man den Namen der Variable nimmt, einen Punkt anhängt und dann den Namen des gewünschten Feldes dazuschreibt. Zum Beispiel:

```
Geburtstag.Monat = 11
```

Kombinationen

Es ist möglich, Arrays in selbstdefinierte Typen einzubauen. Zum Beispiel:

```
TYPE Geburtstagstyp
   Name AS STRING * 50
   Tag AS INTEGER
   Monat AS INTEGER
   Jahr AS INTEGER
   AnzahlGeschenke AS INTEGER
   Geschenk(99) AS STRING * 20
END TYPE
```

Nach

```
DIM Geburtstag AS Geburtstagstyp
```

könnte man dann auf das zehnte Geschenk, das die betreffende Person zum letzten Geburtstag bekommen hat, zugreifen mit

```
PRINT Geburtstag.Geschenk(10)
```

Angenommen, man möchte zu den Geschenken auch noch deren Wert speichern. Dann legt man einen Typ für Geschenke an und verfährt wie folgt:

```
TYPE GeschenkTyp
   Name AS STRING * 20
   Wert AS CURRENCY
END TYPE

TYPE Geburtstagstyp
   Name AS STRING * 50
   Tag AS INTEGER
   Monat AS INTEGER
   Jahr AS INTEGER
   AnzahlGeschenke AS INTEGER
   Geschenk(1 TO 99) AS GeschenkTyp
END TYPE
```

Nach dem entsprechenden DIM-Befehl könnte man nun mit

```
PRINT Geburtstag.Geschenk(10).Wert
```

den Wert des zehnten Geburtstagsgeschenks ermitteln. Um das Ganze auf die Spitze zu treiben, stellen wir uns vor, daß wir die Geburtstage von 99 Menschen in einem großen Array speichern wollen. Dann hieße der DIM-Befehl zum oben genannten Typ:

```
DIM Geburtstag(99) AS Geburtstagstyp
```

Und um auf den Preis des 10. Geschenks der 27. gespeicherten Person zu kommen, gäbe man nun ein:

```
PRINT Geburtstag(27).Geschenk(10).Wert
```

Weitere Experimente überlasse ich Ihrer Phantasie. Aber behalten Sie den zur Verfügung stehenden Speicherplatz im Auge!

2.4 Top-Down-Programmierung und Struktogramme

Mit „Top-Down“ ist nicht etwa gemeint, daß Sie Ihr Programm verächtlich „von oben herab“ betrachten, sondern daß Sie es von oben nach unten entwickeln, also von der höchsten Abstraktionsstufe langsam herabsteigen.

Grau ist alle Theorie...

Wenn Sie zum Beispiel ein Programm schreiben, das mit dem Benutzer „Zahlen raten“ spielt, ist die höchste Abstraktionsstufe einfach: ZahlenRaten. Auf der nächsten Stufe würde man vielleicht schon zwischen zwei Versionen unterscheiden: Entweder der Computer soll eine vom Benutzer ausgedachte Zahl erraten, oder der Benutzer errät eine Zufallszahl des Computers. Man könnte formulieren:

```
DIM Text AS STRING
INPUT "Wer denkt sich eine Zahl aus (Benutzer/Computer)? ", Text
SELECT CASE UCASE$(LEFT$(Text, 1))
CASE "B": ZahlErraten
CASE "C": ZahlErratenLassen
END SELECT
```

Das ist aber dafür, daß wir uns noch auf einer der höchsten Abstraktionsebenen befinden, viel zu konkret. Man sollte versuchen, erst möglichst spät solche BASIC-spezifischen Konstrukte wie oben zu erstellen und vorerst noch mit „Symbolen“ arbeiten, zum Beispiel so:

```
Modus = ModusErfragen
IF Modus = Erraten THEN ZahlErraten ELSE ZahlErratenLassen
```

Aus einem solchen abstrakten Programmteil können später ohne Schwierigkeiten (durch Einführung von einer Konstante namens „Erraten“ und einer Funktion namens „ModusErfragen“) lauffähige BASIC-Zeilen werden.

Auf der dritten Abstraktionsebene teilt sich unser Programm nun schon in die Teile „ZahlErraten“ und „ZahlErratenLassen“. Letzteres könnte zum Beispiel so konkretisiert werden:

```
Zahl = ZahlErmitteln
DO
   RateZahl = ZahlEingeben
   IF RateZahl < Zahl THEN
      Nachricht "Zu klein."
   ELSEIF RateZahl > Zahl THEN
      Nachricht "Zu groß."
   ELSE
      Nachricht "Stimmt!"
   END IF
LOOP UNTIL RateZahl = Zahl
```

Schließlich würden auf einer vierten Ebene die Prozeduren ZahlEingeben, ZahlErmitteln, Nachricht und ModusErfragen programmiert, und damit wäre das Programm lauffähig.

...doch der Praktiker hackt drauflos

Dieses Konzept ist so streng, wie ich es hier vorgeführt habe, in der Praxis kaum vertreten. Das liegt daran, daß Programmieren nicht bloß ein Handwerk, sondern auch eine Kunst ist. Der strenge Top-Down-Ansatz verleugnet jedoch künstlerische Elemente wie Intuition und Ideen. Er stellt die Programmentwicklung als rein logischen, rationalen Vorgang dar – und das ist sie selten.

Trotzdem ist die Top-Down-Arbeitsweise zuweilen ein nützliches Hilfsmittel. Sie erreichen so sehr schnell Programmskelette, die man testen kann. Hierzu verzichten Sie darauf, alle Prozeduren der tieferen Abstraktionsebenen detailliert zu programmieren. Zum Beispiel schreiben Sie in eine Prozedur „LiesDatenAusDatei“, zunächst einfach Code, der immer feste Daten (oder auch Zufallswerte – je nach Anwendung) zurückgibt. Eine Prozedur „ErfrageDateinamen“ könnte zu Testzwecken zunächst immer den gleichen Namen zurückgeben. Die Variablen „Protokolldatei“ und „Druckeranschluß“ könnten während der Testphase zunächst als Konstanten statt als Variablen ausgeführt sein. Die Prozedur „SchreibeInProtokolldatei“ kann zunächst (obwohl sie von überall, wo sie benötigt wird, aufgerufen wird) überhaupt nichts tun und erst später mit Leben gefüllt werden, und so läßt sich die Liste fortsetzen.

Vielen mag ein solches Vorgehen selbstverständlich scheinen und inzwischen in Fleisch und Blut übergegangen sein; andere halten sich aber wirklich während der Konstruktionsphase eines Programms zu sehr mit Details auf, die man leicht vertagen kann. Man sollte zwar erkennen, wo später noch Detailarbeit erforderlich ist, aber anstatt diese Arbeit sofort zu tun, einfach einen Prozeduraufruf einbauen und sich später darum kümmern.

Das Prozedurkonzept ist hier natürlich extrem hilfreich, weil man durch die Verwendung von Prozeduraufrufen, durch dieses „so tun, als gäbe es schon eine Prozedur, die genau das kann, was hier erforderlich ist“, das funktionsfähige Programmskelett später nicht mehr verändert, sondern nur noch die Prozeduren gestaltet.

Struktogramme

Einigen mögen sie verhaßtes Übel sein, für andere sind sie ein nützliches Hilfsmittel zur Gliederung der eigenen Ideen. Insbesondere, wenn man Programme entwerfen möchte, ohne gleich am Rechner zu arbeiten, sind Struktogramme und Flußdiagramme durchaus sinnvoll.

Zwar hat sicherlich jeder schon einmal ein Struktogramm gezeichnet, ohne sich dabei an vorgegebene Symbole zu halten; es gibt jedoch einen Standard, der verblüffend einfach ist, da er mit nur vier Symbolen und vor allem ohne Pfeile auskommt: Die sogenannten „Nassi-Shneiderman-Diagramme“. Für das vorher gezeigte Listing „ZahlErratenLassen“ sähe das Struktogramm danach folgendermaßen aus:

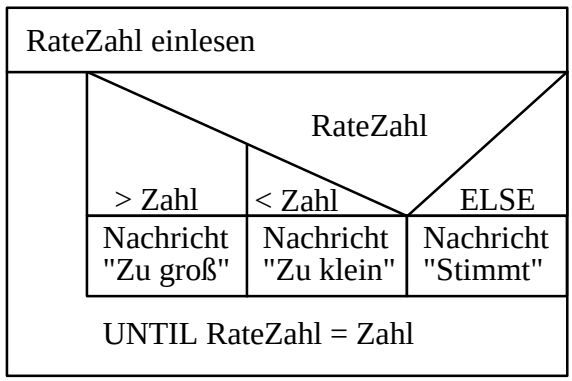
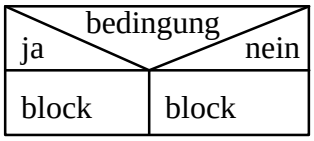
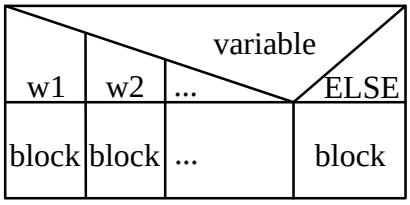


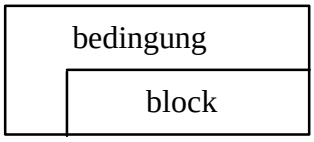
Abbildung 2–2: Nassi-Shneiderman-Diagramm

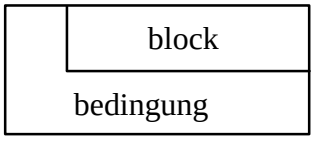
Diese Art von Struktogrammen ist sozusagen rekursiv definiert. Basis ist dabei der „Block“. Ein Block ist ein Rechteck im Diagramm; er kann entweder weitere Blöcke, ein „Atom“ oder eine Struktur enthalten. Ein Atom ist ein nicht weiter unterteilter Befehl oder Prozeduraufruf (in der Abbildung z.B. „RateZahl einlesen“). Eine Struktur ist ein Symbol aus der Liste auf der nächsten Seite, und jede Struktur enthält wiederum mindestens einen Block.

Struktursymbol	*Bezeichnung*	*BASIC-Äquivalent*

	Einfache Bedingung	`IF bedingung THEN`
		`  block`
		`ELSE`
		`  block`
		`END IF`

	Fallunterscheidung	IF-Konstrukt oder
		`SELECT CASE variable`
		`CASE w1: block`
		`CASE w2: block`
		`...`
		`CASE ELSE: block`
		`END SELECT`

	Schleife mit Prüfung am Anfang	`DO WHILE/UNTIL bedingung`
		`  block`
		`LOOP`

	Schleife mit Prüfung am Ende	`DO`
		`  block`
		`LOOP WHILE/UNTIL bedingung`

(Auf den ersten Blick mögen Sie hier ein Äquivalent für BASICs FOR...NEXT vermissen, aber das läßt sich ja auch als Schleife mit Prüfung eines Zählers formulieren.)

Der Kontrollfluß, also die Reihenfolge, in der die Anweisungen ausgeführt werden, verläuft bei dieser Art von Struktogrammen immer von oben nach unten; einzige Ausnahme sind die Schleifensymbole, innerhalb derer ein Block mehrmals ausgeführt werden kann.

Das Faszinierende an diesen Struktogrammen ist, daß man jedes noch so komplexe Programm durch Ineinanderschachteln dieser Symbole ausdrücken kann. Dabei bleibt es einem selbst überlassen, ob man in die „Atome“ schon echten, fertigen BASIC-Code schreibt, oder ob man auf einer abstrakteren Ebene bleibt („RateZahl einlesen“).

2.5 Prozeduren und Funktionen, Kapselung

Die wichtigste Errungenschaft aller Versionen von QuickBASIC und fortgeschrittenen BASIC-Compilern gegenüber GWBasic & Co. ist die Möglichkeit des modularen und damit strukturierten Programmierens.

Prozeduren und Funktionen sind selbständige Programmeinheiten für bestimmte Aufgaben, die im Programm häufiger vorkommen. Früher hat man für solche

Zwecke gerne GOSUB benutzt, das aber nur einen Bruchteil der Fähigkeiten von Prozeduren und Funktionen bietet.

```
GOSUB FrageDateinamen
OPEN DName$ FOR INPUT AS #1
...
FrageDateinamen:
   PRINT "Bitte geben Sie den Dateinamen ein: ";
   LINE INPUT DName$
   RETURN
```

So sieht eine altertümliche, einfache GOSUB-Konstruktion aus. Die „Subroutine“ übergibt nur eine einzige Variable, nämlich *DName$*, an das aufrufende Programm. Als Prozedur formuliert sähe das ganze so aus:

```
FrageDateinamen DName$
OPEN DName$ FOR INPUT AS 31
...
SUB FrageDateinamen(DateiName$)
   PRINT "Bitte geben Sie den Dateinamen ein: ";
   LINE INPUT DateiName$
END SUB
```

Sie bemerken, daß die hier übergebene Variable in der Subroutine anders heißt als beim Prozeduraufruf. Das ist der erste Vorteil von Prozeduren: Von beliebigen Stellen im Programm kann die Prozedur aufgerufen werden, einmal mit *DName$*, einmal mit *a$*, einmal mit *NeuerDateiName$* als Argument, aber in der Prozedur heißt die Variable immer *DateiName$*, und es gibt keine Probleme bei der Übergabe. Man kann natürlich auch mehr als nur eine Variable an eine Subroutine übergeben. Die übergebenen Werte heißen Parameter, oder, wenn man es ganz genau haben will: Innerhalb der Prozedur sind es formale Parameter, außerhalb, also dort, wo die Prozedur aufgerufen wird, sind es Argumente.

Weitere Informationen dazu finden Sie im Disketten-Referenzteil unter den Einträgen zu CALL und zu SUB/FUNCTION.

In unserem Beispiel wäre eine Funktion sogar noch eleganter:

```
OPEN FrageDateinamen FOR INPUT AS #1
...
FUNCTION FrageDateinamen AS STRING
   PRINT "Bitte geben Sie den Dateinamen ein: ";
   LINE INPUT FrageDateiNamen
END FUNCTION
```

Die Verwendung von *FrageDateiNamen* im Programm ruft automatisch die Funktion auf, die dann den eingegebenen Wert zurückgibt.

Lokale Variablen

Ein weiterer großer Vorteil solcher Subroutinen ist, daß sie lokale Variablen haben können. Das bedeutet, daß alle Variablen, die in der Funktion oder Prozedur benutzt werden, völlig unabhängig von denen außerhalb der Funktion oder Prozedur sind. Mit GOSUB war das anders:

```
a% = 80
GOSUB SchreibWas
...
SchreibWas:
FOR a% = 1 TO 20: PRINT "Hallo!": NEXT
RETURN
```

Der Wert 80 in *a%*, der vor dem Aufruf der Routine gespeichert wurde, geht verloren, weil die Routine *a%* für ihre eigenen Zwecke benutzt. In *a%* steht danach vermutlich 21, aber nicht mehr 80.

```
a% = 80
SchreibWas
...
SUB SchreibWas
   FOR a% = 1 TO 20: PRINT "Hallo!": NEXT
END SUB
```

Hier bleibt der Wert von 80 erhalten, denn wir haben mit keinem Befehl erwähnt, daß die Variable *a%* auch der Subroutine zugänglich sein soll, also hat die Subroutine ihr eigenes, lokales *a%* (vgl. „Der Gültigkeitsbereich“ im nächsten Abschnitt).

Globale Variablen

Es gibt insgesamt vier Möglichkeiten, eine Variable nicht lokal zu definieren, sondern dafür zu sorgen, daß sie in der Prozedur wie im Hauptprogramm gleichermaßen benutzt werden kann:

- Man übergibt sie als Parameter (wie *DateiName$* im ersten Beispiel).
- Man erwähnt sie in einem SHARED-Befehl in der betreffenden Prozedur. Dann ist sie nur dieser Prozedur zugänglich.
- Man vereinbart sie im Hauptprogramm mit DIM SHARED. Dann ist sie allen Prozeduren und Funktionen dieses Programms zugänglich.
- Man vereinbart sie im Hauptprogramm mit COMMON SHARED. Dann ist sie nicht nur allen Prozeduren und Funktionen dieses Programms, sondern auch allen Prozeduren und Funktionen anderer Module zugänglich, die denselben COMMON SHARED-Befehl enthalten.

Was im Punkt 4 schon anklingt, ist die Tatsache, daß ein fertiges Programm durchaus verschiedene .BAS-Programme (Module) enthalten kann. So können in einem .BAS-Programm das Hauptprogramm und ein Teil der Prozeduren und Funktionen und in einem zweiten .BAS-Programm der Rest der Prozeduren und Funktionen enthalten sein. Nur dadurch wird es möglich, sehr umfangreiche Programme zu erstellen.

Durch das neue ereignisgesteuerte Programmierkonzept kommen auch noch – neben den .BAS-Codemodulen – die „Formmodule“ hinzu; für jede Form und ihre Ereignisprozeduren wird ein eigenes Modul mit der Erweiterung .FRM erzeugt.

Kapselung

Als Programmierer sollte man stets bemüht sein, all das, was nicht wirklich einer ganz speziellen Aufgabe zuzuordnen ist, „recyclebar“ zu programmieren. Sicherlich haben Sie selbst auch schon öfter beim Programmieren das Gefühl gehabt, etwas ähnliches doch schon einmal aufgestellt zu haben. Und sicher haben Sie sich auch schon das eine oder andere Mal entschlossen, lieber eben die paar Zeilen nochmals einzugeben als jetzt die alte Routine herauszusuchen und all die notwendigen Änderungen vorzunehmen (bzw. erst einmal zu verstehen, was Sie damals eigentlich programmiert haben). Investieren Sie lieber ein bißchen mehr Zeit in die Wiederverwendbarkeit Ihres Codes; Sie werden es später nicht bereuen.

Zu dieser Wiederverwendbarkeit gehören einerseits die vernünftige Archivierung und Dokumentation (auch mit Bemerkungen und „sprechenden“ Variablennamen im Programm), andererseits aber auch einige technische Details:

- Verwenden Sie keine globalen Variablen; wenn unbedingt nötig, verwenden Sie benannte COMMON-Blocks und erstellen Sie eine entsprechende INCLUDE-Datei. Statische lokale Variablen können vielfach anstelle von globalen eingesetzt werden.
- Verwenden Sie in wiederverwendbaren Prozeduren keine festen Dateinummern (`OPEN ... AS #1`), sondern lassen Sie sich mit FREEFILE die nächste freie Dateinummer geben.
- Verwenden Sie lokale Fehlerbehandlungsroutinen (`ON LOCAL ERROR`) und verzichten Sie, wenn möglich, auf ON *event* GOSUB-Aufrufe, da sonst Code auf Modulebene eingefügt werden muß.

Am Ende von Kapitel 21 finden Sie ein Beispiel vorbildlicher Kapselung (die Form „Warten“).

2.6 Der Gültigkeitsbereich

Der Gültigkeitsbereich eines beliebigen Objektes (einer Variable, Konstante, Prozedur, Funktion, Form, eines Steuerelementes oder VB-Spezialobjektes) ist der Bereich des Programms, in dem auf dieses Objekt zugegriffen werden kann. Es ist von Bedeutung, stets die Übersicht über den Gültigkeitsbereich eines Objekts zu haben, um Programmierfehler zu vermeiden. Wenn Sie zum Beispiel nicht wissen, daß eine Konstante nur in dem Modul verwendet werden kann, in dem sie definiert wurde, verwenden Sie vielleicht in einem Modul „TRUE“, in dem diese Konstante gar nicht definiert ist. Sie würden den Fehler wahrscheinlich erst spät bemerken, weil VBDOS „True“ in diesem Modul als Namen einer Variablen auffaßt (die, da nirgendwo ein Wert zugewiesen wird, 0 ist).

Sie können solche Fehler vermeiden, indem Sie OPTION EXPLICIT benutzen. Außerdem können Sie, wenn Sie in VBDOS die F1-Taste benutzen, während der Cursor auf einer Objektbezeichnung steht, erfahren, wo das Objekt wie verwendet wird.

Objekt	*definiert*	*Gültigkeitsbereich*
Variable	mit DIM oder ohne Definition verwendet	die Prozedur/Funktion, in der die Variable auftritt bzw. der Modulcode, wenn sie auf Modulebene auftritt
	mit DIM SHARED oder REDIM SHARED	das gesamte Modul, in dem sie definiert wird (inkl. aller Prozeduren/Funktionen); ausgenommen Prozeduren, die eine gleichnamige Variable mit STATIC oder DIM deklarieren
	mit COMMON	der Modulcode aller Module, in denen der COMMON-Befehl steht (auch Module in Libraries)
	mit COMMON SHARED	Modulcode und alle Prozeduren und Funktionen aller Module, die den COMMON SHARED-Befehl enthalten (auch in Libraries und Quick Libraries); ausgenommen Prozeduren, die eine gleichnamige Variable mit STATIC oder DIM deklarieren
	mit SHARED in einer Prozedur/Funktion	der Modulcode und die Prozedur/Funktion, die den SHARED-Befehl enthält
Prozedur Funktion	in einem Codemodul	das gesamte Projekt (alle Module inkl. Funktionen und Prozeduren); für Funktionen jedoch DECLARE im Modul, das sie verwendet, erforderlich; ausgenommen geladene Quick Libraries
	in einem Formmodul	nur das Modul, in dem sie definiert ist

Objekt	*definiert*	*Gültigkeitsbereich*
Form Steuerelement	in einem Formmodul	das gesamte Projekt (ausgenommen geladene Quick Libraries); in Modulen außer dem, in dem die Form/das Steuerelement definiert ist, muß einem Zugriff auf Eigenschaften oder Methoden der Name der Form vorangestellt sein. Außerdem muß der Metabefehl $FORM verwendet werden.
VBDOS-Spezialobjekte (CLIPBOARD usw.) und Systemvariablen (ERR, TIME$ usw.)		das gesamte Projekt, einschließlich geladener Quick Libraries.

2.7 Dateien

Bevor Sie mit BASIC Änderungen an Dateien vornehmen oder aus diesen auch nur lesen können, müssen sie mit dem OPEN-Befehl geöffnet werden. Dabei wird ihnen eine Dateinummer zugeordnet, durch die sie in Zukunft identifiziert werden. Sie können nun mit verschiedenen Befehlen den Inhalt der Datei manipulieren oder auslesen, immer unter Bezug auf die Dateinummer. Die Bindung von Datei und Dateinummer erlischt mit dem Schließen der Datei durch den CLOSE-Befehl, der unbedingt erforderlich ist, weil unter Umständen erst er Daten wirklich in die Datei schreibt, die schon zu einem früheren Zeitpunkt durch einen BASIC-Befehl „abgeschickt“ wurden.

Das folgende gilt für alle Dateien außer den nur in der professionellen Ausgabe von VBDOS verfügbaren ISAM-Dateien, die in vielem eine Ausnahme bilden und im Kapitel 12 ausführlich behandelt werden.

Beim Öffnen einer Datei müssen Sie den Modus angeben, in dem Sie die Datei öffnen wollen. Durch den Modus legen Sie zugleich fest, um welche Art von Datei es sich handelt. Es gibt drei Typen von Dateien: Sequentielle Dateien, RANDOM-Dateien und BINARY-Dateien. Der Typ ist jedoch nicht in der Datei selbst vermerkt, so daß Sie jede Datei für jeden Typ öffnen können. Manchmal ist es sogar sinnvoll, zum Beispiel eine sequentielle Datei als BINARY-Datei zu öffnen.

Sequentielle Dateien

Sequentielle Dateien sind Dateien, wie sie mit einem einfachen Texteditor erzeugt werden können (man sagt auch ASCII-Dateien, obwohl das nicht so ganz den Punkt trifft). In ihnen sind Daten – meistens Texte – zeilenweise abgespeichert. Würde man eine solche Datei als BINARY-Datei öffnen, so könnte man feststellen, daß am Ende jeder Zeile ein ASCII-Zeichen 13 gefolgt von ei-

nem ASCII-Zeichen 10 steht (der Code CRLF für „neue Zeile“), und daß das letzte Zeichen in der Datei das ASCII-Zeichen 26 ist (der Code für „Dateiende“). Öffnet man die Datei jedoch als sequentielle und bearbeitet sie mit den dafür vorgesehenen Befehlen INPUT, PRINT, LINE INPUT und WRITE, entziehen sich derartige Sonderzeichen dem Zugriff. Der Nachteil an sequentiellen Dateien ist, daß es aufgrund der beliebigen Länge einer Zeile quasi unmöglich ist, beispielsweise die 131. Zeile aus der Datei zu lesen, ohne vorher die davorliegenden 130 Zeilen gelesen zu haben. Zwar gibt es Tricks, um dieses Manko zu umgehen und direkt an bestimmte Stellen der Datei zu springen (die Help-Toolbox von VBDOS – vgl. Kapitel 21 – verwendet SEEK; ebenso das Programmbeispiel „Binäres Suchen in Dateien“ im Kapitel 20), aber dennoch gilt: Sequentiell ist eben nicht „Random Access“, nicht Direktzugriff.

Random Access-Dateien („wahlfreier Zugriff“)

RANDOM-Dateien sind Dateien, die Datenblöcke (Datensätze) mit einer festen Satzlänge aufnehmen. Sie sind gut geeignet, wenn man beispielsweise 100 Variablen (ein Array mit 100 Elementen) eines selbstdefinierten Typs in eine Datei speichern will. Der Vorteil dabei ist die feste Satzlänge, das heißt, daß man mit den dafür vorgesehenen Befehlen (GET und PUT) schnell und problemlos zum Beispiel auf den 131. Datensatz zugreifen kann. RANDOM-Dateien sind aber unflexibel, da sie im Prinzip nur Daten eines einzigen Typs aufnehmen können. Sie ähneln den ISAM-Dateien am meisten, aber bei ISAM-Dateien ist der Programmieraufwand wesentlich geringer, weil ISAM gleich das Sortieren und Verwalten der Daten übernimmt, während eine RANDOM-Datei vom Programmierer verwaltet werden muß.

Man kann in eine RANDOM-Datei auch verschiedene Typen schreiben, allerdings nur, wenn sie die gleiche Länge haben; alles in allem sind für differenzierte Anwendungen die BINARY-Dateien am geeignetsten.

Binäre Dateien

BINARY-Dateien schließlich bieten dem Programmierer die größte Freiheit. Das impliziert allerdings, daß er sich beim Umgang mit diesen Dateien um alles selbst kümmern muß. Aus einer BINARY-Datei kann man von einer beliebigen Position ab beliebig viele Bytes in eine beliebige Variable lesen oder aus einer ebenso beliebigen Variablen in die Datei schreiben. Für viele Zwecke sind BINARY-Dateien die beste Lösung, und RANDOM-Dateien werden angesichts der nur wenig aufwendigeren, dafür aber flexibleren BINARY-Dateien im Prin-

zip überflüssig. BINARY-Dateien gleichen in etwa RANDOM-Dateien mit einer Satzlänge von 1 Byte.

Die BINARY-Methode sollte man außerdem wählen, wenn man in einer bereits vorhandenen Datei unbekannten Formats „herumpfuschen“ will, wenn es zum Beispiel darum geht, eine Programmdatei nach einem bestimmten Text zu durchsuchen, einen bestimmten Text irgendwo einzutragen, ein fremdes Dateiformat zu konvertieren oder einfach eine Datei zu kopieren.

2.8 Grafik und Text – Der Bildschirm

Textmodus

Wenn ein Programm gestartet wird, befindet sich das System zunächst üblicherweise im Textmodus. Der Textmodus ist eine Betriebsart des Bildschirms und der Grafikkarte, die von jedem System unterstützt wird. Im Textmodus kann an jeder Bildschirmposition – je nach Grafikkarte, Bildschirm und Verwendung des WIDTH-Befehls gibt es 80x25, 80x43 oder 80x50 Positionen – genau eines der 256 Zeichen des ASCII-Zeichensatzes in genau einer von 256 möglichen Vorder- und Hintergrundfarbkombinationen dargestellt werden.

Der Textmodus hat den Vorteil, daß er wenig Ansprüche an die Hardware stellt. Zumindest für den 80x25-Modus läßt sich sagen, daß es kein IBM-kompatibles System gibt, das diesen Modus nicht benutzen kann (bei den Farben gibt es natürlich Einschränkungen).

Der Textmodus verbraucht außerdem sehr wenig Speicher. Da ein ganzer 80x25-Bildschirm insgesamt 2.000 Positionen hat, auf denen je eines von 256 Zeichen in je einer von 256 Farben stehen kann, benötigt seine Speicherung insgesamt nur 4.000 Bytes. Im Textmodus ist es einfach, mit Fenstern zu arbeiten und Bildschirmteile aus- und wieder einzublenden, weil der Speicherbedarf äußerst gering ist.

Im Textmodus sind fast immer mehrere Bildschirmseiten verfügbar; dadurch kann man Daten auf „unsichtbare“ Bildschirme schreiben und in Sekundenschnelle diese Bildschirme sichtbar machen (siehe dazu SCREEN-Befehl im Disketten-Referenzteil). Auch im Textmodus können Grafiken dargestellt werden, solange man sich auf die verfügbaren sogenannten Blockgrafikzeichen beschränkt. In puncto Balkengrafiken etc. läßt sich damit schon erstaunlich viel ausrichten, wie viele ausgetüftelte Programme beweisen. Obwohl es schwieriger ist, mit den Blockgrafikzeichen Grafiken zu erzeugen, und obwohl nur einfache Grafiken damit machbar sind (eine Kuchengrafik zum Beispiel ist bereits

unmöglich), haben diese den Vorteil, daß sie auf jedem Rechner laufen und schnell aufgebaut werden können.

Eine Übersicht über alle ASCII-Zeichen, darin eingeschlossen die Blockgrafikzeichen, finden Sie im Anhang D. Diesen Standard-Zeichensatz kann man bei den meisten Grafikkarten umprogrammieren. BASIC bietet dafür zwar direkt keine Methoden an, über Interruptaufrufe läßt sich so etwas jedoch realisieren. Auf diese Weise kann man auch beliebige „Grafik“ im Textmodus anzeigen; einige DOS-Utility-Programme (PC-Tools, Norton Utilities) bedienen sich dieser Technik.

Die Steuerelemente und Formen der ereignisgesteuerten Programmierung können bei VBDOS nur im Textmodus verwendet werden.

Grafikmodus

Im Grafikmodus kann jedes Pixel, also jeder Grafikpunkt auf dem Bildschirm, einzeln angesprochen werden. Bei Pixeln handelt es sich allerdings nicht wirklich um die einzelnen Rasterpunkte des Bildschirms, sondern um eine Einheit, die von der Grafikkarte bestimmt wird. Die maximale Auflösung, die Anzahl der Pixelspalten und -zeilen auf dem Bildschirm, hängt, ebenso wie die Anzahl der Farben, von Bauart und Speicherkapazität der Grafikkarte ab. Für eine Auflösung von 640x200 Punkten in schwarz/weiß (also ein Bit pro Punkt) werden schon 16.000 Bytes benötigt, viermal so viel wie für einen farbigen Textbildschirm. Eine bessere Auflösung von 640x480 Punkten in 16 Farben (vier Bits pro Pixel) benötigt bereits 145.600 Bytes.

Die verschiedenen möglichen Grafikmodi sind vom benutzten System abhängig (mehr dazu siehe SCREEN-Befehl). VBDOS unterstützt keine Super-VGA-Auflösungen (800x600 oder höher); hat man damit zu arbeiten im Sinn, muß man sich ein wenig mit Assembler auskennen und die Karten über Interrupts und den Videospeicher direkt ansprechen.

Im Grafikmodus lassen sich also nicht so schnell wie im Textmodus Bildbereiche abspeichern, verschieben oder verändern. Dafür bestehen viel umfassendere Möglichkeiten, den Bildschirminhalt zu gestalten.

Spezielle Grafikbefehle erleichtern die Erzeugung von Grafiken sowie das Kopieren und Verschieben von Ausschnitten. Mit der Font-Toolbox (professionelle Ausgabe) kann sogar Text in verschiedenen Größen und Schriftarten ausgegeben werden.

Diese Vorteile bezahlt man damit, daß der Grafikmodus langsamer ist und daß ein für einen bestimmten Grafikmodus geschriebenes Programm nur auf Rechnersystemen funktioniert, die denselben Grafikmodus unterstützen. Natürlich

lassen sich Programme entwickeln, die sich an jeden Grafikmodus anpassen – so wie die Presentation Graphics- und die Font-Toolbox aus der professionellen Ausgabe – aber das ist mit großem Aufwand und viel Testarbeit verbunden.

2.9 WINDOWS

Die „Betriebssystemerweiterung“ WINDOWS ist in aller Munde und – inzwischen, nach einigen Anläufen – auch auf fast allen Festplatten. Es gibt zwei Möglichkeiten, die Programme, die Sie mit VBDOS geschrieben haben, unter WINDOWS laufen zu lassen.

VBDOS-Programme als DOS-Anwendung unter WINDOWS

Einerseits kann Ihr VBDOS-Programm so, wie es erstellt wurde (als EXE-Datei) von WINDOWS aus aufgerufen werden. Dann wird es wie eine gewöhnliche DOS-Anwendung behandelt. Alle Grafikmodi funktionieren normal, keinerlei Änderungen am Programm sind notwendig. Das Programm läuft gewöhnlich im Vollbild-Modus und ist auch optisch identisch mit der direkt aus DOS gestarteten Version. Probleme kann es allerhöchstens geben, wenn Ihr Programm sehr viel Speicherplatz benötigt (da WINDOWS selbst auch einen Teil des Speichers belegt), oder wenn Ihr Programm eine Tastenkombination verwendet, die unter WINDOWS belegt ist (Alt-Leertaste, Alt-Tab und einige andere). Solche Probleme lassen sich meistens durch Anfertigen einer WINDOWS-PIF-Datei (Program Information File) beheben, in der man eintragen kann, wieviel Speicherplatz das Programm benötigt und welche Tastenkombinationen reserviert werden sollen.

Außerdem besteht, sofern WINDOWS im erweiterten 386er-Modus läuft und die Bildschirmauflösung hoch genug ist, die Möglichkeit, das Programm als DOS-Programm in einem Fenster unter WINDOWS laufenzulassen. Die Maus kann dann von Ihrem Programm nicht verwendet werden; trotzdem funktioniert der Textmodus in jedem Fall, und die Grafikmodi sind je nach WINDOWS-Einstellung auch verfügbar.

Solange Ihr Programm als DOS-Anwendung unter WINDOWS läuft, brauchen Sie am Quellcode überhaupt nichts zu verändern; es ist allein der Benutzer, der bestimmt, wie er das Programm starten will.

Mittels der ENVIRON$-Funktion (vgl. Referenzteil auf Diskette) läßt sich feststellen, ob Ihr Programm aus WINDOWS heraus aufgerufen wurde oder nicht.

VBDOS-Programme nach VBWIN portieren

Wenn Sie hingegen alle Vorteile von WINDOWS nutzen möchten – also die größenveränderlichen und verschiebbaren Fenster, das gleichartige Aussehen aller Programme, die Möglichkeiten, mit anderen WINDOWS-Programmen zu kommunizieren und Daten auszutauschen usw. – dann müssen Sie von VBDOS Abschied nehmen und Ihr Programm unter VB für WINDOWS kompilieren. VBDOS macht Ihnen diesen „Abschied“ leicht, denn mit dem ereignisgesteuerten Programmierkonzept können Sie schon unter VBDOS Programme im „WINDOWS-Stil“ erstellen, die dann fast ohne Änderungen nach VBWIN übernommen werden können.

Natürlich haben Sie in VBWIN mehr Möglichkeiten als mit der ereignisgesteuerten Programmierung von VBDOS: Da WINDOWS stets im Grafikmodus läuft, können Sie auf Formen auch beliebige Grafiken, Zeichnungen und Symbole darstellen. Sie können die Steuerelemente wesentlich genauer positionieren, verschiedene Schriftarten auswählen und Druckausgaben machen, ohne sich um den angeschlossenen Drucker zu kümmern. Dafür müssen Sie WINDOWS aber auch einige Vorteile des VBDOS opfern.

Der Disketten-Referenzteil enthält genaue Informationen über die Kompatibilität der einzelnen Befehle; das Kapitel 24 beschreibt einige Richtlinien für die „WINDOWS-freundliche“ Programmierung in VBDOS.

Einführung in die ereignisgesteuerte Programmierung 3

Im Vorwort ist bereits angeklungen, daß die ereignisgesteuerte Programmierung die wesentliche Neuerung von Visual BASIC für DOS gegenüber früheren BASIC-Versionen darstellt. Unter „Ereignis“ versteht man dabei – von wenigen Ausnahmen abgesehen – eine Handlung des Benutzers, der mit Ihrem Programm arbeitet.

Wie aber äußert sich dieses neue „Paradigma“, diese Programmiertechnik, in einem ganz gewöhnlichen Alltagsprogramm? Dieses Kapitel führt im Tutorial-Stil und anhand eines Beispiels in die ereignisgesteuerte Programmierung ein.

Wenn Sie einige Erfahrung mit BASIC besitzen, haben Sie sicherlich schon einmal ein Programm geschrieben, das dem Benutzer ein Auswahlmenü präsentiert und dann, entsprechend seiner Wahl, eine Prozedur oder einen Programmteil aufruft. Stark vereinfacht könnte das so ausgesehen haben:

```
DIM Taste AS STRING * 1
PRINT "Hauptmenü"
PRINT "1   Daten erfassen"
PRINT "2   Datenausgabe"
PRINT "3   Listenausdruck"
PRINT "0   Programmende"
PRINT
PRINT "Bitte wählen Sie!"
DO
   Taste = UCASE$(INPUT$)
   SELECT CASE Taste
   CASE "1": DatenErfassen
   CASE "2": DatenAusgabe
   CASE "3": ListenAusdruck
   CASE "0": EXIT DO
LOOP

SUB DatenErfassen
   ...
END SUB
```

```
SUB DatenAusgabe
   ...
END SUB

SUB ListenAusdruck
   ...
END SUB
```

Dieses Programm besteht im wesentlichen aus zwei Teilen: Der Menüroutine, die auf Eingaben des Benutzers reagiert, und den eigentlichen Arbeitsroutinen, die die gewünschte Aktion ausführen. Die Menüroutine benötigt hier nur wenige Zeilen, aber es ist offensichtlich, daß sie recht kompliziert werden kann, wenn man nun noch für eine ansprechende Optik, für die Möglichkeit der Mausbedienung usw. sorgen will.

Wenn Sie stattdessen die ereignisgesteuerte Programmierung verwenden wollen, um solch ein Menü zu schaffen, müssen Sie nur noch die Arbeitsroutinen wirklich programmieren. Den Rest erledigen Sie quasi als Zeichner: Mit dem Form-Designer erstellen Sie eine sogenannte „Form“, das ist ein Fenster auf dem Bildschirm, das Objekte enthalten kann, und arrangieren vier sogenannte „Schaltflächen“ darauf.

Das Ergebnis sieht nach wenigen Schritten etwa so aus (abgebildet ist die Form während der Bearbeitung im Form-Designer; die Punkte verschwinden später):

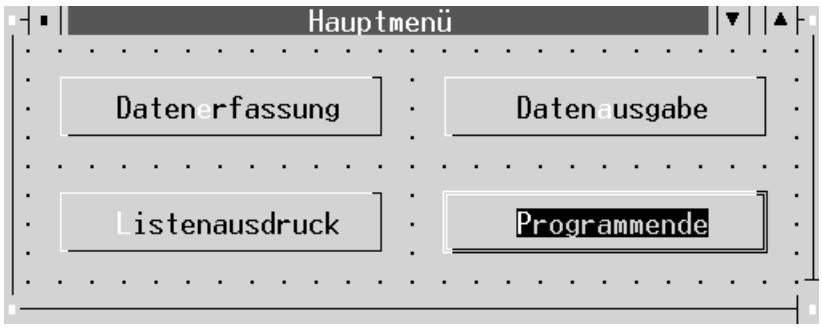


Abbildung 3–1: Vier Schaltflächen auf einer Form im Form-Designer

(Natürlich haben Sie vielfältige Möglichkeiten, das Aussehen zu beeinflussen, und später werde ich noch ganz andere Methoden vorstellen, ein Menü aufzubauen. Dies also nur als einführendes Beispiel.)

Die Form wird automatisch in einem sogenannten „Formmodul“ gespeichert. Das zugehörige Programm ist trivial:

```
SUB Datenerfassung_Click
   ...
END SUB
SUB Datenausgabe_Click
   ...
END SUB
SUB Listenausdruck_Click
   ...
END SUB
SUB Programmende_Click
   END
END SUB
```

Wenn Sie das Programm starten, wird die Form auf den Bildschirm gebracht. Der Benutzer kann nun mit Pfeil- oder Tab-Taste, mit den hervorgehobenen Buchstaben oder mit der Maus eine der vier Auswahlen treffen, und Visual BASIC ruft dann automatisch die zugehörige Click-Prozedur auf.

Im Gegensatz zu einem klassischen BASIC-Programm enthält das oben abgedruckte ereignisgesteuerte Programm keine Schleife, in der auf irgendetwas gewartet wird. Bei einem klassischen Programm kann man jederzeit unterbrechen und sieht dann den Befehl, der gerade ausgeführt wurde; ein ereignisgesteuertes Programm (zumindest ein so einfaches) wartet auf Ereignisse, ohne daß Sie dafür eine Schleife vorsehen.

3.1 Steuerelemente

„Datenerfassung“, „Datenausgabe“, „Listenausdruck“ und „Programmende“ sind in unserem Beispielprogramm *Steuerelemente*, und zwar alles S*chaltflächen*. Welche Steuerelemente es in einer Form gibt und wie sie heißen sollen, legen Sie bei der Gestaltung der Form im Form-Designer fest. Die verschiedenen Steuerelement-Typen sind hingegen von Visual BASIC vorgegeben. (Später werde ich allerdings eine Möglichkeit vorstellen, eigene Steuerelemente zu implementieren).

Neben den Befehlsschaltflächen gibt es noch 14 weitere Steuerelement-Typen, darunter zum Beispiel:

- Kontrollfelder, die entweder „an-“ oder „aus-“geschaltet sein können;
- Optionsfelder, bei denen aus einer Mehrzahl von Möglichkeiten eine ausgewählt werden muß;
- Listenfelder, bei denen aus einer Liste von Text ein Eintrag ausgewählt wird;
- Textfelder, in die ein beliebiger Text eingetragen werden kann;

- Dateilistenfelder, die wie Listenfelder funktionieren, aber von selbst eine Liste aller vorhandenen Dateien anzeigen.

Steuerelemente sind eine Untermenge der *Objekte*; zu den Objekten zählen neben Steuerelementen auch Formen (die ja nichts weiter als ein „Container“ für Steuerelemente sind) und die drei Spezialobjekte CLIPBOARD, PRINTER und SCREEN, die ich später im Detail beschreibe.

Ein Steuerelement hat einen eigenen Datenbereich, in dem seine Eigenschaften gespeichert werden. Die möglichen Eigenschaften eines Steuerelements hängen von seinem Typ ab. Ein Steuerelement kann auf bestimmte Ereignisse reagieren, indem es eine Ereignisprozedur aufruft.

3.2 Eigenschaften

Ein Objekt hat eine Vielzahl von *Eigenschaften*. Darunter fallen zum Beispiel die Position, die Größe, die Farbe und die Art des Rahmens. Der Form-Designer setzt alle Eigenschaften auf die Standardwerte, wenn Sie ein neues Steuerelement einfügen, und erlaubt Ihnen, die Eigenschaften schon bei der Erstellung der Form zu verändern. Im Beispiel habe ich lediglich die Werte der Eigenschaften *Caption* (der Text, der im Steuerelement angezeigt wird) und *CtlName* (der Name des Steuerelements) geändert, und zwar habe ich für die erste Befehlsschaltfläche beide Eigenschaften auf „Datenerfassung“ gesetzt, für die zweite beide auf „Datenausgabe“ etc.; ferner habe ich die Eigenschaft *Default* der vierten Schaltfläche auf TRUE eingestellt und so den doppelten Rahmen erreicht.

Jede Eigenschaft eines Objektes kann in den meisten Fällen wie eine Variable verwendet werden; es ist jedoch nicht möglich, sie „by reference“ an eine Prozedur oder Funktion zu übergeben. Stattdessen müßte man entweder die Prozedur/Funktion so deklarieren, daß sie einen BYVAL-Parameter erwartet, oder man sorgt durch das Setzen von Klammern um die Eigenschaftsbezeichnung dafür, daß sie als Ausdruck behandelt wird.

Welche Eigenschaften ein Steuerelement eines bestimmten Typs hat, ist von Visual BASIC vorgegeben. Einige Eigenschaften müssen unbedingt schon beim Programmentwurf feststehen; die meisten können jedoch auch später, wenn das Programm läuft, geändert werden. Dabei greift man auf die Eigenschaften zu wie auf Elemente eines selbstdefinierten Datentyps:

```
Programmende.Caption = "Programm verlassen"
```

Diese Zeile würde, irgendwo im Beispielprogramm auftauchend, die Beschriftung des „Programmende“-Knopfes in „Programm verlassen“ ändern.

Neben der *CtlName*- und der *Caption*-Eigenschaft gibt es noch etwa 60 weitere, die jedoch jeweils nicht für alle Typen von Steuerelementen verfügbar sind. Zum Beispiel wird durch Eigenschaften festgelegt:

- die Position und Größe;
- die Rahmenart;
- die Farbe;
- das Aussehen des Mauszeigers, während er sich über dem Steuerelement befindet;
- ob das Objekt mit der Maus verschoben werden kann;
- ob das Objekt als ausgewählt gilt, wenn der Benutzer ESC drückt;
- ob das Objekt überhaupt sichtbar ist;
- ob das Objekt mit der Tab-Taste anwählbar sein soll.

3.3 Ereignisse

„Click“ ist ein *Ereignis*, das für eine Befehlsschaltfläche eintreten kann. Es gibt noch etwa 20 weitere Ereignisse, aber nicht jedes Objekt kann alle Ereignisse auslösen.

Prozeduren, die geschrieben werden, um auf ein Ereignis zu reagieren, heißen *Ereignisprozeduren.* Alle vier Prozeduren unseres Programms sind solche Ereignisprozeduren.

Der Name einer Ereignisprozedur besteht aus dem Namen eines Objektes, einem Unterstrich (_) und dem Namen eines Ereignisses, das für dieses Objekt eintreten kann. Die so benannte Ereignisprozedur wird dann automatisch aufgerufen, wenn das betreffende Ereignis für das genannte Objekt eintritt.

Bestimmte Ereignisse werden zum Beispiel ausgelöst, wenn

- der Benutzer eine Taste drückt oder losläßt;
- mit der Maus ein Klick oder Doppelklick ausgeführt wird;
- ein Objekt mit der Maus verschoben wird;
- der Benutzer den Text in einem Textfeld verändert;
- ein Objekt den „Fokus“ erhält (dadurch, daß der Benutzer es mit der Tab-Taste oder der Maus anwählt) oder verliert.

Sie müssen nicht für alle Ereignisse eine Ereignisprozedur schreiben; in unserem Beispiel hatten wir ja auch nur das Ereignis „Click“ verwendet. Ausgelöste Ereignisse verhallen ungehört, wenn keine Ereignisprozedur für sie existiert.

3.4 Methoden

Methoden kommen in unserem Beispielprogramm noch nicht vor. Methoden sind festgelegte Befehle, die auf ein Objekt angewandt werden können. Es gibt nur wenige Methoden, und die meisten sind auch gar nicht für alle Objekte verfügbar.

Die REFRESH-Methode zum Beispiel zeichnet ein Objekt neu; um in unserem Programm den „Programmende“-Knopf neu zeichnen zu lassen, würde man den Befehl `Programmende.REFRESH` verwenden.

Methoden können nicht selbst programmiert werden; man ist auf die vorgegebenen angewiesen.

3.5 Weitere Termini

Zum Abschluß dieser ersten Einführung noch einige Begriffdefinitionen, die Ihnen beim Verständnis der späteren Kapitel helfen werden:

Fokus

Eine Form kann mehrere Steuerelemente enthalten, und am Bildschirm können auch gleichzeitig mehrere Formen angezeigt werden. Deshalb muß VBDOS immer genau wissen, auf welches Objekt sich eine Aktion des Benutzers (z. B. das Drücken einer Taste) bezieht.

Zu diesem Zweck ist immer ein Objekt aktuell – es „hat den Fokus“. Dieses Objekt ist gewöhnlich speziell hervorgehoben; eine Befehlsschaltfläche, die den Fokus hat, wird zum Beispiel mit doppeltem Rahmen angezeigt.

Der Benutzer kann den Fokus mit der Maus oder der Tab-Taste auf ein anderes Objekt setzen. Dann lösen das Objekt, das den Fokus verliert, und das, das ihn erhält, ein spezielles Ereignis (*LostFocus* bzw. *GotFocus*) aus; durch Programmierung einer Ereignisprozedur hierfür können Sie z. B. überprüfen, ob der Benutzer in einem Textfeld, das er gerade verlassen will, gültige Eingaben gemacht hat.

Außerdem kann der Fokus nicht nur vom Benutzer, sondern auch vom Programm aus auf ein bestimmtes Objekt gesetzt werden: Hierzu bedient man sich der SETFOCUS-Methode.

Gebunden – ungebunden

Eine Form, die „gebunden“ angezeigt wird, stoppt die Programmausführung so lange, wie sie auf dem Bildschirm sichtbar ist. Es können weder andere Formen (die vielleicht ebenfalls sichtbar sind) aktiviert werden, noch geht die Programmausführung hinter dem SHOW-Befehl, der das Anzeigen der Form verursachte, weiter. Dazu muß die Form erst wieder geschlossen werden. Eine ungebundene Form hingegen wird auf den SHOW-Befehl hin zwar angezeigt, aber die Programmausführung wird sofort hinter SHOW fortgesetzt. Benutzt werden kann die ungebundene Form nur, wenn DOEVENTS aufgerufen wird, oder wenn das Programm zu Ende ist (dann nämlich wird nicht ins DOS zurückgesprungen, solange noch Formen sichtbar sind). Wenn mehrere ungebundene Formen angezeigt werden, kann der Benutzer in beliebiger Folge Steuerelemente aus allen Formen aktivieren.

Projekt – Startdatei

Zu einem *Projekt* gehören alle Dateien, die zum Schluß gemeinsam ein EXE-Programm ergeben sollen. Das können beliebig viele Form- und Codemodule (.FRM, .BAS) sein. VBDOS führt, wie schon das BASIC PDS, die Namen aller beteiligten Module in einer Datei mit der Extension .MAK auf, um zu wissen, welche Dateien geladen werden müssen, wenn das Projekt kompiliert werden soll.

Die *Startdatei* ist die erste Datei in dieser Projektliste; sie kann in VBDOS mit dem Befehl „Startdatei festlegen“ aus dem Menü „Ausführen“ bestimmt werden. Wenn Sie auf der Befehlszeile „von Hand“ kompilieren (vgl. Kapitel 6), ist die Startdatei diejenige, die Sie beim LINK-Befehl zuerst angeben.

Die Startdatei kann ein Form- oder ein Codemodul sein.

Formmodul – Codemodul

Ein *Codemodul* ist ein ganz gewöhnliches BASIC-Programm, das Prozeduren, Funktionen und Code auf Modulebene enthalten darf. Es wird mit der Extension .BAS gespeichert. Alle Programme, die Sie vor VBDOS mit BASIC geschrieben haben, sind Codemodule.

Jede Form, die Sie erzeugen, wird mit allen ihren Ereignisprozeduren in einer eigenen Datei gespeichert. Eine solche Datei heißt nicht .BAS, sondern hat die Erweiterung .FRM und wird *Formmodul* genannt.

Ein Formmodul enthält immer genau eine Form und darf dazu beliebig viele Ereignisprozeduren sowie andere („gewöhnliche“) Prozeduren und Funktionen

enthalten. Alle Prozeduren und Funktionen in einem Formmodul sind jedoch „privat“.

Öffentlich – privat

Öffentliche Prozeduren und Funktionen sind solche, die von jedem Modul innerhalb eines Projektes aufgerufen werden können (vorausgesetzt, das Modul enthält einen entsprechenden DECLARE-Befehl). Alle Prozeduren und Funktionen in Codemodulen sind öffentlich. Deshalb dürfen in demselben Projekt keine zwei Codemodule vorkommen, die eine Prozedur mit gleichem Namen definieren.

Private Prozeduren und Funktionen dürfen nur innerhalb des Moduls aufgerufen werden, in dem sie definiert sind, nicht aber von einem anderen Modul. Alle Prozeduren und Funktionen – inklusive der Ereignisprozeduren – in einem Formmodul sind privat. Deshalb darf zum Beispiel jedes Formmodul die Prozedur *Form_Load* enthalten, was bei Codemodulen nicht erlaubt wäre.

Modulebene – Prozedurebene – Beginn der Programmausführung

„Code auf *Modulebene*“ sind (ausführbare) BASIC-Befehle, die sich nicht in einer Prozedur oder Funktion befinden. Nur Codemodule dürfen Code auf Modulebene enthalten; Formmodule sind auf die Prozedurebene beschränkt (sie dürfen allenfalls nicht ausführbare Befehle auf der Modulebene haben).

Modulcode wird nur ausgeführt, wenn er

- in der Startdatei steht (diese muß dann ein Codemodul sein) oder
- durch einen ON *event* GOSUB- oder einen ON ERROR GOTO-Befehl direkt angesprungen wird.

Ist die Startdatei ein Codemodul, dann beginnt die Programmausführung bei der ersten Zeile des Modulcodes der Startdatei. Ist die Startdatei ein Formmodul, das keinen Modulcode haben darf, beginnt die Ausführung sofort mit der Anzeige der Form, so, als wäre die Methode SHOW für die Form aufgerufen worden.

Ausführbarer Code

Als *ausführbar* gelten alle BASIC-Befehle, die nicht schon beim Kompilieren, sondern erst beim Programmablauf wirken. Alle ausführbaren Befehle verlängern das EXE-Programm; bei nicht ausführbaren Befehlen ist das nicht notwendigerweise der Fall. Folgende Befehle gelten als *nicht* ausführbar:

COMMON, CONST, DATA, DECLARE, DEF*xxx*, DIM, $DYNAMIC, $FORM, $INCLUDE, OPTION BASE, OPTION EXPLICIT, REM, SHARED, $STATIC, STATIC, TYPE...END TYPE

Der DIM-Befehl ist jedoch ausführbar, wenn er benutzt wird, um ein dynamisches Array zu deklarieren (dies kann nämlich erst zur Laufzeit geschehen).

Die Entwicklungsumgebung VBDOS 4

Dieses Kapitel beginnt mit einigen Bedienungshinweisen zur Entwicklungsumgebung VBDOS und erklärt die wichtigen Menüpunkte und Tastenbelegungen. Später folgen einige Techniklen zur Fehlersuche sowie eine kurze Anleitung zur Erstellung von EXE-Dateien mit VBDOS.

4.1 Übersicht

Der Editor der Entwicklungsumgebung VBDOS ist optimal auf die Bedürfnisse des BASIC-Programmierers zugeschnitten. Er kompiliert eingegebene Programmzeilen sofort vor und bemerkt viele Fehler gleich bei der Eingabe. Er sorgt automatisch für eine einheitliche Schreibung der Variablen und wandelt BASIC-Befehle in Großbuchstaben um.

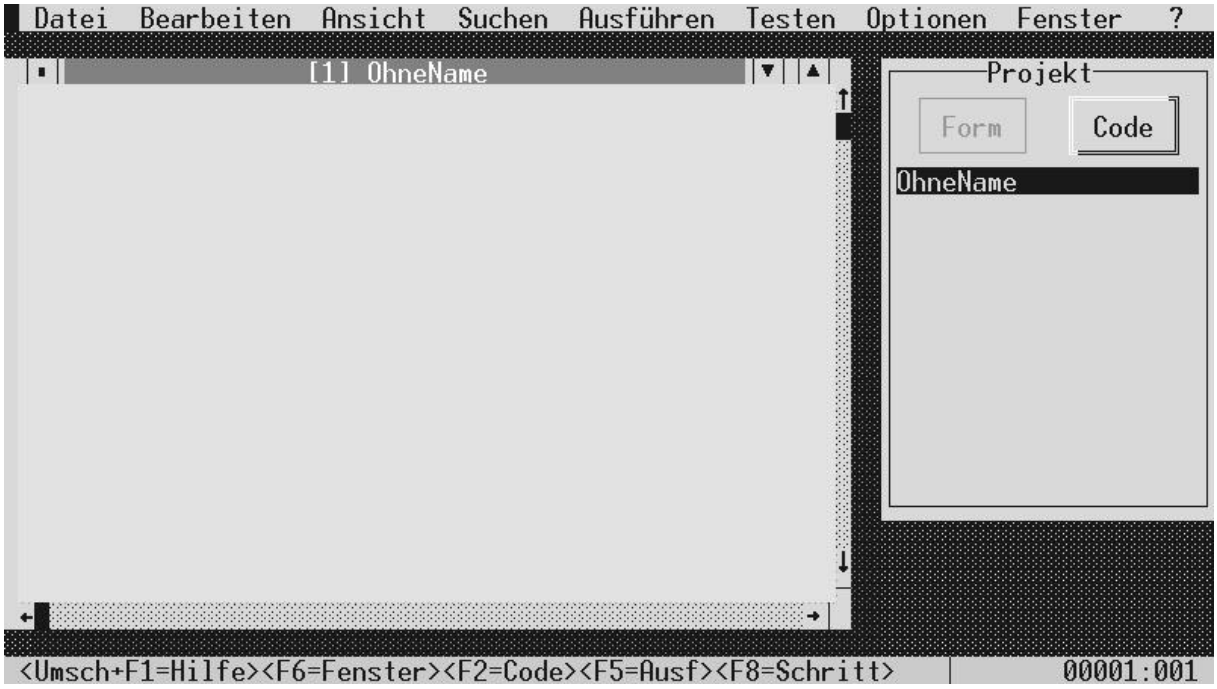


Abbildung 4–1: Der Eröffnungsbildschirm von VBDOS

Beim Start von VBDOS sehen Sie zwei Fenster: Das große Codefenster ist zunächst leer; hier wird der Programmcode angezeigt, den Sie eingeben. Im Codefenster wird zwischen verschiedenen Abschnitten unterschieden; jede Prozedur und Funktion hat ihren eigenen Abschnitt. Benutzen Sie die Tasten F2 und F12, um zu verschiedenen Abschnitten innerhalb des aktiven Codefensters zu schalten. In der Kopfzeile wird angezeigt, um das wievielte Code-Fenster es sich handelt und welcher Abschnitt darin gerade zu sehen ist.

Sie können mehrere Codefenster gleichzeitig öffnen und sie neben- oder übereinander anzeigen.

Das kleinere Fenster rechts gibt einen Überblick über alle geladenen Module (die in ihrer Gesamtheit als „das Projekt“ bezeichnet werden, weil sie beim Kompilieren eine einzige EXE-Datei ergeben). Wenn das Projektfenster Formmodule enthält, können Sie durch Auswahl eines Formmoduls und Mausklick auf „Form“ den Form-Designer starten, um das Aussehen der Form zu bearbeiten. Ein Click auf „Code“ öffnet die „Code“-Dialogbox, in der dann wiederum alle Prozeduren des Moduls zur Auswahl stehen.

4.2 Der Aufruf von VBDOS

Die VBDOS-Befehlszeile sieht folgendermaßen aus:

`VBDOS [switches] [/RUN] programmname [/CMD command$]`

programmname ist der Name des Programms, das Sie bearbeiten wollen. Das Programm mit dem angegebenen Namen wird geladen, als ob Sie es mit dem „Neues Projekt“-Befehl aus dem „Datei“-Menü (siehe später in diesem Kapitel) veranlaßt hätten. Sie können den Namen auch weglassen, dann wird zunächst überhaupt nichts geladen.

Wo innerhalb der Zeile die *switches* stehen, ist ohne Belang. Ausnahmen: /RUN muß, wenn er angegeben wird, immer direkt vor dem Programmnamen stehen, und /CMD darf nur als letztes auf der Zeile stehen.

Die Bedeutung der Switches ist im folgenden aufgeführt.

Switch	*Wirkung*
/AH	Ermöglicht „Huge Arrays“ (siehe Kapitel 8). Alle EXE-Programme, die aus VBDOS heraus erstellt werden, werden ebenfalls mit /AH kompiliert, wenn dieser Switch angegeben wird.
/B	Die VBDOS-Bildschirmanzeige erfolgt nur schwarz/weiß.
/C:*b*	Setzt den Standard-Kommunikations-Buffer auf *b* Bytes (für das Empfangen von Daten über OPEN COM).
/CMD:*command$*	Alles, was hinter /CMD folgt, wird in die Systemvariable COMMAND$ geschrieben und kann von Ihrem Programm abgefragt werden. Ein bequemerer Weg, COMMAND$ zu setzen, ist jedoch der Menüpunkt „COMMAND$ ändern“ im „Ausführen“-Menü.
/Ea	Ermöglicht das Auslagern von Arrays mit einer Größe zwischen 512 und 16.384 Bytes in den EMS-Speicher. Wenn Sie zusammen mit /Ea eine Quick Library laden, muß diese mit /D oder /AH kompiliert worden sein. /Ea ist nicht mit /Es kompatibel.

Switch	*Wirkung*
/E:*code,vbdos*	VBDOS kann einerseits den verfügbaren EMS-Speicher verwenden, um Teile von sich selbst auszulagern, andererseits können auch Arrays (mit /Ea) und Quellcode-Fragmente ausgelagert werden. *code* gibt an, wieviel EMS maximal für Quellcode und Arrays verwendet wird; *vbdos* legt die Obergrenze für den von VBDOS belegten Speicher fest (beide Angaben in KB). Ohne /E kann VBDOS u.U. den gesamten EMS belegen. Weitere Hinweise siehe Kapitel 8.
/Es	Teilt EMS zwischen VBDOS und Routinen anderer Sprachen auf (nur für gemischtsprachliches Programmieren). /Es ist nicht mit /Ea kompatibel.
/G	(nur für CGA-Karten): Beschleunigt den VBDOS-Bildschirmaufbau, kann bei manchen CGA-Karten aber zu unruhigem Bild führen.
/H	VBDOS benutzt, wenn es die Hardware erlaubt, eine kleinere Schrift, so daß mehr als 25 Zeilen auf den Schirm passen (43 bei EGA-, 50 bei VGA-Karten).
/L [*quicklibrary*]	Lädt die angegebene Quick Library. Es kann immer nur eine Quick Library geladen werden; wenn Sie den Namen *quicklibrary* weglassen, lädt VBDOS die Library VBDOS.QLB, die die Routinen für Interruptaufrufe und absolute Funktionsaufrufe enthält.
/MBF	Ersetzt die Funktionen *MKS$*, *MKD$*, *CVS* und *CVD* durch ihre Pendants mit dem Anhängsel MBF, um Kompatibilität mit Uralt-Datenbanken zu gewährleisten. Wenn Sie beim Aufruf von VBDOS /MBF angeben, werden auch alle aus VBDOS heraus erzeugten EXE-Files mit /MBF kompiliert.
/NOHI	Verwendet keine hervorgehobenen Farben (für LCD-Bildschirme)
/RUN	Das genannte Programm wird sofort gestartet. Wenn es einen SYSTEM-Befehl enthält, wird VBDOS dadurch sofort verlassen.
/S:*konv*	Legt fest, wieviel konventionellen Speicher (im Bereich bis 640 KB) VBDOS selbst verwendet. *konv* ist im Bereich 280 bis 534 sinnvoll einstellbar. Je größer *konv*, desto schneller arbeitet VBDOS; je kleiner *konv*, desto mehr Speicherplatz steht für das Projekt zur Verfügung.
/X:*maxxms*	Begrenzt die Verwendung von XMS-Speicher durch VBDOS auf *maxxms* KB. Weitere Hinweise siehe Kapitel 8.

Beispiele zum VBDOS-Programmaufruf

```
VBDOS PROG1
```

Startet VBDOS und lädt PROG1. Dabei wird in folgender Reihenfolge geprüft:

- Wenn PROG1.MAK vorhanden ist, wird dieses Projekt mit allen darin enthaltenen Dateien geladen.
- Wenn PROG1.FRM vorhanden ist, wird dieses Formmodul geladen.

- Wenn PROG1.BAS vorhanden ist, wird dieses Codemodul geladen.
- Ansonsten wird ein leeres Codemodul namens PROG1.BAS geöffnet.

```
VBDOS PROG1 /L
```

Startet VBDOS, lädt PROG1 – wie oben – und dazu die Quick Library VBDOS.QLB.

```
VBDOS /RUN PROG1.MAK /L FONT /CMD C:\*.*
```

Startet VBDOS, lädt die Quick Library FONT.QLB und das Projekt PROG1.MAK, startet das Programm sofort und schreibt in COMMAND$ den Text „C:*.*“ hinein.

```
VBDOS /NOHI/B
```

Hier wird VBDOS gestartet und erhält die Anweisung, alles in schwarz/weiß auszugeben und keine hervorgehobenen (hellen) Farbattribute zu benutzen.

4.3 Editor und Tastenbelegung

Die Tastaturbelegung in VBDOS ist im wesentlichen immer noch die gleiche wie bei den Vorgängern QBX und QB.

Bewegen des Cursors

Die Tasten im folgenden Kasten können auch mit Shift zusammen betätigt werden; dann dienen sie der Markierung von Text. Markierter Text wird üblicherweise invers dargestellt.

Taste	*Funktion*
Pfeiltasten	Cursorbewegung
Einfg (Insert)	Zwischen Einfüge- (kleiner Cursor) und Überschreibmodus (großer Cursor) umschalten
Strg + ←	Wort links
Strg + →	Wort rechts
Bild ↑ (PgUp)	Seite aufwärts
Bild ↓ (PgDn)	Seite abwärts (eine Seite hat soviele Zeilen, wie im Fenster dargestellt werden können).

Taste	*Funktion*
Strg + Bild ↑	Bildschirm links
Strg + Bild ↓	Bildschirm rechts
Pos1	zum ersten Zeichen im Fenster
Ende	zum letzten Zeichen im Fenster

Weitere Tasten zum Bearbeiten sind:

Taste	*Funktion*
Entf (Del)	Markierten Text löschen
Shift + Entf oder Strg + X	Markierten Text löschen, aber zuvor ins Clipboard kopieren (Das Clipboard ist ein Zwischenspeicher, den man zum Verschieben von Text benutzen kann.)
Strg + Y	Aktuelle Zeile löschen, aber zuvor ins Clipboard kopieren
Strg + Einfg oder Strg + V	Text aus dem Clipboard an der Cursorposition einfügen
Shift + Einfg oder Strg + C	Markierten Text ins Clipboard kopieren, ohne ihn zu löschen
Tab	(im Programmtext, wenn Text markiert ist) Den markierten Text um eine Tab-Position weiter einrücken
Shift + Tab	(im Programmtext, wenn Text markiert ist) Den markierten Text auf die vorhergehende Einrück-Position ausrücken. Es wird von der obersten markierten Zeile aus nach oben so lange gesucht, bis eine Zeile gefunden wird, die weniger als die am weitesten links stehende markierte eingerückt ist; um die Differenz zwischen beiden Einrückungen wird der ganze markierte Text nach links gezogen.
Alt + Rückschritt (←) oder Strg + Z	Die letzte Veränderung rückgängig machen. VBDOS behält die letzten 20 geänderten Zeilen, so daß bis zu 20 Änderungen rückgängig gemacht werden können. Diese Tastenkombination kann auch Ersetzen-Vorgänge oder das Löschen von ganzen Prozeduren rückgängig machen. Es gibt jedoch keine Möglichkeit, das Rückgängigmachen rückgängig zu machen (war in früheren Versionen mit Strg + Rückschritt möglich).

Der Editor prüft eingegebene Zeilen sofort auf korrekte Syntax und protestiert, wenn er sie nicht versteht. Außerdem wandelt er alle BASIC-Befehls- und Funktionsnamen sofort in Großbuchstaben um und fügt Leerzeichen, Anführungszeichen und von Zeit zu Zeit auch ein Semikolon ein, wenn es nötig ist. Darüber hinaus sorgt er dafür, daß dieselbe Variable überall im Programm gleich geschrieben ist. Das heißt, wenn in Ihrem Programm bisher dreimal die Variable „Zeilennummer“ erwähnt ist und Sie die Schreibweise an einer Stelle

in „ZeilenNummer“ ändern, werden alle anderen Stellen sofort entsprechend modifiziert.

Fenster und Abschnitte

Für jede Routine Ihres Programms (also jedes SUB und jede FUNCTION) wird ein eigener Abschnitt angelegt. In dem Augenblick, in dem Sie „SUB“ (gefolgt von einem Namen für die Routine) eingeben, eröffnet VBDOS sofort einen neuen Abschnitt im Code-Fenster und schreibt als letzte Zeile gleich „END SUB“ hinein.

Taste	*Funktion*
F6	Ins nächste Fenster springen
Shift + F6	Ins vorherige Fenster springen (bei nur zwei Fenstern identisch mit F6)
Strg + F10	Das aktuelle Fenster auf Bildschirmgröße „aufblasen“; alle anderen Fenster verschwinden, bis Strg + F5 gedrückt wird. Diese Option ist identisch mit der Betätigung des ▲-Symbols rechts oben auf dem Code-Fenster durch die Maus.
Strg + F9	Das aktuelle Fenster auf Symbolgröße verkleinern
Strg + F5	Das aktuelle Fenster wieder auf seine ursprüngliche Größe bringen
F1	(im Programmtext) Öffnet das Hilfsfenster zum BASIC-Befehl oder zum Variablen- oder Prozedurnamen, auf dem der Cursor steht
Shift + F1	Öffnet das Hilfsfenster, von dem aus man (über „Index“) sämtliche Hilfstexte abrufen kann
Strg + F4	Das aktuelle Fenster schließen

Die Online-Hilfe

Während das Hilfe-Fenster aktiv ist, gelten einige spezielle Funktionstasten:

Taste	*Funktion*
F1 oder ENTER	(im Hilfsfenster) Aktiviert das Hyperlink, auf dem der Cursor steht (Hyperlinks werden die in Dreiecke [◄►] eingeschlossenen Verweise auf andere Hilfsseiten genannt)
Tab	(im Hilfsfenster) Setzt den Cursor auf das nächste Hyperlink im gerade angezeigten Hilfstext
Alt + F1	(im Hilfsfenster) Zeigt die zuletzt angezeigte Hilfsseite (die 20 letzten werden zwischengespeichert) wieder an
Strg + F1	(im Hilfsfenster) Zeigt, nachdem man Strg + F1 benutzt hatte, wieder die danach gewählte Hilfsseite an
ESC	schließt das Hilfsfenster

An dieser Stelle sei mir ein Hinweis auf die „Wissensbasis“ erlaubt, die in der Online-Hilfe ein wenig zu kurz kommt. Über das Feld „Microsoft Software Service“ im Inhaltsverzeichnis oder aber über „Wissensbasis“ im Index der Hilfe erreichen Sie das Inhaltsverzeichnis der Microsoft Wissensbasis zu VBDOS. Die Artikel der Wissensbasis beschäftigen sich im Frage-und-Antwort-Stil mit einigen kniffligen Problemen, aber auch mit allgemeinen Tips der VBDOS-Programmierung. Unter anderem finden Sie dort Hinweise, wie man einen Grafikbildschirm auf dem Drucker ausgibt, Informationen zur Genauigkeit von Fließkommaberechnungen oder die Anleitung zur Definition eines eigenen Mauscursors im Grafikmodus.

Die Artikel der Wissensbasis sind nicht ins Deutsche übersetzt.

Auf der Diskette, die diesem Buch beiliegt, befinden sich die Dateien REFERENZ.HLP und REFERENZ.BAT. REFERENZ.HLP enthält ein Verzeichnis aller Standard-BASIC-Befehle, für die Sie hier im Buch keinen Nachschlageteil finden werden. Mittels der Prozedur REFERENZ.BAT können Sie dieses Verzeichnis an Ihre VBDOS-Hilfe anfügen lassen, so daß Sie es während der Arbeit mit VBDOS abrufen können.

Wir haben uns zu dieser etwas unkonventionellen Methode entschlossen, weil der Standard-Referenzteil, der ohnehin größtenteils die „altbekannten“ Befehle enthält, das Buch noch dicker gemacht hätte, als es schon ist. Die neuen Befehle von VBDOS (ereignisgesteuerte Programmierung, Tools der professionellen Ausgabe) sind im Gegensatz dazu komplett abgedruckt, da Sie in diesen Seiten sicherlich nicht nur bei Bedarf nachschlagen, sondern auch aus Interesse blättern werden.

4.4 Die Übersicht über Ihr Programm

Die Code-Dialogbox

Die Taste F2 öffnet eine Dialogbox auf dem Bildschirm, die alle geladenen Dateien mit ihren Namen und sämtliche FUNCTION- oder SUB-Prozeduren (alphabetisch sortiert unter dem Dateinamen, zu dem sie gehören) anzeigt. Mit einem Leuchtbalken kann man dann einen beliebigen Namen anwählen (in einer der untersten Zeilen der Box wird angezeigt, worum es sich bei dem Namen handelt, auf dem gerade der Leuchtbalken steht). Wenn Sie hier eine Buchstabentaste drücken, springt der Balken zum nächsten Eintrag, der mit diesem Buchstaben anfängt.

Abbildung 4–2: Die „Code“-Dialogbox, die nach Betätigung von F2 angezeigt wird

Die Dialogbox wird wie eine typische VBDOS-Form bedient.

Taste	*Funktion*
F2	(im Programm) Öffnet die „Code“-Dialogbox; wenn sich der Cursor im Programm auf dem Namen einer Subroutine befindet, steht der Leuchtbalken in der Box auch auf diesem Namen, so daß er schnell gewählt werden kann, ansonsten steht der Balken auf dem Namen der Prozedur, in der der Cursor steht
Shift + F2	(im Programm) Schaltet in die nächste Subroutine (alphabetische Reihenfolge) weiter
Strg + F2	(im Programm) Schaltet in die Subroutine vor der gerade bearbeiteten (alphabetische Reihenfolge)
ENTER oder Alt + A	(in der „Code“-Dialogbox) Wählt die Subroutine/das Programm, auf der/dem der Leuchtbalken steht, zur Bearbeitung in dem Codefenster, aus dem heraus die Dialogbox aufgerufen wurde
Alt + U	(in der „Code“-Dialogbox) Wählt die Subroutine/das Modul, auf der/dem der Leuchtbalken steht, zur Bearbeitung in einem neuen Codefenster
Alt + E	(in der „Code“-Dialogbox) Löscht nach einer Sicherheitsabfrage das Modul bzw. die Subroutine, auf dem/der der Leuchtbalken steht
Alt + V	(in der „Code“-Dialogbox) Ermöglicht das Verschieben der Subroutine, auf der der Leuchtbalken steht, in ein anderes geladenes Modul

Die Ereignisprozeduren-Dialogbox

Wenn Ihr Programm mindestens eine Form enthält, können Sie über die Taste F12 die Dialogbox „Ereignisprozeduren“ abrufen. Diese enthält drei Listen:

Abbildung 4–3: Die „Ereignisprozeduren“-Dialogbox, die mit F12 angezeigt wird

Ganz links sehen Sie ein Liste aller Formen, die das Projekt enthält. In der Mitte stehen alle Objekte, die die links ausgewählte Form besitzt, und ganz rechts werden alle Ereignisse angezeigt, die von dem in der Mitte gewählten Objekt erzeugt werden können. Die Ereignisse, für die bereits eine Ereignisprozedur existiert, sind in Großbuchstaben geschrieben (in der Abbildung „LOAD“).

Sie können durch Auswahl einer Form, eines Objektes und eines Ereignisses einerseits die zugehörige Ereignisprozedur anzeigen lassen, andererseits aber auch den leeren „Prozedurkopf“ für ein Ereignis erzeugen lassen. Wenn Sie in der abgebildeten Dialogbox zum Beispiel ganz rechts das Ereignis „KeyDown“ wählten, würde VBDOS im aktuellen Code-Fenster einen neuen Abschnitt erzeugen und die Zeilen

```
SUB Form_KeyDown (KeyCode AS INTEGER, Shift AS INTEGER)

END SUB
```

hineinschreiben.

4.5 Menüs

Abbildung 4–4: Die Menüs „Datei“ und „Bearbeiten“ in VBDOS

Menü „Datei“

„Neues Projekt“ löscht alle Module aus dem Speicher und fragt vorher nach, wenn Module noch nicht gespeichert sind.

„Projekt öffnen“ lädt ein neues Projekt. Es muß sich nicht unbedingt um eine .MAK-Datei handeln; auch Formen oder Codemodule können so direkt geladen werden. Alle zuvor im Speicher befindlichen Module werden gelöscht. „Projekt speichern“ speichert alle Dateien des geladenen Projekts. Falls nur eine Form/ein Codemodul geladen ist und bisher kein Projektname vergeben wurde, fragt VBDOS nach.

„Neue Form“ fragt nach dem Namen für eine neue Form und verzweigt dann in den Form-Designer zur Gestaltung der Form. „Neues Modul“ fügt dem Projekt ein neues Codemodul hinzu.

„Datei hinzufügen“ erlaubt es, ein beliebiges Form- oder Codemodul dem aktuellen Projekt beizufügen. Die Datei wird geladen, ohne daß das im Speicher befindliche Projekt gelöscht wird. Es können auch Text- oder Include-Dateien geladen werden; diese werden aber nicht in die Projektliste mit aufgenommen.

„Datei entfernen“ öffnet eine Dialogbox, die alle geladenen Dateien enthält, und ermöglicht es, eine davon aus dem Speicher zu löschen. Falls es ein Form- oder Codemodul ist, wird es aus dem Projekt gestrichen. „Datei speichern“ speichert die aktuelle Datei (und nicht das gesamte Projekt). „Datei speichern unter...“ speichert die aktuelle Datei unter einem beliebigen Namen und erlaubt auch die

Angabe, ob die Datei im Text- oder Binärformat gespeichert werden soll. Zur Verwendung mit CodeView oder dem Profiler muß das Textformat verwendet werden; TRNSLATE erfordert das Binärformat, und für alle anderen Zwecke ist es ziemlich egal, welches Format sie wählen.

„Text laden“ ermöglicht es, einen beliebigen Text an die aktuelle Cursorposition einzufügen (als wäre er über Tastatur eingegeben worden) oder das aktuelle Modul komplett durch eine bestimmten Textdatei zu ersetzen.

„Drucken“ bietet die Möglichkeit, das aktuelle Fenster, das aktuelle Modul oder alle Module zu drucken; dabei kann für jede Prozedur auf Wunsch eine neue Seite angefangen werden. Um Formen auszudrucken, muß man zunächst den Form-Designer aufrufen.

Menü „Bearbeiten“

Neben den Standard-Edierfunktionen bietet dieses Menü die Punkte „Neue Prozedur“ und „Neue Funktion“, die je nach Wunsch im aktuellen oder in einem neuen Codefenster einen neuen Abschnitt für eine Prozedur oder Funktion einrichten (gleiches erreicht man durch Eingabe von SUB oder FUNCTION in einem Codefenster). Außerdem kann mit dem Menüpunkt „Ereignisprozeduren“ die weiter oben beschriebene, gleichnamige Dialogbox aufgerufen werden.

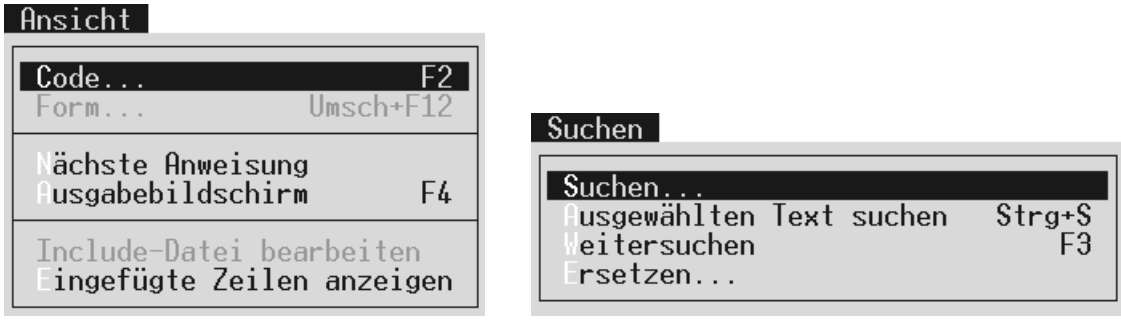


Abbildung 4–5: Die Menüs „Ansicht“ und „Suchen“ in VBDOS

Menü „Ansicht“

Dieses Menü bietet über „Code“ die Möglichkeit, die weiter oben beschriebene „Code“-Dialogbox aufzurufen. „Form“ öffnet die Dialogbox „Form“, die die Auswahl einer der Formen des Projekts ermöglicht und danach in den Form-Designer verzweigt. „Nächste Anweisung“ setzt den Cursor an die Stelle im Programm, die beim Programmstart (mit F5) als nächstes ausgeführt würde.

„Ausgabebildschirm“ zeigt den Programmbildschirm an (nur sinnvoll, wenn ein laufendes Programm zuvor unterbrochen wurde).

„Include-Datei bearbeiten“ ist nur dann anwählbar, wenn der Cursor sich auf einer Zeile mit dem Metabefehl $INCLUDE befindet. Dann führt die Anwahl dieses Punktes dazu, daß die betr. Datei geladen und angezeigt wird. „Eingefügte Zeilen anzeigen“ hingegen kann immer gewählt werden. Dieser Punkt hat Ein-/Aus-Funktion, und solange er gewählt ist, wird der Inhalt aller per $INCLUDE eingebundenen Dateien mit hervorgehobener Farbe an der Stelle angezeigt, an der sie eingebunden werden. Programmänderungen sind jedoch während dieser Anzeige nicht möglich.

Menü „Suchen“

Dieses Menü bietet die üblichen Suchen-und-Ersetzen-Funktionen. Dabei kann stets gewählt werden, ob im aktuellen Fenster, im aktuellen Modul oder in allen Modulen gesucht bzw. ersetzt werden soll, und ob dabei die Groß- und Kleinschreibung beachtet und nur nach ganzen Wörtern gesucht werden soll.

Abbildung 4–6: Die Menüs „Optionen“ und „Fenster“ in VBDOS

Menü „Optionen“

Über „Anzeige“ können Sie die von VBDOS verwendeten Farben, das Bildschirmhintergrundzeichen und die Breite eines Tabulators festlegen. „Zugriffspfade festlegen“ bestimmt, wo VBDOS nach Hilfe-, Include- und ausführbaren Dateien zu suchen hat. Mit „Rechte Maustaste“ können Sie bestimmen, ob die rechte Maustaste wie F1 (Hilfe) oder wie F7 (Programmausführung bis zum Cursor) wirken soll. „Speichern“ erlaubt es, VBDOS anzuweisen, die veränderten Dateien vor jeder Ausführung und vor jedem Wechsel zum Form-Designer automatisch zu speichern oder auch eine Sicherungskopie (.BAK) anzulegen.

„Syntax überprüfen“ hat eine Ein-/Aus-Funktion; solange dieser Punkt abgeschaltet ist, erkennt VBDOS keine Tippfehler, fehlende Klammern usw.; ob VBDOS eine Zeile „schluckt“ oder nicht, erkennt man aber auch dann, da nur dann in einer Zeile Leerzeichen eingefügt und Befehlswörter großgeschrieben werden, wenn sie fehlerfrei ist.

Menü „Fenster“

Dieses Menü bietet die Möglichkeit, zwischen den verschiedenen Code-Fenstern hin- und herzuschalten (alle Code-Fenster werden unterhalb der zweiten Trennlinie angezeigt). Über „Alle anordnen“ können alle derzeit geöffneten Fenster auf dem Bildschirm arrangiert werden, „Neues Fenster“ öffnet ein neues Code-Fenster.

Mit den Optionen im Mittelbereich kann man spezielle Fenster von VBDOS öffnen oder, falls sie bereits geöffnet sind, zu ihnen verzweigen. „Aufrufe“ enthält, wenn ein laufendes Programm unterbrochen wurde, eine hierarchische Liste aller Funktions- und Prozeduraufrufe (mehr dazu später); das „Test“-Fenster zeigt die laufenden Überwachungsausdrücke an (ebenfalls später mehr). Das „Hilfe“-Fenster braucht nicht weiter erläutert zu werden; im „Direkt“-Fenster können BASIC-Befehle eingegeben und sofort ausgeführt werden. „Ausgabe“ zeigt den Programm-Ausgabebildschirm in einem Fenster an (kann aber nicht, wie F4, auch Grafikbildschirme anzeigen), und „Projekt“ zeigt die Liste aller Module im geladenen Projekt an.

Begriffsbestimmungen für das „Ausführen“- und das „Testen“-Menü

Die letzten beiden Menüs erfordern zunächst einige Begriffserklärungen:

In VBDOS kann für beliebige Ausdrücke und Variablen eine **Ausdrucksüberwachung** etabliert werden. VBDOS zeigt dann im „Test“-Fenster stets den aktuellen Wert der Variable oder des Ausdrucks an, die/der überwacht wird. Maximal 16 Ausdrücke können überwacht werden.

Eine **Haltebedingung** geht etwas weiter als die bloße Ausdrucksüberwachung. Wenn Sie eine Haltebedingung für einen bestimmten Ausdruck etablieren, wird der Wert dieses Ausdrucks ebenfalls im „Test“-Fenster angezeigt; das Programm bricht jedoch sofort ab, wenn der Ausdruck den Wert TRUE (–1) annimmt.

Ein **Haltepunkt** ist eine markierte Zeile im Quellcode Ihres Programms, bei deren Erreichen VBDOS die Programmausführung abbricht.

Die Verwendung von Haltebedingungen und Ausdrucksüberwachungen reduziert die Geschwindigkeit der Programmausführung erheblich, weil VBDOS nach jedem Befehl den Wert der Variablen bzw. Ausdrücke bestimmen muß.

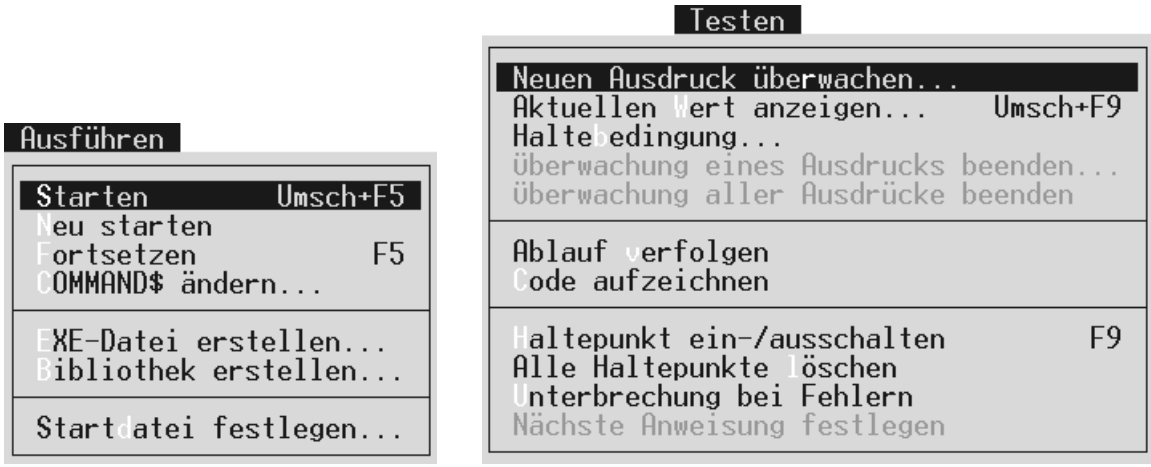


Abbildung 4–7: Die Menüs „Ausführen“ und „Testen“ in VBDOS

Menü „Ausführen“

Der Menüpunkt „Starten“ startet das Programm beim ersten Befehl neu. Im Gegensatz dazu setzt „Fortsetzen“ die Programmausführung am aktuellen Befehl (die Stelle, an der das Programm unterbrochen wurde) fort. „Neu Starten“ setzt alle Variablen zurück, schließt alle Dateien und macht den ersten Befehl des Programms zum aktuellen, beginnt aber im Gegensatz zu „Starten“ nicht sofort mit der Programmausführung.

„COMMAND$ ändern“ ermöglicht die Veränderung der Umgebungsvariable COMMAND$ für das Programm. „EXE-Datei erstellen“ und „Bibliothek erstellen“ erzeugen aus allen geladenen Dateien eine EXE-Datei bzw. eine Library (.LIB) und eine Quick Library (.QLB) – mehr dazu später in diesem Kapitel.

Mit „Startdatei festlegen“ können Sie aus den geladenen Modulen die Startdatei für das Projekt bestimmen. Zu den Auswirkungen der Startdatei-Festlegung vgl. den letzten Abschnitt im Kapitel 3.

Menü „Testen“

Mit „Neuen Ausdruck überwachen“ können Sie die Ausdrucksüberwachung für einen Ausdruck etablieren. Der Gültigkeitsbereich des Ausdrucks muß sich dabei über den im aktuellen Codefenster gezeigten Ausschnitt erstrecken; sie können also nicht, während die Prozedur „LiesDaten“ angezeigt wird, eine

Ausdrucksüberwachung für eine lokale Variable aus „SchreibeDaten“ aktivieren.

„Aktuellen Wert anzeigen“ zeigt den Wert der Variablen oder des Ausdrucks, auf der oder dem der Cursor steht, in einem Fenster an. Mit „Überwachung eines/aller Ausdrücke beenden“ lassen sich Ausdrucksüberwachungen deaktivieren.

„Ablauf verfolgen“ schaltet den Trace-Modus ein oder aus. Im Trace-Modus ist eine Markierung neben „Ablauf verfolgen“. Das Programm läuft extrem langsam. Die gerade bearbeitete Programmzeile wird hervorgehoben, so daß man den Ablauf beobachten kann. „Ablauf verfolgen“ schließt außerdem die Funktion „Code aufzeichnen“ mit ein.

„Code aufzeichnen“ hat ebenfalls eine Ein-/Aus-Funktion. Im Aufzeichnungsmodus ist eine Markierung neben „Code aufzeichnen“ sichtbar. Wenn das Programm unterbrochen wird, können mit Shift+F8 (rückwärts) und Shift+F10 (vorwärts) die letzten 20 ausgeführten Befehle des Programms durchlaufen werden. „Ablauf verfolgen“ und „Unterbrechung bei Fehlern“ aktivieren die Codeaufzeichnung automatisch, ohne daß eine Markierung hier sichtbar ist.

„Haltepunkt ein-/ausschalten“ macht die Zeile, auf der der Cursor steht, zu einer Haltepunktzeile oder wieder zu einer normalen Zeile, wenn sie bereits eine Haltepunktzeile ist. Vor der Ausführung einer Haltepunktzeile wird das Programm angehalten. „Alle Haltepunkte löschen“ wandelt alle Haltepunktzeilen wieder in normale Zeilen um.

„Unterbrechung bei Fehlern“ kann ein- und ausgeschaltet werden. Solange die Funktion aktiv ist, wird das Programm bei Auftreten eines Fehlers immer angehalten, auch dann, wenn der Fehler im Programm von einer Fehlerbehandlungsroutine (einem Error-Handler) aufgefangen wird.

„Nächste Anweisung festlegen“ macht die erste Anweisung der Zeile, auf der der Cursor steht, zur aktuellen Anweisung. Hier wird das Programm weiter ausgeführt, wenn Sie die Taste F5 drücken. Diese Funktion kann nur benutzt werden, wenn zuvor das laufende Programm unterbrochen wurde, und man darf als aktuelle Anweisung keine Anweisung außerhalb der Prozedur oder Funktion, in der das Programm unterbrochen wurde, wählen.

4.6 Fehlersuche

Die Fehlersuche gehört zu den wichtigsten Aufgaben eines Programmierers. Selbst kleine Programme werden selten gleich im ersten Anlauf fehlerfrei erstellt. Glücklicherweise muß man heute dank VBDOS nicht mehr dauernd zwischen Compiler und Editor hin- und herspringen, eine Zeile ändern, kompilie-

ren, Fehler notieren, wieder ändern, kompilieren, linken, wieder einen Fehler finden, einen PRINT-Befehl zur Kontrolle einbauen, kompilieren... und zwischendurch warten, warten, warten.

Ein fertiges Programm startet man in VBDOS einfach mit Shift+F5. VBDOS prüft dann das Programm auf Fehler und zeigt sie an. Der fehlerhafte Befehl wird dabei farbig oder invers dargestellt und läßt sich zumeist einfach verbessern. Nur noch in wenigen Fällen bleibt es zunächst im dunkeln, welcher Fehler eigentlich gemeint ist. Ein „END IF ohne IF“- oder ähnliche Fehler, die das Fehlen von Strukturanweisungen bemängeln, erfordern meist eine etwas längere Suche, da VBDOS nicht anzeigen kann, wo sich der Fehler genau befindet. Das folgende Programm erzeugt zum Beispiel einen „LOOP ohne DO“-Fehler, anstatt daß VBDOS merkt, daß ein END IF fehlt:

```
DO
   a$ = INKEY$
   IF LEN(a$) THEN
      PRINT a$;
LOOP
```

Beschäftigen wir uns aber nun mit den wirklich schwierigen Fehlern. Das sind die Fälle, die VBDOS gar nicht als Fehler erkennt. Fehler, die VBDOS bemerkt und meldet, sind vergleichsweise einfach zu beheben. Fehler aber, die daraus resultieren, daß Sie beim Programmieren schon etwas müde waren und anstatt „Zaehler%“ irgendwo „Zeahler%“ geschrieben haben, können richtige Kopfnüsse werden. Es gibt kein Patentrezept, wie man solche logischen Fehler finden kann, aber es gibt eine Anzahl von guten Ratschlägen, die ich Ihnen hier nicht vorenthalten will.

Verwenden Sie OPTION EXPLICIT

Der in VBDOS neu eingeführte Befehl OPTION EXPLICIT weist VBDOS an, für jede Variable einen Deklarationsbefehl (DIM, SHARED, STATIC oder COMMON) zu verlangen. Damit geben Sie zwar einen der BASIC-Vorteile gegenüber anderen Sprachen auf: Sie können nicht mehr „einfach so“ eine Schleife mit `FOR i% = 1 TO ...` schreiben, sondern müssen auch solche „Bagatellvariablen“ mit DIM deklarieren. Allerdings entdecken Sie auf diese Weise Fehler wie den im vorigen Absatz erwähnten vergleichsweise leicht, und insbesondere bei großen Programmen lohnt sich die Mühe sicher.

Eingrenzen des Fehlers

Versuchen Sie, den Bereich, innerhalb dessen „irgendwo“ der Fehler liegen muß, soweit als möglich einzugrenzen. Dazu können Sie das Programm an verschiedenen Stellen mit Haltepunkten unterbrechen und Variablenwerte mit Shift + F9 oder PRINT abfragen. Wenn Sie immer einen Haltepunkt haben, von dem Sie wissen, daß bis dorthin noch alles o.k. ist, und einen zweiten, bei dem der Fehler bereits aufgetreten ist, können Sie die beiden immer näher zusammenrücken, bis Sie die fehlerhafte Stelle gefunden haben.

Verfolgen von Variablenwerten

Wenn Sie herausfinden, daß irgendetwas falsch läuft, weil eine bestimmte Variable einen falschen Wert hat, dann setzen Sie entweder eine Haltebedingung, oder versuchen Sie herauszufinden, wo dieser Variablen ein Wert zugewiesen wird. Mit der „Suchen“-Funktion aus dem gleichnamigen Menü können Sie feststellen, wo die betreffende Variable überall vorkommt; dann bietet es sich eventuell an, alle Stellen, an denen der Variablen ein Wert zugewiesen wird, mit einem PRINT-Befehl zu versehen, damit Sie beim Programmablauf sehen können, welche Werte wo in die Variable eingetragen werden.

Wenn Sie mit Formen arbeiten und deshalb nicht einfach einen PRINT-Befehl verwenden möchten, fügen Sie dem Projekt einfach eine Form namens „Test“ hinzu, die keine Steuerelemente enthält, schreiben Sie in die Form_Click-Prozedur „CLS“ und verwenden Sie im Programm überall für Testausgaben Aufrufe der Form `Test.PRINT`. Dann werden alle Test-Ausgaben auf die Test-Form ausgegeben. Damit die Form Ihrem Programm nicht im Weg ist, können Sie das Programm während der Testphase in einem 43- oder 50-Zeilen-Modus ablaufen lassen und die Testform unterhalb der Zeile 25 anzeigen. Außerdem können Sie, falls Sie nicht alle `Test.PRINT`-Aufrufe entfernen, aber trotzdem eine vorläufige Programmversion ohne Testinformation (z.B. als Betaversion für den Kunden) erstellen möchten, durch die Anweisung `Test.HIDE` sehr einfach die Testinformationen ausblenden.

Wenn Sie nicht gerade einen Laserdrucker haben, können Sie auch den LPRINT-Befehl zur Fehlersuche einsetzen (zum Beispiel `LPRINT "PRÜFSTELLE 1 ERREICHT  Variable z$ = ";z$` o. ä.), um den Ablauf einfach verfolgen zu können. Auch mit Breakpoints läßt sich das bewerkstelligen, aber manchmal ist es auf dem Papier übersichtlicher.

Die von VBDOS vorgesehene Form der Ausdrucksüberwachung (in einem eigenen Fenster, Ausdrücke werden über das „Testen“-Menü in die Liste aufgenommen) halte ich für etwas unkomfortabel, da sie die am laufenden Band Pro-

grammunterbrechungen erfordert und auch nur die aktuellen Werte, nie aber den Verlauf der Wertveränderungen in der Vergangenheit anzeigt.

Prozeduraufrufe

Wenn Sie den Verdacht haben, daß eine bestimmte Prozedur aufgerufen oder nicht aufgerufen wird, obwohl das Gegenteil der Fall sein sollte, bauen Sie einfach einen Breakpoint in die erste Zeile der Prozedur ein. Wenn die Prozedur dann aufgerufen wird, können Sie im „Aufrufe"-Fenster, das Sie über das „Fenster"-Menü öffnen, ablesen, von wo aus das geschah. Das Fenster zeigt alle Prozeduraufrufe bis zur obersten Ebene an, also sehen Sie auch, von wo aus die Prozedur oder Funktion aufgerufen wurde, die die betreffende Funktion aufrief usw. Wenn Sie im Aufrufe-Fenster eine Prozedur anwählen, wird diese im Programmfenster angezeigt, und der Cursor steht an der Stelle, von der aus der Aufruf erfolgte, der im Fenster abzulesen ist.

Wird stattdessen eine Prozedur nicht aufgerufen, die eigentlich aufgerufen werden sollte, so können Sie an die Stelle, an der die Prozedur eigentlich aufgerufen werden müßte, einen Breakpoint setzen und diesen so lange nach oben verschieben, bis das Programm an der Stelle anhält. Dann wissen Sie, daß diese Stelle ausgeführt wird, und müssen nun prüfen, warum die folgenden Zeilen nicht mehr ausgeführt werden. Grund dafür könnte neben IF- und GOTO-Befehlen auch eine Fehlerbehandlungsroutine sein.

Fehlerbehandlungsroutinen...

können Ihre Bemühungen, einen Fehler zu finden, tückisch unterlaufen. In umfangreichen Programmen ist es nicht immer leicht, den Überblick zu behalten, welches ON ERROR im Augenblick aktiv ist. Gerade dann, wenn Sie sich wundern, warum bestimmte Befehle offenbar grundlos nicht ausgeführt werden, kann der Grund dafür sein, daß einer der vorangehenden Befehle einen Fehler verursacht und das Programm dann in die Fehlerbehandlungsroutine springt und von dort vielleicht an eine andere Stelle zurückkehrt o.ä.

Sie sollten deshalb entweder „Unterbrechung bei Fehlern" aktivieren, damit VBDOS grundsätzlich das Programm unterbricht, wenn ein Fehler auftritt, oder einen PRINT-Befehl oder zumindest ein BEEP oder etwas Vergleichbares an den Anfang der Fehlerbehandlungsroutine setzen, damit Sie bei der Fehlersuche immer sehen können, welcher Fehler wann auftritt.

Verfolgen des Programmablaufs

Wenn innerhalb eines nicht allzugroßen Programmabschnittes etwas unklar ist, können Sie den Programmablauf genau verfolgen. Dazu lassen sich die Trace- und die Aufzeichnungsfunktion einsetzen. Noch flexibler geht es allerdings im „Handbetrieb“: Setzen Sie einen Haltepunkt an die Stelle, von der an es interessant wird, und starten Sie das Programm. Wenn es am Haltepunkt anhält, haben Sie neben der Taste F5, die die Programmausführung normal fortsetzt, auch noch folgende Möglichkeiten:

Taste	*Bedeutung*
F8 „Trace In“	Führt den aktuellen Befehl aus; der nächste Befehl wird zum aktuellen. Wenn der aktuelle Befehl der Aufruf einer Funktion oder Prozedur ist, wird der erste Befehl in dieser Prozedur zum aktuellen Befehl. So kann man die Programmausführung bis ins kleinste Detail verfolgen.
F10 „Trace Over“	Funktioniert wie F8, behandelt aber Prozeduren und Funktionen wie übliche BASIC-Befehle. Sie sehen nicht, was in der Prozedur vorgeht. So lassen sich Prozeduren, an deren Korrektheit kein Zweifel besteht, übergehen.
F7 „Execute to Cursor“	Setzt die Programmausführung am aktuellen Befehl fort, hält aber wieder an, wenn die Zeile erreicht ist, in der der Cursor gerade steht.

So kann man Funktionen und Prozeduren, von denen man glaubt, daß sie keinen Fehler enthalten, mit F10 überspringen und andere, kompliziertere Routinen mit F8 einer genaueren Betrachtung unterziehen. Programmpassagen, die mit hoher Wahrscheinlichkeit fehlerfrei sind (z. B. eine einfache FOR-NEXT-Schleife, die man am Bildschirm nicht 238mal durchlaufen will) überspringt man, indem man den Cursor hinter die Passage setzt und F7 benutzt.

4.7 Quick Libraries in VBDOS

Eine Quick Library ist, genau wie eine gewöhnliche Library, eine Sammlung von Routinen, die man häufiger benutzt und deshalb zweckmäßigerweise in eine Library gepackt hat. Der Unterschied ist, daß die Routinen in einer Quick Library auf den Gebrauch in VBDOS zugeschnitten sind.

Eine Quick Library kann beim Start von VBDOS geladen werden, so daß die in ihr enthaltenen Routinen dem Projekt, das unter VBDOS läuft, zur Verfügung stehen. Dadurch besteht mit Quick Libraries die Möglichkeit, Routinen, die in BASIC-fremden Sprachen geschrieben sind, unter VBDOS einzusetzen.

Allerdings können Routinen, die in einer Quick Library stehen, nicht mehr in Fehlersuche-Aktionen innerhalb VBDOS einbezogen werden. Wenn in einer

Quick Library ein Fehler auftritt, kann VBDOS lediglich anzeigen, in welcher Routine das passierte. Sie können in der Quick Library keine Haltepunkte setzen etc., weil der Quellcode nicht mehr verfügbar ist.

Weitere Informationen zum Inhalt und zur Erstellung von Quick Libraries finden Sie im Kapitel 6.

4.8 EXE-Programme und Libraries

Sie können aus VBDOS heraus EXE-Programme und Libraries erstellen, und zwar mit den Befehlen „EXE-Datei erstellen“ bzw. „Library erstellen“ aus dem „Ausführen“-Menü.

Libraries

Die Erstellung von Libraries mit VBDOS ist fast gleichwertig mit der separaten Methode, bei der BC, LINK und LIB benutzt werden müssen, um die gleichen Resultate zu erzielen. Sie unterliegt jedoch einigen Einschränkungen:

- VBDOS kann nur Libraries für die Emulator-Library erzeugen. Für Quick Libraries ist das zwar die einzige Möglichkeit, aber die Objektcode-Libraries könnten auch mit dem /FPa (professionelle Version) erzeugt werden.
- Es ist nicht möglich, Routinen einzubinden, die in anderen Sprachen geschrieben sind und als OBJ-Dateien vorliegen. Deshalb müssen zum Beispiel die Quick Libraries der Font- und der Präsentationsgrafik-Toolbox immer noch auf separatem Wege produziert werden.

Wenn Sie die „Bibliothek erstellen“-Funktion benutzen, erstellt VBDOS sowohl eine Quick Library (.QLB) als auch eine gewöhnliche Objectcode-Library (.LIB). Die Quick Library wird für VBDOS benötigt, die Objectcode-Library werden Sie immer dann brauchen, wenn Sie ein EXE-Programm erzeugen möchten, das die betreffenden Routinen enthält.

Laden Sie alle Module, die in den Libraries enthalten sein sollen. Eventuell vorhandener Modulcode kann gelöscht werden, da er die Libraries nur unnötig verlängern würde. Wählen Sie dann „Bibliothek erstellen“. Nachdem Sie die gewünschten Optionen (wie beispielsweise Code-Erzeugung für 286er Prozessoren usw.) eingestellt und ENTER gedrückt haben, werden die neuen Libraries erzeugt. Probleme können auftreten, wenn Sie die Betriebssystemvariable LINK benutzen und in ihr einen Switch wie /E eingetragen haben, der für die Herstellung von Quick Libraries nicht benutzt werden darf. Sie müssen dann zunächst diese Variable ändern.

EXE-Programme

Bei der Herstellung von EXE-Programmen mit VBDOS, dem *integrierten Kompilieren*, bestehen gegenüber dem separaten Kompilieren (siehe Kapitel 6) eine Anzahl von Einschränkungen, die jedoch fast ausschließlich Funktionen der professionellen Version betreffen:

- Wenn Sie mit einer Quick Library arbeiten, nimmt VBDOS an, daß diese unter gleichem Namen als gewöhnliche Library (.LIB) existiert, und gibt diesen Namen automatisch beim Linken an. Das führt dazu, daß Sie, wenn Sie eine Quick Library benutzen, nur Programme für die Emulator-Library erstellen können. Bei der separaten Kompilierung und bei VBDOS ohne Quick Library haben Sie dagegen die Wahl zwischen Emulator- und Alternate Math-Library, wenn Sie die professionelle Version verwenden.
- Sie können mit der Funktion „EXE-Datei erstellen“ keine EXE-Programme erstellen, die mit eigenen Runtime-Modulen (siehe Kapitel 14) arbeiten.
- Für Programme, die ohne Runtime-Modul funktionieren, kann man beim Linken Verzicht-Files angeben (siehe Kapitel 23), die das Programm verkürzen. Das ist mit „EXE-Datei erstellen“ nicht möglich.
- LINK kann durch die Switches /F/PACKC und /NON schnellere bzw. kürzere Programme erstellen (siehe Kapitel 23). VBDOS unterstützt diese Switches nicht, man kann sie höchstens in die Betriebssystemvariable LINK eintragen.
- Es lassen sich mit der „EXE-Datei erstellen“-Funktion keine Programme erstellen, die mit Overlays arbeiten.
- VBDOS verwendet beim Aufruf von LINK immer den Switch /E. Dadurch können Sie keine Programme, die zur Weiterverarbeitung mit CodeView oder dem Profiler vorgesehen sind, erzeugen.

Wenn Sie sich von alledem nicht schrecken lassen: Laden Sie alle Module, die Bestandteil des EXE-Programms werden sollen. Nur der Modulcode in der Startdatei wird ausgeführt. Der restliche Modulcode ist – bis auf Deklarationsanweisungen wie DIM, COMMON etc. und eventuell Fehlerbehandlungs- oder Event-Trapping-Routinen – überflüssig und sollte gelöscht werden. Wählen Sie dann „EXE-Datei erstellen“, und stellen Sie die gewünschten Optionen ein. Switches, die Sie hier nicht direkt einstellen können und die VBDOS auch nicht selbst setzt (wie zum Beispiel /S), können Sie in das „Zusätzliche Optionen“-Feld eintragen. VBDOS erzeugt dann das EXE-Programm unter dem gewünschten Namen.

Die dabei entstehenden .OBJ-Dateien werden, obwohl sie nicht mehr benötigt werden, nicht gelöscht.

Der Form-Designer 5

5.1 Übersicht

Der Form-Designer kann zwar auch einzeln von der DOS-Befehlszeile aus gestartet werden, aber zumeist wird VBDOS ihn für Sie aufrufen. Dies geschieht, wenn Sie im VBDOS-Projektfenster auf „Form“ klicken, im VBDOS-Menü „Ansicht“ den Befehl „Form“ wählen, mit Shift+F12 in die Form verzweigen oder mit „Neue Form“ aus dem „Datei“-Menü eine neue Form anlegen.

Wenn Sie noch keine Form erstellt haben, präsentiert sich der Form-Designer so:

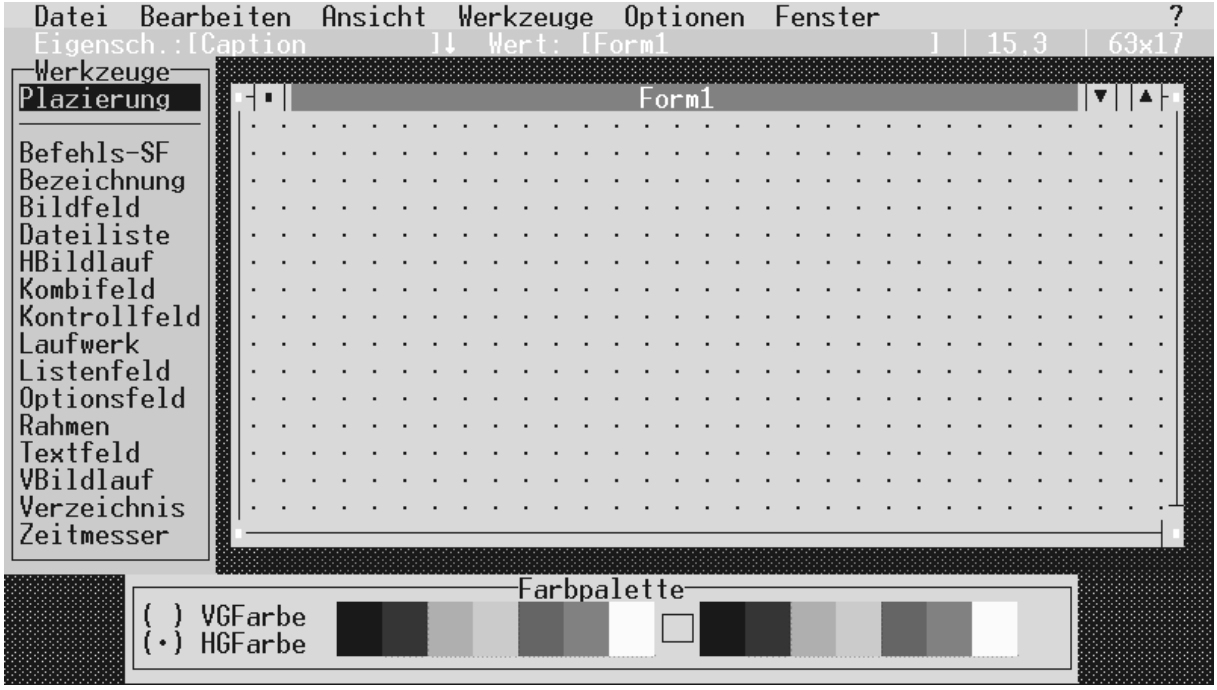


Abbildung 5–1: Der Form-Designer mit leerer Formenanzeige

Oben wird die Menüleiste angezeigt, darunter die Eigenschaftenleiste. Links sehen Sie die „Werkzeugsammlung“, und am unteren Bildschirmrand befindet sich die Farbpalette mit allen 16 Vorder- und Hintergrundfarben. All diese Elemente können auf Wunsch auch ausgeblendet werden, aber solange Ihre Formen nicht über die angezeigte Größe hinauswachsen, läßt sich mit dieser Einstellung sehr gut arbeiten.

Sie können die Form mit der Maus vergrößern und verschieben. Die Werkzeugsammlung erweitert sich automatisch, falls benutzerdefinierte Steuerelemente in Quick Libraries geladen sind.

Eine Form entwerfen

Das Entwerfen einer Form im Form-Designer ist denkbar einfach. Ich empfehle Ihnen dringend, eine Maus zu benutzen; zwar lassen sich alle Funktionen auch mit der Tastatur ausführen, aber das ist zumeist wesentlich umständlicher.

Zuerst wählen Sie aus der Werkzeugsammlung links das gewünschte Steuerelement. Dann bewegen Sie die Maus auf die Form, drücken die linke Maustaste und halten sie gedrückt, während Sie die Maus so lange weiterbewegen, bis das Steuerelement die gewünschte Größe erreicht hat.

Später läßt sich die Größe des Steuerelements noch durch Klicken auf den Rand verändern. Die Posititon verändert man durch Klicken in die Mitte eines Objekts. Während Sie Position oder Größe des Objekts manipulieren, werden diese Maße in der Eigenschaftenleiste oben rechts in Zeilen und Spalten angezeigt.

Sie können ein markiertes Objekt durch Drücken der „Entf“-Taste löschen.

Entwerfen Sie so zunächst alle gewünschten Objekte, und passen Sie die Größe und Position der Form an.

Eigenschaften

Der Form-Designer wählt für alle Eigenschaften Standardwerte. Insbesondere die Eigenschaften *Caption* und *CtlName* werden Sie jedoch bei den meisten Objekten auf eigene Werte setzen wollen. Wählen Sie dazu das betroffene Objekt an, und klicken Sie dann auf das „Eigensch.“-Feld in der Eigenschaftenleiste oder drücken Sie Shift+F2. Nun können Sie mit den Pfeiltasten aus der Liste der für dieses Objekt verfügbaren (und zur Entwurfszeit einstellbaren) Eigenschaften wählen. Die Taste F4 zeigt die Liste aller möglichen Auswahlen an. Drücken Sie die Tab-Taste, wenn Sie die gewünschte Eigenschaft gefunden haben, und wählen Sie den gewünschten Wert oder geben Sie ihn ein.

Wenn Sie eine Eigenschaft ausgewählt haben und nacheinander verschiedene Objekte anwählen, wird immer der Inhalt der gewählten Eigenschaft angezeigt, so daß Sie (durch F2 oder Klick in das „Wert“-Feld der Eigenschaftenleiste) sehr schnell z.B. die *Caption*-Eigenschaften aller Objekte durchgehen können.

In den beiden Kombifeldern „Eigensch.“ und „Wert“ in der Eigenschaftenleiste kann auch durch Drücken des Anfangsbuchstaben des gewünschten Listeneintrags ausgewählt werden. Zuweilen geht das flotter als das Durchblättern der Liste mit der Maus, weil man mit den Augen nicht dauernd am Schirm hängen muß. Meine Standard-Operation beim Erstellen von Formen, nämlich *MinButton*, *MaxButton* und *ControlBox* auf FALSE und *BorderStyle* auf 1 zu setzen,

bewerkstellige ich inzwischen mit folgender Tastenfolge: Shift+F2, M, Tab, F, Tab, M, Tab, F, Tab, C, C, Tab, F, Tab, B, B, Tab, 1.

Die Farbpalette

Für die Änderung der Vorder- und Hintergrundfarben können Sie zwar auch die Eigenschaftenleiste verwenden, aber viel bequemer geht es mit der Farbpalette.

Wählen Sie einfach ein Objekt an, bestimmen Sie durch Mausklick, ob sie die Vorder- oder Hintergrundfarbe ändern wollen (nicht alle Objekte unterstützen beide Eigenschaften), und klicken Sie auf die Farbe. Voilà!

(Wenn Sie mit der Tastatur arbeiten, können Sie mit der Taste F6 in die Farbpalette wechseln und gelangen mit Shift+F6 wieder zur Form zurück.)

5.2 Die Menüs des Form-Designers

Abbildung 5–2: Die Menüs „Datei“ und „Bearbeiten“ im Form-Designer

Das Datei-Menü entspricht im wesentlichen dem Datei-Menü in VBDOS. Bei „Form speichern unter...“ können Sie – wie auch in VBDOS – zwischen Text- und binärer Speicherung wählen. Wählen Sie die Text-Speicherung, wenn Sie mit einem Editor oder einem anderen Programm auf die Datei zugreifen möchten oder wenn die Form mit dem WINDOWS-Formenübersetzer übersetzt werden soll.

Die weiteren Menüpunkte erklären sich wohl selbst; bei „Drucken“ druckt der Form-Designer die Form so aus, wie sie auch von der Routine PRINTFORM (siehe Objekt-Referenzteil) gedruckt wird, und auf Wunsch werden auch noch die Eigenschaften der Objekte (in einem ziemlich unübersichtlichen Kraut-und-Rüben-Format) ausgedruckt.

Im Menü „Bearbeiten“ sind die üblichen Operationen zusammengefaßt: „Ausschneiden“ löscht das markierte Objekt und steckt es in die Zwischenablage, „Kopieren“ kopiert das Objekt in die Zwischenablage, „Einfügen“ fügt den Inhalt der Zwischenablage ein und „Löschen“ entfernt das markierte Objekt. Wenn Sie ein Objekt kopieren und danach einfügen, entstehen dadurch zwei Objekte mit gleicher *Caption*-Eigenschaft. Das veranlaßt den Form-Designer, zu fragen, ob Sie ein Steuerelemente-Array erzeugen wollen (vgl. Kapitel 7). Antworten Sie mit „Nein“, wird der eingefügten Kopie ein neuer *CtlName* zugewiesen.

Abbildung 5–3: Die Menüs „Ansicht“ und „Werkzeuge“ im Form-Designer

Das Ansicht-Menü ermöglicht mit der Auswahl „Code“ den Wechsel nach VBDOS; durch „Form“ können Sie, falls das Projekt noch andere Formen enthält, zu einer anderen Form wechseln.

Der an- und abschaltbare Punkt „Menüleiste“ bestimmt, ob die Menü- und die Eigenschaftenleiste auch dann angezeigt wird, wenn die Alt-Taste nicht gerade gedrückt ist; „Rasterlinien“ legt fest, ob die Form am Bildschirm alle zwei Spalten in jeder Zeile Orientierungspunkte enthält.

Das Menü „Werkzeuge“ benötigen Sie nur, wenn Sie die Werkzeugsammlung am linken Bildschirmrand nicht anzeigen lassen; hier können Sie, genauso wie in der Werkzeugsammlung, einen Steuerelemente-Typ auswählen.

Abbildung 5–4: Die Menüs „Optionen“ und „Fenster“ im Form-Designer

Das Menü „Optionen“ ermöglicht die Einstellung der Bildschirmfarben und des Hintergrundes (über „Anzeige“) und die Festlegung des Hilfedatei-Verzeichnisses. In der Dialogbox „Speichern“ können Sie den Form-Designer anweisen, bei jeder Speicherung einer geänderten die alte Form als .BAK-Datei aufzuheben und die geänderte Form bei Verlassen des Form-Designers automatisch zu speichern. Letzteres würde ich unbedingt empfehlen, weil Sie dadurch die lästige Meldung „Projekt- oder Quelldateien wurden geändert. Jetzt speichern?“, die der Form-Designer sonst beim Übergang zu VBDOS anzeigt, vermeiden können.

Mit dem Menü „Fenster“ können Sie den Cursor in verschiedene Fenster setzen (F6 geht aber schneller). Außerdem läßt sich das Menüentwurfsfenster öffnen (siehe 5.4).

Im hier nicht abgebildeten Hilfe-Menü finden sich die Standard-Hilfethemen („Index“, „Inhalt“, „Tastatur“ und andere).

5.3 Wichtige Tastaturabkürzungen

Ich möchte hier nicht all die Abkürzungstasten, die der Form-Designer kennt, aufzählen, sondern nur in einer Tabelle die wichtigsten zusammenstellen. Wie bereits gesagt, ist die Arbeit mit der Maus empfehlenswert; in Zeiten, da Sie für den Preis dieses Buches schon vier Mäuse kaufen können, sollte es daran nicht mangeln.

Taste	*Funktion*
F2	Setzt den Cursor in das Feld „Wert“ der Eigenschaftenleiste
Shift+F2	Setzt den Cursor in das Feld „Eigensch.“ der Eigenschaftenleiste
F4	Klappt die Liste des Kombifeldes „Wert“ bzw. „Eigensch.“ aus
Alt+F4	Beendet den Form-Designer und kehrt zurück zu VBDOS
F6	Schaltet in das nächste Fenster
F10	Schaltet die Anzeige der Menü- und Eigenschaftenleiste ein bzw. aus
F12	Kehrt zurück zu VBDOS und zeigt dort das „Ereignisprozeduren“-Fenster an
Shift + F12	Ermöglicht das Bearbeiten einer anderen Form desselben Projekts

Taste	*Funktion*
Strg + V	Fügt den Inhalt der Zwischenablage ein
Entf	Löscht das markierte Objekt
Tab	Schaltet die Markierung zum nächsten Objekt weiter

5.4 Das Menüentwurfsfenster

Menü-Steuerelemente sind die einzigen, die nicht mit der Maus über die Werkzeugsammlung eingefügt werden können. Größe und Position der Menü-Steuerelemente werden automatisch festgelegt.

Wenn Sie ein Menü erstellen wollen, wählen Sie „Menüentwurfsfenster“ aus dem Menü „Ansicht“. Sie erhalten folgende Anzeige (allerdings mit noch leeren Feldern):

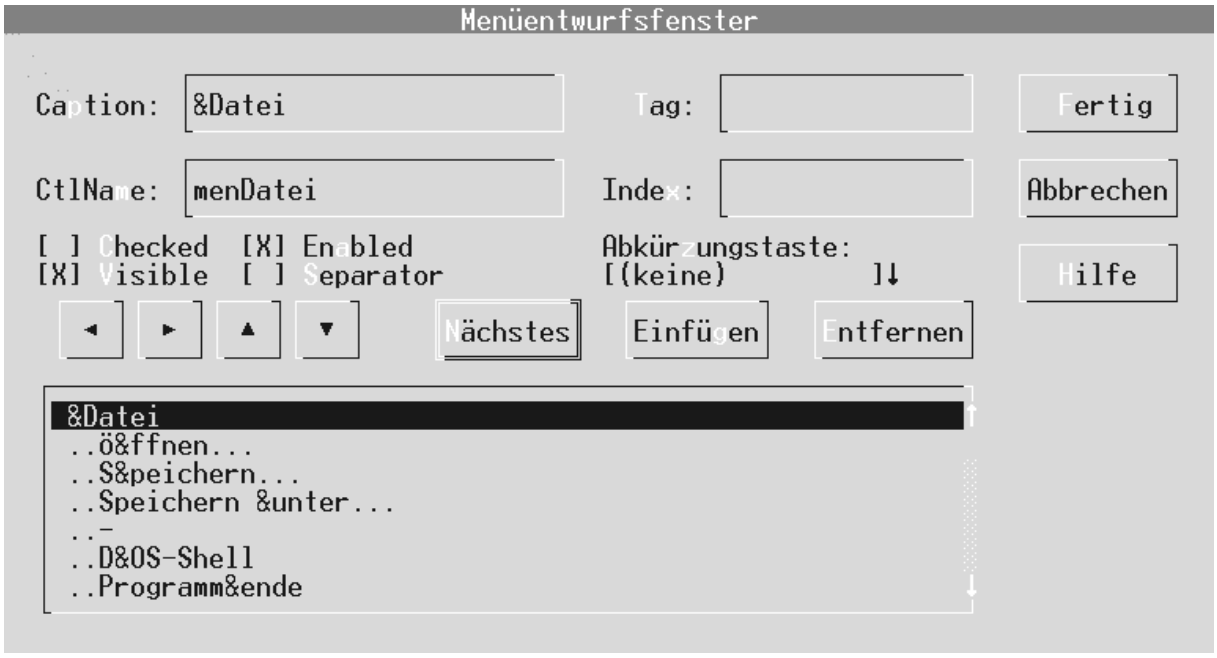


Abbildung 5–5: Das Menüentwurfsfenster mit Ausschnitt aus selbsterstelltem Menü

Das große Fenster in der unteren Hälfte der Anzeige beinhaltet die von Ihnen bereits entworfenen Menüeinträge. Mit den Pfeiltasten können Sie den Balken in dieser Liste bis maximal ein Feld hinter dem letzten Eintrag bewegen.

In die Felder „Caption“, „Ctlname“, „Tag“ und „Index“ werden die Werte für die jeweiligen Eigenschaften des Menüeintrags, auf dem der Balken steht, eingetragen. Bedenken Sie, daß jeder Eintrag eine eindeutige *CtlName*-Eigenschaft haben muß.

Mittels der Ankreuzfelder „Checked“, „Visible“, „Enabled“ und „Separator“ können Sie die gleichnamigen Eigenschaften des Menüeintrags auf TRUE setzen.

Das Feld „Abkürzungstaste“ ermöglicht das Festlegen einer Abkürzungstaste (eine Tastenkombination ohne die Alt-Taste) für einen Menüeintrag; wenn der Benutzer dann später – egal, welches Feld den Fokus hat – diese Taste drückt, wird ein Click-Ereignis für das entsprechende Menü-Steuerelement ausgelöst. Die **Abkürzungstaste** unterscheidet sich von der **Zugriffstaste** dadurch, daß ein Menüeintrag über die Abkürzungstaste auch aufgerufen werden kann, wenn das Menü gar nicht angezeigt wird und ein völlig anderes Steuerelement den Fokus hat. Die Zugriffstaste hingegen ist nur wirksam, wenn der betreffende Menüeintrag auf der ersten Menüebene steht oder die darüberliegende Menüebene angezeigt wird.

Mit den Symbolen ▲ und ▼ können Sie den Menüeintrag, auf dem der Balken steht, in der Liste nach oben oder unten verschieben (alternativ können die Tastenkombinationen Alt+O, Alt+U verwendet werden). Die Symbole ◀ und ▶ (oder Alt+L, Alt+R) ermöglichen es, die Ebene des Menüpunktes zu ändern. Zunächst sind alle eingegebenen Menüeinträge auf der Ebene 1, würden also in der Menüzeile Ihres Programmes angezeigt. Einträge, die Sie eine Ebene herabsetzen, erscheinen in dem Menü, das ausklappt, wenn der Benutzer den darüberliegenden Menüpunkt der Ebene 1 wählt. Analog erscheinen Menüpunkte, die Sie zwei Ebenen herabstufen, nur dann, wenn der darüberliegende Menüpunkt der Ebene 2 gewählt wird. Ein Bildbeispiel für eine Menüstruktur mit drei Ebenen finden Sie im Abschnitt über Menüsteuerelemente in Kapitel 7.

Menüpunkte, die sich nicht auf der ersten Ebene befinden, werden im Entwurfsfenster mit führenden Punkten angezeigt (zwei Punkte pro eingerückter Ebene). Sie können Menüs mit bis zu fünf Ebenen erstellen; VBDOS verwendet selbst allerdings nur zwei Ebenen, und das empfehle ich Ihnen auch. Zumindest vier und fünf Ebenen halte ich für indiskutabel, weil zu unübersichtlich. Arbeiten Sie lieber mit „Dialogboxen“, also Abfragefenstern, die sich öffnen, wenn der Benutzer einen bestimmten Menüpunkt wählt.

Die Schaltflächen „Einfügen“ und „Entfernen“ ermöglichen das Einfügen eines neuen Menüeintrags mitten in der Liste bzw. das Streichen eines Eintrags aus dieser.

Kompilieren von der Befehlszeile 6

In vielen Fällen ist es notwendig oder zumindest praktischer, seine Programme nicht aus dem VBDOS-Menü heraus, sondern direkt mit dem eigentlichen Compiler BC.EXE zu erzeugen, den VBDOS ja ebenfalls benutzt, wenn es EXE-Dateien erstellen soll. Dies wird vor allem dann notwendig, wenn man größere Programme erzeugen will. Arbeitet man mit BC und LINK, um EXE-Programme zu erstellen, hat man sehr viel mehr Flexibilität als mit VBDOS allein.

Das manuelle Erstellen von EXE-Programmen mit BC und LINK wird *separates Kompilieren* genannt, weil alle Schritte einzeln und „von Hand“ durchgeführt werden. Im Gegensatz dazu steht das *integrierte Kompilieren*, das von VBDOS automatisch auf Befehl ausgeführt wird.

6.1 Die verschiedenen Dateiarten

Zum Arbeiten mit BC und LINK ist zunächst das Verständnis der sechs Dateiarten erforderlich, mit denen man dabei zu tun bekommt:

.BAS und .FRM sind die Namenserweiterungen der BASIC-Quelldateien, die mit VBDOS oder mit einem beliebigen Texteditor erzeugt werden. Für jedes lauffähige EXE-Programm ist mindestens ein BAS- oder FRM-Modul notwendig; wenn es ein Codemodul (.BAS) ist, muß es Code auf Modulebene enthalten (also Befehle, die nicht zu einer Prozedur gehören). Dieses Programm kann darüber hinaus natürlich auch Prozeduren enthalten. Ich nenne dieses Programm in den folgenden Beispielen HAUPT.BAS, obwohl es ebenso ein Formmodul sein könnte. Außerdem können beliebig viele weitere BAS- oder FRM-Module verwendet werden, die nur Prozeduren enthalten. Eventueller Modulcode von weiteren BAS-Programmen wird nicht berücksichtigt, es sei denn, es handelt sich um „nicht ausführbare Befehle“ (vgl. Kapitel 3).

.OBJ heißen die Objektdateien, die der Compiler BC aus BAS-Programmen macht. (Der Name hat mit den Objekten in VBDOS – Formen und Steuerelementen – nichts zu tun.) Eine solche Datei ist kein lauffähiges Programm, sondern nur ein Zwischenprodukt auf dem Wege dahin. Eine Objektdatei ist sprachunabhängig, das heißt, daß auch Pascal, C, Assembler und andere (Microsoft-kompatible) Sprachen solche Dateien erzeugen. Zwar ist es nicht ohne weiteres möglich, Objektdateien verschiedener Compiler zu einem lauffähigen Programm zusammenzubinden, aber wenn man gewisse Regeln beachtet, kann man „multilinguale“ Programme erstellen. Das OBJ-File ist die einfache Übersetzung des Programms in Maschinensprache. Es enthält viele Funktions-

aufrufe für BASIC-Hilfsroutinen, die erst beim Linken aus einer Compiler-Library in das EXE-File eingefügt werden. Deshalb werden beim Linken die Compiler-Libraries benötigt.

.QLB ist der Name einer sogenannten Quick Library. Eine Quick Library wird mit dem LINK-Programm aus einer oder mehreren Objektdateien zusammengesetzt und ist ausschließlich zur Nutzung innerhalb von VBDOS zu gebrauchen. In VBDOS können nie mehrere Quick Libraries gleichzeitig benutzt werden.

.LIB ist die Bezeichnung für eine gewöhnliche Library, die mittels des LIB-Programmes aus einer oder mehreren Objektdateien erzeugt wird. Eine Library ist nichts weiter als eine Sammlung von OBJ-Dateien. Sie kann nicht in VBDOS benutzt werden. Sie findet Verwendung bei der Herstellung von EXE-Files mit LINK, obwohl man dabei auch auf sie verzichten kann – dann wird der Vorgang allerdings zuweilen etwas umständlich. Viele der BASIC-Hilfsroutinen, die im Handbuch gar nicht erwähnt sind, sondern nur intern Verwendung finden, sind in den diversen mitgelieferten Libraries enthalten.

.EXE werden schließlich die ausführbaren Programme genannt, die am Ende eines erfolgreichen Kompilier- und Link-Prozesses stehen. Ein EXE-File muß immer konsistent sein, das heißt, es muß alle Routinen, die es aufrufen will, selbst enthalten. Einzige Ausnahme hiervon ist die Verwendung eines Runtime-Moduls; dann holt sich das EXE-Programm zur Laufzeit die benötigten Routinen aus einer zweiten EXE-Datei, dem Runtime-Modul. Mehr dazu weiter unten in diesem Kapitel und im Kapitel 14.

6.2 Beispiele für das separate Kompilieren

Das folgende Schaubild zeigt, wie ein simples EXE-File erzeugt werden kann. Eingerahmt sind jeweils Dateinamen, ohne Rahmen stehen Programmaufrufe. (BC und LINK können wesentlich differenzierter verwendet werden, als es hier gezeigt wird. Mehr dazu später.)

Abbildung 6–1: Simples separates Kompilieren

Um auf diese Weise EXE-Files zu erzeugen, bedarf es nicht des separaten Kompilierens mit BC und LINK, denn es handelt sich ja nur um eine einzige Quelldatei. HAUPT.BAS enthält alle Routinen, die benötigt werden. Man könnte hier ohne weiteres auch VBDOS einsetzen, es sei denn, man möchte beim Kompilieren oder Linken bestimmte Switches benutzen, die von VBDOS aus nicht aktiviert werden können, oder Verzicht-Files (vgl. Kapitel 23) einbinden.

Betrachten wir einen etwas komplizierteren Fall:

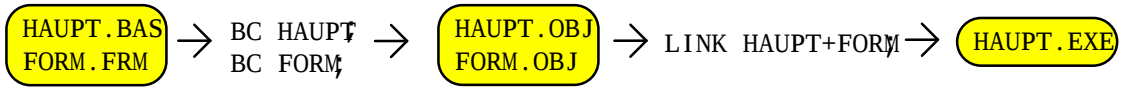


Abbildung 6–2: Separates Kompilieren mit zwei Modulen

Hier handelt es sich um zwei Quelldateien, die beide getrennt kompiliert und dann mit LINK zu einem einzigen Programm vereint werden. Ein Fall wie dieser tritt nicht nur dann ein, wenn man ein Form- und ein Codemodul verwendet, sondern auch, wenn ein Programm so groß wird, daß man einige seiner Subroutinen in eine zweite Datei auslagern muß, weil es sich sonst nicht mehr kompilieren läßt, oder weil man Wert auf ordentliches Modularisieren und Strukturieren legt.

Hat man mehrere Dateien, die Hilfsroutinen enthalten, kann man auch mit einer Library arbeiten, wie es die folgende Abbildung zeigt:

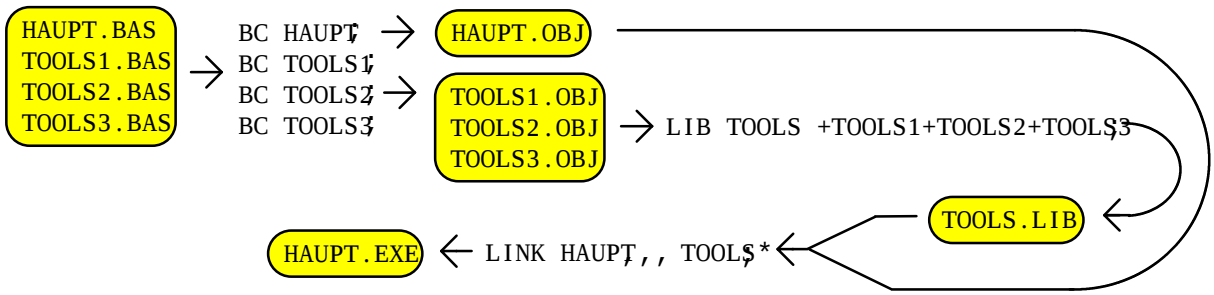


Abbildung 6–3: Separates Kompilieren unter Verwendung von LIB

Auf diese Weise vereinigt man alle Routinen aus den TOOLS-Quelldateien in der Library TOOLS.LIB, die nun auch für jedes beliebige andere Programm benutzt werden kann. Man muß nur ihren Namen beim Linken angeben, damit der Linker sich aus der Library alle benötigten Routinen holen kann.

Die alternative Möglichkeit wäre gewesen, auf die Library zu verzichten und die Namen aller drei Tools-Files (gemäß Abbildung 6–2) beim Linken anzugeben.

Weitere Informationen über das Erstellen von eigenen Libraries mit LIB finden Sie im Abschnitt über LIB weiter unten in diesem Kapitel. Das Beispiel aus Abbildung 6–3 werde ich außerdem noch zur Erläuterung der Runtime-Module in Kapitel 14 heranziehen.

---

* Die drei Kommata im LINK-Befehl sind erforderlich, weil LINK die Library-Angabe als vierten Parameter erwartet. LINK wird später in diesem Kapitel genauer erläutert.

6.3 Der Compiler BC.EXE

BC hat die folgende Aufrufsyntax:

```
BC [modulname [, [objektname] [, listname]]] [switches] [;]
```

Am Ende des Abschnittes finden Sie einige Beispiele hierzu.

modulname ist der Name des BASIC-Moduls, das kompiliert werden soll. Wird keine Extension angegeben, prüft der Compiler erst, ob ein FRM-Modul mit diesem Namen vorhanden ist; findet er keines, nimmt die Erweiterung .BAS an. Geben Sie als Modulnamen USER an, wird die Eingabe direkt von der Tastatur entgegengenommen.

objektname ist der Name des OBJ-Files, das erzeugt werden soll. Standard (wenn Sie keinen Namen explizit angeben) ist derselbe Name wie *modulname*, nur mit der Extension .OBJ.

Hinweis: Sie sollten nie ein Formmodul und ein Codemodul mit dem gleichen „Vornamen" (z.B. HAUPT.BAS und HAUPT.FRM) erstellen. Diese werden vom Compiler nämlich beide zur Objektdatei HAUPT.OBJ kompiliert, wenn Sie nicht explizit einen Namen für die Objektdatei angeben.

listname ist der Name einer Datei, in die der Compiler ein komplettes Modullisting mit Fehlermeldungen, Original-Programmzeile und Adresse jeder Zeile ausgibt. Standard ist die Extension LST; wird kein Dateiname angegeben, erzeugt der Compiler auch keine Liste.

Das optionale Semikolon verhindert ein Nachfragen des Compilers: Bei fehlendem Namen für Objekt- oder Listdatei fragt der Compiler gewöhnlich erst noch einmal nach, welcher Name gewünscht ist, bevor er zu arbeiten beginnt. Ein Semikolon sorgt dafür, daß er für alle Namen die Standardvorgabe wählt und weitermacht. `BC PROGR;` würde beispielsweise dafür sorgen, daß PROGR.BAS in PROGR.OBJ kompiliert und keine Liste erzeugt wird.

Nun zu den *switches*. Ein Switch, ein Compiler-Schalter, muß immer mit einem Schrägstrich (/) anfangen. Bis auf /C, /Ib, /Ie und /Ii haben alle Switches nur eine Ein/Aus-Funktion, das heißt, sie können entweder angegeben oder weggelassen werden. In der folgenden Aufstellung ist jeder Switch mit seiner Funktion aufgelistet; dabei ist auch vermerkt, wie er eingestellt ist, wenn man aus VBDOS heraus kompiliert. „Nicht automatisch" bedeutet, daß Sie diesen Switch bei VBDOS in das Feld „Zusätzliche Optionen" eintragen müssen, wenn Sie ihn verwenden möchten.

Switch	*Bedeutung*

/A Wenn ein *listname* zusammen mit diesem Switch angegeben wird, erzeugt der Compiler zusätzlich ein Listing des Assembler-Codes, den er produziert. In VBDOS: nicht automatisch.

/Ah „Huge Arrays“ – Ermöglicht dynamischen Arrays, größer als 64 K zu sein. Für die Herstellung von Quick Libraries, die mit /Ea in VBDOS verwendet werden sollen, muß entweder /Ah oder /D beim Kompilieren verwendet werden. In VBDOS: Wird zwangsläufig gesetzt, wenn VBDOS mit /Ah aufgerufen wird, sonst nicht automatisch.

/C:*b* Setzt den Standard-Kommunikationspuffer auf *b* Bytes (für das Empfangen von Daten über OPEN COM). /C muß nicht benutzt werden, wenn man mit der Standardeinstellung von 512 Bytes für beide Schnittstellen zusammen zufrieden ist. In VBDOS: Wird automatisch eingestellt.

/D Produziert „debugging“-Code: Mit /D kompilierte Module können jederzeit mit Strg+Break abgebrochen werden, andere nur während einer INPUT-Anweisung. /D-Module achten auch darauf, daß nicht auf Array-Elemente mit ungültigen Indizes zugegriffen wird; bei ISAM führen sie nach jedem DELETE-, INSERT- und UPDATE-Befehl ein CHECKPOINT aus. Mit /D kompilierte Module sind länger. Dieser Switch hat mit dem CodeView-Debugger nichts zu tun. In VBDOS: wählbar („Fehlerüberprüfung zur Laufzeit“).

/E Muß angegeben werden, wenn das Modul die Befehle ON ERROR und RESUME mit Zeilennummer enthält. /E ist eine Untermenge von /X, ist also bei Verwendung von /X überflüssig. In VBDOS: wird automatisch eingestellt, wenn erforderlich.

/Es Teilt Expanded Memory zwischen BASIC und Routinen anderer Sprachen auf (nur für gemischtsprachliches Programmieren). In VBDOS: wird automatisch gesetzt, wenn VBDOS mit /Es aufgerufen wurde, sonst nicht automatisch.

/FPa Benutzt die „Alternate Math“-Libraries; keine Coprozessorunterstützung, dafür aber etwas schnellere Ausführung auf Rechnern ohne Coprozessor. Unterstützt nicht den CURRENCY-Datentyp. Zur Alternate Math-Library siehe Kapitel 15. In VBDOS: wählbar.

/FPi (Standard) Benutzt die „Emulator“-Libraries; Fließkomma-Berechnungen werden Coprozessor-konform formuliert, und wenn kein Coprozessor vorhanden ist, wird einer emuliert. Die entstehenden EXE-Programme unterstützen automatisch Coprozessoren. /FPa und /FPi schließen sich gegenseitig aus. In VBDOS: wählbar.

/G2 Produziert Code, der nur ab 80286-Prozessoren aufwärts funktioniert (das führt zu höherer Effizienz für diese Prozessoren). In VBDOS: wählbar.

/G3 Produziert Code, der nur ab 80386-Prozessoren aufwärts funktioniert. In VBDOS: wählbar.

/Ib:*x* Spezifiziert die Anzahl von ISAM-Puffern; 9 sind für vollen Funktionsumfang (6 für den reduzierten PROISAM-Betrieb, siehe Kapitel 12) mindestens erforderlich, 512 maximal erlaubt. Das Minimum wird benutzt, wenn der Switch nicht angegeben wird. In VBDOS: nicht automatisch.

Switch	*Bedeutung*
/Ie:*x*	Setzt die Menge an EMS (in KB), die für Nicht-ISAM-Anwendungen freigelassen werden soll; wenn /Ie weggelassen wird, benutzt ISAM bis zu 1,2 MB EMS. Ein Wert von –1 verbietet ISAM den EMS-Zugriff. In VBDOS: nicht automatisch.
/Ii:*x*	Setzt die maximale Anzahl von ISAM-Indizes (Null-Index wird nicht mitgerechnet), die in einem Programm benutzt werden können. Standard und Minimum ist 28, Maximum 500. In VBDOS: nicht automatisch.
/MBF	Ersetzt die Funktionen MKS$, MKD$, CVS und CVD durch ihre Pendants mit dem Anhängsel MBF, um Kompatibilität mit Uralt-Datenbanken zu gewährleisten. In VBDOS: wird zwangsläufig gesetzt, wenn VBDOS selbst mit /MBF aufgerufen wurde, sonst nicht automatisch.
/O	Sorgt dafür, daß beim Linken ein Stand-Alone-EXE-Programm erzeugt wird, ein Programm, das ohne das Runtime-Modul funktioniert. Arbeitet man ohne /O, dann werden kürzere EXE-Programme erzeugt, die allerdings nicht ohne das Runtime-Modul VBDRT10.EXE (für Profi-Version A oder E anhängen) funktionieren. Der Switch /O hat nur Einfluß auf die Standard-Runtime-Module. Wenn Sie ein eigenes Runtime-Modul benutzen und beim Linken die entsprechenden Angaben machen, wird der Switch /O ignoriert. Details über Runtime-Module siehe in Kapitel 14. In VBDOS: wählbar.
/R	Speichert Arrays nach Zeilen und nicht, wie üblich, nach Spalten. Ist nur sinnvoll, wenn es zur Kompatibilität mit anderen Sprachen benötigt wird. In VBDOS: nicht automatisch.
/S	Schreibt Strings direkt in die Objektdatei und nicht in die Symboltabelle; spart Platz bei Modulen mit wenig String-Konstanten (vgl. Kapitel 23).
/T	Unterdrückt die Warnungen, die der Compiler üblicherweise bei weniger schweren Fehlern ausgibt (zum Beispiel „Datenfeld nicht dimensioniert“ o. ä.) In VBDOS: immer gesetzt.
/V	Ermöglicht das Event Trapping; prüft nach jedem Befehl, ob eine der Trap-Bedingungen (Timer, Strig, Play, Pen, Uevent, Key) auftritt. Siehe bei ON *event* GOSUB im Disketten-Referenzteil. In VBDOS: Automatisch gesetzt, wenn ein ON *event* GOSUB-Befehl im Programm vorkommt.
/W	Wie /V, prüft aber nur an jeder Zeilennummer bzw. an jedem Zeilenlabel (!). /W ist eine Untermenge von /V und wird deshalb bei Einsatz von /V unnötig. In VBDOS: Nicht möglich, da VBDOS immer /V setzt, wenn das Programm Event Trapping enthält.
/X	Erlaubt ON ERROR und RESUME NEXT (erweitertes /E). In VBDOS: automatisch gesetzt, wenn erforderlich.
/Zd	Erzeugt eine Objektdatei, die (zur Bearbeitung mit Codeview oder SYMDEB) die Zeilennummern des Source-Files enthält. In VBDOS: nicht automatisch.
/Zi	Fügt neben den /Zd-Informationen noch zusätzliche Daten für den Microsoft CodeView-Debugger oder den Profiler in die Objektdatei ein (Achtung: später nicht mit /E linken). In VBDOS: nicht automatisch.

Wenn einer der Switches /E, /X, /W oder /V weggelassen wird, obwohl die Beschaffenheit des Programmes ihn erfordert, erzeugt der Compiler Fehlermeldungen.

Die schattierten Switches sind nur in der professionellen Ausgabe nutzbar.

Hinweis: Der Compiler BC.EXE unterscheidet sich bei beiden Versionen nur durch zwei Byte: An der Dateiposition 211h steht bei der Profi-Ausgabe der Code CDh 05h, während die Standardausgabe hier die Bytes 9Fh F3h aufweist. Das auf der Diskette beiliegende Programm BCPATCH.BAS „patcht“ Ihr BC.EXE und wandelt die Standard- in die Profiausgabe und umgekehrt um. Natürlich sind dadurch (da in der Standardversion viele Dateien fehlen) nicht alle Features der Profi-Ausgabe verfügbar. Der einzige Gewinn, den Sie als Standardversion-Benutzer haben, ist die Möglichkeit des Einsatzes der Switches /G2 und /G3.

Damit hätte ich BC abgehandelt; wie Sie aber schon an den Beispielen gesehen haben, ist das erst die halbe Miete. Der Compiler erzeugt ja nur ein .OBJ-File, das nicht gestartet werden kann. Es muß erst mit dem Programm LINK zu einem EXE-File gemacht werden, bevor man es aufrufen kann.

Anstatt direkt ein EXE-File erzeugen zu lassen, können Sie allerdings auch aus einer oder mehreren OBJ-Dateien eine Library zusammenstellen (siehe weiter unten in diesem Kapitel) oder ein eigenes Runtime-Modul produzieren (siehe Kapitel 14).

Beispiele zu BC-Aufrufen

```
BC PROG1;
```

PROG1.BAS wird zu PROG1.OBJ kompiliert. Kein Switch wurde angegeben, also wird standardmäßig mit der Emulator-Library (als hätten Sie /FPi angegeben) gearbeitet. PROG1.BAS darf, wenn Sie so kompilieren, keine ON ERROR- und keine RESUME-Befehle und außerdem auch kein ON *event* GOSUB enthalten.

Das entstehende Programm wird später das BASIC-Runtime-Modul benötigen (in diesem Fall, VBDRT10.EXE in der Standard- bzw. VBDRT10E.EXE in der professionellen Ausgabe).

```
BC PROG1 /S;
```

Hat prinzipiell dieselbe Wirkung wie der o.g. Befehl, produziert aber oft etwas kleinere und schnellere Programme.

```
BC PROG1,PROGRAMM,LISTE /A/O/V/X/S;
```

Kompiliert PROG1.BAS in PROGRAMM.OBJ, erzeugt dabei ein Programmlisting mit Assembler-Code (/A) in LISTE.LST; das Programm darf sowohl die Befehle ON ERROR/RESUME NEXT als auch ON event GOSUB enthalten (/X und /V) und wird außerdem später ohne das Runtime-Modul funktionieren, weil die BASIC-Routinen beim Linken direkt eingebunden werden (/O).

6.4 LINK

LINK dient dazu, aus OBJ-Dateien (die der BASIC-Compiler, ein beliebiger anderer Microsoft-kompatibler Compiler oder ein Assembler erstellt hat) und Libraries (die eigentlich auch nur umstrukturierte OBJ-Dateien sind), ein lauffähiges EXE-Programm oder eine Quick Library für den Gebrauch in VBDOS zu machen.

Eine Bemerkung vorweg: Es sind die verschiedensten Versionen des LINK-Programms im Umlauf. Ich beziehe mich hier auf die Version von LINK, die mit VBDOS geliefert wird. Es ist möglich, daß ältere oder neuere LINK-Versionen in kleinen Details, vornehmlich im einen oder anderen Switch, von der hier beschriebenen Version abweichen. Im Großen und Ganzen sind jedoch alle LINK-Versionen weitestgehend kompatibel, und alle können benutzt werden, um BASIC-Programme zu linken. VBDOS warnt Sie bei der Installation, wenn Sie eine andere LINK-Version auf der Festplatte haben.

Was tut LINK?

Während des LINK-Vorgangs werden unter anderem die BASIC-Hilfsroutinen zu Ihrem Programm hinzugefügt; deshalb benötigen Sie zum Linken immer auch die Library VBDRT10.LIB (mit Runtime-Modul) bzw. VBDCL10.LIB (selbständiges EXE-Programm). (In der professionellen Ausgabe ist jede Library doppelt vorhanden, einmal mit dem Appendix „E“ für Emulator-, einmal mit „A“ für Alternate Math-Library. Mehr dazu in Kapitel 15.)

Es werden nur die Routinen eingebunden, die Ihr Programm braucht. Welche das genau sind und wieviele und welche (es sind über 1000) Routinen in diesen Libraries überhaupt enthalten sind, wird Sie kaum interessieren*. Da Sie ohnehin nicht über die Dokumentation dieser internen Routinen verfügen, können

* Falls doch, finden Sie weiter unten im Abschnitt über LIB die Beschreibung, wie Sie eine Liste aller in einer Library enthaltenen Routinen ausgeben lassen können. Außerdem beschäftigen sich die Kapitel 18 und 23 unter anderem mit den BASIC-eigenen Routinen.

Sie sie nicht direkt aufrufen. Der Compiler übersetzt die meisten der Befehle in Ihrem Programm in Aufrufe an solche internen Funktionen, ohne daß Sie etwas davon merken.

Der LINK-Aufruf

Die Syntax des LINK-Programms ist folgende:

```
LINK objektname [+objectname...] [,[exename] [,[listname] [,[libname]
     [+libname...] [, definition]]]] [switches] [;]
```

oder, im Aufruf einfacher:

```
LINK @steuerungsdatei
```

Der Linker stellt zuerst alle Prozedur- und Funktionsaufrufe fest, die in den OBJ-Dateien vorkommen, und prüft dann, ob auch alle dazu benötigten Funktionen und Prozeduren vorhanden sind. Wenn in den OBJ-Dateien selbst nicht alle benötigten Routinen gefunden werden, sucht der Linker in den angegebenen Libraries danach und kopiert die Routinen von dort, bis alle geforderten Funktionen und Prozeduren gefunden sind. Ist ihm das nicht möglich, gibt er eine Fehlermeldung aus, die den Namen der fehlenden Prozedur enthält.

Beschäftigen wir uns zunächst mit der oberen Aufrufsyntax.

Hinter dem LINK-Befehl müssen Sie alle OBJ-Dateien – mit „+“ oder Leerzeichen verbunden – angeben, die in das spätere EXE-File eingebunden werden sollen. Mindestens eine muß angegeben werden, und die erste angegebene Datei muß entweder aus einem Formmodul (.FRM) entstanden sein oder Modulcode enthalten. Eine Ausnahme davon ist nur die Erstellung von Quick Libraries, bei er es nicht sinnvoll ist, überhaupt Modulcode zu verwenden, weil eine Quick Library niemals ausgeführt wird (siehe Switch /Q).

Jede angegebene OBJ-Datei wird vollständig eingebunden, egal ob sie benötigt wird oder nicht. Wenn mehrere der angegebenen OBJ-Dateien Modulcode enthalten, wird aller Modulcode bis auf den des ersten OBJ-Files ignoriert, aber trotzdem eingebunden. Er verbraucht so unnötig Platz.

Die Extension .OBJ können Sie sich sparen, LINK nimmt sie automatisch an. Sie können in diese Liste auch Libraries aufnehmen, diese dann allerdings mit ihrer Extension .LIB. Sie werden vollständig eingebunden, wenn sie an dieser Stelle aufgeführt sind.

Nach den Namen der OBJ-Dateien folgt – optional – der Name des EXE-Files. Wenn hier keiner steht, wird der Name des ersten OBJ-Files genommen und .EXE angehängt.

Als drittes in der Liste wird der Name einer List-Datei angegeben, in die LINK eine komplette Liste aller Routinen, Symbole und ihrer Adressen ausgibt. Wenn Sie mit Maschinensprache nichts im Sinn haben und kein Interesse für Adressen und Speicherzuordnung aufbringen, können Sie auf diese Liste getrost verzichten. Standard ist in diesem Feld ohnehin NUL, also keine Listenerzeugung.

Der vierte Parameter ist wieder von größerer Bedeutung. Hier werden Libraries (.LIB) angegeben, in denen nach Funktionen und Prozeduren gesucht werden soll, wenn sie in den OBJ-Dateien fehlen. Hier müßte eigentlich stets mindestens der Name einer BASIC-Library stehen, aus der die internen BASIC-Routinen entnommen werden können (VBDRT10.LIB o.ä.); diesen Standard-Namen, der abhängig davon ist, ob Sie die Standard- oder die professionelle Ausgabe verwenden und ob Sie /FPa angegeben haben oder nicht, notiert aber schon der Compiler in der OBJ-Datei, so daß LINK automatisch die richtige Library verwendet, ohne daß Sie sie explizit angeben müssen.

Für gewöhnliche Programme müssen Sie in diesem Feld überhaupt nichts angeben; Sie werden das Library-Feld zum Beispiel dann benutzen, wenn Sie ein Programm erstellen, das eine oder mehrere der Toolboxen benötigt. In der Beschreibung der einzelnen Toolboxen steht genau, welche Libraries Sie zu erwähnen haben, wenn Sie die jeweilige Toolbox verwenden.

Das fünfte Feld auf der Befehlszeile, *definition*, gibt den Namen einer speziellen Definitionsdatei an, die zur Steuerung des Overlay-Managers verwendet werden kann, für BASIC-Programmierer aber sonst keine weitere Bedeutung hat. Auch für Overlays stelle ich hier eine Technik vor, die das Feld nicht benötigt – siehe Kapitel 9.

Wenn Sie ein Semikolon an das Ende der LINK-Zeile setzen, bedeutet das, daß LINK bei Parametern, die Sie weglassen, nicht extra nachfragt. Würden Sie zum Beispiel einfach `LINK PROGRAMM` eingeben, so müßten Sie noch Fragen nach dem Namen der EXE-Datei, der Listendatei, den Libraries und der Definitionsdatei beantworten. Benutzen Sie stattdessen `LINK PROGRAMM;`, dann entfällt die Fragerei, und LINK verwendet die Standardvorgaben (in diesem Fall also: keine List-Datei, keine Definitionsdatei, keine zusätzlichen Libraries, Name für die EXE-Datei ist PROGRAMM.EXE).

LINK-Switches

Nun zu den *switches*. LINK kennt eine ganze Anzahl von Schaltern, die auch hier alle mit einem Schrägstrich (/) beginnen müssen. Sie können die Schalter entweder auf der Befehlszeile angeben (wie in der Syntax-Beschreibung), oder Sie setzen eine Betriebssystemvariable namens LINK mit den Switches, die Sie

verwenden wollen. Wenn Sie zum Beispiel den DOS-Befehl `SET LINK=/EX/NOE` eingeben, hat das dieselbe Wirkung, als würden Sie von nun an bei jedem LINK-Befehl die Switches /EX und /NOE angeben. Im folgenden eine Übersicht über die für BASIC-Programmierer interessanten Switches:

Switch	*Bedeutung*
/BA	Batch-Modus. LINK gibt keine Copyright-Meldung und auch nicht den Inhalt einer Steuerungsdatei auf dem Bildschirm aus. Wenn Libraries oder Objektdateien nicht gefunden werden können, fragt LINK nicht nach.
/CO	„Codeview“ – Bindet zusätzliche Information in das EXE-Programm ein, die mit CodeView benötigt wird (nur sinnvoll, wenn zuvor mit /Zi oder /Zd kompiliert wurde).
/DY:*n*	„Dynamic“ – Setzt die maximale Zahl von Prozeduraufrufen zwischen Overlays; siehe Kapitel 9.
/E oder /EX	„EXE pack“ – Komprimiert das entstehende EXE-File. Kleinere EXE-Files können schneller geladen werden, deshalb sollte man diesen Switch immer verwenden. In manchen – äußerst seltenen – Sonderfällen kann es passieren, daß das komprimierte File länger ist, als es das nichtkomprimierte wäre; LINK gibt dann eine entsprechende Meldung. /E entfernt unnötige Informationen und kann deshalb nicht mit /CO benutzt werden. /E ist ebenfalls inkompatibel zu /Q.
/F/PACKC	„Far call translation & Pack code“ – Es handelt sich hier um zwei verschiedene Switches. Sie sollten sie nur gemeinsam anwenden, da /F ohne /PACKC nicht so effizient arbeiten kann. Durch eine leichte Umstrukturierung des EXE-Files erreichen diese Switches insbesondere bei größeren Programmen eine weitere Verkleinerung des entstehenden Programms. Programme, die mit diesem Switch erzeugt wurden, laufen auch zumeist etwas schneller – insbesondere auf Rechnern mit 80286- oder höherem Prozessor. Laut Microsoft gibt es ein „kleines Risiko“, daß LINK mit dem Switch /F unerlaubte Optimierungen durchführt und Ihr Programm dann aus scheinbar unklärbaren Gründen abstürzt. In solchen Fällen, die aber bei BASIC eigentlich nicht möglich seien, müsse man ohne diesen Switch linken (mir ist allerdings kein derartiger Fall bekannt).
/HE	„Help“ – Zeigt eine Liste der Switches an.
/INF	„Information“ – Zeigt beim Linken genau an, was in welcher Reihenfolge abgearbeitet wird.
/LI	„Line numbers“ – Schließt in die Listendatei, falls sie überhaupt erzeugt wird, die Adressen von Zeilennummern ein. /LI ist nur sinnvoll, wenn Sie beim Kompilieren /Zi oder /Zd angegeben haben.
/M	„Map“ – Schließt in die Listendatei, falls sie überhaupt erzeugt wird, eine sortierte Liste aller globalen Symbole ein. Standard für das Feld *listname* ist dann *exename* mit angehängtem .MAP (und nicht länger NUL).

Switch	*Bedeutung*
/NOD	„No default library search“ – Wie erwähnt, benutzt LINK die Standard-Library, deren Namen der Compiler schon in die OBJ-Datei einträgt. Mit /NOD können Sie verhindern, daß solche in OBJ-Dateien erwähnten Libraries benutzt werden. Wenn Sie dann allerdings nicht für Ersatz sorgen, werden Fehler beim Linken auftreten, da der Linker die BASIC-Hilfsroutinen nicht mehr findet.
/NOE	„No extended dictionary“ – LINK wendet zum Heraussuchen der benötigten Prozeduren kein „extended dictionary“ an. Das macht den LINK-Prozeß langsamer, ist aber bei komplizierten Link-Vorgängen manchmal notwendig (LINK gibt selbst die Meldung „... benutzen Sie /NOE“ aus), und zwar dann, wenn dieselbe Routine in mehreren angegebenen Libraries vorhanden ist.
/NOF/NOP	„No far call translation & No pack code“ – Ist die Gegenfunktion zu /F/PACKC. /NOF/NOP ist ausschließlich dann sinnvoll, wenn Sie in die Betriebssystemvariable LINK (mit SET LINK=...) den Switch /F/PACKC eingetragen haben, ihn aber für einen einzelnen LINK-Vorgang nicht benutzen wollen, so daß es nicht lohnt, die Variable zu ändern. Dann können Sie /NOF/NOP auf der Befehlszeile angeben. Dadurch wird das /F/PACKC in der Systemvariable ignoriert.
/NOL	„Nologo“ – Verhindert die Anzeige der Copyright-Meldung beim Start von LINK.
/NON	Verringert die Größe des EXE-Files um 16 Bytes, da mit /NON vor dem _TEXT-Segment keine 16 Null-Zeichen eingefügt werden, wie das sonst üblich ist. Für BASIC-Programme können Sie getrost auf diese 16 Null-Zeichen verzichten.
/O:*nr*	„Overlay interrupt“ – Benutzt statt des Interrupts 63 (Standard) den Interrupt *nr* als Overlay-Interrupt. /O ist nur relevant, wenn Sie mit Overlay-Technik arbeiten, und sollte auch nur verwendet werden, wenn Ihr Programm nicht korrekt funktioniert, weil ein anderes aktives Programm auch den Interrupt 63 benutzt. Sie sollten dann (und nur dann) mit /O eine beliebige andere freie Interrupt-Nummer auswählen (75 bis 103 sind zum Beispiel normalerweise ungenutzt).
/Q	„Quick Library“ – Erzeugt eine Quick Library. Mehr dazu im Abschnitt 7 dieses Kapitels.
/SE:*x*	„Segments“ – Setzt die maximal erlaubte Anzahl von Segmenten für das Programm. Der Standard ist 128, und wenn Sie beim Linken die Meldung *Zu viele Segmente* erhalten, sollten Sie /SE mit einem Wert größer 128 angeben, bis es klappt. Für kleine Programme können Sie die Segmentzahl auch herabsetzen. Das Maximum für *x* ist 3072.
/ST:*x*	„Stack“ – setzt die Stackgröße (in Byte, 16 bis 65.534). Dieser Switch hat die gleiche Wirkung wie der STACK-Befehl von VBDOS; Sie sollten /ST nur dann verwenden, wenn Sie ein Formmodul kompilieren, das mehr Stack benötigt als Standard ist. In Formmodulen ist nämlich kein Modulcode erlaubt, und der VBDOS-Befehl STACK kann nur im Modulcode eingesetzt werden.

Sie ahnen vielleicht, daß man durch vernünftigen Einsatz der verschiedenen Schalter – sowohl bei LINK als auch beim Compiler BC – einiges an Bytes und Sekunden einsparen kann. Näheres dazu finden Sie in Kapitel 23.

Eine Übersicht über Fehlermeldungen, von denen LINK eine ganze Reihe parat hat, finden Sie in Anhang C.

LINK-Steuerungsdatei

Bei komplexeren Systemen kann es vorkommen, daß die Befehlszeile (in DOS können Sie maximal 128 Zeichen auf einer Zeile eingeben) nicht ausreicht, um alle gewünschten Optionen anzugeben. Dann können Sie entweder LINK ohne Parameter oder nur mit Switches aufrufen. Sie werden in diesem Fall nach den Namen für Objektdateien, EXE-File, Listendatei, Libraries und Definitionsdatei gefragt. Eine andere Möglichkeit ist die, eine Steuerungsdatei zu erstellen, die in fünf Zeilen die Eintragungen für die fünf Felder (Objects, EXE, Liste, Libraries und Definition) enthält. Um diese Datei übersichtlicher zu machen, ist es auch möglich, die Zeile für die OBJ-Dateien (erste Zeile) und die für LIB-Dateien (vierte Zeile) auszudehnen. Schreiben Sie einfach als letztes Zeichen auf der Zeile ein +, dann nimmt LINK an, daß die nächste Zeile auch noch dazugehört.

Eine korrekte, vollständige Steuerungsdatei sollte also *genau fünf Zeilen* enthalten, die *nicht mit einem + aufhören*. Switches können übrigens an beliebiger Stelle untergebracht werden – vorzugsweise jedoch als erstes in der OBJ-Zeile.

Sie können statt

```
LINK HAUPT+HALLO+PRINTER+ENDE,,,CHRTBEFR+FONTBEFR/EX;
```

auch schreiben:

```
LINK @HAUPT.CMD
```

wenn in HAUPT.CMD folgende Zeilen (ohne Nummern!) stehen:

```
(1) /EX HAUPT+
(2) HALLO+PRINTER+ENDE
(3)
(4)
(5) CHRTBEFR+FONTBEFR
(6)
```

Die Extension .CMD ist hier willkürlich gewählt und könnte auch eine beliebige andere sein.

Die dritte und vierte Zeile sind leer, denn auch im LINK-Befehl wurden das zweite und dritte Feld leer gelassen. Die Zeilennummer ist hier um eins größer als die Feldnummer, weil die Zeilen eins und zwei dasselbe Feld beschreiben (+

am Ende der Zeile 1). Auch für das fünfte Feld steht in der Steuerungsdatei eine Leerzeile, weil es ebenfalls beim LINK-Befehl leer gelassen wurde. Dort hatte ich mir das vierte Komma gespart und stattdessen mit dem Semikolon angedeutet, daß LINK keine Fragen mehr stellen soll.

Man kann eine Steuerungsdatei auch vorzeitig beenden, indem man, wie auf der Befehlszeile, ein Semikolon verwendet. Hätte ich an das Ende der Zeile 5 ein Semikolon geschrieben, hätte ich mir Zeile 6 sparen können.

6.5 LIB und Libraries

Der „Microsoft Library Manager“ LIB.EXE dient zum Erstellen und Verändern von gewöhnlichen Libraries (.LIB), die Sie nicht mit Quick Libraries (.QLB) verwechseln dürfen. Nichtsdestotrotz wird häufig parallel zu jeder Library, die man erzeugt, auch eine Quick Library erstellt, um die in der Library enthaltenen Routinen auch innerhalb von VBDOS nutzen zu können.

Gleich vorweg: LIB ist nur ein Hilfsprogramm, und es gibt nichts, was man nicht auch ohne LIB tun könnte. Libraries sind einfach Sammlungen von mehreren OBJ-Dateien, also von kompilierten Routinen. Indem man alle OBJ-Dateien, die man beim Linken eines Programms üblicherweise angibt (bis auf die eine mit Modulcode), in einer Library vereint, erspart man sich das Auflisten aller OBJ-Dateien. Man muß nur noch die Library angeben, und der Linker sucht sich dann selbst die benötigten OBJ-Dateien aus der Library heraus.

Es lohnt sich, OBJ-Dateien, die man häufig braucht, aber kaum noch verändert (zum Beispiel eine Sammlung von Routinen zur Plottersteuerung), in einer Library zusammenzufassen. Microsoft liefert die Toolboxen als fertige Libraries auf den Disketten mit.

Sie sollten in Libraries keine OBJ-Dateien einbinden, die Modulcode besitzen. Dadurch werden die Library und evtl. das spätere EXE-Programm unnötig verlängert, denn der Modulcode wird nie ausgeführt.

LIB-Aufruf

Das Programm LIB wird wie folgt angewandt:

```
LIB library[switches] [befehle] [, [listfile] [, neulib] [;]
```

oder

```
LIB @steuerungsdatei
```

library ist dabei der Name einer bestehenden oder zu erzeugenden Library (.LIB muß nicht angegeben werden). Üblicherweise speichert LIB beim Verändern einer bestehenden Library die neue Version unter dem gleichen Namen ab wie die ursprüngliche und bewahrt die alte Version unter der Extension .BAK. Wenn jedoch *neulib* angegeben wird, speichert LIB die erzeugte Library unter diesem Namen.

listfile kann ebenfalls weggelassen werden; trägt man hier einen Namen ein, wird in der betreffenden Datei eine Liste aller Routinen der Library mit dem Namen der OBJ-Datei erzeugt, in der LIB sie gefunden hat (siehe dazu Kapitel 23) weiter unten in diesem Kapitel). *befehle* ist eine Kette von beliebig vielen LIB-Befehlen. Wenn Sie überhaupt keinen Befehl angeben, erzeugt LIB nur eine Liste in *listfile*. Wenn Sie auch das *listfile*-Feld leer lassen, dann prüft LIB lediglich, ob die angegebene Library keine Fehler enthält.

LIB-Befehle bestehen immer aus einem oder zwei Befehlszeichen und einem nachfolgenden OBJ-Dateinamen.

Befehl	*Wirkung*
+	Die angegebene OBJ-Datei muß auf der Platte vorhanden sein und wird an die Library angefügt. Hinter dem Befehl + kann auch statt einer OBJ-Datei eine andere Library angegeben werden (die Extension LIB darf dann nicht weggelassen werden). Dann werden alle OBJ-Dateien aus dieser Library in die gerade bearbeitete übernommen, so als hätten Sie sie erst einzeln aus der einen Library herausgeholt und dann einzeln in die gerade bearbeitete aufgenommen.
–	Die angegebene OBJ-Datei wird aus der Library gelöscht (sie muß nicht auf der Platte vorhanden sein).
–+	Die angegebene OBJ-Datei muß auf der Platte sein. Wenn eine OBJ-Datei gleichen Namens in der Library ist, wird sie aus der Library gelöscht. Dann wird das angegebene File in die Library aufgenommen.
*	Die angegebene OBJ-Datei wird aus der Library in eine echte OBJ-Datei auf der Platte kopiert (die Library bleibt dabei unverändert)
–*	Die angegebene OBJ-Datei wird in eine OBJ-Datei auf der Platte verwandelt und aus der Library entfernt.

Durch Verwendung des *-Befehls können Sie aus fertigen Libraries Teile extrahieren, um sie in anderen Libraries zu verwenden (so können Sie z.B. aus VBDOS.LIB die Routinen für Interruptaufrufe nehmen und in eine eigene Library kopieren, ohne die anderen in VBDOS.LIB enthaltenen Routinen ebenfalls übernehmen zu müssen).

Als *switches* sind für den BASIC-Programmierer nur drei interessant:

Switch	*Bedeutung*
/PA:	Setzt die „Seitengröße“ der Library auf *n*. Die Standard- und kleinstmögliche Seitengröße ist 16 Bytes. Die maximale Größe einer Library ist das 65.536fache der Seitengröße (bei der Standard-Seitengröße von 16 Bytes also genau 1 MB). Erhöhen Sie die Seitengröße, wenn Sie größere Libraries erzeugen möchten.
/NOE	Benutzen Sie diesen Switch nur, wenn LIB meldet „...benutzen Sie /NOEXTDICTIONARY“. Er erzeugt eine etwas kleinere Library, die aber langsamer zu linken ist.
/NOLOGO	Zeigt die LIB-Copyright-Meldung nicht an.

Wenn Sie am Ende der LIB-Zeile ein Semikolon angeben, wird dadurch weiteres Nachfragen von LIB verhindert. Üblicherweise fragt LIB, wenn Sie mindestens eines der Felder leer lassen, im Dialog die fehlenden Inhalte ab. Außerdem fragt LIB, wenn Sie eine Library angeben, die noch nicht existiert, ob diese erzeugt werden soll. Durch das Anhängen eines Semikolons werden alle von Ihnen leergelassenen oder nicht angegebenen Felder auf ihre Standardwerte gesetzt. Standard ist NUL für *listfile*, der gleiche Name wie *library* für *neulib* und „j“ als Antwort auf die Frage, ob die Library neu erstellt werden soll.

Beispiele für LIB-Aufrufe

```
LIB MODULE +INHALT;
```

fügt INHALT.OBJ an die Library MODULE.LIB an.

```
LIB MODULE +TOOLS.LIB,,MODULNEU;
```

kombiniert die Libraries MODULE.LIB und TOOLS.LIB in eine neue Library namens MODULNEU.LIB. Die zwei Kommata müssen angegeben werden, damit LIB bemerkt, daß das *listfile*-Feld leer ist und nicht etwa den Inhalt MODULNEU hat.

```
LIB MODULE -+INHALT,MODULE.LST;
```

entfernt das alte INHALT.OBJ aus der Library MODULE.LIB und setzt anstelle dessen das neue INHALT.OBJ ein; eine Liste aller Routinen wird in MODULE.LST geschrieben. Was Sie mit dieser Liste anfangen können, lesen Sie im Kapitel 23.

Steuerungsdateien bei LIB

Wenn Sie sich Arbeit ersparen wollen, können Sie das, was normalerweise in der Befehlszeile angegeben wird, auch in eine Steuerungsdatei schreiben (ähnlich, wie es auch bei LINK möglich ist).

Eine Steuerungsdatei sollte bei LIB aus vier Zeilen bestehen: Die erste enthält den Library-Namen, die zweite die LIB-Befehle, die dritte den Namen für die List-Datei und die vierte den Namen für die neue Library. Natürlich können Sie auch hier Zeilen leer lassen. Wenn die Library, die in der ersten Zeile erwähnt wird, noch nicht vorhanden ist, muß nach der ersten noch eine weitere Zeile mit einem J darin eingefügt werden als Antwort auf die Frage, ob die Library neu erstellt werden soll.

Wenn die Steuerungsdatei weniger als 4 Zeilen hat, fragt LIB die fehlenden Informationen im Dialog ab, es sei denn, Sie beenden die letzte Zeile Ihrer Steuerungsdatei mit einem Semikolon – dann entfallen weitere Fragen, einschließlich der Frage, ob die Library neu erstellt werden soll, wenn die angegebene Library noch nicht existiert.

Falls die zweite Zeile (die mit den LIB-Befehlen) zu lang wird, können Sie diese auch ausdehnen, indem Sie am Ende der Zeile ein kaufmännisches Und-Zeichen (&) schreiben. Dann wird der Inhalt der Folgezeile auch noch dem Befehlsfeld zugerechnet. Dies läßt sich beliebig oft wiederholen.

Statt

```
LIB TEST +RTC+ABD+HALLO+HILFE,,TESTNEU
```

könnte man also schreiben

```
LIB @TEST.CMD
```

wenn TEST.CMD den folgenden Inhalt hat (ohne Zeilennummern und unter der Voraussetzung, daß TEST.LIB schon existiert):

```
(1) TEST
(2) +RTC+ABD &
(3) +HALLO+HILFE
(4)
(5) TESTNEU
```

Wenn TEST.LIB noch nicht existierte, müßte zwischen (1) und (2) noch eine Zeile mit einem J eingefügt werden, als Antwort auf die Frage, ob die Library neu erstellt werden soll.

Switches werden in der Steuerungsdatei am Ende der ersten Zeile angegeben, wobei /NOLOGO keine Wirkung hat, da die Copyright-Meldung schon angezeigt wird, bevor LIB die Datei liest.

Linken mit Libraries

Sie kennen bereits die Arbeitsweise des LINK-Programms. Nehmen wir an, Sie linken mit folgender Zeile ein Programm:

```
LINK TEST+TOOLS1+TOOLS2+TOOLS3;
```

Dabei werden die Dateien TOOLS1.OBJ, TOOLS2.OBJ und TOOLS3.OBJ zusammen mit TEST.OBJ, das Modulcode enthalten muß, zu TEST.EXE gelinkt. Die TOOLS-Dateien werden vollständig eingebunden.

Stattdessen können Sie auch alle TOOLS-Dateien in eine Library zusammenfassen, und zwar mit diesem LIB-Befehl:

```
LIB TOOLS +TOOLS1+TOOLS2+TOOLS3;
```

Der äquivalente LINK-Befehl zum oben erwähnten hieße dann:

```
LINK TEST+TOOLS.LIB;
```

Es ist ja möglich, zusammen mit OBJ-Dateien auch LIB-Dateien bei LINK anzugeben. Was ist der Unterschied zwischen dieser Schreibweise und der folgenden?

```
LINK TEST,,,TOOLS;
```

Hier wurde die TOOLS-Library im für Libraries vorgesehenen Feld bei LINK angegeben, deshalb kann auch die Extension .LIB weggelassen werden (vgl. Abbildung 6–3 auf Seite 67).

Der Unterschied ist, daß bei der letztgenannten Schreibweise aus TOOLS.LIB nur das entnommen wird, was von TEST.OBJ wirklich aufgerufen wird, während bei `LINK TEST+TOOLS.LIB;` einfach die gesamte Library eingebunden wird.

Es ist also in den meisten Fällen vorteilhaft, die Libraries als viertes in der LINK-Zeile anzugeben, damit LINK selektieren kann, welche Teile daraus eingebunden werden und welche nicht.

Libraries optimieren

Im Lieferumfang von VBDOS sind einige Libraries mit zugehörigem Quellcode enthalten: CMNDLG.LIB, FONT.LIB und CHART.LIB. Für diese und auch alle selbsterstellten Libraries gilt, daß Sie beim Kompilieren der Module, die Sie in die Library aufnehmen möchten, den Switch /G2 oder /G3 angeben können, wenn die Programme später nur auf 286- bzw. 386- und größeren Rechnern laufen. Die mitgelieferten Libraries sind nicht mit diesen Switches erstellt; Sie

können sie also etwas optimieren, indem Sie sie neu erstellen (vgl. auch Kapitel 13 und 23).

6.7 Quick Libraries

Quick Libraries werden wie gewöhnliche EXE-Programme unter Verwendung des LINK-Programms erzeugt – mit zwei Ausnahmen:

- Modulcode in Quick Libraries wird nie ausgeführt, während bei EXE-Dateien zumindest der Modulcode des ersten angegebenen Moduls (der Startdatei) ausgeführt wird.
- Quick Libraries werden mit dem Switch /Q und der Library VBDOSQLB.LIB gelinkt.

Eine Quick Library ist ausschließlich für die Verwendung mit VBDOS gedacht. Sie wird erstellt, um die Routinen (SUBs, FUNCTIONs und evtl. auch Unterprogramme in anderen Sprachen) aus OBJ-Dateien und Libraries in VBDOS verwenden zu können.

Wenn Sie zum Beispiel meine Warteanzeige WARTEN.FRM aus Kapitel 21 in Ihren Programmen nutzen wollen, werden Sie womöglich auf die Idee kommen, diese Warte-Anzeige zusammen mit anderen nützlichen Routinen in einer Library zu verbinden. Nehmen wir an, diese Library hieße TOOLS.LIB.

Haben Sie diese Library zur Verfügung, können Sie sie in Zukunft beim Linken jedes Ihrer Programme angeben. Sobald ein Programm die Warte-Anzeige (oder andere Routinen, die in der Library stehen) verwendet, wird LINK die benötigten Teile aus der Library in Ihr Programm übernehmen.

Die Library TOOLS.LIB allerdings kann von VBDOS nicht gelesen werden, und so würde VBDOS ein Programm, das Routinen aus der Warte-Anzeige aufruft, so lange nicht laufen lassen, bis Sie die Form WARTEN.FRM explizit hinzuladen. Ebenso müßten Sie alle anderen „Universalroutinen“, die Sie in TOOLS.LIB zusammengefaßt hatten, in VBDOS laden, um sie dort zu verwenden.

Um dies zu vermeiden, können Sie eine Quick Library erstellen. Dazu verwenden Sie üblicherweise die OBJ-Dateien, z.B. also so:

```
LINK WARTEN.OBJ,TOOLS.QLB,,VBDOSQLB/Q;
```

(Sie würden natürlich hinter WARTEN.OBJ noch weitere OBJ-Dateien anfügen, wenn Sie mehr als die Warte-Anzeige in der Library haben möchten.)

Der Switch /Q ist der Befehl, eine Quick Library zu erstellen, und VBDOSQLB.LIB (eine mitgelieferte Library) enthält Routinen, die LINK dazu braucht. Am Ende dieses Vorgangs steht die Quick Library TOOLS.QLB, und in Zu-

kunft können Sie durch Eingabe von `VBDOS /L TOOLS` alle in TOOLS.QLB enthaltenen Routinen in VBDOS aufrufen, ohne die zugehörigen Quelltexte zu laden. Außerdem wird VBDOS, wenn Sie „EXE-Datei erstellen“ aus dem „Ausführen“-Menü wählen, automatisch beim LINK-Befehl die Library TOOLS.LIB angeben (in der Annahme, daß sie diese ebenfalls erzeugt haben), damit die Routinen von dort auch in das erzeugte EXE-File übernommen werden können.

/Q sollte nicht zusammen mit anderen Switches verwendet werden, die auf das zu erzeugende EXE-File einwirken (wie /LI, /F/PACKC oder /E).

Hinweis: OBJ-Dateien, die in eine Quick Library gebunden werden, dürfen nicht mit dem Switch /FPa kompiliert worden sein. Sie erhalten sonst beim Laden der Quick Library eine Fehlermeldung.

Quick Libraries aus Libraries erstellen

Sie können zu jeder existierenden Library leicht eine passende Quick Library erstellen, indem Sie den Befehl

`LINK` *`libname`*`.LIB,`*`libname`*`.QLB,,VBDOSQLB/Q;`

verwenden, wobei Sie für *libname* den Namen der Library – z. B. FONT – einsetzen. Dies ist eine Alternative zur oben gezeigten Methode, bei der Sie erst die OBJ-Dateien mit dem Compiler BC.EXE neu erstellen müssen.

Quick Libraries optimieren

Libraries enthalten die Routinen, die in Ihre fertigen EXE-Programme eingebunden werden; Quick Libraries enthalten dieselben Routinen zur Verwendung in VBDOS. Wenn Sie also Programme erstellen, die später auf jedem Rechner verwendbar sein sollen, selbst aber mit einem 386er arbeiten, können Sie die Quick Libraries – nicht allerdings die Objektcode-Libraries – mit dem Compiler-Switch /G3 erstellen. Dadurch erreichen Sie eine etwas bessere Leistung in Ihrer VBDOS-Umgebung, ohne die entstehenden Programme zu verändern. Dieses Verfahren können Sie auch auf die mitgelieferten Quick Libraries anwenden, die bisher alle ohne /G2 oder /G3 erzeugt wurden.

Inhalt von Quick Libraries

Das mitgelieferte Programm QLBVIEW.FRM ermöglicht es Ihnen, den Inhalt einer Quick Library zu prüfen. Es ist jedoch weder möglich, einzelne Routinen aus einer Library zu entnehmen, noch können Sie Routinen zu einer Quick Library hinzufügen oder mehrere Quick Libraries verbinden. Stattdessen muß eine Quick Library für solche Veränderungen stets neu erstellt werden.

Auch deshalb ist es sinnvoll, parallel zu jeder Quick Library eine gewöhnliche Library zu erzeugen; diese kann nämlich jederzeit mit LIB verändert werden, und nach einer solchen Änderung läßt sich mit LINK schnell wieder eine Quick Library erstellen.

Ereignisgesteuerte Programmierung im Detail 7

7.1 Vom Standardprogramm zur Ereignissteuerung

Damit Ihnen der Einstieg in die ereignisgesteuerte Programmierung leichter fällt als mir seinerzeit, habe ich mich entschlossen, zu Anfang dieses Kapitels einmal ganz deutlich und Schritt für Schritt an einem Beispiel vorzuexerzieren, was ein ereignisgesteuertes Programm von einem gewöhnlichen BASIC-Programm unterscheidet.

Das Beispielprogramm

Als Studienobjekt habe ich ein kleines Programm vorgesehen, das einen gegebenen Text in mehreren Dateien durch einen anderen Text ersetzen kann. Die Dateien sind dabei in der Größe auf 20 KB begrenzt, damit sie auf einmal in den Speicher geladen werden können, aber das ist Nebensache. In der Hauptsache geht es hier ja um die Benutzerschnittstelle, und die sieht bei der Standardversion so aus:

```
Suchen und Ersetzen in mehreren Dateien
=======================================

Bitte geben Sie eine Dateibezeichnung an (* und ? erlaubt): h:\vbdos\*.bas
40 Dateien passen.
Bitte geben Sie den Suchtext ein: PRINT
Bitte geben Sie den Ersatztext ein: FROSCH
Groß-/Kleinschreibung beachten (J/N)? N
Alle Eingaben korrekt (J/N/A=Abbrechen): J

40 Dateien geprüft
18 Ersetzungen durchgeführt
13 Dateien größer als 20KB

Nochmal (J/N)? N
```

Abbildung 7–1: Die Bildschirmausgabe von ERS_NORM.BAS

In der Standardversion arbeitet das Programm ganz gewöhnlich mit PRINT und LINE INPUT; zwei große DO-LOOP-Schleifen sorgen dafür, daß man das Programm mehrmals benutzen und seine Eingaben außerdem korrigieren kann.

Wenn die Benutzereingabe (während der auch ein Verzeichnis eingelesen wird) abgeschlossen ist, muß das Programm erst noch die Pfadangabe aus der Dateibezeichnung des Benutzers extrahieren, da diese vor jeden Dateinamen gesetzt wird. Dann wird für jeden gefunden Dateinamen die Prozedur „ErsetzeInDatei“

aufgerufen, deren Innereien hier nicht von Bedeutung sind, weil sie auch später im ereignisgesteuerten Programm unverändert sein wird.

```
REM $INCLUDE: 'CONSTANT.BI'

CONST DateiZuGross = 1, Gefunden = 2, NichtGefunden = 3 ' Rückgabewerte Funktion
CONST MaxDateien = 999                                  ' ErsetzeInDatei

DIM DateiBezeichnung AS STRING, Suchen AS STRING, Ersetzen AS STRING
DIM AnzahlDateien AS INTEGER, GrossKlein AS INTEGER, Zeichen AS STRING
DIM DateiName(1 TO MaxDateien) AS STRING, Pfadangabe AS STRING
DIM Zaehler AS INTEGER, Ersetzungen AS INTEGER, ZuGross AS INTEGER

PRINT "Suchen und Ersetzen in mehreren Dateien"
PRINT "======================================="

DO ' Hauptschleife (damit Programm mehrmals benutzt werden kann) ------------------
   PRINT
   DO ' Eingabeschleife (damit Benutzer Eingaben korrigieren kann) ----------------
      DO
         PRINT "Bitte geben Sie eine Dateibezeichnung an (* und ? erlaubt): ";
         LINE INPUT DateiBezeichnung
         VerzeichnisLesen DateiBezeichnung, DateiName(), AnzahlDateien
         IF AnzahlDateien = 0 THEN
            PRINT "Keine Dateien gefunden!"
         ELSE
            PRINT FORMAT$(AnzahlDateien); " Dateien passen.": EXIT DO
         END IF
      LOOP

      DO
         PRINT "Bitte geben Sie den Suchtext ein: ";: LINE INPUT Suchen
         IF LEN(Suchen) = 0 THEN
            PRINT "Suchtext darf nicht leer sein!"
         ELSE
            EXIT DO
         END IF
      LOOP

      PRINT "Bitte geben Sie den Ersatztext ein: ";: LINE INPUT Ersetzen

      PRINT "Groß-/Kleinschreibung beachten (J/N)? ";
      DO: Zeichen = UCASE$(INPUT$(1)): LOOP UNTIL INSTR("JN", Zeichen)
      PRINT Zeichen: GrossKlein = (Zeichen = "J")

      PRINT "Alle Eingaben korrekt (J/N/A=Abbrechen): ";
      DO: Zeichen = UCASE$(INPUT$(1)): LOOP UNTIL INSTR("JNA", Zeichen)
      PRINT Zeichen: IF Zeichen = "A" THEN SYSTEM
   LOOP UNTIL Zeichen = "J" ' Ende Eingabeschleife -------------------------------
   Pfadangabe = ExtrahierePfad(DateiBezeichnung)

   FOR Zaehler = 1 TO AnzahlDateien
```

⇨

```
      SELECT CASE ErsetzeInDatei(Pfadangabe + DateiName(Zaehler), Suchen,
                                 Ersetzen, GrossKlein)
      CASE Gefunden: Ersetzungen = Ersetzungen + 1
      CASE DateiZuGross: ZuGross = ZuGross + 1
      END SELECT
   NEXT
   PRINT
   PRINT FORMAT$(AnzahlDateien); " Dateien geprüft"
   PRINT FORMAT$(Ersetzungen); " Ersetzungen durchgeführt"
   PRINT FORMAT$(ZuGross); " Dateien größer als 20KB"
   PRINT
   PRINT "Nochmal (J/N)? ";
   DO: Zeichen = UCASE$(INPUT$(1)): LOOP UNTIL INSTR("JN", Zeichen)
   PRINT Zeichen
LOOP UNTIL Zeichen = "N" ' Ende Hauptschleife
```

```
' Öffnet die angegebene Datei, führt die Ersetzung aus und schreibt die Datei
' wieder, wenn eine Ersetzung stattgefunden hat
' Gibt als Funktionswert die Konstante "DateiZuGross" zurück, wenn Datei > 20 KB;
' ansonsten wird "Gefunden" oder "NichtGefunden" zurückgegeben.
FUNCTION ErsetzeInDatei (Datei AS STRING, Such AS STRING, Ersetz AS STRING,
         GKlein AS INTEGER) AS INTEGER

' Diese Funktion ist hier nicht abgedruckt, da sie für das Beispiel ohne Belang
' ist. Sie ist jedoch im Programm auf der Diskette enthalten.
```

```
FUNCTION ExtrahierePfad (Bezeichnung AS STRING) AS STRING

   DIM Zaehler AS INTEGER

   FOR Zaehler = LEN(Bezeichnung) TO 1 STEP  1        ' von hinten : oder \ suchen
      SELECT CASE ASC(MID$(Bezeichnung, Zaehler, 1))
      CASE 92, 58                                     ' 92=\, 58=:
         ExtrahierePfad = LEFT$(Bezeichnung, Zaehler) ' wenn gefunden: alles bis
                                                      ' hier ist Pfadangabe
         EXIT FUNCTION                                ' Funktion verlassen
      END SELECT
   NEXT
   ExtrahierePfad = ""                                ' Dann bleibt's leer

END FUNCTION
```

```
' Liest mittels DIR$ ein Dateiverzeichnis in ein STRING-Array ein
SUB VerzeichnisLesen (PathSpec AS STRING, FileName() AS STRING, Number AS INTEGER)

   FileName(1) = DIR$(PathSpec): Number = 1
   DO WHILE LEN(FileName(Number))
      Number = Number + 1
      FileName(Number) = DIR$
   LOOP
   Number = Number   1

END SUB
```

Listing 7–1: ERS_NORM.BAS

Der Entwurf einer geeigneten Form

Da wir das Programm in eine ereignisgesteuerte Version übersetzen wollen, brauchen wir (mindestens) eine Form, über die die Kommunikation mit dem Benutzer abgewickelt wird. Die Befehle PRINT und LINE INPUT sind, wie Sie dem Referenzteil entnehmen können, in ereignisgesteuerten Programmen nicht zulässig (oder nur, während keine Formen angezeigt werden).

Um die Form zu erstellen, gilt es zunächst zu bestimmen, welche Eingaben durch den Benutzer erfolgen müssen und welche Ausgaben das Programm tätigt. Bei den Ausgaben ist zu unterscheiden zwischen solchen, die nur in besonderen Fällen kurz angezeigt werden und dann wieder verschwinden, und solchen, die dauernd sichtbar sein sollen.

Als Eingaben benötigen wir
- die Dateibezeichnung,
- den Suchtext,
- den Ersatztext,
- die Information, ob Groß- und Kleinschreibung beachtet werden sollen.

Als „ständige“ Ausgabe ist die Anzahl der gefundenen Dateien sinnvoll.

Als „einmalige“ Ausgaben benötigen wir evtl. Fehlermeldungen und die Statistikmeldung am Schluß einer Ersetzung.

Für die ersten drei Eingaben kommt nur ein Textfeld in Frage, da hier ein beliebiger Text eingegeben werden soll. Für die Groß-/Klein-Eingabe könnte man zwar auch ein Textfeld verwenden, aber ein Kontrollfeld ist hierfür prädestiniert, da es nur die Zustände „Ein“ und „Aus“ annehmen kann.

Die „einmaligen“ Ausgaben können wir, wenn wir nicht allzuviele Sonderwünsche bezüglich des Designs haben, über MSGBOX abwickeln. Der MSGBOX-Befehl zeigt eine Meldung an und wartet, bis der Benutzer darauf mit einem Tastendruck reagiert (vgl. Referenzteil).

Für die „ständige“ Ausgabe müssen wir ein Steuerelement auf der Form vorsehen. Ein Bezeichnungsfeld ist hier ideal: Es stellt einen Text dar, der nicht vom Benutzer angewählt oder verändert, vom Programm aus aber jederzeit neu beschrieben werden kann. Zwar kann man auf eine Form auch mit der PRINT-Methode Texte ausgeben, aber wenn man Bezeichnungsfelder verwendet und ihre Inhalte im Programm verändert, hat man jederzeit die Möglichkeit, die Form im Form-Designer völlig umzustellen, ohne irgendwelche Änderungen am Programm vorzunehmen.

Am besten sehen wir gleich zwei Bezeichnungsfelder vor: Eines für die Zahl der gefundenen Dateien und eines mit dem Text „Datei(en) gefunden“. Das letztere bleibt dann immer konstant.

Ferner benötigen wir noch für jedes der drei Textfelder ein Bezeichnungsfeld, das quasi als „Überschrift“ für das Textfeld fungiert, damit der Benutzer auch sieht, wofür das Textfeld, in das er Daten eingibt, eigentlich da ist.

Unser aktueller Stand im Form-Designer sieht etwa so aus (alle Namen sind zunächst so belassen, wie der Form-Designer sie vorgibt, und die Positionen und Feldlängen sind willkürlich gewählt):

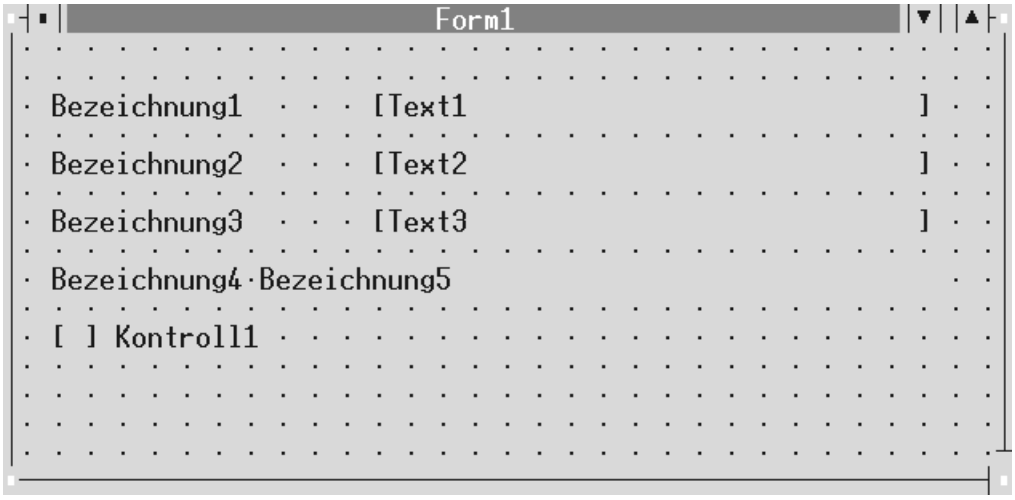


Abbildung 7–2: Erstes Entwurfsstadium im Form-Designer

Der nächste Schritt ist das Zuordnen der Eigenschaften *Caption*, *CtlName* und *Text* (wo vorhanden). Im *Caption*-Text kann ein &-Zeichen vorkommen, um eine Zugriffstaste zu definieren. Die *Text*-Eigenschaften der Textfelder werden auf Leerstrings gesetzt.

Die Größe der Form kann schon einmal angepaßt werden, und da unsere Form nicht verkleinert oder vergrößert werden soll, können die Form-Eigenschaften *ControlBox*, *MinButton* und *MaxButton* auf FALSE gesetzt werden. *BorderStyle* können wir aus dem gleichen Grund auf 1 setzen.

Der Zwischenstand ist jetzt (die *CtlName*-Eigenschaften habe ich hinzugefügt; sie sind für den Programmcode später wichtig):

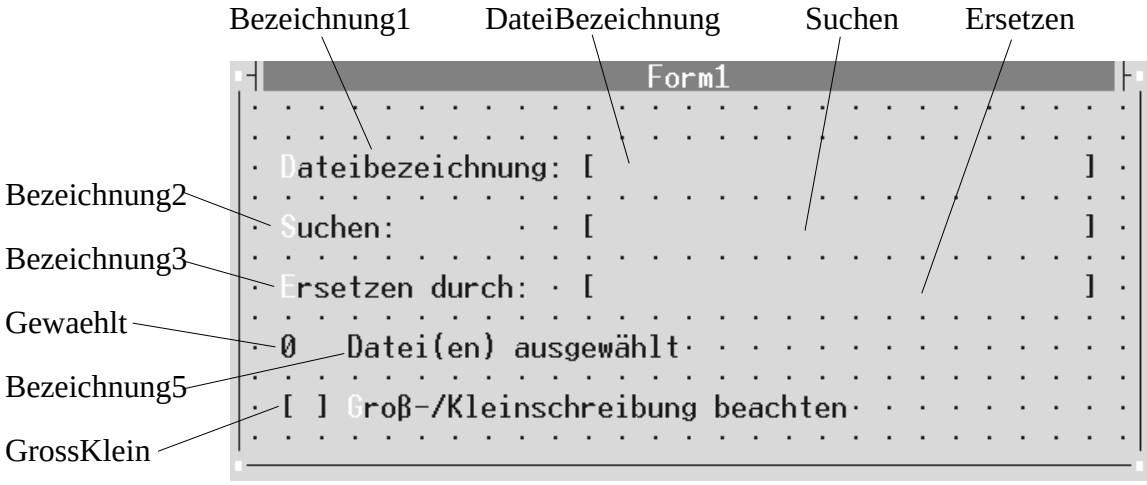


Abbildung 7–3: Zweites Entwurfsstadium im Form-Designer

Sie sehen, daß ich die Eigenschaft *CtlName* nur für einige Steuerelemente auf einen neuen Wert gesetzt habe. Bei den Bezeichnungsfeldern, die ich später im Listing nicht benötige, spielt die *CtlName*-Eigenschaft überhaupt keine Rolle, deshalb kann ruhig der voreingestellte Wert beibehalten werden.

Damit wäre die Form schon fast fertig. Wenn Sie bereits einen Überblick über die ereignisgesteuerte Programmierung haben, dann wundern Sie sich bestimmt schon: „Wo bleibt die Schaltfläche?“ – aber langsam:

In einem ereignisgesteuerten Programm müssen Sie als Programmierer den Anspruch aufgeben, alles selbst zu steuern. Wenn die Form fertig ist, werden Sie nur noch den Befehl geben, die Form zu laden und anzuzeigen (falls das Formmodul das Startmodul ist – und das wird es sein – geschieht das automatisch). Dann hat die Form die Kontrolle. Sie läßt den Benutzer in den Feldern, die wir definiert haben, herumspringen und verschiedene Eingaben machen. Uns fehlt noch eine Möglichkeit für den Benutzer, den Ersetzvorgang wirklich zu starten, nachdem er alles eingegeben hat. Da er ja beliebig zwischen den Feldern umherschalten kann, wäre es äußerst unklug, wie im alten Programm zu verfahren und genau dann mit dem Ersetzen zu beginnen, wenn das Feld „Groß-/Kleinschreibung“ verlassen wird.

Der langen Rede kurzer Sinn: wir führen noch eine Schaltfläche ein, mit der der Benutzer den Ersetzungsvorgang auslösen kann, und gleich noch eine zweite, mit der er das Programm verlassen kann.

Als *CtlName* wird ihnen „Start“ und „Beenden“ zugewiesen.

Damit ist unsere Form komplett. Jetzt kann noch die *Caption*-Eigenschaft der Form etwas freundlicher gestaltet werden („Form1“ liest sich etwas trocken), die Form kann einen eigenen Namen (Eigenschaft *FormName*, bisher auch „Form1“, der Form-Designer-Standard; wir wählen „HauptForm“) erhalten, und

zu guter letzt sollten Sie noch einmal die Werte der *TabIndex*-Eigenschaften überprüfen.

Die *TabIndex*-Eigenschaft ordnet alle Felder in einer Reihenfolge an, und in dieser Reihenfolge springt der Cursor dann durch die Felder, wenn der Benutzer die Tab-Taste benutzt. Je nachdem, wie Sie beim Entwerfen vorgegangen sind, geht es dabei ziemlich durcheinander – stellen Sie also eine vernünftige Reihenfolge ein. Wichtig ist dabei, daß die Bezeichnungsfelder, die wir als „Überschriften“ vor die Textfelder gesetzt haben, einen Wert erhalten, der um eins kleiner als der des zugehörigen Textfeldes ist, damit das Drücken der Abkürzungstaste der Überschrift den Cursor in das zugehörige Textfeld und nicht sonstwohin setzt.

Damit sieht unsere fertige Form wie folgt aus (ich habe noch ein zusätzliches Bezeichnungsfeld einfügen müssen, um den Text des Kontrollfeldes umzubrechen – aber das ist nur Kosmetik):

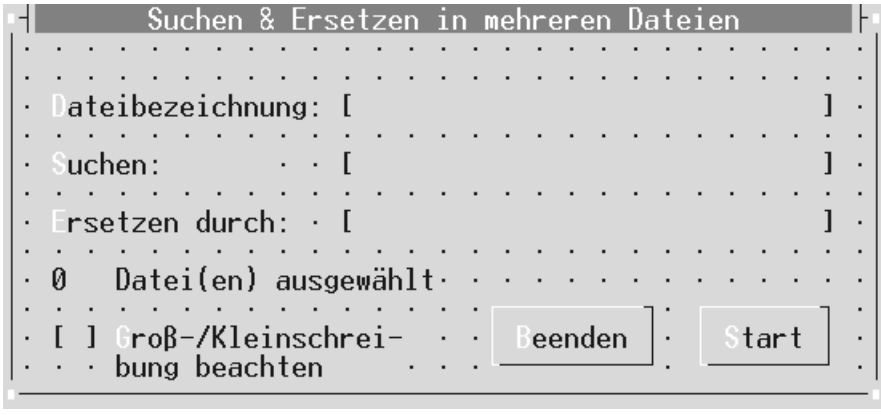


Abbildung 7–4: Die fertige Form im Form-Designer

Erste Ereignisprozeduren

Ohne es genau abgezählt zu haben, schätze ich, daß für unsere Form rund 150 Ereignisprozeduren möglich sind. Immerhin haben wir alles in allem 13 Objekte angelegt, und jedes kann im Schnitt über 10 Ereignisse erkennen.

98% dieser möglichen Ereignisprozeduren sind nutzlos. Zum Beispiel die Prozedur `Form_Click`, die immer aufgerufen wird, wenn der Benutzer auf einen Bereich der Form klickt, der nicht von einem Steuerelement belegt ist. Wenn Sie möchten, können Sie in diesem Falle natürlich als Schmankerl eine kleine Melodie spielen. Notwendig ist das jedoch nicht.

Welche Ereignisprozeduren aber sind notwendig? Auf welche Ereignisse muß das Programm anders reagieren, als es üblich ist?

Diese Frage läßt sich andersherum leichter beantworten: Was kann das Programm? Es kann beendet werden. Es kann einen Ersetzungsvorgang starten.

Und es kann zu einer gegebenen Dateibezeichnung die Anzahl der gefundenen Dateien ermitteln und ausgeben.

Das sind die drei wesentlichen Funktionen, die wir unterbringen müssen. Als nächstes ist zu bestimmen, wann diese Funktionen ausgeführt werden sollen. Die beiden erstgenannten sind kein Problem: Sie erfolgen als Reaktion auf das Drücken einer Schaltfläche. Das ergibt schon die erste Ereignisprozedur:

```
SUB Beenden_Click()
   SYSTEM
END SUB
```

Zugegeben, keine herausragende intellektuelle Leistung, aber immerhin: Wenn der Benutzer auf „Beenden“ klickt, wird das Programm beendet. Die Ereignisprozedur für „Start“ spare ich mir noch einen Augenblick auf; handeln wir erst einmal das Problem der Ermittlung der Dateianzahl ab:

Diese Anzahl muß immer dann (neu) ermittelt werden, wenn der Benutzer eine (neue) Dateibezeichnung angibt. Zuerst denkt man hier sicherlich an die Ereignisprozedur `DateiBezeichnung_Change`, die immer dann aufgerufen wird, wenn sich der Inhalt des Textfeldes `DateiBezeichnung` ändert. Bei näherem Hinsehen entpuppt sie sich jedoch als nicht optimal: Stellen Sie sich vor, der Benutzer gibt in das Textfeld den String „H:\VBDOS*.BAS“ ein. Dann wird nach jedem eingegebenen Zeichen die Change-Ereignisprozedur aufgerufen (weil sich der Inhalt ja wieder geändert hat). Es würde unter Umständen ziemlich lange dauern, wenn man nach jedem Zeichen einen Versuch startet, das Verzeichnis einzulesen.

Die bessere Alternative ist hier die *LostFocus*-Ereignisprozedur. Sie wird aufgerufen, wenn der Fokus vom Feld DateiBezeichnung „abberufen“ wird. Das passiert auch dann, wenn der Benutzer gerade noch einen Namen eintippte und jetzt auf „Start“ klickt – `DateiBezeichnung_LostFocus` wird dann noch vor `Start_Click` aufgerufen. Indem wir das Verzeichnis nur bei Aufruf der *LostFocus*-Prozedur aufbauen, ermöglichen wir dem Benutzer, ungestört Text einzugeben, und reagieren darauf erst nach Abschluß der Eingabe. Solange der Fokus das Feld nicht verläßt, kann ohnehin keine andere Aktion gestartet werden. Die zweite Ereignisprozedur in unserem Programm lautet also:

```
SUB DateiBezeichnung_LostFocus ()
   SHARED DateiName() AS STRING ' Damit die anderen Prozeduren
                                ' etwas damit anfangen können
   DIM Zaehler AS INTEGER       ' Brauchen die anderen nicht, sie
                                ' können ja Gewaehlt.Caption lesen
   VerzeichnisLesen (DateiBezeichnung.Text), DateiName(), Zaehler
                    ' Muß in Klammern gesetzt werden, weil VBDOS sich
                    ' sonst mit "Parametertypen unverträglich" meldet
```

```
   Gewaehlt.Caption = FORMAT$(Zaehler)
END SUB
```

Die Prozedur liest das Verzeichnis in das String-Array *DateiName( )*, das mit SHARED deklariert wurde, damit andere Prozeduren es auch verwenden können, und schreibt die Anzahl der gefundenen Dateien in die *Caption*-Eigenschaft des Bezeichnungsfeldes *Gewaehlt*. Dadurch wird sie sofort auf der Form sichtbar.

Start!

Jetzt fehlt nur noch die Ereignisprozedur für das Klicken auf die „Start"-Schaltfläche. Hier gibt es zwei prinzipiell verschiedene Möglichkeiten.

Die erste ist, die Form „gebunden" anzuzeigen und in die Start-Schaltfläche einfach den Befehl `UNLOAD HauptForm` einzubauen. Dann würden wir das bisherige Programm ungefähr so lassen, wie es ist, und anstelle der Eingabeschleife den Befehl `SHOW HauptForm, 1` einsetzen. VBDOS zeigt dann die Hauptform gebunden an, das heißt, daß hinter dem SHOW-Befehl erst nach Verstecken oder Entfernen der Form fortgefahren wird. Da ein Klick auf den „Start"-Knopf die Form aus dem Speicher löscht, würde also in diesem Moment die Programmausführung hinter SHOW weitergehen, der Ersetzungsvorgang würde starten, und wenn er beendet ist, könnten wir die Form wieder anzeigen lassen. Auf diese Weise kann man Formen recht einfach in bestehende Programme einbinden, ohne allzuviel ändern zu müssen, weil das eigentliche Programm bestehen bleibt.

Dieses Vorgehen hat jedoch den Nachteil, daß während des eigentlichen Ersetzungsvorgangs der Bildschirm gelöscht wird, weil die Form ausgeblendet werden muß, um die Programmfortsetzung zu ermöglichen. Im allgemeinen ist die Bedienung eines Programms umso angenehmer, je weniger gebundene Formen es enthält, weil der Benutzer bei ungebundenen Formen die Freiheit hat, zwischen ihnen umherzuschalten.

Für diese erste Möglichkeit würden wir ein Codemodul (mit dem Programmrumpf wie bisher) und ein Formmodul für die Form benötigen. Ein Beispiel für ein Programm, das auf diese Weise mit einer gebundenen Form arbeitet, finden Sie im Abschnitt 7.

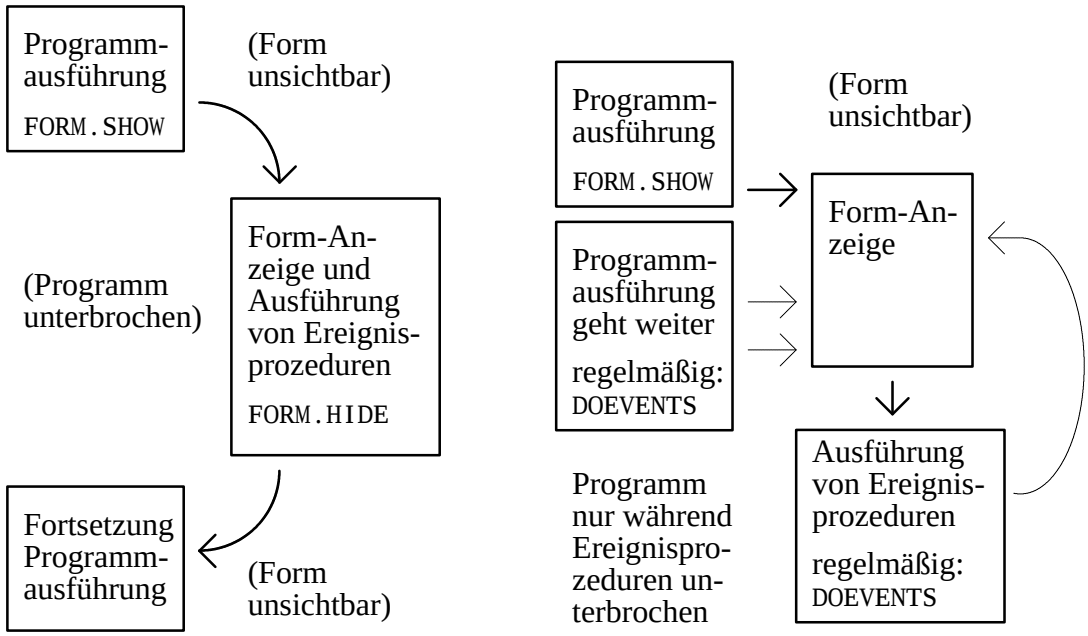


Abbildung 7–5: Gebundene (links) und ungebundene (rechts) Anzeige einer Form

Die zweite Möglichkeit ist die ungebundene Anzeige der Form; dann können wir allerdings nicht einfach wie oben vorgehen, weil dann durch den SHOW-Befehl die Form nur angezeigt wird, die Programmausführung aber sofort weitergeht. Es würde ein Ersetzungsvorgang beginnen, ohne daß der Benutzer Daten eingetragen hat.

Vielmehr muß der Code zur Ausführung der Ersetzungsaktion bei der ungebundenen Anzeige in die Ereignisprozedur `Start_Click` eingebaut werden. Außerdem müßte, falls die Startdatei ein Codemodul ist, dafür gesorgt werden, daß die Form nicht nur angezeigt wird (mit SHOW), sondern daß auch Formereignisse verarbeitet werden. Bei der gebundenen Anzeige geschieht das automatisch; bei der ungebundenen Anzeige muß dazu die DOEVENTS-Funktion aufgerufen werden.

Wir wählen diese zweite Möglichkeit, machen es uns aber noch einfacher und verzichten völlig auf das Codemodul. Wir verwenden nur ein einziges Formmodul, das dann auch Startdatei ist. Dadurch wird die Form beim Programmstart automatisch angezeigt, und Formereignisse werden verarbeitet.

Damit ist das Programm weitgehend fertig. In die Start-Prozedur habe ich noch zwei Abfragen eingebaut, die verhindern, daß der Benutzer einen Ersetzungsvorgang startet, ohne vernünftige Daten eingegeben zu haben.

```
REM $INCLUDE: 'CONSTANT.BI'
CONST DateiZuGross = 1, Gefunden = 2, NichtGefunden = 3
DIM DateiName(1 TO 999) AS STRING ' Auf Modulebene erlaubt: statisches Datenfeld!

SUB Beenden_Click ()
   SYSTEM
END SUB
```

```
SUB DateiBezeichnung_LostFocus ()

   SHARED DateiName() AS STRING ' Damit die anderen Prozeduren zugreifen können
   DIM Zaehler AS INTEGER       ' Brauchen die anderen nicht, sie können ja
                                ' Gewaehlt.Caption lesen
   VerzeichnisLesen (DateiBezeichnung.Text), DateiName(), Zaehler
                    ' Muß in Klammern gesetzt werden, weil VBDOS sich
                    ' sonst mit "Parametertypen unverträglich" meldet
   Gewaehlt.Caption = FORMAT$(Zaehler)

END SUB
```

```
FUNCTION ErsetzeInDatei (Datei AS STRING, Such AS STRING, Ersetz AS STRING,
                         GKlein AS INTEGER) AS INTEGER
' bleibt unverändert wie im letzten Listing
```

```
FUNCTION ExtrahierePfad (Bezeichnung AS STRING) AS STRING
' bleibt unverändert wie im letzten Listing
```

```
SUB Start_Click ()

   SHARED DateiName() AS STRING
   DIM Zaehler AS INTEGER, DateiName AS STRING, Ersetzungen AS INTEGER
   DIM ZuGross AS INTEGER, Message AS STRING, PfadAngabe AS STRING

   IF VAL(Gewaehlt.Caption) = 0 THEN
      MSGBOX "Sie müssen mindestens eine Datei auswählen.", 0, "Fehler"
      DateiBezeichnung.SETFOCUS
   ELSEIF LEN(Suchen.Text) = 0 THEN
      MSGBOX "Sie müssen einen Suchtext angeben.", 0, "Fehler"
      Suchen.SETFOCUS
   ELSE
      ' Wir müssen keine Knöpfe sperren, da wir im folgenden nicht die DOEVENTS()-
      ' Funktion aufrufen. Anderenfalls wäre es notwendig, zu verhindern, daß der
      ' Benutzer während des Prozedurablaufs die Feldinhalte verändert oder einen
      ' neuen Suchvorgang auslöst!
      PfadAngabe = ExtrahierePfad((DateiBezeichnung.Text)) ' Doppelklammern, weil
                                                           ' VBDOS sonst motzt
      FOR Zaehler = 1 TO VAL(Gewaehlt.Caption)
         DateiName = PfadAngabe + DateiName(Zaehler)
         SELECT CASE ErsetzeInDatei(DateiName, (Suchen.Text), (Ersetzen.Text),
                                    (GrossKlein.Value))
```

```
         CASE Gefunden
            Ersetzungen = Ersetzungen + 1
         CASE DateiZuGross
            ZuGross = ZuGross + 1
         END SELECT
      NEXT

      Message = Gewaehlt.Caption + " Dateien geprüft" + CHR$(13) + CHR$(10)
      Message = Message + FORMAT$(Ersetzungen) + " Ersetzungen durchgeführt" +
                          CHR$(13)+ CHR$(10)
      Message = Message + CHR$(10) + FORMAT$(ZuGross) + " Dateien größer als 20KB"
      MSGBOX Message, 0, "Aktion beendet"
   END IF

END SUB
```

```
SUB VerzeichnisLesen (PathSpec AS STRING, FileName() AS STRING, Number AS INTEGER)
' bleibt unverändert wie im letzten Listing
```

Listing 7–2: (Auszug aus) ERS_FORM.FRM

Dieses Listing ist, da wir ohne gebundene Formen arbeiten, „pseudo-multitasking-fähig“. Angenommen, wir programmieren noch ein weiteres Utility, das mit einer Form auskommt, z. B. einen Taschenrechner, sorgen dafür, daß dessen Form beim Start des Programms ebenfalls angezeigt wird und bauen in die `Start_Click`-Prozedur noch einige DOEVENTS-Aufrufe ein, dann kann der Benutzer, während ein Ersetzungsvorgang läuft, beliebige Berechnungen mit dem Taschenrechner ausführen. Mehr dazu weiter unten in diesem Kapitel (Seite 133).

Noch eine Schönheitsoperation...

Einen kleinen Schönheitsfehler hat das (lauffähige) Programm allerdings noch: Es zeigt keine Meldung an, der man entnehmen könnte, daß der Ersetzungsvorgang läuft. Vielmehr drückt man auf den Start-Knopf und wundert sich zunächst einmal, daß scheinbar nichts passiert, bis dann plötzlich die Meldung „Vorgang beendet“ erscheint.

Die eleganteste Möglichkeit, dies zu beheben, ist die Einführung einer zweiten Form. So könnte man zum Beispiel einfach eine Form mit einem großen Bezeichnungsfeld erstellen, die zu Beginn des Vorgangs mit `WarteForm.SHOW` angezeigt und am Ende mit `WarteForm.HIDE` wieder versteckt wird (unter der Voraussetzung, daß Sie *FormName* auf „WarteForm“ gesetzt haben):

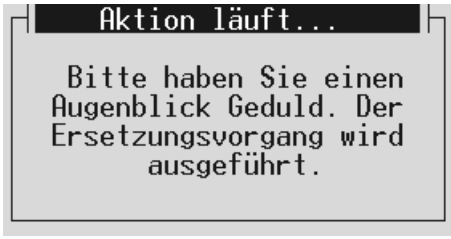


Abbildung 7–6: Eine simple Warte-Anzeige

Man muß selbst für einen solch bescheidenen Zweck eine neue Form nehmen, weil der MSGBOX-Befehl die Ausführung unterbricht, bis der Benutzer eine Taste drückt. Das aber ist hier nicht erwünscht.

Nun kann man noch weiter gehen und den Benutzer – wenn man ohnehin schon eine Warte-Form angelegt hat – wissen lassen, wie weit die Aktion fortgeschritten ist. Eine Prozentanzeige bietet sich an; man bringt einfach auf der Form ein zusätzliches Bezeichnungsfeld (*CtlName* = ProzentZahl) unter und sorgt vom Programm aus für die Aktualisierung:

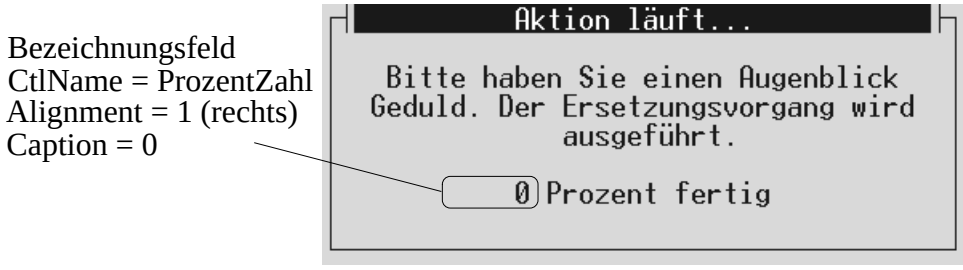


Abbildung 7–7: Warte-Anzeige mit Prozentangabe

Die FOR-Schleife in der `Start_Click`-Prozedur würde dann wie folgt geändert:

```
WarteForm.SHOW
FOR Zaehler = 1 TO VAL(Gewaehlt.Caption)
   WarteForm.ProzentZahl.Caption = FORMAT$((Zaehler * 100) \
                                   VAL(Gewaehlt.Caption))
   ' ... weiter wie bisher
NEXT
WarteForm.HIDE
```

Wenn Sie in einem Programm öfters eine Warte-Anzeige benötigen, läßt sich selbstverständlich die gleiche Form „recyclen“; dann müßte nur der *Caption*-Text jeweils der Situation angepaßt werden.

Darüber hinaus ist es kein Problem – und erfordert auch nicht die kleinste Änderung am eigentlichen Programm – der Form noch einen Prozentbalken hinzuzufügen, der die Prozentzahl visualisiert. Die Routine dazu stelle ich im Kapitel 21 vor.

...und ein Trick für Programmierfaule

Wenn wir ohnehin mit der Ereignissteuerung arbeiten, können wir uns die Verzeichnis-Routinen sparen und einfach eine Dateiliste einsetzen. Sie muß ja nicht unbedingt angezeigt werden, sie kann ja auch im Hintergrund für uns werkeln. Das Listing wird dadurch wesentlich einfacher.

Zeichnen Sie zunächst im Form-Designer eine Dateiliste und versehen Sie diese mit den Eigenschaft *Caption* = „DateiListe“ und *Visible* = FALSE. (Wohin Sie sie zeichnen, spielt keine Rolle, da sie nicht angezeigt werden wird.)

Nun können sie die `DateiBezeichnung_LostFocus`-Prozedur ändern:

```
SUB DateiBezeichnung_LostFocus()
   DateiListe.FileName =  DateiBezeichnung
   Gewaehlt.Caption = FORMAT$(DateilListe.ListCount)
END SUB
```

Die Variable *PfadAngabe*, das Datenfeld *DateiName()*, die Funktion `ExtrahierePfad` und die Prozedur `VerzeichnisErstellen` fallen weg; die FOR-Schleife kann wie folgt geändert werden:

```
FOR Zaehler = 1 TO DateiListe.ListCount
   DateiName = DateiListe.Path + DateiListe.List(Zaehler   1)
               '(weil DateiListe bei 0 zu zählen beginnt)
   ' usw. wie bisher
NEXT
```

Wie Sie sehen, spart man eine ganze Menge Code durch den Einsatz des versteckten Dateilistenfeldes. Noch einige Tips zu versteckten Steuerelementen finden Sie weiter unten in diesem Kapitel (Seite 132).

Krönender Abschluß

Einen erneuten Abdruck des resultierenden Listings spare ich mir hier; aber genießen Sie die Bildschirmausgabe des ereignisgesteuerten Programms, und vergleichen Sie sie mit dem kläglichen Output des ursprünglichen Programms auf Seite 87:

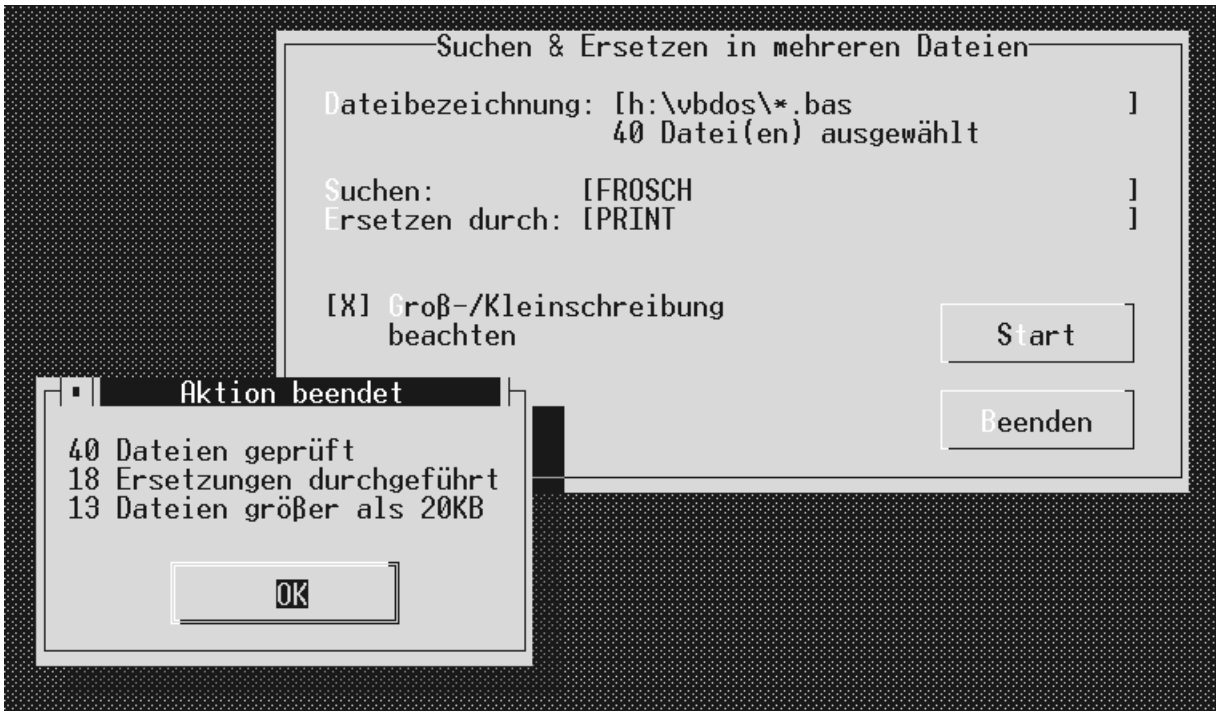


Abbildung 7–8: Die Ausgabe des fertigen ereignisgesteuerten Programms

Gut, solche Resultate ließen sich auch schon früher erzielen – aber dieses hier benötigt weniger Programmzeilen als die alte Version, und wenn Sie etwas Routine im ereignisgesteuerten Programmieren haben, gehen Ihnen diese Programme zweifelsohne schneller von der Hand als das, was Sie früher gemacht haben.

Checkliste

Zum Abschluß dieses Beispiels wiederhole ich in Form einer kurzen Checkliste, was beim Entwickeln eines (einfachen) ereignisgesteuerten Programms zu beachten ist:

Schritt	*Beschreibung*
1. Welche Steuerelemente?	Überlegen Sie, welche Steuerelemente Sie benötigen. Für die meisten Zwecke läßt sich ein optimales Steuerelement finden. Überlegen Sie insbesondere, ob Sie statt einfachen Textfeldern Kombinations- oder Listenfelder einsetzen können, um dem Benutzer Standardauswahlen anzubieten. Ordnen Sie die Steuerelemente auf einer Form an.

Schritt	*Beschreibung*
2. Bezeichnungen und Anfangswerte	Stellen Sie die *CtlName*-Eigenschaft der Steuerelemente, die in Ihrem Programm vorkommen werden, auf einen sinnvollen, „sprechenden“ Wert (Steuerelemente, die Sie nur der Optik wegen einsetzen, brauchen nicht berücksichtigt zu werden).
	Stellen Sie die *Caption*- bzw. *Text*-Eigenschaften auf die Werte, die beim Programmstart in den jeweiligen Feldern enthalten sein sollen. Stellen Sie (durch „&“ in der *Caption*-Eigenschaft) Zugriffstasten ein. Bestimmen Sie evtl. eine *Default*- und eine *Cancel*-Schaltfläche.
3. Formeinstellungen	Stellen Sie die Größe der Form ein, passen Sie ggf. Position und Größe der Steuerelemente an; wenn Sie eine Form mit fester Größe wollen, setzen Sie *BorderStyle* auf 1, *ControlBox*, *MinButton* und *MaxButton* auf FALSE.
4. Letzte optische Korrekturen und Prüfen der *TabIndex*-Reihenfolge	Prüfen Sie, ob die Objekte durch die TabIndex-Eigenschaft so numeriert werden, wie sie auch mit der Tab-Taste angesprungen werden sollen, oder ob hier Durcheinander herrscht.
5. Programmieren der forminternen Ereignisprozeduren	Erstellen Sie zunächst die Ereignisprozeduren, die benötigt werden, um das Verhalten der Steuerelemente festzulegen (z. B. unser „Gewaehlt“-Feld, das sich immer anpassen mußte, wenn „DateiBezeichnung“ geändert wurde).
6. Entscheidung, ob Form gebunden oder ungebunden verwendet wird	Von dieser Entscheidung hängt ab, ob die Form mit ihren Ereignisprozeduren selbst Aktionen auslösen soll (ungebundene Form) oder ob sie bloß als „dumme“ Maske angezeigt wird und die Routine, von der aus die Form aufgerufen wurde, nach dem Entfernen der Form mit den Benutzereingaben weiterhantiert.
7. Programmieren der externen Ereignisprozeduren	Erstellen Sie zuletzt die Ereignisprozeduren, die nicht bloß an der Form herumdoktern, sondern wirklich eine Aktion auslösen (so wie `Start_Click` in unserem Beispiel).

Die nächsten drei Abschnitte behandeln die Eigenarten der Steuerelemente (7.2), Formen (7.3) und Spezialobjekte (7.4) in VBDOS; danach möchte ich auf einige Aspekte der Ereignissteuerung noch näher eingehen.

7.2 Steuerelemente

Die größte Gruppe unter den Objekten bilden die Steuerelemente, die Sie mit dem Form-Designer auf Formen anordnen können. Sie werden in der „Werkzeugkiste“ des Form-Designers in dieser Reihenfolge angeboten, mit Ausnahme des Menüeintrags, den Sie nur über das Menüentwurfsfenster des Form-Designers erstellen können (vgl. Kapitel 5).

Die Beschreibung umfaßt Aussehen, Verwendung und typische Eigenschaften der Steuerelemente. Eine detaillierte Übersicht über die Eigenschaften finden Sie im Referenzteil.

Ich gebe auch die englischen Bezeichnungen an, da diese benötigt werden, wenn man mit der IF TYPEOF-Anweisung arbeiten will (siehe Objekt-Referenzteil).

(Befehls-)schaltfläche	**Command Button**
Eigenschaften	BackColor, Cancel, Caption, CtlName, Default, DragMode, Enabled, Height, Index, Left, MousePointer, Parent, TabIndex, TabStop, Tag, Top, Value, Visible, Width
Ereignisse	Click, DragDrop, DragOver, GotFocus, KeyDown, KeyPress, KeyUp, LostFocus
Methoden	DRAG, MOVE, REFRESH, SETFOCUS

Die Befehlsschaltfläche wird als Auslöser für einfache Aktionen verwendet („Start“, „Abbruch“ usw.). Das wichtigste Ereignis ist denn auch „Click“ und wird ausgelöst, wenn der Benutzer die Schaltfläche mit ENTER oder der linken Maustaste anwählt.

Eine Schaltfläche zeigt den Text ihrer *Caption*-Eigenschaft zentriert an; dieser Text kann aber nicht mehr als eine Zeile umfassen. Einige Beispiele für das Aussehen von Schaltflächen:

Abbildung 7–9: Diverse Schaltflächen

Bezeichnung-Steuerelement	**Label**
Eigenschaften	Alignment, AutoSize, BackColor, BorderStyle, Caption, CtlName, DragMode, Enabled, ForeColor, Height, Index, Left, MousePointer, Parent, TabIndex, Tag, Top, Visible, Width
Ereignisse	Change, Click, DblClick, DragDrop, DragOver, MouseDown, MouseMove, MouseUp
Methoden	DRAG, MOVE, REFRESH

Diese Art von Steuerelement wird gewöhnlich verwendet, um einen festen Text auf einer Form unterzubringen. Eine Bezeichnung kann beliebig viele Zeilen haben und zeigt den Text an, der in ihrer *Caption*-Eigenschaft steht. Dieser Text kann während der Laufzeit des Programms vom Programm selbst geändert werden, nicht jedoch vom Benutzer. Der Text wird, wenn es sich um eine mehrzeilige Bezeichnung handelt, automatisch an Leerzeichen und an CR/LF-Kombinationen (`CHR$(13)+CHR$(10)`) umgebrochen.

Eine Bezeichnung kann mit einem Rahmen versehen werden. Die *Alignment*-Eigenschaft bestimmt, ob der Text linksbündig, rechtsbündig oder zentriert angezeigt wird, und eine *AutoSize*-Eigenschaft ermöglicht es, die Größe des Elements automatisch dem Text anzupassen, der hineingeschrieben wird.

Bildfeld	**Picture Box**
Eigenschaften	AutoRedraw, BackColor, BorderStyle, CtlName, CurrentX, CurrentY, Enabled, ForeColor, Index, Parent, ScaleHeight, ScaleWidth, TabIndex, TabStop, Tag, Top, Visible, Width
Ereignisse	Click, DblClick, DragDrop, DragOver, GotFocus, KeyDown, KeyPress, KeyUp, LostFocus, MouseDown, MouseMove, MouseUp, Paint
Methoden	CLS, DRAG, MOVE, PRINT, REFRESH, SETFOCUS, TEXTHEIGHT, TEXTWIDTH

Ein Bildfeld kann (wie ein Rahmen) als „Container“ für weitere Steuerelemente verwendet werden. Diese Gruppierung ist sinnvoll, wenn man mit Optionsfeldern arbeitet (siehe „Optionsfeld“, „Rahmen“). Ferner ist es möglich, in einem Bildfeld mittels der PRINT-Methode beliebige Textausgaben vorzunehmen. Ein Bildfeld fungiert dann als „kleiner Bildschirm“ mitten auf einer Form.

Der Name „Bildfeld“ ist etwas irreführend: Außer ASCII-Zeichen kann in einem Bildfeld nichts ausgegeben werden; es ist also nicht möglich, ein Bild (eine Grafik) anzuzeigen.

Text, der mit der PRINT-Methode in ein Bildfeld geschrieben wird und die Breite überschreitet, wird abgeschnitten (siehe PRINT-Methode im Objekt-Referenzteil).

Anstatt des LOCATE-Befehls, den man bei einer normalen Bildschirmausgabe verwendet, sind für die Textausgabe in einem Bildfeld die Eigenschaften *CurrentX* und *CurrentY* zu verwenden.

Dateiliste	**File List Box**
Eigenschaften	Archive, BackColor, CtlName, DragMode, Enabled, FileName, ForeColor, Height, Index, ListCount, ListIndex, MousePointer, Normal, Parent, Path, Pattern, ReadOnly, System, TabIndex, TabStop, Tag, Top, Visible, Width
Ereignisse	Click, DblClick, DragDrop, DragOver, GotFocus, KeyDown, KeyPress, KeyUp, LostFocus, MouseDown, MouseMove, MouseUp, PathChange, PatternChange
Methoden	DRAG, MOVE, REFRESH, SETFOCUS

Ein Dateilistenfeld zeigt eine einspaltige Liste mit Dateinamen an (die Optik entspricht der eines Listenfeldes). Dabei kann mit entsprechenden Eigenschaften gewählt werden, welche Dateien in der Liste stehen (Verzeichnismaske, Attribute, Verzeichnis und Laufwerk). Die Liste wird alphabetisch sortiert.

Mit den Eigenschaften *Archive, Hidden, Normal, ReadOnly* und *System* (alle TRUE oder FALSE) wird bestimmt, welche Attribute eine Datei haben darf, um angezeigt zu werden. Attribute sind Informationen, die DOS zu den üblichen Verzeichniseinträgen zusätzlich speichert; sie können durch das DOS-Dienstprogramm ATTRIB (oder durch Interruptaufrufe) geändert werden. Mit diesen Attribut-Eigenschaften ist es nicht ganz so einfach; die folgende Tabelle gibt eine Übersicht über ihre Wirkung:

Wenn Sie nur diese Eigenschaft auf TRUE setzen...	*...werden solche Dateien angezeigt*
Archive	alle, bei denen das Archiv-Attribut, aber nicht das Hidden- oder System-Attribut gesetzt ist
Hidden	alle, bei denen das Hidden-Attribut, aber nicht das System-Attribut gesetzt ist
Normal	alle, bei denen weder Hidden- noch System-Attribut gesetzt sind (ReadOnly- und Archiv-Attribut spielen keine Rolle)
ReadOnly	alle, bei denen das ReadOnly-Attribut, aber nicht das Hidden- oder System-Attribut gesetzt ist
System	alle, bei denen das System-Attribut, aber nicht das Hidden-Attribut gesetzt ist

Wenn Sie mehrere Eigenschaften auf TRUE setzen, wird die Vereinigungsmenge aller Dateien angezeigt, die bei den einzelnen Eigenschaften angezeigt würden.

Die Eigenschaft *Path* gibt an, auf welchem Laufwerk und Verzeichnis nach Dateien gesucht wird. Die Eigenschaft *Pattern* gibt das Muster an, das die Dateien erfüllen müssen (z. B. „*.EXE“).

Bei jeder Änderung der vorgenannten die Liste bestimmenden Eigenschaften wird sofort das Verzeichnis neu eingelesen. Dabei treten die üblichen Fehler auf, wenn z. B. ein Diskettenlaufwerk nicht geschlossen ist. Man muß beim Programmieren also damit rechnen, daß ein Befehl wie

```
Dateiliste.Path = NewPath$
```

durchaus einen Fehler wie „Laufwerk nicht bereit“ o.ä. verursachen kann.

Ein erneutes Einlesen des Verzeichnisses (z. B. wenn eine neue Datei geschrieben wurde) kann außerdem mit der REFRESH-Methode erreicht werden.

Die Eigenschaft *FileName* enthält den aktuellen Dateinamen, sofern einer gewählt ist. Ansonsten (z. B. bevor der Benutzer überhaupt etwas mit der Dateiliste angefangen hat) ist *FileName* leer.

Bildlaufleisten (horizontal, vertikal)	**HScrollBar, VScrollBar**
Eigenschaften	Attached, CtlName, DragMode, Enabled, Height, Index, LargeChange, Left, Max, Min, MousePointer, Parent, SmallChange, TabIndex, TabStop, Tag, Top, Value, Visible, Width
Ereignisse	Change, DragDrop, DragOver, GotFocus, KeyDown, KeyPress, KeyUp, LostFocus
Methoden	DRAG, MOVE, REFRESH, SETFOCUS

Bildlaufleisten erfüllen verschiedene Zwecke. Grundsätzlich können sie in drei Klassen eingeteilt werden:

Automatische Bildlaufleisten, die für Textfelder generiert werden, wenn ihre *ScrollBars*-Eigenschaft auf einen Wert ≠ 0 gesetzt wird. Diese sehen zwar genauso aus wie die Bildlaufleisten-Steuerelemente, sind aber selbst keine Objekte; auf sie können also die hier genannten Methoden und Eigenschaften nicht angewandt werden, und sie lösen auch keine Ereignisse aus. Das Textfeld reagiert automatisch auf Veränderungen, die der Benutzer mit der Maus an diesen Bildlaufleisten vornimmt. Eine automatische vertikale Bildlaufleiste wird außerdem für jedes Listenfeld generiert.

Mit Formen verbundene Bildlaufleisten. Diese Bildlaufleisten sind ganz gewöhnliche Steuerelemente, die jedoch genau auf dem Rand einer Form sitzen

und außerdem in ihrer *Attached*-Eigenschaft den Wert TRUE haben. Sie können unabhängig behandelt werden wie jede andere Bildlaufleiste, ihre Größe ändert sich jedoch automatisch mit, wenn sich die Größe der Form ändert. Solche Bildlaufleisten kommen meistens dann zur Anwendung, wenn man dem Benutzer die Möglichkeit geben will, durch größere Textmengen zu blättern, aber ein Textfeld nicht flexibel genug ist. Eine automatische Reaktion auf Änderungen der Bildlaufleisten findet nicht statt, sie muß auf gewöhnlichem Wege programmiert werden.

Freie Bildlaufleisten schließlich werden wie jedes andere Steuerelement einfach irgendwo auf einer Form plaziert und unterliegen keinen besonderen Bestimmungen. Sie können zum Beispiel zur Eingabe von Werten benutzt werden.

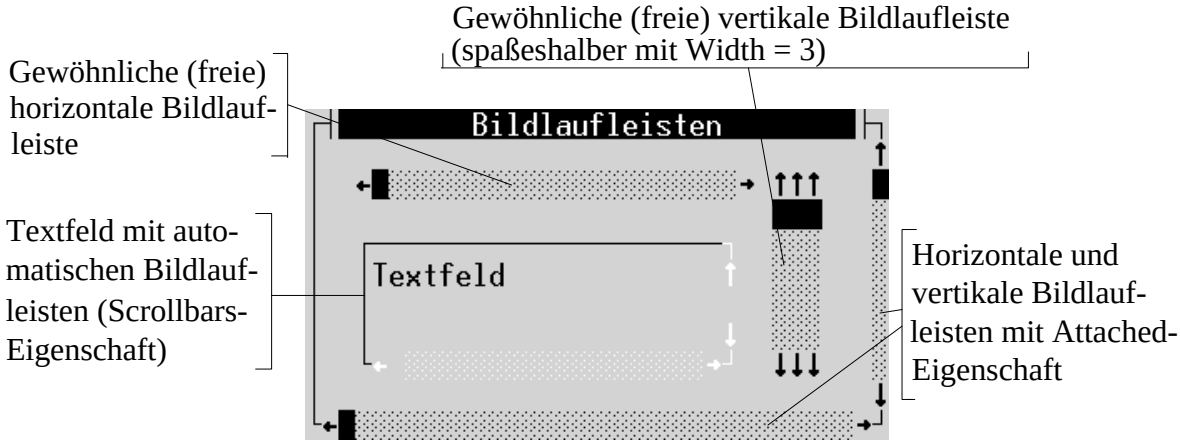


Abbildung 7–10: Verschiedene Bildlaufleisten

Mit den Eigenschaften *Min* und *Max* kann festgelegt werden, welches der kleinste (linker bzw. oberer Anschlag) und der größte (rechter bzw. unterer Anschlag) erreichbare Wert der Bildlaufleiste ist. Soll der größte Wert beim linken/oberen und der kleinste beim rechten/unteren Anschlag angenommen werden, kann man für *Min* einen kleineren Wert als für *Max* wählen.

Die Eigenschaft *Value* gibt an, welcher Wert gerade eingestellt ist bzw. läßt sich dazu verwenden, einen Wert darzustellen. Das Steuerelement errechnet aus seiner Breite bzw. Höhe, den Eigenschaften *Min* und *Max* und aus *Value* selbsttätig, wo die Markierung in der Bildlaufleiste angezeigt werden muß.

Eine Bildlaufleiste kann nur für ganzzahlige Werte verwendet werden.

Die Eigenschaft *SmallChange* gibt an, um wieviel sich *Value* ändern soll, wenn auf einen der Pfeile an den Enden der Bildlaufleiste geklickt wird bzw. wenn die Pfeil-Auf/Links- oder Pfeil-Ab/Rechts-Taste gedrückt wird. Die Eigenschaft *LargeChange* bestimmt die *Value*-Änderung beim Klicken auf die Bildlaufleiste selbst (neben der Markierung) oder beim Betätigen der Bild-Auf- oder Bild-Ab-Taste.

Kombinationsfeld	**Combo Box**
Eigenschaften	BackColor, CtlName, DragMode, Enabled, Index, List, ListCount, ListIndex, MousePointer, Parent, SelLength, SelStart, SelText, Sorted, Style, TabIndex, TabStop, Tag, Text, Top, Visible
Ereignisse	Change, Click, DblClick, DragDrop, DragOver, DropDown, GotFocus, KeyDown, KeyPress, KeyUp, LostFocus
Methoden	ADDITEM, DRAG, MOVE, REFRESH, REMOVEITEM, SETFOCUS

Ein Kombinationsfeld ist eine Verbindung aus Text- und Listenfeld. Es gibt drei Varianten des Kombifeldes; in welcher davon ein Kombifeld auftritt, wird durch seine *Style*-Eigenschaft festgelegt. Beim **Drop-Down-Kombifeld** wird nur ein Textfeld angezeigt, aus dem sich jedoch durch Drücken auf Alt+Pfeil-Ab oder durch Klicken auf einen abgebildeten Pfeil eine Liste öffnen läßt. Das **einfache Kombifeld** zeigt diese Liste ständig an. Beide Kombifelder haben die Eigenschaft, daß man auch von Hand einen Eintrag im Feld vornehmen kann, der nicht in der Liste enthalten ist. Das **Drop-Down-Listenfeld** entspricht dem Drop-Down-Kombifeld, erlaubt aber diese manuelle Eingabe nicht. Beim Drop-Down-Listenfeld *muß* man einen Eintrag aus der Liste auswählen.

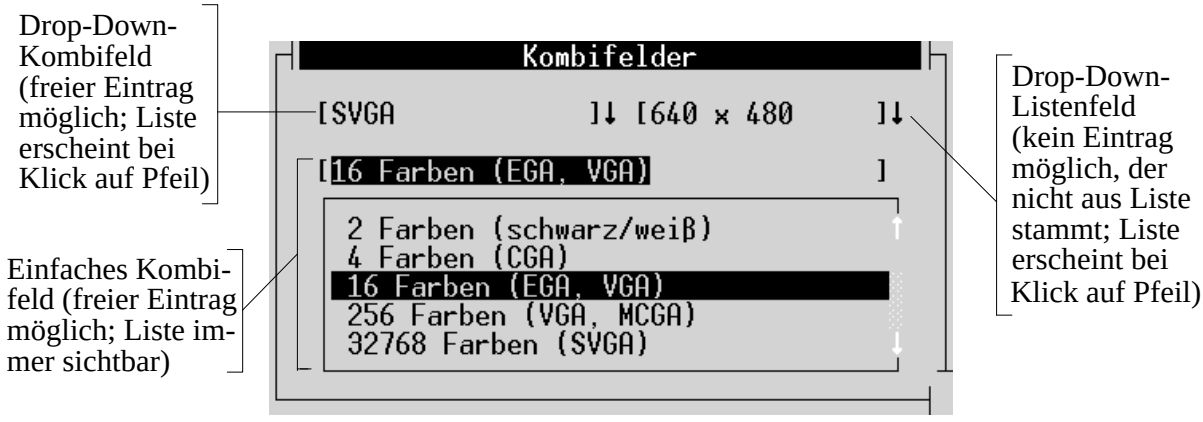


Abbildung 7–11: Die drei Varianten eines Kombifeldes

Wie bei einem Listenfeld müssen die Listeneinträge mit der ADDITEM-Methode zur Laufzeit hinzugefügt werden. Eine Sortierung ist möglich: Setzen Sie die *Sorted*-Eigenschaft auf TRUE.

Bei den zwei Varianten, die nicht ständig, sondern nur auf Anforderung eine Liste anzeigen, kann die Höhe dieser Liste nicht festgelegt werden; Visual BASIC wählt automatisch eine ihm adäquat erscheinende Höhe. Die Breite der Liste hingegen entspricht der Breite des Steuerelements.

Das *Click*-Ereignis tritt bei Kombifeldern nur ein, wenn in den Listenbereich geklickt wird.

Das *DropDown*-Ereignis ist eine Spezialität von Kombifeldern und tritt auf, unmittelbar bevor die Liste angezeigt wird (also wenn der Benutzer auf das Pfeil-Ab-Symbol klickt oder Alt+Pfeil-Ab betätigt). Es existiert nicht bei einfachen Kombifeldern, deren Liste immer angezeigt wird.

Kontrollfeld	**Check Box**
Eigenschaften	BackColor, Caption, CtlName, DragMode, Index, Left, MousePointer, Parent, TabIndex, TabStop, Tag, Top, Value, Visible, Width
Ereignisse	Click, DragDrop, DragOver, GotFocus, KeyDown, KeyPress, KeyUp, LostFocus
Methoden	DRAG, MOVE, REFRESH, SETFOCUS

Ein Kontrollfeld ist ein Steuerelement, das vom Benutzer entweder ein- oder ausgeschaltet werden kann. Der Name ist eine der vielen traurigen Fehlübersetzungen in Visual BASIC: „Check Box“ ist im Englischen ein Kästchen, das man ankreuzen kann (❑), und genau darum geht es hier. Ankreuzfeld wäre richtig gewesen, mit „Kontrolle“ hat das Ganze nichts zu tun.

Kontrollfelder sind immer einzeilig; dem *Caption*-Text gehen eine offene und eine geschlossene eckige Klammer voran, und wenn das Feld angewählt ist, erscheint dazwischen ein X.

Abbildung 7–12: Kontrollfelder

Die Eigenschaft *Value* eines Kontrollfeldes ist 1, wenn es gewählt ist, 0, wenn nicht, und 2, wenn es durchgestrichen ist. Der „Durchgestrichen“-Status ist vom Benutzer nicht einstellbar; nur Ihr Programm kann ihn durch eine Zuweisung an die *Value*-Eigenschaft hervorrufen.

Den Wert eines Kontrollfeldes ändert der Benutzer durch Mausklick oder mit der Leertaste, wenn das Kontrollfeld den Fokus hat.

Laufwerksliste	**Drive List Box**
Eigenschaften	BackColor, CtlName, DragMode, Drive, Index, Left, List, ListCount, ListIndex, MousePointer, Parent, TabIndex, TabStop, Tag, Top, Visible, Width
Ereignisse	Change, DragDrop, DragOver, GotFocus, KeyDown, KeyPress, KeyUp, LostFocus
Methoden	DRAG, MOVE, REFRESH, SETFOCUS

Das Laufwerkslistenfeld ist eine spezielle Form des Kombifeldes. Es hat nur eine Zeile, in der es das aktuelle Laufwerk anzeigt. Wie bei einem Kombifeld kann eine Liste aller Laufwerke, die im System installiert sind, aufgerufen werden. Wenn das Feld breit genug ist, werden hinter den Laufwerksbuchstaben auch die Datenträgerbezeichnungen (die z. B. mit dem LABEL-Befehl in DOS gesetzt werden) angezeigt.

Die Eigenschaft *Drive* enthält das gerade gewählte Laufwerk, gefolgt von einem Doppelpunkt. Gibt man beim Laden der Form nichts an, ist das aktuelle Laufwerk voreingestellt.

Wenn man *Drive* in einer Zuweisung verwendet, ist nur das erste Zeichen relevant. Weist man *Drive* ein ungültiges Laufwerk zu, verursacht das einen Fehler.

Mittels der Eigenschaften *List*, *ListCount* und *ListIndex* kann man wie bei einem gewöhnlichen Listenfeld auf die Laufwerksliste zugreifen.

Obwohl sich ein Laufwerkslistenfeld wie ein Kombifeld verhält, unterstützt es nicht das Ereignis *DropDown*.

In Kombination mit einem Verzeichnis- und einem Dateilistenfeld sollte man dafür sorgen, daß bei Eintritt eines *Change*-Ereignisses in der Laufwerksliste die *Path*-Eigenschaft der Verzeichnis- und evtl. der Dateiliste auf das neue Laufwerk gesetzt werden.

Listenfeld	**List Box**
Eigenschaften	BackColor, CtlName, DragMode, Enabled, Index, List, ListCount, ListIndex, MousePointer, Parent, Sorted, TabIndex, TabStop, Tag, Text, Top, Visible, Width
Ereignisse	Click, DblClick, DragDrop, DragOver, GotFocus, KeyDown, KeyPress, KeyUp, LostFocus, MouseDown, MouseMove, MouseUp
Methoden	ADDITEM, DRAG, MOVE, REFRESH, REMOVEITEM, SETFOCUS

Ein Listenfeld besteht aus einem Rahmen, der am rechten Rand mit einer horizontalen Bildlaufleiste versehen ist und in dem Listeneinträge angezeigt wer-

den. Der Benutzer kann in der angezeigten Liste blättern und einen Eintrag auswählen.

Intern verwaltet ein Listenfeld für sich ein Array mit Strings variabler Länge, in dem es alle Listeneinträge speichert. Eine automatische Sortierung ist möglich. Über spezielle Eigenschaften kann man auf dieses im Listenfeld gespeicherte Array zugreifen.

Die Eigenschaft *List* ermöglicht den Zugriff auf Array-Elemente des Listenfeldes (z. B. `a$ = Listenfeld.List(5)`). Das erste Element hat die Nummer 0, unabhängig von der OPTION BASE-Einstellung.

Die Eigenschaft *ListIndex* enthält die Nummer des Eintrages, der gerade hervorgehoben ist (–1, wenn keiner hervorgehoben ist); *ListCount* gibt an, wieviele Einträge in der Liste stehen.

Mit der ADDITEM- und REMOVEITEM-Methode werden Elemente der Liste hinzugefügt bzw. aus dieser entfernt.

Wenn man die Eigenschaft *Sorted* auf TRUE setzt, werden Listeneinträge automatisch alphabetisch sortiert. Groß- und Kleinschreibung spielt dabei keine Rolle, und Umlaute werden wie die zugrundeliegenden Vokale (ß wie s) einsortiert.

Theoretisch kann eine Liste bis zu 32.767 Elemente enthalten; da jedes Element jedoch zusätzlich zu seiner wahren Länge 10 Bytes an Verwaltungsaufwand benötigt, und da alle Steuerelement-Daten einer Form in einem gemeinsamen Segment (64 KB) gespeichert werden, ist der verfügbare Speicher selbst bei Strings der Länge 1 und einer ansonsten leeren Form spätestens nach etwa 5.800 Elementen voll.

Menüeintrag	**Menu**
Eigenschaften	Caption, Checked, CtlName, Enabled, Index, Parent, Separator, Tag, Visible
Ereignisse	Click
Methoden	keine

Der Menüeintrag fällt ein wenig aus der Rolle der üblichen Steuerelemente. Ein Menüeintrag kennt nur ein einziges Ereignis: *Click*. Es wird ausgelöst, wenn der Menüeintrag durch Mausklick oder durch Drücken der Enter-Taste gewählt wird (Details hierzu siehe *Click*-Ereignis im Objekt-Referenzteil).

Menüs werden auch nicht wie die anderen Steuerelemente im Form-Designer einfach auf die Form gezeichnet, sondern mittels des Menüentwurfsfensters erstellt (siehe dazu den Abschnitt über das Menüentwurfsfenster in Kapitel 5).

Obwohl für jeden Menüeintrag ein *Click*-Ereignis eintreten kann, sollten Sie Ereignisprozeduren nach Möglichkeit nur für die Menüeinträge schreiben, die keine untergeordneten Einträge mehr enthalten. Enthält ein Menüeintrag nämlich untergeordnete Einträge, wird bei einem Klick auf diesen nicht das Menü ausgeblendet, sondern ein Untermenü angezeigt; das heißt, daß beim *Click*-Ereignis eines Menüpunktes mit untergeordneten Steuerelementen das Menü noch angezeigt wird, während es üblicherweise schon wieder ausgeblendet ist, wenn das *Click*-Ereignis eintritt.

Allenfalls können Sie das *Click*-Ereignis eines Menüpunktes mit untergeordneten Einträgen verwenden, um dieses untergeordnete Menü zu verändern. Es ist nicht erlaubt, in einer solchen Prozedur an den übergeordneten Menüpunkten zu manipulieren.

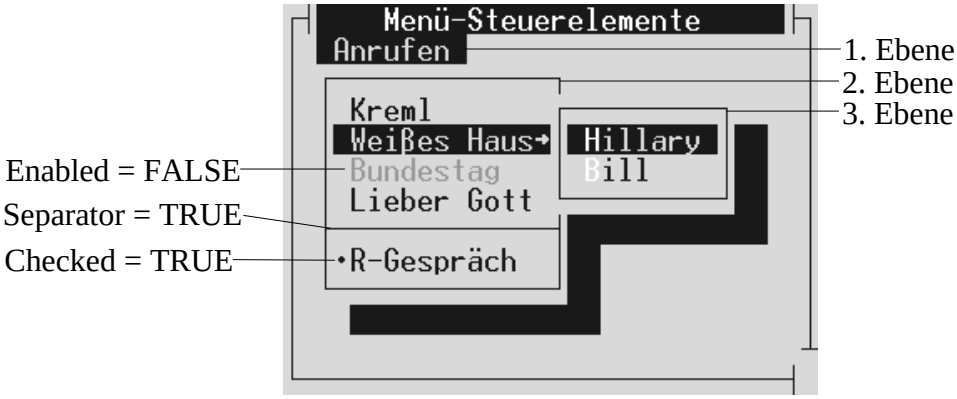


Abbildung 7–13: Ein dreistufiges Menü

Die Eigenschaft *Separator* eines Menüpunktes bestimmt, ob er als Trennlinie dargestellt wird. Ist *Separator* TRUE, wird eine Trennlinie im Menü angezeigt, egal, welcher Text in *Caption* steht.

Die Eigenschaft *Checked* kann verwendet werden, um eine Markierung neben einem Menüpunkt anzuzeigen.

Wenn Sie die Menüs Ihres Programms zur Laufzeit ändern wollen, gibt es dafür zwei Möglichkeiten: Entweder Sie definieren ein Array von Menüeinträgen in jedem Menü; dann können Sie mittels LOAD zur Laufzeit beliebig viele Einträge hinzufügen. Das Menü „Fenster“ in VBDOS wird zum Beispiel zur Laufzeit um die geladenen Module erweitert. Die zweite Möglichkeit ist, alle Menüpunkte schon zur Entwurfszeit zu definieren, einige davon jedoch zu verstekken, indem Sie die *Visible*-Eigenschaft auf FALSE setzen. Viele Programme machen davon Gebrauch, um z. B. in der „Anfänger“-Einstellung nur die wichtigsten, in der „Profi“-Einstellung alle Menüpunkte anzuzeigen.

Optionsfeld	**Option Button**
Eigenschaften	BackColor, Caption, CtlName, DragMode, Enabled, ForeColor, Height, Index, Left, MousePointer, Parent, TabIndex, TabStop, Tag, Top, Value, Visible, Width
Ereignisse	Click, DblClick, DragDrop, DragOver, GotFocus, LostFocus, KeyDown, KeyPress, KeyUp, LostFocus
Methoden	DRAG, MOVE, REFRESH, SETFOCUS

Ein Optionsfeld ist dem Kontrollfeld vergleichbar; es kann entweder markiert (*Value* = TRUE) oder nicht markiert (*Value* = FALSE) sein, und der Status wird vom Benutzer durch einen Mausklick oder die Leertaste geändert.

Im Unterschied zum Kontrollfeld kann jedoch von allen Optionsfeldern, die dem gleichen Objekt untergeordnet sind, immer nur eines markiert sein. Die Markierungen aller anderen Optionsfelder in der gleichen Gruppe werden von VBDOS automatisch gelöscht, wenn einem davon der *Value* TRUE zugewiesen wird.

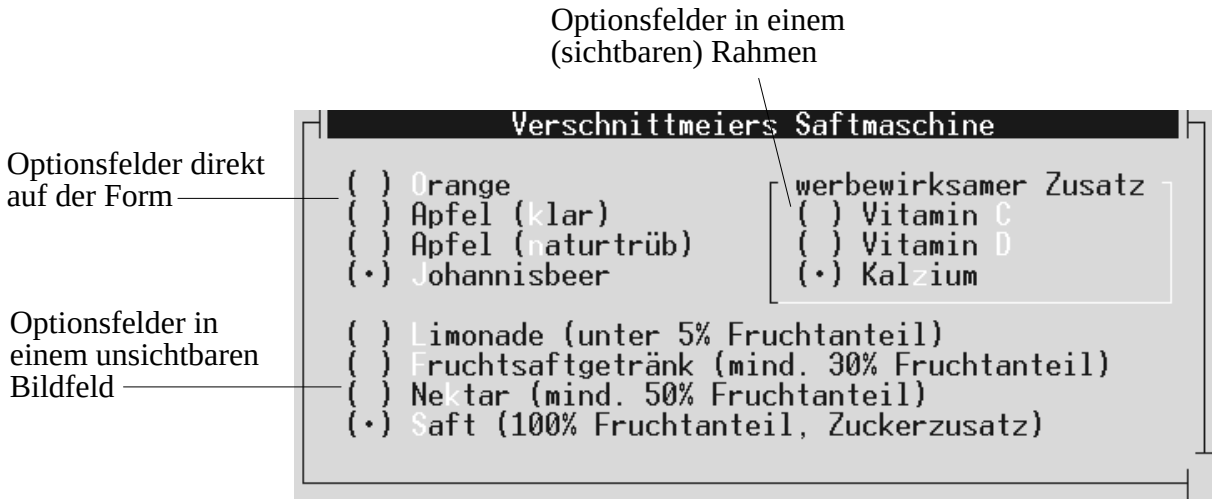


Abbildung 7–14: Drei Gruppen von Optionsfeldern

Daher werden Optionsfelder verwendet, wenn aus verschiedenen Möglichkeiten eine ausgewählt werden soll.

Optionsfelder haben einen zuweilen sehr störenden Nachteil: Sie können nicht ohne weiteres feststellen, ob der Inhalt eines Optionsfeldes verändert wurde. Wenn der Benutzer auf ein nicht markiertes Optionsfeld klickt, wird dieses erst markiert, dann wird ein *GotFocus*- und dann ein *Click*-Ereignis ausgelöst; klickt er auf ein bereits markiertes Optionsfeld, passiert genau das gleiche. Ein *Change*-Ereignis gibt es nicht. Wie es trotzdem geht, zeigt ein Beispiel auf Seite 130.

Rahmen	**Frame**
Eigenschaften	BackColor, Caption, CtlName, Enabled, ForeColor, Height, Index, Left, MousePointer, Parent, TabIndex, Tag, Top, Visible, Width
Ereignisse	DragDrop, DragOver
Methoden	DRAG, MOVE, REFRESH, SETFOCUS

Ein Rahmen wird selten als „aktives“ Steuerelement eingesetzt; vielmehr können Sie unter Verwendung von Rahmen einerseits andere Steuerelemente gruppieren und andererseits Ihre Form ansprechender gestalten.

Rahmen können (wenn Sie die Höhe bzw. Breite auf 1 setzen) als „Linien“ auf der Form angeordnet werden, um diese zu unterteilen.

Die Gruppierung von Steuerelementen dient nicht nur der Optik. Die *Top*- und *Left*-Eigenschaft von Objekten, die in einem Rahmen oder Bildfeld stehen, wird nicht mehr von der oberen linken Ecke der Form aus gemessen, sondern von der oberen linken Ecke des Rahmens bzw. Bildfeldes. Daher werden solche Objekte automatisch mitverschoben, wenn man die Position des Rahmens oder Bildfeldes ändert. Wenn die *Visible*-Eigenschaft des Rahmens auf FALSE gesetzt wird, verschwinden auch alle darin enthaltenen Steuerelemente vom Bildschirm.

Darüber hinaus müssen Sie Optionsfelder in Rahmen oder Bildfeldern gruppieren, wenn Sie erreichen wollen, daß aus verschiedenen Gruppen je ein Feld angewählt sein kann (siehe Abbildung unter „Optionsfelder“ – diese Abbildung enthält auch einen Rahmen).

Textfeld	**Text Box**
Eigenschaften	BackColor, BorderStyle, CtlName, DragMode, Enabled, ForeColor, Height, Index, Left, MousePointer, MultiLine, Parent, ScrollBars, SelLength, SelStart, SelText, TabIndex, TabStop, Tag, Text, Top, Visible, Width
Ereignisse	Change, DragDrop, DragOver, GotFocus, KeyDown, KeyPress, KeyUp, LostFocus
Methoden	DRAG, MOVE, REFRESH, SETFOCUS

Textfelder dienen der Eingabe einer oder mehrerer Textzeilen. Große Textfelder verhalten sich wie ein einfacher Editor (vergleichbar etwa dem VBDOS-Editor).

Die *Text*-Eigenschaft eines Textfeldes enthält alle Zeichen, die im Textfeld stehen. Bei mehrzeiligen Textfeldern, deren *MultiLine*-Eigenschaft auf TRUE gesetzt ist, gibt es zwei Arten von Zeilenumbrüchen: Wenn das Textfeld horizontale automatische Bildlaufleisten besitzt (*Scrollbars*-Eigenschaft), wird nur dort

eine neue Zeile begonnen, wo die Zeichenkombination CR/LF (`CHR$(13) + CHR$(10)`) in der *Text*-Eigenschaft vorkommt oder der Benutzer die Eingabetaste drückt. Besitzt das Textfeld keine horizontalen automatischen Bildlaufleisten, wird der eingegebene Text an Leerzeichen in eine neue Zeile umgebrochen, wenn das nächste Wort nicht mehr auf die Zeile paßt. (Zu „automatischen Bildlaufleisten“ siehe auch S. 106.)

Die Eingabe in einem Textfeld ist auf 32.767 Zeichen begrenzt. Wenn Sie verhindern wollen, daß der Benutzer Zeichen über ein bestimmtes Limit hinaus eingibt, können Sie das in der *KeyPress*-Ereignisprozedur tun, etwa so:

```
SUB Textfeld_KeyPress(KeyAscii AS INTEGER)
   CONST MaxLaenge = 80
   IF LEN(Textfeld.Text) > MaxLaenge AND KeyAscii <> 8 THEN ' Rücktaste ist erlaubt
      Sound 500,1
      KeyAscii = 0 ' Taste verwerfen
   END IF
END SUB
```

Die *KeyPress*-Ereignisprozedur kann auch verwendet werden, um zum Beispiel jedes eingegebene Zeichen sofort in Großbuchstaben umzuwandeln oder auf Gültigkeit zu prüfen. Außerdem ist es möglich, durch die simple Zeile `KeyAscii = 0` in dieser Ereignisprozedur alle Tastendrücke zu verwerfen, so daß im Textfeld nur durch einen Text geblättert werden kann, Änderungen aber unmöglich sind (die Tasten, mit denen man den Cursor im Text bewegen kann, sind ausnahmslos Tasten mit erweitertem Tastencode, die kein *KeyPress*-Ereignis auslösen, deshalb funktionieren sie weiterhin).

In einem mehrzeiligen Textfeld wird mit der Eingabetaste eine neue Zeile begonnen. Befindet sich auf der Form jedoch eine Schaltfläche mit der Eigenschaft *Default* = TRUE, dann „schnappt“ diese sich die Eingabetaste, so daß man in diesem Fall Strg+Eingabe verwenden muß, um eine neue Zeile zu beginnen.

Die Eigenschaft *SelStart* gibt die Cursorposition innerhalb von *Text* an; *SelLength* und *SelText* können verwendet werden, um die Markierung zu setzen oder zu lesen (vgl. Objekt-Referenzteil). Wenn Sie zum Beispiel erreichen wollen, daß der gesamte Text in einem Textfeld markiert wird, wenn das Textfeld den Fokus erhält (ähnlich macht VBDOS es auch), können Sie einfach das Ereignis *GotFocus* abfangen und dort eine Prozedur aufrufen, die die Markierung auf den gesamten Text im Textfeld ausdehnt:

```
SUB Textfeld_GotFocus ()

   VollMarkierung Textfeld ' bzw. stattdessen der Name des jeweiligen Objektes!

END SUB
```

```
SUB VollMarkierung (Objekt AS CONTROL)

   Objekt.SelStart = 0
   Objekt.SelLength = LEN(Objekt.Text)

END SUB
```

Hier habe ich – im Vorgriff auf spätere Ausführungen – schon eine Prozedur geschrieben, die ein Steuerelement als Argument akzeptiert. Dadurch können Sie im *GotFocus*-Prozedurcode jedes Textfeldes einfach *VollMarkierung* mit dem jeweiligen Namen aufrufen und müssen nicht für jedes Steuerelement eine eigene Prozedur schreiben. Weitere Informationen finden Sie im Abschnitt 7.6, „Übergabe von Formen und Steuerelementen als Parameter“.

Timer („Zeitmesser“)	**Timer**
Eigenschaften	CtlName, Enabled, Index, Interval, Parent, Tag
Ereignisse	Timer
Methoden	keine

Der Timer ist das einzige Steuerelement, das von vornherein als unsichtbar konzipiert ist. Sein Zweck ist, das altbekannte ON TIMER(*x*) GOSUB zu ersetzen, das man einsetzen mußte, wenn eine bestimme Aktion in regelmäßigen Zeitabständen ausgeführt werden sollte.

In der Eigenschaft *Interval* hat ein Timer-Objekt die Zeit (in Millisekunden), die bis zum Auslösen des *Timer*-Ereignisses verstreicht. Sobald *Enabled* auf TRUE gesetzt wird, beginnt die Zeit zu laufen, und es wird so lange regelmäßig ein *Timer*-Ereignis ausgelöst, bis *Enabled* wieder auf FALSE gesetzt wird.

Wenn Sie im Form-Designer ein Timer-Element auf die Form setzen, müssen Sie sich nicht um Position und Größe kümmern; sie werden ohnehin nicht beachtet.

Weitere Informationen finden Sie unter *Interval* und *Timer* im Objekt-Referenzteil.

Verzeichnisliste	**Dir List Box**
Eigenschaften	BackColor, CtlName, DragMode, Enabled, ForeColor, Height, Index, Left, List, ListCount, ListIndex, MousePointer, Parent, Path, TabIndex, TabStop, Tag, Top, Visible, Width
Ereignisse	Change, Click, DragDrop, DragOver, GotFocus, KeyDown, KeyPress, KeyUp, LostFocus, MouseDown, MouseMove, MouseUp
Methoden	DRAG, MOVE, REFRESH, SETFOCUS

Ein Verzeichnis-Listenfeld verhält sich im wesentlichen wie ein ganz gewöhnliches Listenfeld, zeigt aber automatisch eine Liste aller Verzeichnisse an. Die *Path*-Eigenschaft enthält stets das gerade gewählte Verzeichnis (inkl. Laufwerksbezeichnung) und kann auch benutzt werden, um die Verzeichnisliste auf ein anderes Laufwerk umzustellen.

Üblicherweise wird man bei einem System mit Laufwerks-, Verzeichnis- und Dateiliste dafür sorgen, daß im *Change*-Ereignis der Laufwerksliste das neue Laufwerk an die *Path*-Eigenschaft der Verzeichnisliste übergeben wird, und daß das *Change*-Ereignis der Verzeichnisliste den neuen Pfad an die *Path*-Eigenschaft der Dateiliste übergibt. Falls Sie ein Textfeld hinzufügen, in das der Benutzer auch direkte Eingaben tätigen kann, muß bei Abschluß einer solchen Eingabe der Text an die Dateiliste übergeben werden; diese löst dann ein *PathChange*-Ereignis aus, wenn der Benutzer im Text ein anderes Laufwerk oder Verzeichnis eingeschlossen hat, und die Laufwerks- und Verzeichnisliste können aktualisiert werden.

Ein Beispiel für diesen etwas komplizierten Vorgang bietet das im Lieferumfang von VBDOS enthaltene Projekt „CMNDLG“ („common dialogues“), eine Sammlung häufig gebrauchter Dialogboxen, die ich im Kapitel 21 ausführlicher behandle.

Im Unterschied zu einem gewöhnlichen Listenfeld kennt die Verzeichnisliste auch negative Indizes. Das Listenelement mit dem Index 0 ist das erste Unterverzeichnis des gewählten Verzeichnisses, von da an aufsteigend finden sich weitere Unterverzeichnisse; absteigend sind die darüberliegenden Verzeichnisse bis hin zum Hauptverzeichnis eingetragen. Auch *ListIndex* kann deshalb einen negativen Wert annehmen. Wenn sie einer Verzeichnisliste z. B. die *Path*-Eigenschaft „C:\PROGRAMM\VBDOS“ zuweisen, enthält sie in den Elementen *List(0)*, *List(1)* usw. die Unterverzeichnisse dieses Verzeichnisses, *List(–1)* ist genau dieses Verzeichnis, *List(–2)* ist „C:\PROGRAMM“, und *List(–3)* ist „C:\“. Auf weitere Negativwerte können Sie zugreifen, aber sie liefern alle einen Leerstring zurück.

7.3 Formen

Form	Form
Eigenschaften	AutoRedraw, BackColor, BorderStyle, Caption, ControlBox, CurrentX, CurrentY, DragMode, Enabled, ForeColor, FormName, FormType, Height, Left, MaxButton, MinButton, MousePointer, Parent, ScaleHeight, ScaleWidth, Tag, Top, Visible, Width, WindowState
Ereignisse	Click, DblClick, DragDrop, DragOver, GotFocus, KeyDown, KeyPress, KeyUp, Load, LostFocus, MouseDown, MouseMove, MouseUp, Paint, Resize, UnLoad
Befehle	LOAD, UNLOAD
Methoden	CLS, DRAG, HIDE, MOVE, PRINT, PRINTFORM, REFRESH, SHOW, TEXTHEIGHT, TEXTWIDTH

Eine Form ist ein Bildschirmfenster, das andere Steuerelemente enthält und auch zur Direktausgabe von Text mit der PRINT-Methode verwendet werden kann. Jedes Formmodul muß und darf (nur) eine Form enthalten.

Die wichtigsten speziellen Eigenschaften einer Form sind *BorderStyle*, *ControlBox*, *MinButton* und *MaxButton* (siehe Referenzteil); damit legen Sie fest, ob die Form einen Rahmen hat (und welchen), ob sie verschoben, verkleinert und vergrößert werden kann. Das Verkleinern und Vergrößern ist eine nette Spielerei, hat aber nicht viel Sinn, wenn Sie Steuerelemente auf der Form anordnen, da diese nach einer Verkleinerung u.U. nicht mehr sichtbar sind (sofern sie nicht ihre Position und Größe automatisch anpassen).

Formen auf Symbolgröße verkleinern

Mit der Eigenschaft *WindowState* läßt sich die Form auf Symbolgröße verkleinern oder vergrößern, bis sie den ganzen Bildschirm einnimmt. Eine Form, die auf Symbolgröße verkleinert wurde, umfaßt zwei Zeilen á 12 Zeichen, und die ersten 10 Buchstaben der *Caption*-Eigenschaft werden angezeigt. Formen in Symbolgröße können eine Alternative zu Schaltflächen sein: Anstatt in der Hauptform Ihres Programms eine Schaltfläche „Standardeinstellungen...“ anzubieten, die eine entsprechende Form öffnet, können Sie genausogut die Standardeinstellung-Form in Symbolgröße anzeigen und es dem Benutzer überlassen, sie bei Bedarf zu vergrößern.

MDI-Formen – ein fragwürdiges Konstrukt

Die Eigenschaft *FormType* bestimmt, ob es sich um eine Standard- oder eine MDI-Form handelt. MDI-Formen („Multiple Document Interface“, also etwa „Mehrdokumentenschnittstelle“) sind die Ausnahme; sie nehmen immer den ganzen Bildschirm ein und können keine Steuerelemente (außer Menüeinträgen) enthalten. Dafür sind auf MDI-Formen untergeordnete Formen möglich. So läßt sich beispielsweise ein Text-Editor programmieren, in den man verschiedene Texte hintereinander laden und diese dann über-, unter- oder nebeneinander anzeigen kann. Diese – an sich leistungsfähige – Idee wird in VBDOS allerdings dadurch ad absurdum geführt, daß es nicht möglich ist, ein Array von Formen zu erstellen, das man zur Laufzeit erweitert. Wollen Sie in Ihrem Texteditor also maximal vier Dokumente bearbeiten lassen, so müssen Sie die MDI-Umgebungsform und vier untergeordnete Formen entwerfen; die untergeordneten Formen sind, bis auf den Namen, völlig identisch. Wenn der Benutzer dann zur Laufzeit eine zusätzliche Datei laden will, kommt es zu solch haarsträubenden Konstrukten wie diesem:

```
SELECT CASE GeladeneFormen
CASE 0: FORM1.SHOW
CASE 1: FORM2.SHOW
CASE 2: FORM3.SHOW
CASE 3: FORM4.SHOW
END SELECT
GeladeneFormen = GeladeneFormen + 1
```

Und so zieht es sich durch das ganze Programm – überall, wo auf die Formen zugegriffen wird (und man nicht gerade mit SCREEN.ActiveForm hantieren kann), findet sich der gleiche Befehl x-mal, nur mit einem anderen Formnamen. Vergessen Sie also lieber die MDI-Formen; mir wäre kein Problem bekannt, das man nicht mit einer Standardform (wird übrigens auch „SDI“ genannt) lösen könnte.

Wenn die Tochterform einer MDI-Form ein eigenes Menü besitzt, wird das Menü der MDI-Form selbst durch das Menü der Tochterform überschrieben, solange diese den Fokus hat.

Namenskonventionen beim Umgang mit Formen

Die Verwendung von Formen gleicht in vielem der Verwendung von Steuerelementen; ein wichtiger Unterschied ist jedoch, daß die Ereignisprozeduren für eine Form nicht aus dem Namen der Form und dem Ereignisnamen, sondern schlicht aus „Form_“ und dem Ereignisnamen bestehen. Egal, wie der Name

Ihrer Form ist – die Prozedur für das *Load*-Ereignis heißt immer „Form_Load“. Greifen Sie hingegen aus anderen Modulen auf Eigenschaften der Form oder auf ein Steuerelement zu, müssen Sie den Namen der Form angeben.

PROGRAMM.BAS	PROGRAMM.FRM (enthält die Form „HauptForm“ und ein Bildfeld „Anzeige“)
`PRINT "Hallo"` gibt den String „Hallo“ unter Verwendung des üblichen PRINT-Befehls auf dem Bildschirm aus; wird eine Form angezeigt, gibt es einen Fehler. `Anzeige.PRINT "Hallo"` verursacht in jedem Fall einen Fehler. Um den gleichen Effekt wie rechts zu erzielen, muß `HauptForm.PRINT "Hallo"` bzw. `HauptForm.Anzeige.PRINT "Hallo"` verwendet werden.	`PRINT "Hallo"` wendet die Methode PRINT auf die Form an. `Anzeige.PRINT "Hallo"` wendet die Methode PRINT auf das Bildfeld an. Die Syntax `HauptForm.PRINT "Hallo"` bzw. `HauptForm.Anzeige.PRINT "Hallo"` führt zum gleichen Resultat.
`Top = 5` weist der Variablen *Top* die Zahl 5 zu. Um den gleichen Effekt wie rechts zu erzielen, muß `HauptForm.Top = 5` angewandt werden.	`Top = 5` verschiebt die Form, so daß sie in der Zeile 5 beginnt (die Eigenschaft *Top* der Form wird auf 5 geändert.)
`Form_Click` führt zu einer Fehlermeldung, weil die Prozeduren des Formmoduls „privat“ sind und von außerhalb nicht aufgerufen werden können.	`Form_Click` ruft die Click-Ereignisprozedur wie eine gewöhnliche Prozedur auf und simuliert so einen Mausklick auf die Form. (Natürlich muß die Ereignisprozedur existieren, sonst wird eine Fehlermeldung erzeugt.)

Abbildung 7–15: Übersicht über Form- und Objektreferenzen aus verschiedenen Modulen

7.4 Spezial-Objekte

VBDOS kennt drei spezielle Objekte: CLIPBOARD, PRINTER und SCREEN. Die ersten beiden wurden implementiert, um wenigstens eine teilweise Kompatibilität zu WINDOWS zu erhalten, das mit ebensolchen Objekten arbeitet. Man kann auf beide getrost verzichten, wenn man nicht plant, sein Programm irgendwann einmal nach VB für WINDOWS zu übertragen.

Das SCREEN-Objekt gibt es in VB für WINDOWS zwar auch, aber von Kompatibilität kann hier nicht die Rede sein; es wurde wohl in VBDOS eingebaut, um dem Programmierer die Einstellung der globalen Variablen für die Formenanzeige (Eigenschaft *ControlPanel*) zu ermöglichen.

CLIPBOARD	
Eigenschaften	keine
Ereignisse	keine
Methoden	CLEAR, GETTEXT, SETTEXT

In VBDOS ist das CLIPBOARD nichts weiter als ein Zwischenspeicher für Text (so etwas wie eine globale String-Variable). Sie können Text hineinschreiben, Text herauslesen und Text löschen. Der Ursprung dieses Objekts ist die mächtige WINDOWS-Zwischenablage, mittels derer man Texte und Grafiken zwischen verschiedenen Anwendungen kopieren kann. Von solcher Funktionalität ist das CLIPBOARD jedoch weit entfernt; es bietet auch dann keinen Zugriff auf die WINDOWS-Zwischenablage, wenn WINDOWS läuft.

PRINTER	
Eigenschaften	PrintTarget
Ereignisse	keine
Methoden	ENDDOC, NEWPAGE, PRINT

Das PRINTER-Objekt ist ein Ersatz für den LPRINT- oder PRINT#-Befehl. Sie können damit Daten an den Drucker schicken; die Eigenschaft *PrintTarget* legt fest, auf welchen Drucker die Ausgabe gesendet wird. Auch dieses Objekt muß vor dem Hintergrund des PRINTER-Objekts in VB für WINDOWS gesehen werden, das Grafiken verarbeiten und sich außerdem über den WINDOWS-Druckertreiber an jeden Drucker automatisch anpassen kann.

SCREEN	
Eigenschaften	ActiveControl, ActiveForm, ControlPanel, Height, MousePointer, Width
Ereignisse	keine
Methoden	HIDE, SHOW

Das Objekt SCREEN ermöglicht es, über die Eigenschaften *ActiveForm* und *ActiveControl* das Steuerelement und die Form zu ermitteln, die gerade den Fokus haben.

Die Eigenschaft *MousePointer* ist hier nützlich, um den Mauszeiger insgesamt – und nicht bloß, während er über einem Objekt steht – zu verändern.

Schließlich ermöglicht das Eigenschaften-Array *ControlPanel()* die Einstellung der Farben für die Anzeige von Objekten (sofern die Objekte dafür nicht selbst

eine Eigenschaft besitzen) und die Festlegung einer Hintergrundanzeige für den Bildschirm (siehe Objekt-Referenzteil).

7.5 Aufrufreihenfolge der Ereignisse

Häufig löst ein und dasselbe Ereignis verschiedene Ereignisprozeduren aus. Wenn Sie zum Beispiel in einer Form ein Textfeld und eine Schaltfläche haben und das Textfeld den Fokus hat, löst ein Mausklick auf die Schaltfläche die Ereignisse *Textfeld_LostFocus*, *Befehl_GotFocus* und *Befehl_Click* (in dieser Reihenfolge) aus. Ein einfacher Tastendruck im Textfeld sorgt schon für bis zu vier Ereignisse: *KeyDown*, *KeyPress*, *Change* und *KeyUp* (wobei kein Ereignis eintritt, falls es sich um die Tab-Taste handelt oder um ESC/Enter, wenn eine Schaltfläche mit *Cancel*- bzw. *Default*-Eigenschaft existiert; *KeyPress* tritt nur ein, wenn es sich um keine Sondertaste handelt, und *Change* tritt nur ein, wenn der Text durch den Tastendruck verändert wird).

Wenn eine Form erstmals geladen und angezeigt wird (z. B. mit der SHOW-Methode), treten die Ereignisse `Form_Load`, `Form_Resize`, `Form_GotFocus` und `Form_Paint` ein (*Paint* nur, wenn *AutoRedraw* FALSE ist; statt `Form_GotFocus` tritt das *GotFocus*-Ereignis für das Steuerelement mit der niedrigsten *TabIndex*-Eigenschaft ein, wenn Steuerelemente auf der Form existieren, die den Fokus erhalten können).

Die möglichen Kombinationen und Situationen lassen sich hier wegen des Umfangs nicht in einer Tabelle zusammenfassen; in den meisten Fällen spielt es auch für das Programm keine Rolle, welches Ereignis vor welchem ausgelöst wird. Wenn es einmal darauf ankommt, probieren Sie es aus – und rechnen Sie mit allem...

7.6 Übergabe von Formen und Steuerelementen als Parameter

Mit dem ereignisgesteuerten Programmieren haben auch zwei neue Datentypen Einzug in BASIC gehalten: „FORM“ und „CONTROL“. Sie sind jedoch nur eingeschränkt nutzbar:

- Sie können keine eigenen Variablen vom Typ FORM oder CONTROL definieren (z. B. mit DIM).
- Sie können Variablen des Typs FORM oder CONTROL weder als ganzes ausgeben oder speichern noch einander zuweisen.
- Variablen des Typs FORM oder CONTROL können auch nicht miteinander verglichen werden.

Einziger Zweck dieser Datentypen ist es, die Übergabe von Formen oder Steuerelementen als Parameter an Prozeduren und Funktionen zu ermöglichen. Die Bezeichnungen „FORM“ und „CONTROL“ tauchen also *ausschließlich* im Parameter-Bereich einer SUB-, FUNCTION- oder DECLARE-Anweisung auf.

Ein Beispiel dafür ist das Ereignis *DragDrop*, das ich im folgenden kurz vorstellen will. Betrachten Sie folgende Form:

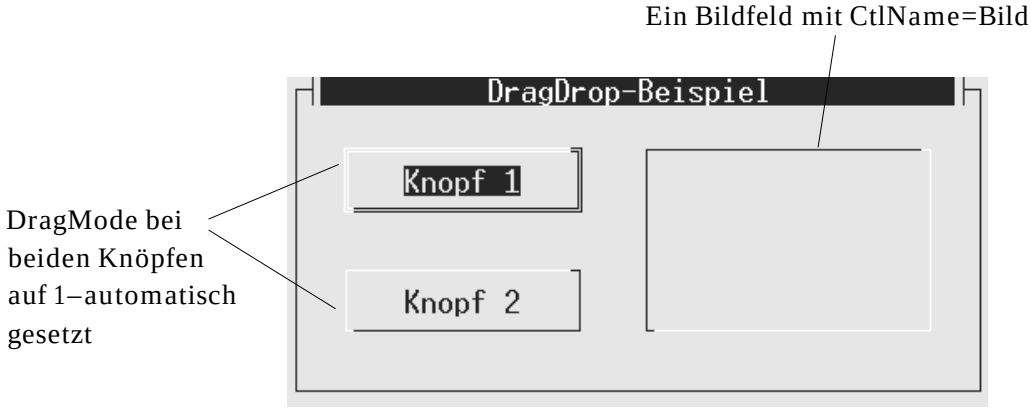


Abbildung 7–16: DragDrop-Beispielform

Da die Eigenschaft *DragMode* der beiden Knöpfe (Befehlsschaltflächen) auf 1 gesetzt wurde, können sie bei Gedrückthalten der Maustaste verschoben werden. Werden sie über einem anderen Steuerelement „fallengelassen“, wird für dieses Steuerelement – im Beispiel nehme ich an, es handele sich um das Bildfeld „Bild“ – ein *DragDrop*-Ereignis mit folgender Code-Schablone ausgelöst:

```
SUB Bild_DragDrop (Source AS CONTROL, X AS SINGLE, Y AS SINGLE)
...
END SUB
```

Dem Steuerelement „Bild“ wird also durch den Parameter *Source* das Steuerelement übergeben, das über ihm fallengelassen wurde. Über diesen Parameter kann nun auf die Eigenschaften des verschobenen Steuerelements zugegriffen werden; soll zum Beispiel das Bildfeld als Papierkorb dienen, in den man nicht benötigte Steuerelemente schiebt, um sie verschwinden zu lassen, könnte die DragDrop-Prozedur so aussehen:

```
SUB Bild_DragDrop (Source AS CONTROL, X AS SINGLE, Y AS SINGLE)
   Source.Visible = FALSE
END SUB
```

Ein solcher Parameter kann jedoch *nur* zum Zugriff auf die Eigenschaften eines Objektes verwendet werden. Es gibt keine direkte Möglichkeit, festzustellen, um welches Objekt es sich eigentlich handelt. Ein Konstrukt der Form `IF Source = Befehl1 THEN...` ist ungültig. Die *CtlName*-Eigenschaft ist zur Laufzeit

nicht verfügbar. Im obigen Beispiel könnte man allerdings die *Caption*-Eigenschaft verwenden, um zu prüfen, welcher der beiden Knöpfe verschoben wurde:

```
SUB Bild_DragDrop (Source AS CONTROL, X AS SINGLE, Y AS SINGLE)
   Bild.PRINT Source.Caption; "wurde verschoben."
END SUB
```

Wenn einmal auch diese Möglichkeit ausfällt – zum Beispiel, weil ein Objekt keine *Caption*-Eigenschaft hat oder sich ihr Wert im Programmablauf ändert – kann man die „Joker-Eigenschaft“ *Tag* verwenden, um seine Objekte eindeutig zu kennzeichnen. Mehr dazu in der Eigenschaften-Übersicht im Referenzteil.

7.7 Die Eigenschaften ActiveForm, ActiveControl und Parent

Neben der Möglichkeit, Steuerelemente oder Formen als Parameter zu übergeben, gibt es auch noch eine andere Weise, auf die Sie mit Variablen vom Typ „FORM“ oder „CONTROL“ in Berührung kommen können: Es gibt drei Eigenschaften, die solche Typen zurückgeben. Die *Parent*-Eigenschaft jedes Steuerelements ist ein Verweis auf die Form, auf der es sich befindet; die *ActiveControl*-Eigenschaft des Spezialobjektes SCREEN ist ein Verweis auf das Steuerelement, das den Fokus hat, und *ActiveForm* bezeichnet die aktuelle Form.

Extensiven Gebrauch von der Parent-Eigenschaft macht zum Beispiel die folgende Routine, die einen Rahmen um ein Objekt zeichnet. Die meisten Objekte, bei denen das sinnvoll ist, können das zwar selbst (mit der *BorderStyle*-Eigenschaft) – aber wer weiß, vielleicht kommen Sie ja einmal in die Situation, einen Rahmen um ein Kontrollfeld zeichnen zu wollen...

```
SUB ZeichneRahmen (Objekt AS CONTROL)

   ' Objekt.Parent ist die "Mutterform"
   Objekt.Parent.CurrentX = Objekt.Left   1:Objekt.Parent.CurrentY = Objekt.Top   1
   Objekt.Parent.PRINT "Ú"; STRING$(Objekt.Width, 196); "¿";

   FOR i% = Objekt.Top TO Objekt.Top + Objekt.Height   1
      Objekt.Parent.CurrentY = i%: Objekt.Parent.CurrentX = Objekt.Left   1
      Objekt.Parent.PRINT "³";
      Objekt.Parent.CurrentX = Objekt.Left + Objekt.Width
      Objekt.Parent.PRINT "³"
   NEXT

   Objekt.Parent.CurrentX = Objekt.Left   1
   Objekt.Parent.CurrentY = Objekt.Top + Objekt.Height
   Objekt.Parent.PRINT "À"; STRING$(Objekt.Width, 196); "Ù";

END SUB
```

Listing 7–3: RAHMEN.BAS

Der Routine wird ein Steuerelement übergeben (zum Beispiel könnten Sie in die Click-Ereignisprozedur der Schaltfläche „Abbrechen“ den Befehl `ZeichneRahmen Abbrechen` schreiben, um die Schaltfläche mit einem „Trauerrand“ versehen zu lassen). Dann werden Position und Größe des benötigten Rahmens ermittelt, und durch Anwendung der Eigenschaften *CurrentX* und *CurrentY* sowie der Methode PRINT auf die *Parent*-Form des Steuerelements wird der Rahmen angezeigt.

Hinweis: Die Routine funktioniert nur, wenn sie auf Steuerelemente angewandt wird, die nicht ihrerseits einem anderen Steuerelement untergeordnet sind. Sonst ist nämlich deren *Top*- und *Left*-Eigenschaft nicht vom Rand der Form, sondern vom Rand des Mutterobjekts aus gezählt, und der Rahmen wird an der falschen Stelle ausgegeben. Es gibt in VBDOS keine Möglichkeit, ähnlich wie mit der *Parent*-Eigenschaft auf das Mutterobjekt eines untergeordneten Objektes zuzugreifen.

ActiveControl ist ebenfalls vom Typ CONTROL; diese Eigenschaft ist dann interessant, wenn man in einer *LostFocus*-Ereignisprozedur feststellen möchte, welches Objekt den Fokus erhalten hat. *ActiveControl* wird nämlich schon geändert, bevor die *LostFocus*-Ereignisprozedur des Steuerelementes, das bisher den Fokus hatte, aufgerufen wird. Dadurch wird es möglich, z. B. bei einer Gruppe von zusammengehörigen Elementen erst dann zu reagieren, wenn der Benutzer den Fokus aus der Gruppe herausbewegt.

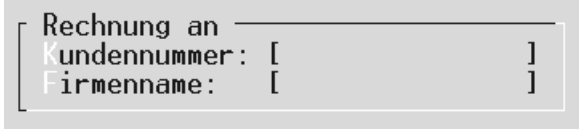


Abbildung 7–17: Hier kann ActiveControl sinnvoll sein

Angenommen, Sie möchten bei dem abgebildeten Form-Ausschnitt eine Fehlermeldung anzeigen, wenn der Benutzer den Rahmen verläßt, ohne in eines der beiden Textfelder einen Eintrag gemacht zu haben. Dazu reicht die normale *LostFocus*-Prozedur nicht aus; würden Sie in `KundenNummer_LostFocus` prüfen, ob beide Felder leer sind, und ggf. einen Fehler anzeigen, so könnte es ja sein, daß der Benutzer gerade zu *FirmenName* wechseln wollte, um dort einen Eintrag zu machen. Stattdessen würden Sie schreiben:

```
SUB KundenNummer_LostFocus ()
   IF ActiveControl.Tag <> "FName" THEN
      IF KundenNummer.Text = "" AND FirmenName.Text = "" THEN
         FehlerMeldung: SETFOCUS KundenNummer
      END IF
   END IF
END SUB
```

```
SUB FirmenName_LostFocus ()
   IF ActiveControl.Tag <> "KNummer" THEN
      IF KundenNummer.Text = "" AND FirmenName.Text = "" THEN
         FehlerMeldung: SETFOCUS KundenNummer
      END IF
   END IF
END SUB
```

Hierbei wird vorausgesetzt, daß Sie die *Tag*-Eigenschaften der beiden Textfelder entsprechend gesetzt haben; da auf die *CtlName*-Eigenschaft zur Laufzeit nicht zugegriffen werden kann, muß man sich zur Identifikation der Felder mit einer anderen Eigenschaft behelfen. Wenn Sie diszipliniert programmieren (und nicht dauernd die Form umgestalten), können Sie die Felder auch anhand der *TabIndex*- oder einer anderen Eigenschaft identifizieren. Es muß jedoch für diesen Zweck eine Eigenschaft sein, die von allen Objekten, die den Fokus erhalten können, unterstützt wird – sonst erzeugen Sie mit der Abfrage womöglich einen Fehler 422 (*Eigenschaft nicht gefunden*).

ActiveForm schließlich ist am ehesten in MDI-Anwendungsprogrammen von Nutzen, z. B. wenn in einem Texteditor drei Dateien geladen sind und festgestellt werden soll, in welcher davon der Benutzer gerade Änderungen vornimmt.

7.8 Eine Form – mehrere Gesichter

Wenn Sie sich näher mit der ereignisgesteuerten Programmierung beschäftigen und einmal ein größeres Projekt in Angriff nehmen, werden Sie sehr schnell eine große Zahl von Formen in der Projektliste stehen haben. Das ist nicht immer wünschenswert, weil ein Projekt dadurch leicht an Übersichtlichkeit verliert.

Sie können, um die Gesamtzahl an Formen gering zu halten, mehrere Formen zusammenfassen und jeweils (indem Sie *Visible* auf FALSE setzen) die Objekte verstecken, die nicht gebraucht werden.

Betrachten Sie zum Beispiel folgende beiden Formen:

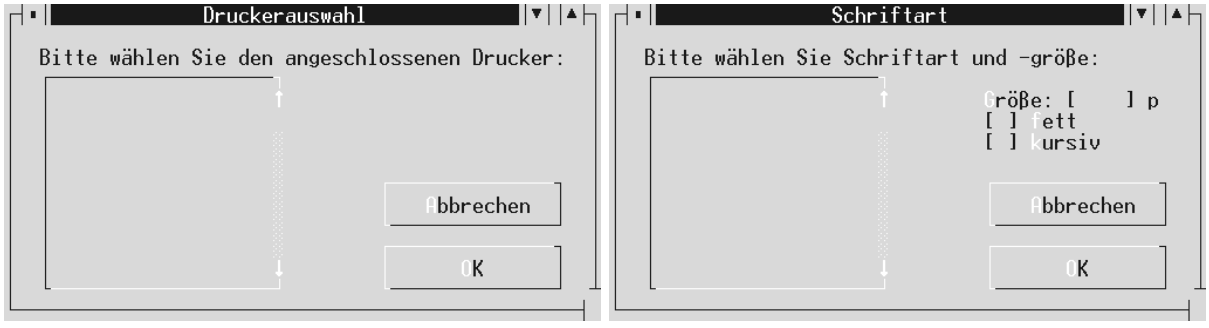


Abbildung 7–18: Zwei durchaus ähnliche Formen

Nehmen wir an, Sie verwenden in einem Programm die Prozeduren „Waehle-Drucker“ und „WaehleSchrift“, die ihrerseits diese Formen anzeigen:

```
SUB WaehleDrucker (Drucker AS STRING)

   DruckerForm.SHOW 1
   Drucker = DruckerForm.Liste.List(ListCount) ' Das gewählte Element

END SUB
```

```
SUB WaehleSchrift (SchriftName AS STRING, Groesse AS INTEGER, Fett AS INTEGER,
                   Kursiv AS INTEGER)

   SchriftForm.SHOW 1
   SchriftName = SchriftForm.Liste.List(ListCount) ' Das gewählte Element
   Groesse = VAL(SchriftForm.Groesse.Text)         ' Wert des Textfeldes
   Kursiv = (SchriftForm.Kursiv.Value = 1)         ' TRUE, wenn Value = 1
   Fett = (SchriftForm.Fett.Value = 1)             ' TRUE, wenn Value = 1

END SUB
```

Dann können Sie sich die erste Form auch sparen und stattdessen schreiben (ich nehme an, die verbliebene Form hieße jetzt „GesamtForm“):

```
SUB WaehleDrucker (Drucker AS STRING)

   GesamtForm.Groesse.Visible = FALSE: GesamtForm.Kursiv.Visible = FALSE
   GesamtForm.Aufforderung = "Bitte wählen Sie den angeschlossenen Drucker:"
   GesamtForm.Bezeichnung2.Visible = FALSE    ' Der Text "Größe:"
   GesamtForm.Bezeichnung3.Visible = FALSE    ' Das kleine p hinter dem Größenfeld
   GesamtForm.SHOW 1
   Drucker = GesamtForm.Liste.List(ListCount) ' Das gewählte Element

END SUB
```

```
SUB WaehleSchrift (SchriftName AS STRING, Groesse AS INTEGER, Fett AS INTEGER,
                   Kursiv AS INTEGER)

   GesamtForm.Groesse.Visible = TRUE: GesamtForm.Kursiv.Visible = TRUE
   GesamtForm.Aufforderung = "Bitte wählen Sie den angeschlossenen Drucker:"
   GesamtForm.Bezeichnung2.Visible = TRUE    ' Der Text "Größe:"
   GesamtForm.Bezeichnung3.Visible = TRUE    ' Das kleine p hinter dem Größenfeld
   SchriftForm.SHOW 1
   SchriftName = SchriftForm.Liste.List(ListCount) ' Das gewählte Element
   Groesse = VAL(SchriftForm.Groesse.Text)         ' Wert des Textfeldes
   Kursiv = (SchriftForm.Kursiv.Value = 1)         ' TRUE, wenn Value = 1
   Fett = (SchriftForm.Fett.Value = 1)             ' TRUE, wenn Value = 1

END SUB
```

So haben Sie eine Form, die jeweils den Gegebenheiten angepaßt wird. Das Verfahren läßt sich sogar noch verfeinern, indem man alle die Operationen, die das Formaussehen beeinflussen, in das *Paint*-Ereignis der Form anstatt in die Aufrufprozeduren einbaut. Man kann dann die *Tag*-Eigenschaft der Form benutzen, um ihr mitzuteilen, wie sie denn nun auszusehen hat:

```
 SUB WaehleDrucker (Drucker AS STRING)

   GesamtForm.Tag = "DRUCK"
   GesamtForm.SHOW 1
   Drucker = GesamtForm.Liste.List(ListCount) ' Das gewählte Element

END SUB
```

```
SUB WaehleSchrift (SchriftName AS STRING, Groesse AS INTEGER, Fett AS INTEGER,
                   Kursiv AS INTEGER)

   GesamtForm.Tag = "SCHRIFT"
   SchriftForm.SHOW 1
   SchriftName = SchriftForm.Liste.List(ListCount) ' Das gewählte Element
   Groesse = VAL(SchriftForm.Groesse.Text)         ' Wert des Textfeldes
   Kursiv = (SchriftForm.Kursiv.Value = 1)         ' TRUE, wenn Value = 1
   Fett = (SchriftForm.Fett.Value = 1)             ' TRUE, wenn Value = 1

END SUB
```

```
SUB Form_Paint ()                                  ' Im Formmodul!

   SELECT CASE Tag
   CASE "DRUCK"
      ' Hier die Zeilen aus der alten WaehleDrucker-Prozedur hinkopieren
   CASE "SCHRIFT"
      ' Hier die Zeilen aus der alten WaehleSchrift-Prozedur hinkopieren
   END SELECT

END SUB
```

Nun habe ich hier ein recht einfaches Beispiel gewählt. Diese Methode erlaubt es allerdings, sehr viel in einer Form unterzubringen. Man kann auch noch weiter gehen und die Größe der Form je nach Verwendungszweck anpassen. Wenn Sie sich einmal ansehen möchten, was passiert, wenn man diese „Formkompaktierung“ auf die Spitze treibt, betrachten Sie die mitgelieferte Datei CMNDLG.FRM. Hier sind sage und schreibe sechs völlig verschiedene Formen zusammengepfercht, und das Ganze läßt sich mit dem Form-Designer nur noch unter Entbehrungen bearbeiten. Um nicht bei jedem Aufruf allzuviele *Visible*-Eigenschaften ändern zu müssen, hat man hier einfach jede der sechs zusammengewürfelten Formen auf ein eigenes Bildfeld (ohne Rahmen) gesetzt. Dadurch braucht man nur das Bildfeld auf *Visible* = FALSE zu stellen, und schon sind auch alle untergeordneten Steuerelemente verschwunden.

7.9 Arrays von Steuerelementen

Anstatt jedem Steuerelement einen eigenen, eindeutigen Namen (*CtlName*) zu geben, können Sie auch mehrere Steuerelemente mit dem gleichen Namen versehen und sie durch ihre *Index*-Eigenschaft (von 0 an aufsteigend) unterscheiden. Die *Index*-Eigenschaft bleibt bei normalen Steuerelementen ungenutzt.

Die Bedingung für eine solche Zusammenfassung ist, daß die Objekte sich auf der gleichen Form befinden und den gleichen Typ haben (Sie können nicht ein Textfeld und eine Schaltfläche mit der gleichen *CtlName*-Eigenschaft versehen).

Derart zusammengefaßte Steuerelemente verhalten sich wie ein Array, und der Zugriff verläuft ebenso: Wenn Sie fünf Bildfelder mit der *CtlName*-Eigenschaft „Bild“ haben, ist *Bild(0).Top* die *Top*-Eigenschaft des ersten, *Bild(1).Top* die *Top*-Eigenschaft des zweiten usw. Die Steuerelemente sind weiterhin unabhängig in ihren Eigenschaften und werden wie alle anderen behandelt, mit dem Unterschied, daß man immer in Klammern angeben muß, welches man meint.

Der große Unterschied findet sich bei den Ereignisprozeduren. Hier hat nicht mehr jedes Steuerelement seine eigene Prozedur, sondern es gibt eine Ereignisprozedur pro Ereignis für das gesamte Array, der (vor anderen Parametern, die sie möglicherweise empfängt) stets der Index des betroffenen Steuerelements übergeben wird.

Steuerelemente-Arrays sind in zwei Situationen sinnvoll: Wenn eine Form viele der Funktion nach ähnliche Steuerelemente enthält, oder wenn zur Laufzeit neue Steuerelemente erstellt werden sollen.

Viele ähnliche Steuerelemente

Häufig kommt es vor, daß eine Anzahl von Steuerelementen auf einer Form ähnlich agieren. Insbesondere bei Optionsfeldern lohnt sich schnell ein Array; stellen Sie sich vor, es gibt auf der Form 4 Optionsfelder wie diese:

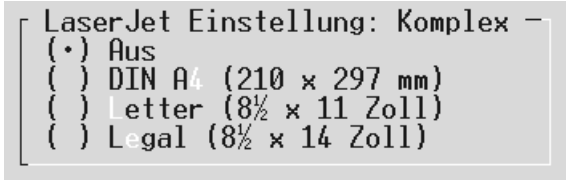


Abbildung 7–19: Optionsfelder, für die sich ein Steuerelemente-Array lohnt

Angenommen ferner, Sie möchten bei jeder Änderung eines dieser Felder eine Meldung anzeigen, die dem Benutzer mitteilt, daß dadurch alle geladenen Schriftarten im Drucker gelöscht werden. Dann können Sie, wenn Sie die vier

Optionsfelder als ein Steuerelementefeld deklariert haben, in der *Click*-Ereignisprozedur schreiben:

```
SUB Komplex_Click(Index AS INTEGER)

   STATIC AlteEinstellung AS INTEGER ' wird gebraucht, um festzustellen, ob der
                                     ' Benutzer auf ein schon markiertes Feld
   DIM Taste AS INTEGER              ' geklickt hat oder nicht

   IF Index <> AlteEinstellung       ' Nur bei Änderung...
      WarnMeldung Taste              ' Angenommen, Taste wäre TRUE, wenn der
                                     ' Benutzer trotz Warnung weitermacht, FALSE,
                                     ' wenn er abbricht
      IF Taste = FALSE THEN          ' Im Falle eines Abbruchs muß die alte Ein-
         Komplex(AlteEinstellung).Value = TRUE ' stellung wiederhergestellt werden.
      ELSE                           ' Dabei wird wieder ein Click ausgelöst!
         AlteEinstellung = Index     ' Das macht aber nichts, weil dieses "Tocher-
      END IF                         ' ereignis" gleich bei der ersten IF-Abfrage
   END IF                            ' beendet wird, da Index = AlteEinstellung.

END SUB
```

Die Verwendung von Steuerelement-Arrays muß nicht immer so kompliziert aussehen. Das folgende Code-Fragment sorgt dafür, daß beim Klick auf ein Bildfeld der Text „Guten Tag“ darin angezeigt wird, egal, wieviele Bildfelder es sind (sie müssen alle in einem Array mit *CtlName* = „Gruss“ zusammengefaßt sein):

```
SUB Gruss_Click (Index AS INTEGER)
   Gruss(Index).PRINT "Guten Tag"
END SUB
```

Hinzufügen von Steuerelementen zur Laufzeit

Das zweite große Einsatzgebiet von Steuerelemente-Arrays ist das Erstellen neuer Steuerelemente zur Laufzeit. Es ist zum Beispiel kein Problem, eine Anwendung zu programmieren, die auf Benutzerwunsch neue Schaltflächen erzeugt. Die Beispielanwendung BEFARRAY.FRM ermöglicht das Erzeugen beliebiger Befehlsschaltflächen, die zwei verschiedene Funktionen haben können:

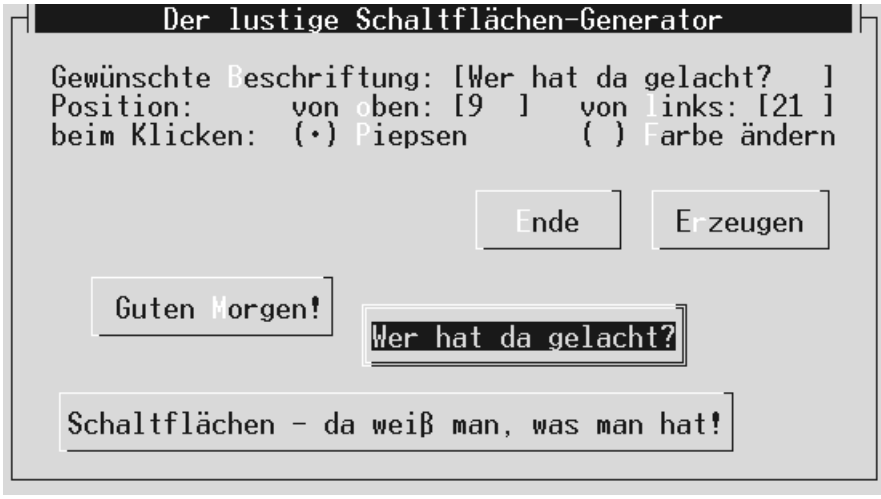


Abbildung 7–20: Der „lustige Schaltflächengenerator“ mit drei selbsterzeugten Schaltflächen

Das Listing hierzu ist wesentlich einfacher, als es zunächst den Anschein hat. In der Form ist neben den im oberen Bereich sichtbaren Steuerelementen noch eine unsichtbare Befehlsschaltfläche (*CtlName* = Array, *Index* = 0) enthalten, die quasi als Wurzel aller neu erzeugten Schaltflächen fungiert:

```
DIM SHARED Anzahl AS INTEGER   ' für die Anzahl der geladenen Schaltflächen
REM $INCLUDE: 'CONSTANT.BI'

SUB Array_Click (Index AS INTEGER)     ' Click-Prozedur für die neuen Schaltflächen
   IF Array(Index).Tag = "PIEP" THEN   ' Piepsen
      BEEP
   ELSE
      Array(Index).BackColor = (Array(Index).BackColor + 1) MOD 16   ' Farbe ändern
   END IF
END SUB
```

```
SUB Ende_Click ()
   SYSTEM
END SUB
```

```
SUB Erzeugen_Click ()
   IF Anzahl > 0 THEN       ' Array(0) ist schon da (im Form-Designer erzeugt);
      LOAD Array(Anzahl)    ' wenn aber ein höherer Index gebraucht wird: neues
   END IF                   ' Element mit LOAD erzeugen

   ' Eigenschaften gemäß den Einträgen einstellen
   Array(Anzahl).Top = VAL(VonOben.Text): Array(Anzahl).Left = VAL(VonLinks.Text)
   Array(Anzahl).Height = 3: Array(Anzahl).Width = LEN(Beschriftung.Text) + 2
   Array(Anzahl).Caption = Beschriftung.Text
   IF Piepsen.Value THEN Array(Anzahl).Tag = "PIEP" ELSE Array(Anzahl).Tag = "FARBE"
   END IF
   Array(Anzahl).Visible = TRUE  ' Neues Element sichtbar machen
   Anzahl = Anzahl + 1           ' Zähler erhöhen
END SUB
```

Listing 7–4: BEFARRAY.FRM

Wenn man im „lustigen Schaltflächen-Generator“ auf „Erzeugen“ klickt, wird das Array „Array“, für das im Form-Designer zunächst nur ein Element vorgesehen wurde, um ein Element vergrößert (außer beim ersten „Erzeugen“, denn da kann das im Form-Designer erstellte Element Nr. 0 verwendet werden). Dann werden die Eigenschaften des neuen Elementes gemäß den Einträgen in den Steuerelementfeldern vorgenommen, und schließlich wird das Element sichtbar gemacht.

Beim LOAD-Befehl erhält das neu erzeugte Element zunächst automatisch alle Eigenschaften des im Form-Designer erzeugten Elements Nr. 0; daher ist es auch so lange versteckt, bis die Eigenschaft *Visible* auf TRUE gesetzt wird.

Man kann Steuerelemente-Arrays natürlich auch für ernsthaftere Zwecke einsetzen. Zum Beispiel kommt es häufig vor, daß Menüs zur Laufzeit um gewisse Einträge ergänzt werden müssen. Die zur Laufzeit mit LOAD erzeugten Elemente sind völlig gleichberechtigt mit denen, die schon im Form-Designer erstellt wurden; der Benutzer Ihres Programms kann beide Typen nicht voneinander unterscheiden. Die mit LOAD erzeugten Steuerelemente können zur Laufzeit auch mit UNLOAD wieder entfernt werden; versuchen Sie das allerdings mit einem im Form-Designer erstellten Steuerelement, erhalten Sie die Meldung *Zur Entwurfszeit erzeugte Steuerelemente können nicht entfernt werden* (Nr. 362).

7.10 Unsichtbare Steuerelemente

Im Abschnitt 7.7 habe ich Steuerelemente erwähnt, die „versteckt“ werden, um eine Form mehrfach benutzen zu können. Es kommt aber auch vor, daß Steuerelemente niemals angezeigt werden, obwohl das Programm regen Gebrauch davon macht. Jedes Steuerelement hat ja seinen eigenen Datenbereich, so daß Steuerelemente auch als Variablen zweckentfremdet werden können. Man setzt einfach ihre *Visible*-Eigenschaft auf FALSE (notfalls erzeugt man eine ganz unsichtbare Form, die nur solche „Dummy-Steuerelemente“ enthält).

Listenfelder zu Arrays...

So läßt sich zum Beispiel ein Listenfeld vorzüglich anstelle eines Arrays zum Speichern von Strings einsetzen. Man kann dann ohne großen Aufwand Elemente an eine bestimmte Stelle einfügen (indem man die gewünschte Position bei ADDITEM angibt) und Elemente an einer bestimmten Position streichen

(mit REMOVEITEM). Diese Routinen sind deutlich schneller als eigene Konstrukte.

... und Laufwerkslisten zu Systemspionen

Auch das Laufwerkslistenfeld ist nicht ohne. Es ermöglicht Ihnen – auch, wenn es die ganze Zeit über versteckt bleibt und der Benutzer überhaupt nichts damit zu schaffen hat –, festzustellen, welche Laufwerke im System installiert sind, zum Beispiel so:

```
DIM LaufwerkInstalliert(1 TO 26) AS INTEGER ' 1 = A:, 26 = Z:
FOR i% = 0 TO LaufwerksListe.ListCount
   LaufwerkInstalliert(ASC(LaufwerksListe.List(i%))   63) = TRUE
NEXT
```

Dadurch können Sie den besonders lästigen Zwischenfall verhindern, der eintritt, wenn der Benutzer auf das Laufwerk B: zugreift, obwohl es nicht installiert ist (DOS meldet sich, ungeachtet der schönsten Bildschirmmaske, mit seinem penetranten „Diskette in Laufwerk B: einlegen und eine Taste drücken“). Prüfen Sie einfach, ob das Laufwerk B: in der Laufwerksliste vorkommt.

Wie man ein Dateilistenfeld „verborgen“ einsetzt, wurde schon im Rahmen des Programmierbeispiels am Anfang dieses Kapitels erläutert.

7.11 Multitasking

Um es gleich vorweg zu sagen: Ein echtes „Multitasking“ ist natürlich mit VBDOS nicht möglich. Trotzdem ergeben sich verblüffende Effekte, oft sogar, ohne daß man es bei der Programmentwicklung geplant hätte.

Nehmen wir zum Beispiel an, daß Sie unserem Beispielprogramm aus Abschnitt 7.1 („Ersetzen in mehreren Dateien“) noch eine Schaltfläche „Einstellungen“ spendieren, die eine zweite Form öffnet, auf der einige Standard-Einstellungen für das Programm vorgenommen werden können (z. B. Farbeinstellung, Protokolldatei). Wenn Sie dann noch während der zeitintensiven Vorgänge (also in der Schleife in *Start_Click* und eventuell sogar in *ErsetzeInDatei*) häufige Aufrufe an die Funktion DOEVENTS einbauen (`Dummy = DOEVENTS()`), erreichen Sie ein „Pseudo-Multitasking“. Der Benutzer kann den Cursor zwischen den Feldern in der Einstellungsform umherbewegen und die Feldinhalte verändern; wenn Sie mit einem Ereignis, das dabei auftritt, umfangreichere Aktionen verbinden, können auch diese ausgeführt werden. Dann allerdings

liegt der Ersetzen-Prozeß so lange auf Eis. Solange der Benutzer sich ruhig verhält, wird der Ersetzen-Prozeß weiter ausgeführt.

Die Aufrufhäufigkeit von DOEVENTS muß für das „Pseudo-Multitasking“ hoch sein (2 Aufrufe pro Sekunde sollten als absolute Untergrenze gelten), um den Benutzer nicht warten zu lassen. Da VBDOS einfache Mausaktionen (Klicken, aber nicht Drag-and-Drop) und alle Tastaturaktionen puffert, kommen bei einem DOEVENTS-Aufruf alle Ereignisse zum Tragen, die seit dem letzten DOEVENTS in die Warteschlange gereiht wurden. (Eine Sonderregelung gilt für Timer-Ereignisse; siehe Referenzteil).

Sie sollten die DOEVENTS-Aufrufe so plazieren, daß sie häufig ausgeführt werden. Gefährlich kann es jedoch sein, einen DOEVENTS-Aufruf an einer empfindlichen Stelle – etwa während gerade eine Datei beschrieben wird – auszuführen; bedenken Sie stets, daß ein einziges DOEVENTS jede beliebige Funktion Ihres Programmes, die ereignisgesteuert ist, auslösen kann. Es muß verhindert werden, daß der Benutzer z. B. in einer solch empfindlichen Phase durch Klicken auf „Beenden“ einfach einen SYSTEM-Befehl auslöst.

Neben dem regelmäßigen DOEVENTS-Aufruf erfordert das „Pseudo-Multitasking“ auch, daß Formen nicht gebunden angezeigt werden. Während der Anzeige gebundener Formen kann keine andere Aktion ablaufen.

Reentrante Prozesse

Obwohl solches „Pseudo-Multitasking“ ein sehr interessanter und benutzerfreundlicher Effekt ist, gibt es auch Nebenwirkungen, die man sorgfältig voraussehen und vermeiden muß.

Zum Beispiel wäre es dem Benutzer in unserem mit DOEVENTS modifizierten Beispielprogramm möglich, durch Klicken auf den Start-Knopf einen zweiten Ersetzungsprozeß auszulösen, während bereits einer aktiv ist. In seltenen Fällen kann das wünschenswert sein; hier aber würde solches Vorgehen nur Verwirrung stiften. Daher sollte man verhindern, daß ein Prozeß gestartet wird, wenn er bereits („im Hintergrund“) läuft.

Um das zu bewerkstelligen, können Sie gleich zu Anfang der `Start_Click`-Prozedur den Befehl `Start.Enabled = FALSE` ausführen lassen, so daß der Start-Knopf nicht mehr betätigt werden kann.

Eine andere Möglichkeit wäre es, eine globale Variable (oder eine geeignete Eigenschaft) zu setzen, an der alle anderen Programmteile erkennen können, daß gerade ein Ersetzen-Vorgang läuft.

So wäre es zum Beispiel denkbar, am Anfang der Start_Click-Prozedur den Befehl Start.Tag = "LÄUFT" einzubauen, und vor dem Ende der Prozedur die Eigenschaft wieder zurückzusetzen: Start.Tag = "". Dann könnten auch andere Objekte feststellen, ob gerade ein Ersetzungsvorgang läuft; zum Beispiel könnte in der Beenden_Click-Prozedur folgendes Codefragment untergebracht werden:

```
IF Start.Tag = "LÄUFT" THEN
   IF MSGBOX("Wollen Sie den Vorgang abbrechen?", MB_YESNO, "") = IDYES THEN
      ' MB_YESNO, IDYES sind Konstanten aus CONSTANT.BI
      Start.Tag = "ABBRUCH"
   END IF
ELSE
   SYSTEM
END IF
```

In der Start_Click-Prozedur würden Sie dann regelmäßig prüfen, ob sich die *Start.Tag*-Eigenschaft inzwischen auf „ABBRUCH“ geändert hat, und falls ja, den Vorgang geordnet abbrechen.

Andere unerwünschte Nebenwirkungen

Außerdem muß sichergestellt werden, daß der Benutzer keine Feldeinstellungen verändert, die für den laufenden Prozeß benötigt werden.

In unserem speziellen Fall lauert die Gefahr in allen Feldern auf der Form: Wenn der Benutzer während eines laufenden Ersetzungsvorgangs plötzlich das Feld „Ersetzen“ ändert, werden von nun an alle Dateien mit dem neuen Ersatztext bearbeitet. Die Verwirrung wird etwas dadurch begrenzt, daß der ErsetzeInDatei-Prozedur der Wert von *Ersetzen.Text* „by value“ übergeben wird. Daher wird bei Aufruf der Prozedur eine temporäre Variable erzeugt, die sich nicht mehr verändert, auch wenn der Benutzer während des Ablaufs der Prozedur am „Ersetzen“-Feld herumwurstelt. Hätten wir stattdessen unsere ErsetzeInDatei-Prozedur so konzipiert, daß sie direkt den Feldinhalt von „Ersetzen“ ausliest, so könnte es bei ungünstigem Timing vorkommen, daß der Suchtext in ein- und derselben Datei mal durch den einen, mal durch den anderen Ersatztext ersetzt wird.

Das gilt genauso für die Felder Suchen und GrossKlein; bei Änderung der Dateibezeichnung wären sogar schwereren Fehlern Tür und Tor geöffnet, denn wenn der Benutzer mitten im Ersetzungsprozeß die Dateiliste verändert (verkürzt), wird möglicherweise auf ein nicht vorhandenes Element zugegriffen.

Solche Probleme können Sie vermeiden, indem Sie

- „von Hand“ einen Schutz einbauen, der (z. B. in der `Suchen_KeyPress`-Prozedur) jeden Tastendruck annuliert, solange ein Ersetzungsvorgang läuft;
- einfach die ganze Form oder die Steuerelemente, deren Änderung unerwünscht ist, deaktivieren (*Enabled* = FALSE);
- die Inhalte der betroffenen Steuerelemente zu Anfang eines Ersetzungsprozesses in Variablen kopieren, so daß eine Änderung der Steuerelemente den laufenden Prozeß nicht mehr beeinträchtigt.

Welches der drei Verfahren sinnvoll ist, hängt ganz vom Aufbau Ihres Programms ab. Die letztgenannte Methode läßt dem Benutzer am meisten Freiheit, ist aber auch mit am aufwendigsten in der Realisation.

Formen können nur eine Instanz haben

Bei allen Experimenten mit dem Aufrufen mehrerer Prozesse müssen Sie eins im Auge behalten: Eine Form kann nie in zwei „Instanzen“, also in doppelter Ausführung, quasi als Original und Kopie, auf dem Bildschirm erscheinen. Das bedeutet, daß insbesondere bei der Verwendung von Hilfsformen wie der „Bitte warten“-Anzeige mit Prozentbalken aus Kapitel 21 Vorsicht geboten ist.

Wenn Ihr Programm zwei länger dauernde Prozesse ermöglicht, die beide durch einen entsprechenden Start-Knopf aus der Hauptform gestartet werden, und Sie „Pseudo-Multitasking“ durch DOEVENTS-Aufrufe betreiben, können beide Prozesse nicht einfach die Warte-Form verwenden. Der Benutzer könnte nämlich theoretisch den zweiten Prozeß starten, während der erste noch läuft, und dieser würde die Form dem ersten „wegschnappen“; nach Beendigung des zweiten Prozesses würde der erste zwar fortgesetzt, müßte aber zuvor die Warte-Form mit seinen Werten wiederherstellen – und woher soll er wissen, daß sie ihm inzwischen entwendet worden war?

Diese Information ließe sich zwar z. B. in der *Tag*-Eigenschaft der Warteform speichern, aber nichtsdestotrotz wäre die Warteanzeige von Prozeß 1 unsichtbar, während Prozeß 2 läuft, und das ist nicht erwünscht.

7.12 Interaktion

Im Abschnitt über „reentrante Prozesse“ sind wir bei einem sehr wichtigen Thema angelangt: Der Verständigung zwischen Objekten und Prozessen.

Einfache Programme wie unser Ersetzen-Beispiel haben einen relativ strengen Ablauf: Man kann ziemlich genau vorhersagen, was der Benutzer in welcher

Reihenfolge tun wird, und es ist auch legitim, ihn in seinen Wahlmöglichkeiten etwas einzuschränken.

Wenn Sie aber größere Systeme mit vielen Formen entwickeln, steigt die Anzahl der möglichen Aktionen und Ausführungsreihenfolgen stark an, und als guter Programmierer werden Sie Ihr Programm so konzipieren, daß der Benutzer alles tun kann, was sinnvoll ist, ohne sich dabei an vorgegebene Reihenfolgen oder Strukturen halten zu müssen. Er wird Formen öffnen und schließen (mit Befehlsschaltflächen auf der Startform oder durch das Systemmenüfeld); er wird Formen auf Vollbildanzeige vergrößern oder zum Symbol werden lassen, Feldeinstellungen ändern und Aktionen auslösen, die ihm sinnvoll erscheinen.

Wenn Sie durch den Aufruf der DOEVENTS-Funktion „Multitasking“ realisieren, können theoretisch unbegrenzt viele Prozesse gleichzeitig laufen. In Wahrheit wird natürlich immer nur der zuletzt aufgerufene Prozeß ausgeführt, aber nach dessen Beendigung wird der jeweils unterbrochene Prozeß fortgesetzt.

Datenübertragung zwischen Steuerelementen

Es wird immer wieder Stellen im Programm geben, an denen es wichtig ist, zu wissen, ob ein anderer Prozeß möglicherweise gerade läuft oder schon abgeschlossen ist. Solche Informationen können über globale Variablen oder Eigenschaften der betroffenen Steuerelemente verteilt werden – die ansonsten ungenutzte *Tag*-Eigenschaft bietet sich an.

Diese Art der „Datenübertragung“ muß auch angewandt werden, wenn ein laufender Prozeß auf Benutzeraktionen reagieren soll. Angenommen, Sie haben auf der Form eine „Abbruch“-Schaltfläche installiert, mit der der Prozeß, der durch Klick auf „Start“ ausgelöst wurde, beendet werden soll. Nun wird es in den meisten Fällen nicht zweckmäßig sein, bei Klick auf „Abbruch“ einfach das gesamte Programm mit SYSTEM abzubrechen; vielmehr müßte dem laufenden Prozeß die Information übermittelt werden, daß die Schaltfläche betätigt wurde.

Eine mögliche Lösung dieses Problems habe ich schon im Abschnitt über „reentrante Prozesse“ gezeigt: Der laufende Vorgang prüft ständig den Wert einer bestimmten Eigenschaft – z. B. der *Tag*-Eigenschaft der „Abbrechen“-Schaltfläche – und erkennt daran, ob der Benutzer einen Abbruch veranlaßt hat.

Gegenseitige Aufrufe

Es ist möglich, Ereignisprozeduren direkt (wie normale Prozeduren) aufzurufen. Dabei wird dann allerdings nur der Code ausgeführt, der wirklich in der Er-

eignisprozedur steht, und nicht das, was VBDOS üblicherweise tut, wenn dieses Ereignis eintritt. Die Zeile

```
Textfeld_KeyPress 65
```

würde zwar die `KeyPress`-Prozedur des Textfeldes aufrufen und ihr mitteilen, ein großes A sei gedrückt worden, aber dieses A würde keinesfalls im Textfeld erscheinen.

Indirekte Aufrufe zwischen Ereignisprozeduren hingegen kommen recht häufig vor. Wenn zum Beispiel der Befehl `Text1.Text = "Standard"` ausgeführt wird, tritt ein *Text1_Change*-Ereignis ein (es sei denn, in Text1.Text hätte schon vorher „Standard“ gestanden). Ändert man auf ähnliche Weise „gewaltsam“ die *Value*-Eigenschaft einer Befehlsschaltfläche auf TRUE, tritt für diese ein *Click*-Ereignis ein.

Gegenseitige Aufrufe werden oft benutzt, um eine zusammengehörige Gruppe von Steuerelementen zu synchronisieren. Ein Textfeld zum Beispiel ist mit der Maus nicht bedienbar; sollen nur Zahlen eingegeben werden, bietet es sich an, das Feld durch Beiordnung einer Bildlaufleiste „mausfreundlich“ zu machen:

Abbildung 7–21: Textfeld mit beigeordneter Bildlaufleiste

Dann würde man die Ereignisprozeduren so formulieren:

```
SUB GehaltWert_LostFocus
   GehaltBalken.Value = VAL(GehaltWert.Text)
END SUB

SUB GehaltBalken_Change
   GehaltWert.Text = FORMAT$(GehaltBalken.Value)
END SUB
```

Dadurch kann der Wert im Gehaltsfeld sowohl durch direkte Eingabe einer Zahl als auch durch den Balken manipuliert werden. (Ich habe das Ereignis `Gehalt-Wert_LostFocus` und nicht *Change* gewählt, damit der Balken bei der Eingabe von „120“ nicht erst auf die 1, dann auf die 12 und schließlich auf die 120 springt. Das Beispiel setzt natürlich die angemessene Einstellung der *Min*- und *Max*-Eigenschaft der Bildlaufleiste voraus.)

Endlose Rekursionen

Dieser indirekte, oft unbedachte gegenseitige Aufruf von Ereignisprozeduren kann auch unangenehme Folgen haben. Im folgenden Beispiel...

```
SUB Text1_Change()
   Text2.Text = " " + Text1.Text
END SUB
SUB Text2_Change()
   Text1.Text = "*" + Text2.Text
END SUB
```

...wird bei der kleinsten Änderung an Text1 oder Text2 eine endlose Rekursion ausgelöst, die schließlich den Stack überlaufen läßt und zu einer Meldung „Stapelspeicher voll“ o.ä. führt. Ganz klar: Wenn Sie in Text1 eine Änderung vornehmen, wird der Text mit einem führenden Leerzeichen in *Text2.Text* geschrieben. Dadurch wird `Text2_Change` ausgelöst, was wiederum Text1 verändert usw.

Zum Vergleich: Dieses Beispiel...

```
SUB Text1_Change()
   Text2.Text = UCASE$(Text1.Text)
END SUB
SUB Text2_Change()
   Text1.Text = LCASE$(Text2.Text)
END SUB
```

wird keine endlose Rekursion auslösen, weil spätestens nach zwei gegenseitigen Aufrufen in Text1 ein klein- und in Text2 ein großgeschriebener Text steht. Dann wird aber durch den Befehl `Text1.Text = LCASE$(Text2.Text)` Text1 nicht mehr verändert, und es tritt auch kein *Change*-Ereignis mehr ein.

(Microsoft behauptet in der VBDOS-Dokumentation, daß auch dieses Beispiel zu einer endlosen Rekursion führe, weil jede Zuweisung an die *Text*-Eigenschaft die *Change*-Prozedur aufrufe. In Wahrheit wird die *Change*-Prozedur aber nur aufgerufen, wenn sich durch die Zuweisung an der Eigenschaft etwas ändert.)

7.13 Grenzen der ereignisgesteuerten Programmierung

Sie haben jetzt einen relativ weiten Einblick in die Möglichkeiten der ereignisgesteuerten Programmierung mit Visual BASIC für DOS erhalten. Es ist an der Zeit, auch einmal ein paar Worte über die Grenzen dieser Einrichtung zu verlieren.

Kein Grafikmodus

Sie können die ereignisgesteuerte Programmierung nur verwenden, wenn sich das System im Textmodus befindet. Sobald Sie (mit dem SCREEN-Befehl) in einen grafiktauglichen Bildschirmmodus umschalten, können Sie keine Formen und Steuerelemente mehr anzeigen. Schalten Sie in einen Grafikmodus, solange noch Formen am Bildschirm sichtbar sind, wird eine Fehlermeldung erzeugt.

Sie können zwar problemlos zwischen beiden Modi hin- und herschalten (vgl. z. B. die Anwendung CHRTDEMO, die im Lieferumfang von Visual BASIC für DOS enthalten ist), aber Sie können nie Formen im Grafikmodus anzeigen.

Falls Sie also, was bei besseren Grafik- und Zeichenprogrammen meist unerläßlich ist, Benutzereingaben im Grafikmodus verarbeiten wollen, steht Ihnen die ereignisgesteuerte Programmierung nicht zur Verfügung.

Nur für dialogintensive Programme sinnvoll

Die Verwendung der ereignisgesteuerten Programmierung vereinfacht Dialoge mit dem Benutzer wesentlich. Wenn es sich jedoch um Programme handelt, die relativ wenig mit dem Benutzer kommunizieren, sollte man zweimal nachdenken, bevor man mit Kanonen auf Spatzen schießt. Denn:

- Das ereignisgesteuerte Programmieren erfordert aufgrund der vielen Funktionen, die es selbst zur Verfügung stellt, viel Speicher. Erinnern Sie sich noch an das Primitiv-Beispielprogramm aus Kapitel 3? Das zugehörige EXE-Programm ist in der ereignisgesteuerten Version etwa 120 KB länger als in der alten, wenn man nicht mit Runtime-Modulen arbeitet. (Kapitel 23 zeigt Möglichkeiten, die Größe der EXE-Programme zu reduzieren, wenn man Teile der ereignisgesteuerten Programmierung nicht verwendet.)
- Programme, die kaum mit Benutzereingaben zu tun haben, werden durch Verwendung des ereignisgesteuerten Programmierens möglicherweise unübersichtlich. Üblicherweise sind ereignisgesteuerte Programme zwar leichter zu lesen als gewöhnliche, aber wenn die Ereignis-Struktur dem Programm aufgezwängt werden muß, kann das auch von Nachteil sein.

Speicherverwaltung und Variablen 8

8.1 Boolesche Variablen

Häufig werden in einem Programm sogenannte Flags benötigt, Variablen, die entweder „ein“ oder „aus“ sein können. Andere Sprachen stellen dafür einen speziellen Datentyp zur Verfügung, BOOLEAN, der entweder TRUE oder FALSE sein kann. Wenn BASIC auch diesen Typ nicht kennt, ist es empfehlenswert, sich diese Praxis anzugewöhnen: Definieren Sie am Anfang jedes Programms die Konstanten TRUE mit dem Wert –1 und FALSE mit 0. Verwenden Sie für Flag-Variablen den Typ INTEGER, obwohl das eigentlich Verschwendung ist, denn ein INTEGER könnte 16 Flags enthalten. Dennoch sollte hier die Übersichtlichkeit vorgehen.

Die Werte –1 und 0 werden auch vom Compiler verwendet, um das Ergebnis von Ausdrücken zu repräsentieren. Man kann zum Beispiel schreiben:

```
Ungleich = Zahl1 <> Zahl2
```

Dann wird die Variable *Ungleich* TRUE, wenn der Ausdruck (in diesem Fall die Behauptung `Zahl1 <> Zahl2`) wahr ist, und FALSE, wenn er falsch ist. Außerdem können auch Konstrukte wie `IF Ungleich THEN...` oder `LOOP UNTIL Ungleich` benutzt werden, wenn *Ungleich* nur die Werte TRUE oder FALSE annimmt. Durch die Verwendung der Konstanten wird Ihr Programm sehr viel verständlicher (zum Thema TRUE und FALSE siehe auch „Logische und bitweise Operatoren“ im Kapitel 2).

Ausnahmen sollten Sie nur in Notfällen machen. Wenn Sie zum Beispiel ein Array mit 1.000 Elementen definieren wollen und jedes Element darin 10 Flags enthält, dann können Sie schon auf die folgende, weniger übersichtliche Methode zurückgreifen, da sie in diesem Falle 18 KB Speicher spart.

Die Routinen auf der folgenden Seite ermöglichen es Ihnen, ein einzelnes Bit innerhalb einer INTEGER-Zahl zu setzen, zu löschen und abzufragen. So können in einer INTEGER-Zahl 16 Flags untergebracht werden. Dabei müssen Sie jeweils die betreffende Zahl und das Bit, mit dem Sie arbeiten wollen, angeben. Das kleinste Bit hat die Nummer 1, das größte die Nummer 16. Wegen dieses sechzehnten Bits müssen die Routinen *SetBit* und *DelBit* einige Kopfstände machen, da es in BASIC den Wert –32.768 (anstatt, konsequenterweise, 32.768) hat, um negative Integers zu ermöglichen. In *CheckBit* ist der Umstand hingegen nicht notwendig, weil BASIC, wenn es nur um das Abfragen geht, toleranter ist.

```
SUB SetBit (Zahl AS INTEGER, Bit AS INTEGER)
   SELECT CASE Bit
   CASE 1 TO 15: Zahl = Zahl OR 2^(Bit   1)
   CASE 16:      Zahl = Zahl OR  32768
   END SELECT
END SUB
```

```
SUB DelBit (Zahl AS INTEGER, Bit AS INTEGER)
   SELECT CASE Bit
   CASE 1 TO 15: Zahl = Zahl AND ( 1 - 2^(Bit   1))
   CASE 16:      Zahl = Zahl AND 32767
   END SELECT
END SUB
```

```
FUNCTION CheckBit (Zahl AS INTEGER, Bit AS INTEGER) AS INTEGER
   CheckBit% = (Zahl AND 2^(Bit   1) <> 0
END FUNCTION
```

Listing 8–1: BITMAN.BAS

Sie werden diese Routinen zum Teil auch gebrauchen können, wenn Sie mit Interrupts arbeiten (vgl. Kapitel 22). Sie können diese Routinen umschreiben, so daß die Variable *Zahl* ein LONG-Integer wird; dann haben Sie 32 Bits zur Verfügung. Ersetzen Sie jede 15 durch eine 31, jede 16 durch eine 32, 32.767 durch $2^{31}-1$ und –32.768 durch -2^{31}. Eine sinnvolle Erweiterung wäre auch, die Zweierpotenzen in einem Array zu speichern und nicht jedes Mal neu zu berechnen – das spart Zeit.

8.2 Strings mit fester und variabler Länge

Im Namen ist es nur ein kleiner Unterschied, in der Verwaltung handelt es sich um zwei völlig verschiedene Konzepte. Während der String mit fester Länge sich in die Gesellschaft aller anderen Datentypen (numerische und selbstdefinierte Typen) einreiht, eben weil er eine feste Länge hat, fällt sein Kollege mit variabler Länge völlig aus der Rolle.

Für alle „normalen“ Datentypen muß der Compiler nur speichern, wo sie sich im Speicher befinden (die Startadresse) und um welchen Typ es sich handelt (zum Beispiel INTEGER oder STRING*80; also ein String mit fester Länge 80). Bei Strings mit variabler Länge kommt jedoch noch die Information hinzu, wie lang sie sind, da sich hier die Länge zur Laufzeit verändern kann. In der Interpreter-Version von BASIC wurde für diese Längenangabe nur ein Byte reserviert, also konnte ein String maximal 255 Zeichen lang sein. VBDOS benutzt

allerdings zwei Bytes für die Längeninformation und erlaubt Strings mit einer Länge von bis zu 32.767 Zeichen.

Durch diesen zusätzlichen Verwaltungsaufwand sind Strings mit variabler Länge langsamer als solche mit fester Länge; zugleich sind sie aber sparsam mit Speicher, da sie nur so viele Bytes im Speicher belegen, wie wirklich gebraucht werden, während Strings mit fester Länge immer ihre volle Länge benötigen. Eine Ausnahme sind ISAM-Datenbanken in der professionellen Ausgabe: Hier können Sie nur Strings mit fester Länge speichern, aber diese werden komprimiert und benötigen auf der Platte nur so viel Platz, wie ihr Inhalt lang ist.

Bei der Arbeit mit Strings von fester Länge sind einige Punkte zu berücksichtigen:

- Strings mit fester Länge können nicht als solche an Subroutinen übergeben werden; sie werden dafür automatisch in Strings mit variabler Länge umgewandelt, die aber exakt die Länge des ursprünglichen Strings (und nicht nur die seines Inhalts) haben. Siehe dazu den Eintrag zu CALL im Referenzteil.
- Nachdem sie mit DIM oder COMMON oder einem anderen Deklarationsbefehl initialisiert wurden, bestehen Strings mit fester Länge aus NUL, aus Zeichen mit dem ASCII-Wert 0. Wenn *Text* ein String mit beliebiger fester Länge ist, wird die Abfrage `IF Text = ""` bzw. `IF LEN(Text) = 0` natürlich niemals TRUE ergeben.
- Sobald einem String mit fester Länge irgendetwas – selbst wenn es nur ein Leerstring ("") ist – zugewiesen wird, werden alle verbleibenden Zeichen des Strings mit Leerzeichen (ASCII 32) aufgefüllt. Von der ersten Zuweisung an kann also die Funktion RTRIM$ benutzt werden, um überflüssige Zeichen am Ende des Strings zu entfernen, nicht jedoch vorher, da RTRIM$ auf das Zeichen ASCII 0 nicht reagiert. Das folgende Beispiel veranschaulicht diese Tatsache:

  ```
  DIM Text AS STRING * 10
  PRINT LEN(Text); ASC(Text); LEN(RTRIM$(Text))
  Text = ""
  PRINT LEN(Text); ASC(Text); LEN(RTRIM$(Text))
  ```

  Es gibt auf dem Bildschirm aus:

  ```
  10  0  10
  10  32  0
  ```

- Werden einem String mit fester Länge mehr Zeichen zugewiesen, als er aufnehmen kann, nimmt er nur so viele auf, wie hineinpassen – der Rest geht verloren, aber es wird kein Fehler erzeugt.

Details zur Speicherung von Strings mit variabler Länge

Strings mit variabler Länge werden in VBDOS grundsätzlich als „Far Strings“ gespeichert. Far Strings werden verwaltet, indem im Near-Speicher (im DGROUP-Segment, das auch alle einfachen Variablen und den Stack speichert) ein 4 Byte langer „Stringdeskriptor“ erzeugt wird, der auf zwei Variablen im Far-Speicher zeigt, die wiederum den String lokalisieren. Jeder String benötigt dadurch 4 Bytes im Near- und 6 Bytes im Far-Speicher an Verwaltungsinformation, zuzüglich seiner eigentlichen Länge.

Der Near-Speicher umfaßt ein Segment (64 KB); die Segmente im Far-Speicher stehen auch nicht beliebig zur Verfügung, sondern werden (laut offiziellen Angaben, vgl. aber Programmausgabe weiter unten) wie folgt verteilt:

- Alle im Hauptprogramm und in benannten COMMON-Anweisungen deklarierten Strings erhalten ein Segment.
- Alle in unbenannten COMMON-Blocks deklarierten Strings erhalten ein zweites Segment.
- Alle automatischen und statischen Strings, die in Prozeduren deklariert werden, erhalten ein drittes Segment.
- Jede Prozedur erhält ein eigenes Segment für ihre lokalen String-Arrays.
- Jede Form speichert ihre Eigenschaften-Variablen in einem eigenen Segment.

In jedem Segment werden 64 Bytes an Basis-Verwaltungsinformation sowie 6 Bytes pro String belegt.

Wenn Sie die Speicherbelegung durch Verwaltungsinformation bedenken, ist es insbesondere bei kurzen Strings oft lohnend, Strings mit fester Länge zu verwenden. Beispiel: Ein Datenfeld mit Dateinamen. Unter DOS hat ein Dateiname maximal 12 Zeichen (inklusive Punkt). Selbst wenn die durchschnittliche Länge eines Dateinamens nur 7 Zeichen beträgt, würde ein Feld mit 100 Dateinamen in variabler Form 1.000 Byte Verwaltungsinformation und 700 Byte Stringdaten, also 1.700 Bytes, benötigen. Die „verschwenderischen“ Strings mit fester Länge, die für jeden String die volle Länge belegen, könnten dieses Rennen bei einem Speicherbedarf von insgesamt 1.200 Bytes klar für sich entscheiden.

Far String-Speicher für Großverbraucher

Wenn Sie besonders viel String-Speicherplatz benötigen, können Sie also zunächst das String-Segment des Codemoduls auffüllen und danach ein Segment mit Strings, die Sie per COMMON deklariert haben. Dies ist der einfachste Weg, bis zu 128 KB an Stringdaten zu speichern. Reicht das nicht aus, können

Sie eine Prozedur aufrufen. Diese benutzt dann das Segment für Prozedur-Stringdaten, das macht zusammen mit den 64 KB aus dem COMMON-Block und 64 KB aus dem Hauptprogramm (die der Prozedur, zum Beispiel über SHARED, ja auch zur Verfügung stehen können) schon 192 KB.

Das folgende Beispiel veranschaulicht die Segmentzuteilung an Far Strings:

```
REM $DYNAMIC
' das erste Segment für den COMMON-Block: (Wir brauchen zwei Strings, um es zu fül-
' len, denn ein String darf nur 32.767 Zeichen haben, ein Segment hat 65.536 Byte!)
COMMON SHARED ErstesSegment1 AS STRING, ErstesSegment2 AS STRING

' das zweite Segment für's Hauptprogramm:
DIM SHARED ZweitesSegment1 AS STRING, ZweitesSegment2 AS STRING

' nun das SUB aufrufen
StringDemo

' Programmende
END
```

```
SUB StringDemo ()
   ' das dritte Segment für alle Prozeduren und Funktionen
   DIM DrittesSegment1 AS STRING, DrittesSegment2 AS STRING
   ' ein weiteres Segment für jede Prozedur, die String-Arrays hat
   DIM WeiteresSegment(1 TO 2) AS STRING
   ' nun alles vollschreiben
   MaxAlloc ErstesSegment1, "COMMON-Segment (1)"
   MaxAlloc ErstesSegment2, "COMMON-Segment (2)"
   MaxAlloc ZweitesSegment1, "Modul-Segment (1)"
   MaxAlloc ZweitesSegment2, "Modul-Segment (2)"
   MaxAlloc WeiteresSegment(1), "Prozedur-Array-Segment (1):"
   MaxAlloc WeiteresSegment(2), "Prozedur-Array-Segment (2)"
   MaxAlloc DrittesSegment1, "Prozedur-Stringsegment (1)"
   MaxAlloc DrittesSegment2, "Prozedur-Stringsegment (2)"

END SUB
```

```
' Setzt den übergebenen String auf die maximal mögliche Länge und gibt diese aus
SUB MaxAlloc (Zeichenfolge AS STRING, Bezeichnung AS STRING)
   ON LOCAL ERROR RESUME NEXT
   PRINT "Speicher für "; Bezeichnung; ": ";
   FOR i% = 1000 TO 31000 STEP 1000
      Zeichenfolge = SPACE$(i%)
      IF ERR THEN PRINT FORMAT$(i% - 1000); : EXIT FOR
   NEXT
   IF ERR = 0 THEN PRINT "31000";
   PRINT " temporäre Strings:"; FRE("")
END SUB
```

Listing 8–2: SEGFSTR.BAS

Bei 192 KB beginnen dann allerdings die Schwierigkeiten. Wem 192 KB Stringspeicherplatz nicht ausreichen, der kann nicht einfach aus der ersten Prozedur heraus eine zweite aufrufen, in der Hoffnung, weitere 64 KB zu erhalten. Alle Prozeduren teilen sich ja das Prozeduren-Stringsegment. Man müßte in diesem Falle zur letzten Möglichkeit greifen und in einer Prozedur ein String-Array deklarieren, dies dann an eine zweite Prozedur übergeben, die ihrerseits wieder ein String-Array vereinbart und so fort, denn für den „edleren“ Zweck der String-Arrays erhält jede Prozedur ein eigenes Segment.

Das Segment, das im Beispiel „DrittesSegment“ heißt, wird nicht ohne Grund als letztes gefüllt; es beherbergt nämlich auch den Speicher für temporäre Strings, und ein Befehl der Form `a$ = SPACE$(i%)` benötigt immer (kurzfristig) i% Bytes in diesem Bereich; wenn wir das Segment als erstes gefüllt hätten, wären alle zukünftigen Zuweisungen daran gescheitert. Allerdings scheinen auch andere String-Anweisungen diesen Speicher zu reduzieren, wie die Ausgabe dieses Programms zeigt:

```
Speicher für COMMON-Segment (1): 31000 temporäre Strings: 65388
Speicher für COMMON-Segment (2): 31000 temporäre Strings: 65366
Speicher für Modul-Segment (1): 31000 temporäre Strings: 65344
Speicher für Modul-Segment (2): 31000 temporäre Strings: 65322
Speicher für Prozedur-Array-Segment (1): 31000 temporäre Strings: 34288
Speicher für Prozedur-Array-Segment (2): 17000 temporäre Strings: 17254
Speicher für Prozedur-Stringsegment (1): 9000 temporäre Strings: 8220
Speicher für Prozedur-Stringsegment (2): 4000 temporäre Strings: 4186
```

Ganz deutlich erkennen Sie allerdings, daß mit dem Schrumpfen des Speichers für temporäre Strings auch die Möglichkeit großer Zuweisungen nicht mehr gegeben ist.

8.3 Statische und dynamische Arrays

Der wesentliche Unterschied zwischen statischen und dynamischen Feldern ist, daß der Speicherplatz für dynamische Felder erst belegt wird, wenn das Programm läuft, während für statische Felder schon beim Kompilieren Speicher reserviert wird.

Das hat zur Folge, daß dynamische Felder ungleich flexibler sind. Man kann sie während des Programmablaufs völlig löschen und den von ihnen belegten Speicherplatz für andere Daten benutzen, und man kann ihre Dimension ändern oder die Arrays genau in der benötigten Größe dimensionieren. All das ist mit statischen Feldern nicht möglich.

Welche Arrays werden nun statisch, welche dynamisch angelegt? Die folgende Übersicht versucht, Ordnung in das Durcheinander zu bringen:

Array-Typ	*statisch / dynamisch*
Impliziert deklarierte Arrays*	immer statisch
Arrays, die in einem COMMON-Befehl auftauchen, bevor sie dimensioniert werden	ohne Metabefehl: dynamisch wenn $STATIC aktiv: statisch wenn $DYNAMIC aktiv: dynamisch
Arrays in einer Prozedur (SUB/FUNCTION) mit automatischen Variablen	immer dynamisch
Arrays in einer Prozedur (STATIC SUB/ STATIC FUNCTION) mit statischen Variablen	ohne Metabefehl: statisch wenn $STATIC aktiv: statisch wenn $DYNAMIC aktiv: dynamisch
Sonstige Arrays, die mit numerischen oder symbolischen Konstanten dimensioniert werden	ohne Metabefehl: statisch wenn $STATIC aktiv: statisch wenn $DYNAMIC aktiv: dynamisch
Sonstige Arrays, die mit Variablen dimensioniert werden	immer dynamisch

Wie Sie sehen, können Sie die Metabefehle REM $STATIC und REM $DYNAMIC dazu verwenden, die Zuordnung zu beeinflussen.

Manchmal ist es sinnvoll, ein Array, das sonst als dynamisch angelegt worden wäre, per Metabefehl statisch werden zu lassen, da Verwaltungsroutinen für dynamische Speicherzuordnung unter Umständen bis zu 5 KB zusätzlich im EXE-File benötigen.

Auf der anderen Seite kann auch Interesse daran bestehen, für ein Array dynamische Speicherzuordnung zu erzwingen, um es mit ERASE wieder löschen oder mit REDIM verändern zu können (siehe „Dynamische Speicherverwaltung“ weiter unten).

Arrays aus Strings mit variabler Länge

Ein Array, das aus Strings mit variabler Länge besteht, darf maximal 64 KB umfassen und wird, egal ob statisch oder dynamisch, im Far-Speicher abgelegt. Für jedes Array-Element werden jedoch in DGROUP vier Bytes an Verwaltungsinformationen benötigt.

---

* „Impliziert deklariert“ bedeutet, daß das Array niemals mit DIM dimensioniert wurde. Es hat dann den Bereich 0 bis 10 (oder 1 bis 10; je nach OPTION BASE), und beim Kompilieren wird eine Warnung ausgegeben; das Programm wird jedoch fehlerlos laufen, wenn der o.g. Bereich nicht überschritten wird. Diese Toleranz ist dem Compiler eingebaut worden, um noch mit BASIC-Interpretern kompatibel zu sein.

Numerische Arrays

Unter diesen Terminus fallen hier nicht nur Arrays aus wirklich numerischen Daten, sondern alles, was eine feste Länge hat: Erstens die numerischen Datentypen, zweitens die Strings mit fester Länge und drittens sämtliche selbstdefinierten Datentypen.

Diese Arrays werden in jedem Falle im Far-Speicher abgelegt. Die Obergrenze ist auch hier 64 KB – mit einer Ausnahme: Den „Huge Arrays“. Um sie zu verwenden, muß man (beim Compiler oder bei VBDOS) den Switch /Ah angeben, der für die Huge Arrays („Riesengroße Felder“) zuständig ist. Huge Arrays dürfen in jedem Falle bis zu 128 KB groß sein; selbst diese Grenze können sie übersteigen, wenn die Größe der Elemente, aus denen das Feld besteht, eine Zweierpotenz ist. Ein selbstdefinierter Typ also, der aus zwei INTEGER-, einem LONG- und einem STRING * 8-Wert besteht, würde mit einer Gesamtlänge von 16 dieser Bedingung entsprechen. Arrays, die die Bedingung erfüllen, können so groß sein, daß sie den gesamten Far-Speicher auffüllen. Wer nicht lange nachrechnen möchte, kann mit LEN die Länge eines beliebigen Datentyps ermitteln.

Vorsicht aber mit den Huge Arrays: Nur dynamische Arrays können „Huge“ werden, und wenn man /Ah angibt, werden alle dynamischen Arrays mit „Huge“-Methoden verarbeitet, egal wie groß sie wirklich sind. Huge Arrays sind etwas langsamer als normale dynamische Arrays, ebenso wie normale dynamische Arrays etwas langsamer als statische sind.

Dynamische Speicherverwaltung mit REDIM

Bevor mit dem BASIC PDS 7.1 der REDIM PRESERVE-Befehl eingeführt wurde, kam sicher auch in Ihren Programmen häufig folgendes Konstrukt vor:

```
CONST MaxDateiNamen = 299
DIM DateiNamen(1 TO MaxDateiNamen AS STRING)
```

Man mußte sehr oft vorher festlegen, wieviel Daten maximal gespeichert werden konnten. Heute ist das ganz anders: Mit REDIM PRESERVE kann man immer so viel Speicher belegen, wie gerade notwendig ist. Darüber hinaus spart man eine Variable, die anzeigt, wie viele Elemente bereits gespeichert sind. Hierfür kann man jetzt UBOUND verwenden. Das Beispiel auf der folgenden Seite illustriert die alte und die verbesserte, neue Programmiertechnik, und auch die Routinen aus INHALT.BAS im Kapitel 22 machen regen Gebrauch von REDIM PRESERVE.

Alte Methode (mit statischem Datenfeld)

```
CONST MaxNamen = 999
DIM Namen(1 TO MaxNamen) AS STRING
DIM AnzahlNamen AS INTEGER
DIM Eingabe AS STRING
DO
   PRINT "Name oder Q für Ende: ";
   LINE INPUT Eingabe
   IF Eingabe = "Q" THEN EXIT DO
   AnzahlNamen = AnzahlNamen + 1
   Namen(AnzahlNamen) = Eingabe
   IF AnzahlNamen = MaxNamen THEN
      PRINT "Array voll!": EXIT DO
   END IF
LOOP
PRINT "Eingegebene Namen:"
FOR i% = 1 TO AnzahlNamen
   PRINT i%;"."; Namen(i%)
NEXT
```

REDIM PRESERVE-Methode

```
REM $DYNAMIC
DIM Namen(0 TO 0) AS STRING

DIM Eingabe AS STRING
DO
   PRINT "Name oder Q für Ende: ";
   LINE INPUT Eingabe
   IF Eingabe = "Q" THEN EXIT DO
   REDIM PRESERVE Namen(0 TO UBOUND ⮎
                  (Namen) + 1) AS STRING
   Namen(UBOUND(Namen)) = Eingabe
   ' "Array voll" gibt es hier nicht.
   ' höchstens "Speicher voll"
LOOP
PRINT "Eingegebene Namen:"
FOR i% = 1 TO UBOUND(Namen)
   PRINT i%;"."; Namen(i%)
NEXT
```

Listings 8–3, 8–4: REDIM PRESERVE

In zeitkritischen Applikationen sollte man das rechte Listing noch etwas erweitern, so daß es den REDIM-Befehl verwendet, um immer gleich eine größere Portion neuen Speicher zu reservieren – z. B. in 50-Elemente-Blöcken. Da REDIM PRESERVE immer einen zusammenhängenden Speicherbereich „freischaufeln“ muß, geht die blockweise Allokation meist schneller.

8.4 Statische und automatische Variablen

Spätestens hier wird es vielleicht Zeit, die Nebel um das Wort „statisch“ ein bißchen zu lüften: Zunächst gibt es den eben abgehandelten Unterschied zwischen statischen und dynamischen Feldern. In diesen Bereich gehört auch der Metabefehl $STATIC. Der gewöhnliche Befehl STATIC hingegen hat mit Prozeduren zu tun und gehört in den Bereich der statischen und automatischen Variablen, die nun behandelt werden sollen – ebenso wie der Zusatz STATIC bei der Prozedur oder Funktionsdefinition.

Wenn im Kopf einer Prozedur oder Funktion (SUB bzw. FUNCTION) kein STATIC angegeben wird und man auch den Befehl STATIC nicht verwendet, sind alle Variablen, die in der Prozedur mit DIM vereinbart und/oder benutzt werden, sogenannte automatische Variablen. Das bedeutet, daß sie alle bei jedem Aufruf der Prozedur mit 0 bzw. mit Leerstrings initialisiert und nach Beendigung der Prozedur völlig gelöscht werden. Dadurch ist sichergestellt, daß die Prozedur keinerlei Variablenspeicherplatz benötigt, solange sie nicht aktiv ist.

Die Prozeduren/Funktionen ohne statische Variablen sind auch die einzigen, mit denen man sinnvoll Rekursionen programmieren kann, da sie sich direkt oder auf Umwegen beliebig oft selbst aufrufen können und dabei für jeden Aufruf eine neue Gruppe von Variablen zur Verfügung gestellt wird, wobei die Daten aus übergeordneten Aufrufen erhalten bleiben.

Dazu im Gegensatz stehen die statischen Prozeduren und Funktionen. Auch heute noch sind sie durchaus sinnvoll: Einerseits werden sie etwas schneller ausgeführt als ihre Kollegen mit automatischen Variablen, andererseits ermöglichen sie es, weitgehend selbständige Subroutinen zu programmieren, die nicht nur Hilfsroutine, sondern regelrecht „Programm im Programm“ sind. Statische Variablen innerhalb einer Subroutine behalten ihre Werte zwischen zwei Prozeduraufrufen, anstatt bei jedem Prozeduraufruf neu initialisiert zu werden. Dadurch kann eine Prozedur einen eigenen Datenbestand verwalten, dessen Lebensdauer nicht mit dem Ende der Prozedur endet.

Der Befehl STATIC, dessen Anwendung nur in automatischen Subroutinen sinnvoll ist, ermöglicht es, ausgewählte Variablen in einer automatischen Subroutine statisch zu machen. In einer statischen Subroutine sind *alle* Variablen statisch.

Selbständige Subroutinen

Mit statischen Variablen kann man Subroutinen programmieren, die ein regelrechtes Eigenleben führen, ihre eigenen Variablen und Datenbestände auf der Festplatte verwalten und andere praktische Dinge tun.

Zum Beispiel könnten Sie, wenn Sie ein Programm schreiben, das viel mit Druckern arbeiten muß, sich einen Drucker-Befehlsinterpreter als Subroutine schreiben, der nur einen einzigen String-Parameter hat:

```
SUB Drucker (Kommando AS STRING) STATIC
```

Dann könnten Sie in Ihrem Programm vielleicht Befehle wie `DRUCKER "INIT: EPSONFX80"`, `DRUCKER "RAND LINKS 2CM"`, `DRUCKER "ZENTRIERT: " + Text$` und so weiter verwenden, anstatt sich dauernd mit LPRINT-, WIDTH- und PRINT#-Befehlen herumzuschlagen. Diese Drucker-Kontrollroutine könnte dann Druckertreiber für verschiedene Modelle auf der Festplatte haben. Sie könnte, wenn sie das erste Mal aufgerufen wird, automatisch einen Standardtreiber in ihren eigenen statischen Speicher laden; sobald sie einen INIT-Befehl erhält, könnte sie einen neuen Treiber laden, und wenn sie feststellt, daß sie zuletzt vor über fünf Minuten aufgerufen wurde, selbständig feststellen, ob der Drucker noch bereit ist oder vielleicht inzwischen eine Fehlermeldung vorliegt.

Sie könnte vielleicht auch einen Befehl `"VERZEICHNIS"` unterstützen, der sie dazu veranlaßt, in einem einzigen String (32.767 Bytes dürften dafür ausreichen) eine Liste aller installierten Druckertreiber zurückzugeben, und vieles mehr.

Mit einer solchen Druckerkontrollroutine hätten Sie ein universelles Hilfsmittel für alle zukünftigen Programme geschaffen, das Ihnen nicht nur die Programmierung erleichtert. Alle Ihre Programme wären von nun an untereinander „druckerkompatibel“, das heißt, einmal definierte Druckertreiber könnten für jedes Programm benutzt werden. Mit Bedacht eingesetzt, läßt sich mit solcher Technik ein wirklich modulares Programmieren realisieren, zu dem die statischen Variablen einen nicht unerheblichen Beitrag leisten.

STATIC-Variablen und Formen

Der Vergleich zu Formen liegt nahe, und in der Tat werden Formen intern ähnlich wie statische Prozeduren behandelt: Alle Eigenschaften der Form und ihrer Steuerelemente bleiben auch zwischen den Aufrufen erhalten. Das gilt jedoch nicht für Variablen in Ereignisprozeduren; diese unterliegen den gewöhnlichen Regeln. Wenn Sie in einer Ereignisprozedur statische Variablen benötigen, müssen Sie diese entweder mit STATIC einzeln vereinbaren oder die ganze Prozedur als STATIC SUB deklarieren.

8.5 Extended & Expanded Memory

Dieser Abschnitt handelt von Expanded Memory (EMS) und von Extended Memory (XMS). Beide Begriffe beziehen sich auf Speicherbereiche oberhalb der 1-MB-Grenze, und bei beiden reicht es nicht einfach aus, die zusätzlichen RAM-Chips im Rechner installiert zu haben, sondern man muß auch einen entsprechenden Treiber durch einen Aufruf in der CONFIG.SYS-Datei laden.

Unter „Expanded Memory“ versteht man RAM-Speicher, den Ihr Rechner über 1 MB hinaus besitzt, und der durch einen speziellen Treiber den EMS-Spezifikationen (genauer: Lotus-Intel-Microsoft Expanded Memory Specification 4.0 oder auch LIM 4.0) gemäß verfügbar gemacht wird. Häufig ist dies der von Microsoft mit DOS und WINDOWS gelieferte „EMM386.EXE“, aber auch andere Speichermanager wie „QEMM386“ oder „386MAX“ stellen EMS zur Verfügung, und manchmal ist auch beim Kauf eines Motherboards bzw. eines Komplettrechners eine Treiberdiskette beigelegt.

EMS kann in jedem PC-kompatiblen Rechner vorhanden sein, da EMS den Zusatzspeicher so verwaltet, daß auch 8086-Prozessoren ihn adressieren können.

Extended Memory ist nur mit 80286- oder neueren Prozessoren verträglich und bezieht sich auf Speicher oberhalb 1 MB, der durch einen Treiber gemäß XMS-Spezifikation 2.0 verfügbar gemacht wird. Ein solcher XMS-Treiber ist HIMEM.SYS.

EMS und XMS innerhalb von VBDOS

Wenn XMS vorhanden ist, kann VBDOS etwa 60 KB seines eigenen Programmcodes in den XMS-Bereich jenseits 1 MB schieben, so daß diese 60 KB nun für größere Programme etc. zur Verfügung stehen. Mehr nutzt Ihnen XMS innerhalb von VBDOS nicht.

EMS kann bei der Arbeit mit VBDOS dafür genutzt werden, bestimmte Programm- und Datenbereiche, die andernfalls den 640 KB-Standard-Arbeitsspeicher belasten würden, auszulagern.

VBDOS lagert generell nur Datenbereiche aus, die eine Größe von mindestens 512 und höchstens 16.384 Bytes haben. Als solche Bereiche gelten einerseits SUBs oder FUNCTIONs und andererseits sämtliche Arrays (ausgenommen String-Arrays mit variabler Länge), die innerhalb der angegebenen Größen liegen.

Die Auslagerung von Arrays erfordert, daß man beim Aufruf von VBDOS den Switch /Ea angibt; Subroutinen werden automatisch ausgelagert. Mit dem Switch /E:*vbdos,programm* kann man die maximale EMS-Menge festlegen, derer sich VBDOS zur Auslagerung seiner eigenen Komponenten (*vbdos*) bzw. von Programmkomponenten (*programm*) bemächtigen darf; läßt man ihn weg, kann VBDOS unter Umständen den gesamten zur Verfügung stehenden EMS-Speicher belegen.

Wenn Sie eine Quick Library benutzen, die ihrerseits Zugriffe auf das EMS unternimmt, müssen Sie überdies den Switch /Es angeben, der VBDOS veranlaßt, vor jedem Quick-Library-Aufruf den aktuellen Status des EMS zu sichern. Das verhindert, daß sich die Quick-Library-Routinen und VBDOS in die Quere kommen, kostet aber etwas Rechenzeit.

Wenn es Speicherprobleme bei der Arbeit mit VBDOS gibt und man EMS besitzt, sollte man also darauf achten, daß möglichst alle Prozeduren und Arrays in ihrer Größe zwischen 512 und 16.384 Bytes liegen, damit sie ausgelagert werden können. Außerdem wird der EMS-Speicher – nicht nur in VBDOS – auch als ISAM-Buffer (Zwischenspeicher) benutzt, um schnellere Zugriffe zu ermöglichen (ISAM ist nur in der professionellen Ausgabe verfügbar).

EMS und XMS bei kompilierten Programmen (EXE-Files)

Kompilierte Programme können EMS weder zur Auslagerung von Subroutinen noch zur Speicherung von Arrays nutzen. Durch direkte Aufrufe des EMS-Treibers wird es jedoch möglich, beliebige Daten im EMS zu speichern und so Programme zu schreiben, die mehrere Megabyte an Daten im Speicher verwalten. Weitere Informationen hierüber finden Sie in Kapitel 18.

In der professionellen Ausgabe können XMS und EMS zum Auslagern von Overlays verwendet werden (die Auslagerung geht dann wesentlich schneller als bei Verwendung der Festplatte). Weitere Informationen zu Overlays finden Sie im Kapitel 9.

Ebenso wie innerhalb von VBDOS kann in kompilierten Programmen EMS-Speicher als Zwischenspeicher für ISAM dienen. Näheres dazu im Kapitel 12.

Falls Sie Overlays ins EMS oder XMS auslagern lassen und mit nicht in BASIC geschriebenen Zusatz-Routinen arbeiten, die EMS benutzen, müssen Sie beim Kompilieren den Switch /Es angeben, der dafür sorgt, daß der Compiler vor jedem Library-Aufruf den Zustand des EMS-Speichers sichert, damit es keine Interferenzen zwischen dem BASIC-Programm und den anderen Routinen gibt. Der Switch /Es sorgt allerdings für längere EXE-Files und bei Library-Zugriffen für etwas langsamere Ausführungsgeschwindigkeit.

8.6 Zeitcodes

VBDOS kennt eine Anzahl von Funktionen, die mit Zeitcodes umgehen können. Als Beispiel seien NOW und YEAR genannt: NOW gibt den Zeitcode des aktuellen Datums zurück, und YEAR ermittelt zu einem gegebene Zeitcode das Jahr.

Zeitcodes werden von VBDOS wie Fließkommazahlen doppelter Genauigkeit gehandhabt (DOUBLE-Zahlen, Typbezeichner „#“). Sie haben gegenüber der gewöhnlichen Schreibweise für Datum und Uhrzeit den entscheidenden Vorteil, fortlaufend zu sein. Das bedeutet, daß man zwei Zeitcodes einfach voneinander subtrahieren kann, um aus dem entstehenden Differenz-Zeitcode dann zu entnehmen, wie weit die beiden Zeitpunkte voneinander entfernt waren.

Es ist mit Zeitcodes kein Problem mehr, den Wochentag zu einem bestimmten Datum zu ermitteln, und viele andere Probleme (Schaltjahre etc.) fallen ebenfalls weg.

Ein Zeitcode ist, wie gesagt, eine Fließkommazahl doppelter Genauigkeit, die auf die Sekunde genau einen ganz bestimmten Zeitpunkt beschreibt. Im ganz-

zahligen Teil enthält sie das Datum (Tag, Monat und Jahr), während im fraktionalen Teil die Uhrzeit (Stunde, Minute und Sekunde) gespeichert werden kann. Das Datum kann im Bereich 1.1.1753 bis 31.12.2078 bearbeitet werden, die Uhrzeit selbstverständlich von 00:00.00 bis 23:59.59.

Durch die Aufteilung in Vor- und Nachkommastellen ist es leicht möglich, einen kompletten Zeitcode in einen reinen Datums-Zeitcode und einen reinen Uhrzeit-Zeitcode zu teilen:

```
NurDatum# = INT(ZeitCode#)
NurZeit# = ZeitCode#   NurDatum#
```

Ebenso kann durch Addition aus reinem Datums- und reinem Uhrzeit-Zeitcode wieder ein kompletter Zeitcode gewonnen werden.

Umfangreiche Programme 9

9.1 Übersicht

In bezug auf Speicherknappheit gibt es bei der Programmentwicklung verschiedene Stadien. Die erste Grenze, auf die die meisten von uns wahrscheinlich schon einmal gestoßen sind, ist der Arbeitsspeicher des Compilers BC.EXE. Wenn man ihm eine einzige Quelldatei vorsetzt, die zu groß ist – zu viele Variablen, Prozeduren, Funktionen enthält oder schlicht zu viele Zeilen hat – kann er sie nicht mehr korrekt kompilieren. Man erkennt das schon daran, daß die Zahl vor der Meldung „Byte frei“ mit jeder neuen Programmversion etwas kleiner wird...

Es kommt häufig vor, daß ein Programm in VBDOS anstandslos läuft, für BC aber schon zu groß ist. Das liegt daran, daß BC anders als VBDOS intern mehrere Querverweistabellen usw. für das Programm aufbauen muß und sich so leichter „verschluckt“.

Die erste Hürde

Dieses erste Problem kann man recht leicht überwinden, indem man einige Teile des Programms als SUB oder FUNCTION formuliert und in ein zweites Modul auslagert. Viele Programmierer, die ich kenne, haben sich bei dieser Gelegenheit eher unfreiwillig mit dem Prozedurkonzept, mit lokalen und globalen Variablen und allem, was dazugehört, auseinandergesetzt – besser spät als nie.

Bei einem solchen aus mehreren Modulen bestehenden Programm wird jedes Modul für sich kompiliert, und LINK wird benutzt, um die entstehenden OBJ-Dateien zu einem einzigen EXE-Programm zusammenzufügen. VBDOS erledigt das automatisch.

Mit diesem Konzept fährt man erfahrungsgemäß etwa so lange gut, bis die Größe der einzelnen EXE-Dateien zwischen 400 und 500 KB liegt. Dann beginnt LINK zuweilen, ominöse Fehler zu melden („Fixup overflow near segment...“), und das EXE-Programm ist nicht so stabil, wie man das vielleicht erhofft.

EXE-Programme über 400 KB

Um dieses Problem zu lösen, hatte man früher ausschließlich die Möglichkeit, das Programm in mehrere Komponenten aufzuteilen, die sich gegenseitig mit RUN oder CHAIN aufrufen. Der größte Nachteil dabei war, daß bestimmte

Routinen in allen Programmen enthalten sein mußten (bei mir gehörten zum Beispiel eine Menü- und eine Eingaberoutine immer zum „harten Kern“) und so mehrfach Platz verbrauchten.

Welche Methoden mit VBDOS zur Verfügung stehen, werde ich im folgenden detailliert erklären. Vorab sei noch erwähnt, daß man natürlich nicht warten muß, bis das EXE-Programm 400 KB groß ist, bevor man eine dieser Methoden anwendet. Insbesondere die Overlay-Methode kann auch genutzt werden, um den Speicherplatz, den ein Programm benötigt, drastisch zu reduzieren. Es soll sie ja noch geben, die PC-XTs mit 640 oder gar 512 KB Hauptspeicher, diversen speicherresidenten Programmen...

9.2 Programmsysteme mit CHAIN

Diese Methode war bisher die einzige, die es ermöglichte, Programme mit großem Gesamtumfang zu erstellen: Man schreibt einzelne, voneinander unabhängige Programme und kompiliert bzw. linkt sie getrennt. Es bietet sich an, eine Library mit den in allen Programmteilen benötigten Routinen zusammenzustellen, die dann bei jedem einzelnen Link-Vorgang benutzt werden kann (siehe Kapitel 6). Die fertigen, ausführbaren Module liegen schließlich als einzelne EXE-Files vor und können sich gegenseitig mit `CHAIN "dateiname"` oder `RUN "dateiname"` aufrufen.

Eine Datenübergabe zwischen solchen Modulen – sofern sie nötig ist – kann entweder über eine temporäre Datei laufen (Modul 1 erzeugt die Datei mit allen wichtigen Daten, startet dann Modul 2; Modul 2 lädt die Daten und löscht die Datei) oder über den COMMON-Befehl.

Die Variante mit der temporären Datei hat den Nachteil, daß sie zusätzliche Disketten- bzw. Festplattenzugriffe benötigt und dadurch länger dauert. Sie ist aber wesentlich „robuster“ als die COMMON-Methode, denn

- COMMON funktioniert nur, wenn man ohne den /O-Switch kompiliert, d.h. wenn man mit einem Runtime-Modul arbeitet;
- nur namenlose COMMON-Blocks werden übergeben (siehe Eintrag zu COMMON im Disketten-Referenzteil) und
- bei der COMMON-Übergabe müssen die Variablentypen exakt übereinstimmen, der Compiler kann keine Prüfung vornehmen (wie er es zum Beispiel bei Prozeduraufrufen tut), und kleinste Fehler führen zu nicht vorhersagbaren Programmabstürzen während der Laufzeit.

Wenn eine Datenübergabe erforderlich ist, sollte man also ernsthaft erwägen, ob man nicht lieber eine temporäre Datei statt des COMMON-Befehls verwenden will. Auch der gegenseitige Aufruf von VBDOS und Form-Designer wird über eine temporäre Datei gesteuert.

Es empfiehlt sich bei solchen Programmsystemen, mit Runtime-Modul (also ohne /O-Switch) zu arbeiten. So wird verhindert, daß die BASIC-Routinen in jedes einzelne EXE-File eingebunden werden. Das Runtime-Modul muß außerdem nicht bei jedem CHAIN neu geladen werden. Es bleibt im Speicher, wenn alle Module so erstellt sind, daß sie mit demselben Runtime-Modul arbeiten. Achten Sie also als Anwender der professionellen Ausgabe von VBDOS darauf, nicht ein Modul mit der Emulator- und ein anderes mit der Alternate Math-Library zu erstellen, sonst sind die Vorteile dahin, und Sie müssen Ihr Programm gleich mit zwei Runtime-Modulen ausliefern.

Theoretisch ist auch der RUN-Befehl geeignet, um Programme sich gegenseitig aufrufen zu lassen. Im Gegensatz zum CHAIN-Befehl schließt er zuvor alle Dateien, erlaubt keinerlei COMMON-Datenübergabe und lädt in jedem Fall die Runtime-Library neu. Das macht ihn zwar relativ ungeeignet; sollten Sie jedoch einmal daran verzweifeln, daß ein großes CHAIN-System (selbst dann, wenn Sie kein COMMON verwenden) von Zeit zu Zeit scheinbar grundlos abstürzt, kann es die Rettung sein, CHAIN durch RUN zu ersetzen. RUN räumt das Speicher-Schlachtfeld vor jedem Programmaufruf auf und organisiert den Speicher neu, was man von CHAIN nicht gerade behaupten kann.

Runtime-Module

Programmsysteme mit CHAIN haben oft den Nachteil, daß häufig benötigte Routinen (Menü-Routine, Eingaberoutine etc.) in jedem Modul vorhanden sind und so nicht nur Ladezeit, sondern auch Festplatten-Speicherplatz kosten. Dem kann (schon seit BASIC 6.0, bei VBDOS allerdings nur in der professionellen Ausgabe) abgeholfen werden, indem man eigene universelle Routinen in Runtime-Module einbindet. Sie müssen dann nur einmal auf der Festplatte vorhanden sein und auch nur einmal geladen werden. Details über Runtime-Module und deren Verwendung finden Sie im Kapitel 14.

9.3 Overlays – der Schlüssel zum Megabyte-Programm

Wer von uns BASIC-Programmierern hat nicht schon einmal etwas neidisch auf Softwarepakete geblickt, die ihren gesamten Leistungsumfang in einem Zwei-

Megabyte-File konzentrierten, anstatt aus unzähligen Einzelprogrammen zu bestehen?

Solche Riesenprogramme haben gewiß auch Nachteile. Sie laufen nur auf einer Festplatte, und die Installation mit Hilfe von Disketten ist nicht ganz unproblematisch, weil die Datei in einzelne Häppchen gesplittet und erst beim Installieren auf der Platte zu einem Ganzen zusammengesetzt werden muß. Dennoch sind solche Programme wesentlich eleganter als ein Haufen einzelner Programme und überdies auch kompakter, da im EXE-File keine einzige Routine mehrfach vorliegen muß.

Derart große Programme lassen sich mit Overlay-Technik erzeugen. Das Overlay-Konzept hat schon bei vielen anderen Programmiersprachen erfolgreich Verwendung gefunden. Der Microsoft-Linker unterstützt es schon lange, aber die fehlende Anpassung des BASIC-Compilers strafte noch bis zur Version 6.0 den experimentierfreudigen Programmierer mit Systemabstürzen, wenn er sich an Overlays heranwagte. Mit VBDOS wurde die im BASIC 7.1 eingeführte Overlay-Technik verändert; die professionelle Ausgabe von VBDOS unterstützt jetzt „MOVE“ (Microsoft Overlay Virtual Environment), eine angeblich recht fortschrittliche Methode.

Overlay-Technik bedeutet, daß sich sämtliche Routinen, die ein Programm benötigt, in einem einzigen EXE-File befinden, daß aber nur ein Teil davon im Speicher arbeitet und die benötigten Parts automatisch nachgeladen werden.

Die Betonung liegt hierbei auf „automatisch“ – man braucht in seine Programme keinerlei Anweisungen einzubinden, die für das Nachladen von Routinen sorgen. Dieses automatische Nachladen geht zumeist auch wesentlich schneller als das Starten eines anderen Moduls mit CHAIN oder RUN.

Overlays und erweiterter Speicher

Unter DOS können Programme nur abgearbeitet werden, wenn sie sich innerhalb der ersten 640 KB des Hauptspeichers befinden. Deshalb können bei Overlay-Programmen auch immer maximal 640 KB (abzüglich des Platzes, der von DOS und residenten Programmen belegt wird) in den Speicher geladen werden. Um dennoch von eventuell vorhandenem EMS- oder XMS-Speicher (jenseits der 1 MB-Grenze) zu profitieren, können Overlays in den EMS bzw. XMS ausgelagert werden, so daß sie bei Bedarf nicht von der Festplatte, sondern aus dem Erweiterungsspeicher nachgeladen werden. Das bringt erhebliche Geschwindigkeitsvorteile. Näheres zur Nutzung des erweiterten Speichers finden Sie im Kapitel 8.

Voraussetzungen für Overlays

Um die Overlay-Technik nutzen zu können, müssen Sie sich zunächst einen Plan machen, welche Ihrer Routinen als Overlays geeignet sind. Dabei werden Sie in den meisten Fällen eine Anzahl von Routinen finden, die dauernd und von allen Programmteilen benötigt werden, und eine Gruppe von Routinen, die gegeneinander austauschbar sind, das heißt, wenn die Routine A gerade läuft, wird die Routine B wahrscheinlich nicht benötigt usw.

Wenn Sie bei dieser Planung Fehler machen, ist das nicht so tragisch; es wird Ihr Programm allenfalls verlangsamen. Abstürzen wird es nicht – zumindest nicht deshalb.

Wenn Sie ein Programmsystem, das zuvor mit CHAIN funktionierte, auf Overlay-Technik umstellen wollen, müssen Sie den Modulcode aller Programme außer dem Hauptprogramm in Subroutinen umformulieren, so daß am Ende nur noch ein einziges Modul überhaupt Modulcode besitzt. CHAIN-Befehle müssen dann durch einen Aufruf der entsprechenden, neu entstandenen Subroutine ersetzt werden, so daß sich ein Programmsystem ergibt, das theoretisch, wenn es nicht so groß wäre, auch ohne Overlays in ein einziges EXE-Programm kompiliert werden könnte.

Bevor ich jetzt fortfahre, noch ein deutliches Wort über die Overlays betreffenden Passagen im Microsoft-Handbuch: Vergessen Sie es. Sie haben vielleicht schon an anderen Stellen gemerkt, daß ich mit dem Original-Handbuch *nicht so ganz* zufrieden war, aber im besagten Kapitel stehen fast ausschließlich Halbwahrheiten und unwichtige Informationen. Halten Sie sich lieber an den folgenden Text.

Linken mit Overlays

Die Overlay-Technik wird aktiviert, indem man beim Linken nicht alle verwendeten OBJ-Dateien nur durch + verbindet, sondern dabei Klammern setzt. Dabei bildet jede Gruppe von OBJ-Dateien in Klammern einen Overlay-Block. Wenn das Programm später läuft, ist nur der Teil der OBJ-Dateien dauernd im Speicher, der außerhalb aller Klammern steht. Jede umklammerte Gruppe bildet ein Overlay, und von diesen Overlays sind immer nur so viele geladen, wie in den dafür reservierten Speicherbereich – den „Overlay Heap“ – passen.

Wird eine Routine aus einem gerade nicht geladenen Overlay aufgerufen, so wird dieses in den „Overlay Heap“ geladen. Falls dort nicht genug freier Platz zur Verfügung steht, wird das Overlay, das schon die längste Zeit nicht benutzt

wurde, aus dem Heap entfernt. Reicht der Speicher immer noch nicht, werden weitere Overlays entfernt, bis das angeforderte geladen werden kann.

Die Größe des Overlay-Heaps ist standardmäßig so groß, daß die drei größten Overlays gleichzeitig hineinpassen. Wenn der verfügbare Speicher kleiner ist, wird seine Größe angepaßt; er kann jedoch niemals kleiner als die Größe des größten Overlays werden.

Indem Sie die DOS-Umgebungsvariable OVERLAY_HEAP setzen, können Sie die Größe des Overlay-Heaps in KB einstellen. Der DOS-Befehl `SET OVERLAY_HEAP=200` würde die Größe des Overlay-Heaps also auf 200 KB setzen. Der Befehl muß jedoch vor dem Start Ihres Programmes ausgeführt werden; es gibt keine Möglichkeit, den Overlay-Heap innerhalb des Programms zu verstellen.

Nehmen wir an, nach dem erfolgreichen Kompilieren liegen fünf Objektmodule vor: EINS.OBJ, das Hauptprogramm; ZWEI.OBJ mit allen Routinen, die dauernd benötigt werden; DREI.OBJ und die Kombination aus VIER.OBJ und FUENF.OBJ, zwei Programmteile, die gegeneinander austauschbar sind (das heißt, DREI.OBJ benötigt keine Routinen aus VIER.OBJ und FUENF.OBJ und umgekehrt). Dann hieße der LINK-Befehl:

```
LINK EINS+ZWEI+(DREI)+(VIER+FUENF);
```

Hier bilden die Module EINS und ZWEI, die nicht mit Klammern versehen sind, die sogenannte „Root“, den Grundstock, der immer geladen ist. Alles, was in Klammern steht, ist jeweils ein Overlay, das heißt, es wird geladen, wenn eine Routine daraus benötigt wird. In unserem Falle würde DREI automatisch geladen, wenn aus EINS oder ZWEI heraus eine Prozedur aufgerufen würde, die in DREI enthalten ist. DREI bliebe, auch wenn es nicht mehr benötigt wird, weiterhin im Speicher, bis eine Prozedur aus VIER oder FUENF gebraucht wird. Dann würde geprüft werden, ob im Overlay-Heap noch genügend Platz ist; wenn ja, würde das zweite Overlay mit VIER und FUENF hinzugeladen, wenn nein, würde DREI zuvor aus dem Speicher entfernt.

Die Leistungsfähigkeit des Programms wird stark davon beeinflußt, wie Sie die Klammern setzen. Es muß stets darauf geachtet werden, daß möglichst selten ein neues Overlay von der Platte nachgeladen wird, denn das kostet Zeit.

Je mehr kleine Overlays Sie verwenden, desto eher verlieren Sie zwar die Übersicht, was wann wie benötigt wird, aber desto größer ist auch die Wahrscheinlichkeit, daß sich im Overlay-Heap irgendwann ein „Gleichgewicht“ einpendelt, das einen relativ ruhigen Betrieb gewährleistet. Microsoft empfiehlt eine durchschnittliche Overlay-Größe von nur 4 KB, und ich kann aus Erfahrung bestätigen, daß zumindest einstellige Overlaygrößen sehr effektiv sind.

Wenn Sie dieser Empfehlung nachkommen, werden Sie sehr viele einzelne OBJ-Dateien haben, und es bietet sich an, mit einer Befehlsdatei für LINK zu arbeiten. Dadurch kann man auch große LINK-Vorgänge gut strukturieren (ein Beispiel aus dem „wirklichen Leben“, die Zeilennummern gehören nicht dazu):

```
( 1) /DY:150/PACKC+
( 2) MAIN+
( 3) EINGABE PARMS COMMON+
( 4) HINWEIS STHINW KFEHLER BINOP SPLITS BUFFER KEYIN+
( 5) PAUSE TON COL GROSS HIGHL DRUCK EXIST UNIF+
( 6) LEVEL COMVAL VIDEO INSCREEN LREIHE+
( 7) SORTSTR ARRAY GRFARB FORMAT+
( 8) NOTAB ALLCENS CALCATTR HELP+
( 9) (ZRVW)+
(10) (MENUE MMENU FINDNAME)+
(11) (AP TKGL TREND GLEIDUR)+
(12) (DREIHE PRPL BOEING)+
(13) (EDITUP SREIHE CONVR)+
(14) (FILEMAN COPYDIR DELDIR RENDIR COPY DIRECT ADDDIR IDENT)+
(15) (ABLEIT UPDIA DIARUN DIASHOW)+
(16) (GRAPHEX GRAPHIK LOGFUNC)+
(17) (ZREIHE)+
(18) (VERZ)+
(19) (MAKESHOW)+
(20) (TKUT)+
(21) (AS)+
(22) (TKIV SAISIND)+
(23) (DOSEX)+
(24) (CENSUS)+
(25) (TKKOM LOTUS DBASE)+
(26) (PF)+
(27) (TKTRANS)+
(28) (AT)+
(29) NOEDIT SMALLERR
(30) MAIN.EXE
(31)
(32) DIVERSE.LIB
(33)
```

In der ersten Zeile stehen nur die Switches für den LINK-Vorgang. Den Switch /DY:150 habe ich angegeben, um ein etwas kleineres EXE-File zu erhalten (siehe weiter unten). Die Zeilen 2–8 enthalten alle OBJ-Dateien, die der „Root“ zugeordnet werden, und von Zeile 9 bis 28 finden sich die Overlays. In Zeile 29 verwende ich zwei Verzicht-Files, die ebenfalls der „Root“ zugeordnet werden. Da die Zeilen 1 bis 28 alle mit einem + enden, betrachtet LINK sie als eine einzige Zeile, und erst die Zeile 30 ist für LINK die Zeile 2, in der der Name des zu erzeugenden EXE-Programms steht.

Wenn diese Datei MAIN.CMD hieße, könnte man durch

```
LINK @MAIN.CMD
```

den LINK-Vorgang starten. Details zu LINK finden Sie im Kapitel 6; dieses Beispiel sollte Ihnen nur zeigen, daß Sie glänzend ohne die von Microsoft empfohlene DEF-Datei auskommen können.

Wenn man mit sehr vielen kleinen Overlays arbeitet, wird man selten mehrere OBJ-Dateien zusammen in ein Overlay stecken, wie ich das hier z. B. in Zeile 14 getan habe. Das ist nur dann sinnvoll, wenn einige Routinen so eng zusammenarbeiten, daß eine fast immer (oder immer) die andere aufruft und es deshalb unsinnig wäre, hiermit den Overlay-Manager zu belasten. In solchen Fällen kann man natürlich auch mehrere Routinen in einem Modul lassen, so daß sich erst gar nicht mehrere OBJ-Dateien ergeben.

Beschränkungen

Wenn Sie mit Overlays arbeiten, können Sie beim Linken nicht den Switch /EX verwenden. Der Switch /F wird automatisch gesetzt; eine zusätzliche Verkleinerung der entstehenden Programme erreichen Sie mit /PACKC.

Die voreingestellte Maximalzahl von Prozeduraufrufen zwischen Overlays ist 255; diese Zahl kann durch den Switch /DY verändert werden. Geben Sie beim Linken den Switch /INF an, um zu sehen, wieviele Aufrufe zwischen Overlays tatsächlich erzeugt werden (die Zahl wird ganz am Ende der langen Liste ausgegeben). Die mit /DY angegebene Zahl muß um eins größer als die angezeigte Anzahl sein. Wenn Sie eine zu große Zahl angeben (oder /DY nicht verwenden und Ihr Programm weniger als 255 Aufrufe zwischen Overlays hat), ist das Verschwendung. Pro Aufruf werden 6 Bytes reserviert, so daß ein Programm mit 100 Aufrufen zwischen Overlays bei Verzicht auf /DY:101 924 Byte zu lang wird. Wenn Sie beim Linken einen Fehler „Zu viele Aufrufe zwischen Overlays“ erhalten, haben Sie einen zu kleinen Wert für /DY angegeben.

Ein einzelner Overlay-Block darf nicht größer als 64 KB sein; die Maximalzahl von Overlays ist 2.047.

Wie bereits erwähnt, werden Overlays bei Vorhandensein von EMS oder XMS zur Laufzeit in das XMS (bevorzugt) oder das EMS ausgelagert. Sie können die EMS-/XMS-Nutzung durch Overlays mit den Betriebssystemvariablen OVERLAY_XMS und OVERLAY_EMS beschränken; geben Sie z.B. vor dem Start Ihres Programms `SET OVERLAY_XMS=50` ein, werden maximal 50 KB XMS verwendet. Beachten Sie dabei, daß EMS und XMS vom Overlay-Manager nicht gleichzeitig verwendet werden können.

Wenn sich zur Laufzeit mehr als 64 Overlays untereinander aufrufen, stürzt das Programm unweigerlich und ohne vorherige An- oder Fehlermeldung ab.

Overlays optimieren

Um die Größe der Overlays optimal einzustellen und Anhaltspunkte zu gewinnen, ob ein bestimmtes Modul besser in der „Root“ (also immer) oder in einem Overlay geladen wird, und auch, um sinnvolle Einstellungen für die Größe des Overlay-Heaps zu erhalten, können Sie das Hilfsprogramm TRACE.EXE verwenden. Geben Sie dazu die Library VBDTRACE.LIB beim Linken im Library-Feld an. Das resultierende EXE-Programm sollten Sie nur zu Testzwecken verwenden. Es protokolliert Overlay-Aufrufe in einer Datei und läuft daher langsamer.

Wenn Sie dieses so erstellte Programm dann starten und damit arbeiten, notiert der Overlay-Manager in der Datei MOVE.TRC penibel, welche Overlays er wann, warum und woher lädt oder wohin auslagert. Um eine möglichst realitätsnahe Analyse zu ermöglichen, sollten Sie während eines solchen Testlaufs Ihr Programm etwa in der Art und Weise benutzen, wie das im „Ernstfall“ auch geschehen wird.

TRACEs kryptischer Output

Nach Beendigung Ihres Programms rufen Sie das Hilfsprogramm TRACE.EXE auf. Es gibt eine Liste der vom Overlay-Manager in MOVE.TRC vermerkten Ereignisse im Klartext aus, etwa so (gekürzt):

```
Microsoft (R) MOVE Trace Utility  Version 1.1
Copyright (C) Microsoft Corp.  1991-1992.  All rights reserved.

(0000) 07c6:0076 - Load from disk           - (0005) [9a97] Call (0005) 10d9
(0000) 07c6:007b - Present                  - (0005) [9a97] Call (0005) 27a0
(0005) 2588:a125 - Load from disk           - (0002) [9e57] Call (0002) 003c
(0005) 2588:a125 - Present                  - (0005) [9a97] Return
(0000) 07c6:00dd - Present                  - (0005) [9a97] Call (0005) 08f2
(0005) 2588:a125 - Discard from heap        - (0003) [9f99] Call (0001) 00fa
                 - Cache to extended memory - (0003) [9f99]
                 - Discard from heap        - (0004) [9fa6]
                 - Cache to extended memory - (0004) [9fa6]
                 - Load from disk           - (0001) [9f99]
(0001) 2588:a125 - Present                  - (0001) [9f99] Call (0001) 01b3

TRACE:  All done
```

Diese Liste zu dechiffrieren, ist nicht ganz so einfach. Die Zahlen in runden Klammern sind Overlaynummern, alle anderen sind Speicheradressen. In der mittleren Spalte ist zu lesen, wie der Overlay-Manager reagierte.

Das erste Overlay (die erste in Klammern gesetzte Gruppe) in Ihrem LINK-Aufruf erhält die Overlaynummer 1; die weiteren sind fortlaufend (hexadezimal) durchnumeriert. Die Overlaynummer 0 steht für die „Root“. Untersuchen wir einen Eintrag aus der Liste einmal genauer:

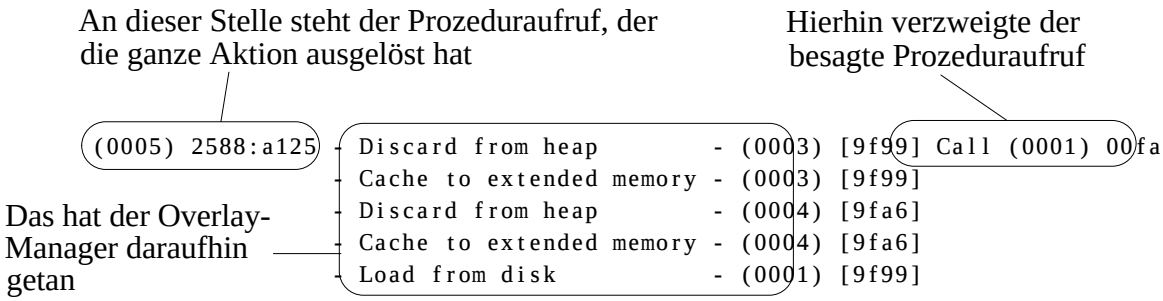


Abbildung 9–1: Die Ausgabe des TRACE-Programms

Zunächst kann man hier ablesen, an welcher Stelle im Programm der Prozeduraufruf stand, der den Vorgang ausgelöst hatte. Es muß sich natürlich um den Aufruf einer Prozedur in einem anderen Overlay handeln – Aufrufe in die „Root“ oder in dasselbe Overlay werden hier nicht berücksichtigt. In diesem Fall ist es ein Prozeduraufruf aus dem Overlay 5 an eine Prozedur im Overlay 1. Der Overlay-Manager mußte daraufhin zunächst die Overlays 3 und 4 aus dem Overlay-Heap auslagern – vermutlich ist Overlay 1 also ein größeres Modul – und konnte dann das Overlay 1 von der Platte laden.

Wenn später wieder das Overlay 3 oder 4 gebraucht wird, wird kein „Load from disk“ mehr hier stehen, sondern ein „Load from extended memory“, da die beiden Overlays ja dorthin ausgelagert wurden. (Ausnahme: Wenn inzwischen des XMS knapp wurde und andere Overlays dorthin ausgelagert werden mußten, können die Overlays 3 und 4 aus dem XMS gelöscht worden sein und müßten dann wieder von der Platte nachgeladen werden.)

Neben den „Call“-Zeilen existieren auch noch „Return“-Zeilen in der Ausgabe von TRACE.EXE; diese bezeichnen die Beendigung einer Prozedur, die von einem Overlay aus in einem anderen Overlay aufgerufen wurde. Hierbei kann es unter Umständen ebenfalls nötig sein, das Overlay, zu dem zurückgekehrt wird, nachzuladen, falls es im Verlaufe der Prozedur gelöscht oder ausgelagert wurde. In den Return-Zeilen kann jedoch nicht abgelesen werden, aus welchem Overlay die Rückkehr erfolgte.

Von der Adresse zum Segment...

Nun ist es für Sie wahrscheinlich nicht gerade besonders interessant, von welcher Adresse und an welche Adresse ein Overlay-Aufruf stattfand: Die Zeilennummer im BASIC-Programm oder zumindest der Name der Prozedur wären Ihnen lieber.

Um an diese Informationen zu gelangen, müssen Sie entweder viel Handarbeit leisten – oder Sie verwenden TRACELST.BAS, das ich später vorstelle. Zunächst aber ein kurzer Exkurs in die Technik des Spurenlesens:

Wenn Sie beim Linken Ihrer Programme einen Namen für eine Listdatei (.MAP) angeben, erzeugt LINK darin eine Aufschlüsselung aller Segmente mit ihren Adressen. Das sieht ungefähr so aus (gekürzt):

```
 Anfang Ende   Länge  Name                   Klasse
 Resident
 00000H 005A7H 005A8H CHRTDEMO_CODE          BC_CODE
 005B0H 08F02H 08953H CHART_CODE             BC_CODE
 08F10H 0C2BEH 033AFH FONT_CODE              BC_CODE
 0C2C0H 0F23FH 02F80H CMNDLG_CODE            BC_CODE
 0F240H 1021EH 00FDFH CMNDLGF_CODE           BC_CODE
...
 Overlay 1H
 00000H 00265H 00266H CHRTATTR_CODE          BC_CODE
 Overlay 2H
 00000H 01415H 01416H CHRTDATA_CODE          BC_CODE
 Overlay 3H
 00000H 000C7H 000C8H CHRTFONT_CODE          BC_CODE
 Overlay 4H
 00000H 000C7H 000C8H CHRTTYPE_CODE          BC_CODE
 Overlay 5H
 00000H 03BF8H 03BF9H CHRTSUBS_CODE          BC_CODE
...
```

Die Sprungadresse in Abbildung 9–1 lautete `(0001) 00fa`. Wir haben also im Bereich des Overlays 1 nach einem Segment zu suchen, zwischen dessen Anfang und Ende die Adresse 00fa liegt, was hier nicht sonderlich schwer fällt, da es nur ein Segment gibt: CHRTATTR_CODE. Es kommt aber auch durchaus vor, daß man eine Adresse nicht entschlüsseln kann. Die auslösende Adresse hieß in der Abbildung `(0005) 2588:a125`. Die vierstellige Zahl vor dem Doppelpunkt, die Segmentadresse, ist für uns bedeutungslos; wir würden also im Overlay 5 nach einem Segment suchen, das die Adresse a125 einschließt, und das existiert nicht. Ich werde später zeigen, warum.

Nun haben wir das Segment unseres Sprungziels entschlüsselt. (Sie sehen, daß ich als Beispielanwendung das Programm CHRTDEMO mit Overlays kompi-

liert habe.) Es heißt CHRTATTR_CODE. Es fand also ein Aufruf einer Prozedur in CHRTATTR (.BAS oder .FRM) statt. Welche Prozedur es ist, kann man anhand der einfachen .MAP-Datei noch nicht feststellen.

...und vom Segment zur Zeile

Um das zu erreichen, müssen Sie Ihre Programme mit dem Switch /zd oder /zi kompilieren und LINK mit dem Switch /LI aufrufen. Die Compiler-Switches veranlassen den Compiler, die Zeilennummern der Quelldateien mit in die OBJ-Dateien einzubinden (die Dateien müssen dabei mit „Speichern unter“ als Textdateien und nicht als Binärdateien gespeichert werden). LINK wird durch /LI angewiesen, die Zeilennummern und ihre zugehörigen Adressen in die .MAP-Datei zu schreiben. In einer solchen .MAP-Datei findet sich dann nach der oben gezeigten Segmentliste ein Zeilennummernbereich (gekürzt):

```
Line numbers for chrtdemo.obj(CHRTDEMO.FRM) segment CHRTDEMO_CODE
   305 0000:003C   307 0000:0047   308 0000:0053   309 0000:0060
   312 0000:0071   315 0000:0076   319 0000:007B   320 0000:0084
...

Line numbers for chrtattr.obj(CHRTATTR.FRM) segment CHRTATTR_CODE
   185 0000:003C   186 0000:0047   187 0000:0058   188 0000:0065
   192 0000:0070   193 0000:007B   194 0000:008C   195 0000:0099
   198 0000:00A4   199 0000:00AF   200 0000:00EF   204 0000:00FA
   205 0000:0105   206 0000:012C   207 0000:0164   208 0000:0167
...
```

Zu jedem Segment finden sich hier die Adressen jeder Zeilennummer des Quellcodes. Wir suchen immer noch die Adresse 00fa im Segment CHRTATTR_CODE und werden hier fündig: Es handelt sich um Zeile 204 im Modul CHRTATTR.FRM. Wenn man jetzt noch weiß, daß die Zeilen hier immer von der ersten Zeile im Modul – und nicht in der Prozedur – gezählt werden, findet man leicht die Zeile in CHRTATTR.FRM, die Zieladresse des Prozeduraufrufs war:

```
SUB lstItems_Click()
```

Hierhin wurde also gesprungen – in die Prozedur *lstItems_Click*. Nun wird auch klar, warum wir die Adresse, von der aus dieser Aufruf getätigt wurde, nicht ermitteln konnten: Es gibt im Programm gar keine Stelle, an der die Prozedur *lstItems_Click* aufgerufen wird. Es handelt sich um eine Ereignisprozedur, die von VBDOS selbst aktiviert wurde, als der Benutzer auf das Steuerelement *lstItems* klickte.

Hätte es sich um einen gewöhnlichen Prozeduraufruf gehandelt, so wäre es uns auch gelungen, eine Zeile für die Aufrufadresse auszumachen. Beim Suchen der Aufrufadresse muß allerdings nicht die Zeilennummer aus der MAP-Datei, die der Adresse entspricht, sondern die davorliegende Zeilennummer gewählt werden.

Das Ganze bitte automatisch...

geht natürlich auch. Um Ihnen allzuviel Sucherei in diversen Dateien und Adressen zu ersparen, habe ich ein Programm geschrieben, das die Ausgabe des TRACE-Programms entschlüsselt und dabei genau so vorgeht, wie ich das hier gezeigt habe. Es ist auf der beiliegenden Diskette enthalten.

Bedingungen für den Einsatz von TRACELST.BAS sind:
- die Quelldateien müssen mit /zi oder /zd compiliert sein;
- beim Linken müssen /LI und die Library VBDTRACE.LIB angegeben werden;
- das Programm muß ausgeführt worden sein, damit MOVE.TRC erzeugt wird;
- das TRACE-Programm muß gestartet worden sein, und seine Ausgabe muß in eine Datei umgeleitet werden, die den gleichen Namen wie die MAP-Datei von LINK, aber die Extension .TRC hat (z.B. `TRACE > CHRTDEMO.TRC`);
- die Quelldateien müssen im aktuellen Verzeichnis vorhanden sein.

TRACELST.BAS ist ein Experimentalprogramm; es ist nicht auf Perfektion und schnellstmögliche Analyse ausgerichtet, sondern vielmehr darauf, daß Sie leicht Änderungen vornehmen können (z. B. wenn Sie auch noch eine Statistik benötigen, welches Overlay wie oft aus welchem anderen Overlay aufgerufen wird).

Die (gekürzte) Ausgabe von TRACELST.BAS sieht für unser Beispiel so aus:

```
Prozedur mnuChartOptions_Click aus Modul CHRTDEMO.FRM (Root):
   CALL GetFonts (8)
   verursacht durch einen Aufruf an:
   Overlay 5, Modul CHRTSUBS.BAS, Prozedur GetFonts
   folgende Aktion(en):
   - Keine, da Overlay 5 bereits geladen ist.

VBDOS an Adresse A125h im Modul CHRTSUBS.BAS (Overlay 5)
   verursacht durch einen Aufruf an:
   Overlay 1, Modul CHRTATTR.FRM, Prozedur lstItems_Click
   folgende Aktion(en):
   - Overlay 3 wird aus dem Heap in das XMS ausgelagert
   - Overlay 4 wird aus dem Heap in das XMS ausgelagert
   - Overlay 1 wird von der Platte nachgeladen
```

⇨

```
VBDOS an Adresse A125h im Modul CHRTATTR.FRM (Overlay 1)
   verursacht durch einen Aufruf an:
   Overlay 1, Modul CHRTATTR.FRM, Prozedur txtTitle_Change
   folgende Aktion(en):
   - Keine, da Overlay 1 bereits geladen ist.

(...)

Overlay  Treffer    Laden          Löschen aus Heap    Löschen aus EMS/XMS
    1       25      1 (1+0+0)      1 (0+0+1)           0
    2        7      1 (1+0+0)      0 (0+0+0)           0
    3        0      2 (1+0+1)      1 (0+0+1)           0
    4        0      2 (1+0+1)      1 (0+0+1)           0
    5       34      1 (1+0+0)      0 (0+0+0)           0

(a+b+c): a ist Anzahl der Platten-, b Anzahl der EMS- und c Anzahl der
         der XMS-Zugriffe; beim Laden wird angegeben, woher geladen wird,
         beim Löschen, ob das Overlay in den EMS/XMS ausgelagert wird oder
         nur noch auf der Platte steht.
```

Falls die Adresse, von der aus der Aufruf erfolgte, ermittelt werden kann, gibt TRACELST die Prozedur und die Zeile (in Klammern dahinter die Zeilennummer innerhalb der Prozedur) aus. Bei ereignisgesteuerten Programmen tritt VBDOS selbst natürlich häufig als Verursacher von Prozeduraufrufen auf; bei Programmen ohne Ereignissteuerung kommt das in der Regel nicht vor.

9.4 Die verschiedenen Konzepte im Vergleich

Abschließend möchte ich die drei Möglichkeiten, große Programmsysteme aufzubauen, noch einmal grafisch vergleichen.

Die erste Abbildung zeigt den Strukturplan eines (fiktiven) integrierten Paketes, das mit BASIC nach dem CHAIN-Modell (ohne Runtime-Modul) erstellt wurde. Es besteht aus vier Hauptfunktionen: Datenbank, Textverarbeitung, Tabellenkalkulation und Grafik.

Jede dieser Funktionen ist in einem eigenen EXE-File realisiert. Außerdem gibt es noch das Hauptprogramm, das die Initialisierung übernimmt, den Benutzer zwischen den vier Programmteilen wählen läßt und mittels CHAIN das entsprechende Programm lädt. Ich nehme an, daß die Programme so aus Source-Dateien zusammengesetzt sind, wie das Schaubild es zeigt:

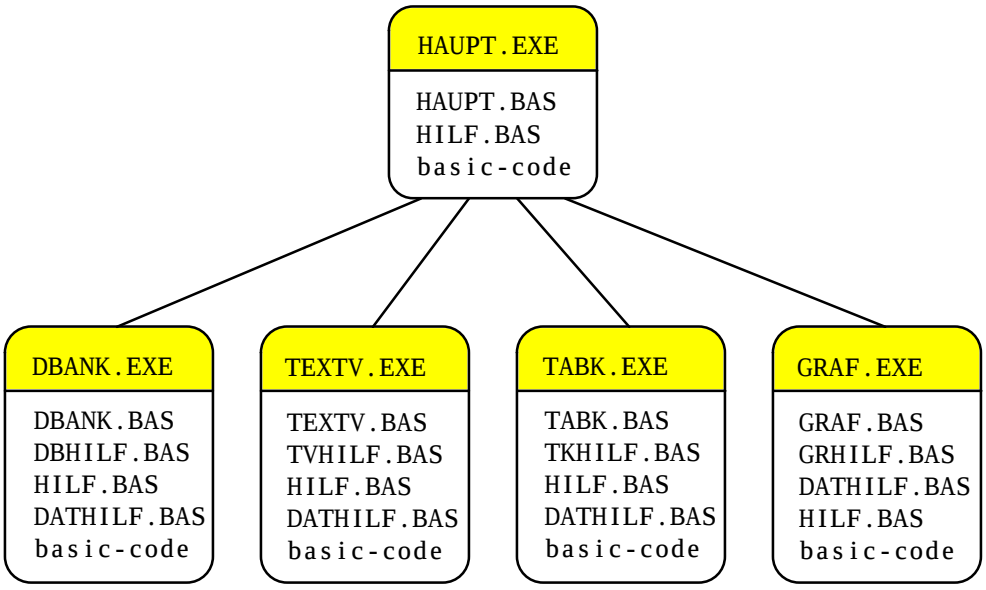


Abbildung 9–2: Programme, die sich mit CHAIN aufrufen

HAUPT.BAS, DBANK.BAS, TEXTV.BAS, TABK.BAS und GRAF.BAS sind die eigentlichen Programme; DBHILF.BAS, TVHILF.BAS, TKHILF.BAS und GRHILF.BAS sind Hilfsroutinen für die entsprechenden Programmteile; HILF.BAS enthält allgemeine Hilfsroutinen (Menü, Mausunterstützung, Eingabe usw.), und DATHILF.BAS enthält einige Routinen zur Dateimanipulation. Als „basic-code“ sind in diesem und den folgenden Schaubildern die BASIC-Hilfsroutinen bezeichnet, die jedem Programm auf irgendeine Weise zur Verfügung stehen müssen.

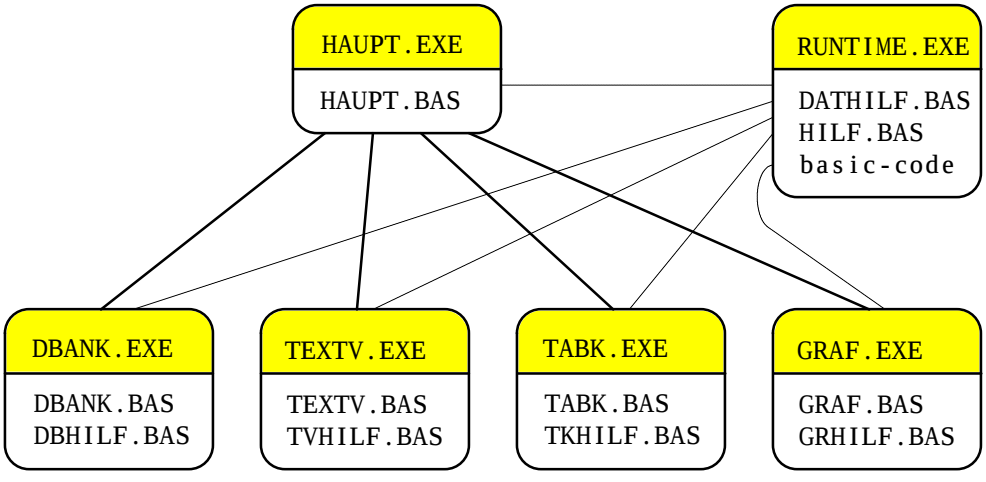


Abbildung 9–3: Programme mit CHAIN und Runtime-Modul.

In dieser Abbildung sehen Sie den Plan für dasselbe Programm, diesmal aber mit einem Runtime-Modul. Es ist leicht zu erkennen, daß diese Version schon sehr viel effizienter arbeitet. Sie benötigt weniger Platz auf der Festplatte und

ist auch schneller, da das Runtime-Modul nicht dauernd nachgeladen wird. Die Routinen, die vorher insgesamt fünfmal – für jedes EXE-Programm einmal – auf der Platte standen, wurden in das Runtime-Modul gesteckt, wo sie nur einmal Platz belegen.

Von der Hauptspeicherauslastung her gesehen ist diese Lösung allerdings etwa identisch mit der zuerst gezeigten, da die Routinen etwa gleichviel Speicher brauchen, egal, ob sie aus dem Runtime-Modul kommen oder direkt im EXE-Programm standen.

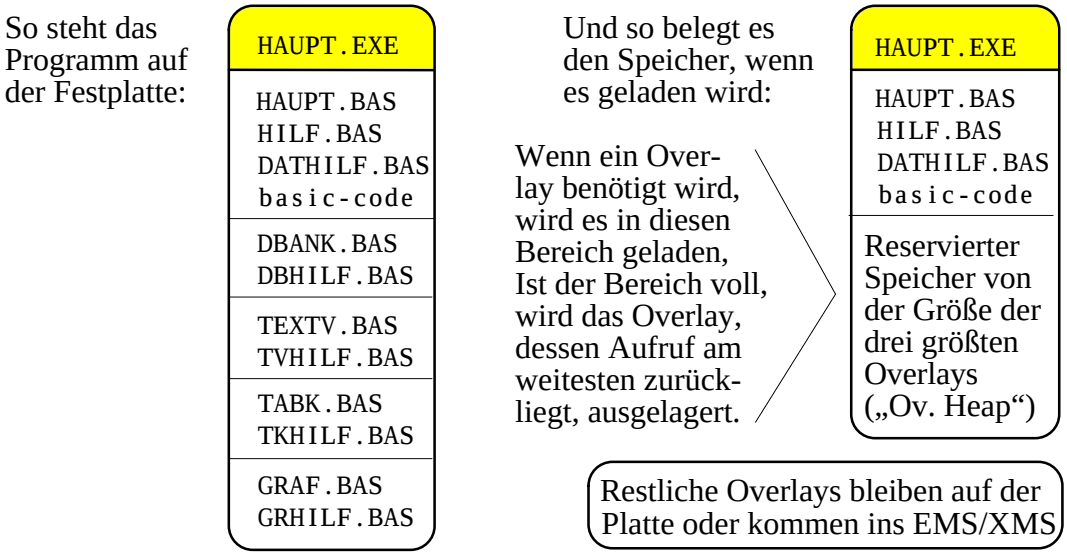


Abbildung 9–4: Programm mit Overlays

Diese Grafik schließlich stellt für diese Anwendung die beste Lösung vor: Nur noch ein einziges EXE-File beinhaltet alle Programmteile. Als Overlays werden die einzelnen Funktionskomplexe bei Bedarf in den Speicher geholt. Der korrekte LINK-Befehl hierfür wäre:

```
LINK HAUPT+HILF+DATHILF+(TEXTV+TVHILF)+(DBANK+DBHILF)+
     (TABK+TVHILF)+(GRAF+GRHILF);
```

Obwohl der Overlay-Manager, den LINK in das EXE-File einbindet, auch einige Bytes belegt, wird dieses Modell das sparsamste unter allen dreien sein, da es statt sechs Dateien (wie das Runtime-Modell) nur eine einzige benötigt und so einiges an EXE-Überhang wegfällt.

Dateien 10

Anhand eines konkreten Beispiels sollen hier verschiedene Möglichkeiten dargestellt werden, Daten auf der Diskette bzw. Festplatte zu verwalten. Betrachten Sie die auf einen konstruierten Fall bezogenen Ideen und Überlegungen als einen Pool von Möglichkeiten, aus dem Sie die freie Auswahl haben, wenn es darum geht, ein bestimmtes Speicherungsproblem zu lösen.

Das ISAM-Datenbanksystem, das in die professionelle Ausgabe von VBDOS integriert ist, wird in diesem Kapitel nicht berücksichtigt. Für viele Anwendungen stellt es sicherlich eine zumindest erwägenswerte Lösung dar; mehr über ISAM erfahren Sie in Kapitel 12.

Ich betrachte als Beispiel ein Programm, das es dem Benutzer ermöglicht, vorgegebene Formulare mit dem Drucker zu beschriften. Es muß gleich vier Datenbestände verwalten: Eine eigene Parameterdatei (auch Initialisierungsdatei genannt), die Druckertreiber, die Formularbeschreibungen (also Informationen, welches Feld auf einem bestimmten Formular wo steht, zum Beispiel „Feld Kontoinhaber, 10 cm von oberer Papierkante, 1 cm von linker Papierkante, Länge 5 cm“) und die Formularbeschriftungen, die einem gegebenen Formular einen bestimmten Text zuordnen und von denen es pro Formular beliebig viele geben kann.

10.1 Parameterdatei

Charakteristika

Eine Parameterdatei wird zumeist Informationen enthalten wie *Welcher Bildschirm ist angeschlossen?, Wo sind die Daten abgespeichert?, Welcher Drucker wird benutzt?* usw. Sie ist also recht kurz, und es bietet sich an, die Datei beim Starten des Programms komplett in den Speicher einzulesen, am besten in Variablen, die mit COMMON SHARED sämtlichen Modulen und Subroutinen verfügbar gemacht werden.

Form als Datenspeicher

In Zusammenhang mit Formen gibt es allerdings eine sehr interessante Alternative hierzu: Zumeist muß ohnehin eine Form namens „Einstellungen“ o.ä. erstellt werden, die es dem Benutzer erlaubt, die Standardeinstellungen zu verändern. Da wäre es ein doppelter Aufwand, globale Variablen und zugleich die

Steuerelemente zu speichern, weil diese ja in ihrer *Text*- bzw. *Value*-Eigenschaft die jeweilige Einstellung ohnehin speichern. Man kann also *anstelle* von globalen Variablen direkt auf die Steuerelementeigenschaften zugreifen und spart sich dadurch einigen Programmieraufwand.

Eine Methode, Steuerelement-Eigenschaften effizient zu speichern, finden Sie weiter hinten in diesem Kapitel.

Zurückschreiben der Datei

Für das Zurückschreiben der Parameterdaten auf die Platte – sofern der Benutzer die Möglichkeit hat, die Daten innerhalb des Programms zu ändern oder das Programm Dinge wie *Welche Datei wurde zuletzt bearbeitet?* darin speichert – gibt es schon zwei Möglichkeiten. Entweder man speichert erst, wenn der Benutzer das Programm beendet, oder man speichert sofort, wenn sich irgendetwas verändert hat.

Die erste Methode ist zweifelsohne schneller, da nur ein einziges Mal gespeichert wird. Wenn allerdings ein Stromausfall oder das unachtsame Abschalten des Rechners das Programm gewaltsam unterbrechen, sind die Änderungen verloren. Die zweite Methode hat diesen Nachteil nicht, dafür aber den, daß sie langsamer ist, weil sie unter Umständen recht häufig auf den externen Datenträger zugreifen muß.

Die beste Lösung ist hier oft ein Kompromiß, z. B. das Speichern immer dann, wenn der Benutzer das Änderungsmenü verläßt bzw. die Form schließt.

Wo speichern?

Eine Parameterdatei zeichnet sich zumeist dadurch aus, daß sie Daten enthält, die so wichtig sind, daß das Programm ohne sie nicht starten kann. Ist das der Fall, dann ist es sinnvoll, in das Programm einen kleinen „Generator“ einzubauen, der eine neue Parameterdatei mit Standardwerten erzeugt, falls die alte nicht gefunden wird.

Ferner stellt sich die Frage, wo die Parameterdatei am günstigsten abzuspeichern ist. Schreibt man einfach mit `OPEN "PROGRAMM.PRM"...` drauflos, dann wird die Datei im aktuellen Directory geschrieben bzw. gesucht – erfolglos, wenn der Benutzer das Programm-Directory im PATH genannt und das Programm von einem anderen Directory aus aufgerufen hat. Die Parameterdatei aus dem Programm-Directory würde dann nur gefunden, wenn dieses auch mit dem DOS-Befehl APPEND verfügbar gemacht wurde; geschrieben würde sie aber auf jeden Fall in das aktuelle Verzeichnis, und das ist nicht erwünscht.

Am einfachsten ist es, einen Standard-Directory-Namen zu benutzen und so den Benutzer zu zwingen, für das Programm genau dieses Directory anzulegen – oder man speichert seine Parameterdatei einfach immer im Hauptdirectory C:\ ab. Beides ist nicht das Gelbe vom Ei, denn es könnte dabei Schwierigkeiten mit dem Laufwerk geben (der Benutzer könnte das Programm ja auf seinem Laufwerk F: installieren wollen...).

Eine Lösung für dieses Problem liegt darin, zunächst das aktuelle Directory auf Vorhandensein der Parameterdatei zu prüfen und – wenn das fehlschlägt – danach alle Directories, die im PATH eingetragen sind (dieser kann ja mit ENVIRON$ ermittelt werden). In einem dieser Verzeichnisse muß zumindest das Programm selbst sein, es sei denn, der Benutzer hat es mit einem Konstrukt wie `C:\MAMA\PROGRAMM` aufgerufen – und das vertragen die meisten Programme nicht.

Die folgende Routine prüft das aktuelle und alle im PATH enthaltenen Verzeichnisse auf Vorhandensein einer Datei und gibt das Verzeichnis zurück, in dem sie zuerst gefunden wurde:

```
FUNCTION FindePfad (DateiName AS STRING) AS STRING

   ' Diese Konstante kann geändert werden, wenn z.B. die mit SET LIB= oder
   ' SET HELPFILES= o.ä. gesetzten Verzeichnisse durchsucht werden sollen
   CONST DurchsucheVariable = "PATH"

   DIM Pfad AS STRING, Path AS STRING, Semikolon AS INTEGER, Datei AS INTEGER
   Pfad = CURDIR$: IF RIGHT$(Pfad, 1) <> "\" THEN Pfad = Pfad + "\"
   Path = UCASE$(ENVIRON$(DurchsucheVariable)) + ";"

   ON LOCAL ERROR GOTO OpenFehler
   Datei = FREEFILE
   OPEN Pfad + DateiName FOR INPUT AS #Datei
   CLOSE Datei
   FindePfad = Pfad: EXIT FUNCTION

OpenFehler:
   Semikolon = INSTR(Path, ";")
   IF Semikolon = 0 THEN ' Alle durchsucht, nix gefunden: leeren String zurückgeben
      FindePfad = "": EXIT FUNCTION
   ELSE
      Pfad = RTRIM$(LEFT$(Path, Semikolon - 1)) ' erste Angabe extrahieren
      Path = LTRIM$(MID$(Path, Semikolon + 1))  ' und aus Path löschen
      IF RIGHT$(Pfad, 1) <> "\" THEN Pfad = Pfad + "\"
   END IF
   RESUME

END FUNCTION
```

Listing 10–1: FINDEPFD.BAS

Geht man davon aus, daß der Benutzer Programm und Parameterdatei ins gleiche Directory kopiert hat, ist das Problem nun gelöst. Das gefundene Verzeichnis wird in einer Variablen aufbewahrt, denn dort können vermutlich auch die meisten anderen Programm-Dateien gefunden werden, und dorthin wird auch die Parameterdatei später zurückgeschrieben.

Datei als Teil des Programms

Man kann sogar völlig verhindern, daß der Benutzer die Parameterdatei beim Kopieren versehentlich „verliert“ oder daß sie sich in einem anderen Verzeichnis als das Programm befindet: Man schreibt die Parameterdaten einfach direkt in das EXE-File des Programms. Oder besser: *an* das EXE-File. Dazu programmiert man den Parameter-Einlese-Teil so, daß er zunächst (nach oben beschriebener Methode) das Programm findet, dann (mit OPEN FOR BINARY und LOF) dessen Größe ermittelt und von dieser Größe die Gesamtlänge der Parameterdaten abzieht. Er erhält dann den Offset innerhalb des EXE-Files, an dem er beginnen kann, die Parameter einzulesen, was mit OPEN FOR BINARY ja kein Problem mehr ist.

Wenn die Gesamtlänge der Parameterdaten nicht konstant ist, muß man in den letzten zwei Zeichen des EXE-Files vermerken, wieviele Bytes vom Programmende aus gesehen zu den Parameterdaten zählen (zwei Zeichen reichen für Zahlen bis 32.768, wenn man PUT direkt auf die INTEGER-Zahl anwendet oder MKI$ benutzt). Die Kürzung der Datei, die erforderlich ist, wenn sich der Umfang des „Parameter-Anhängsels“ verringert, läßt sich mit einem Interruptaufruf durchführen (vgl. Kapitel 22).

Selbstverständlich muß bei solchen Tricks nach einer Neuerstellung der Programm-EXE-Datei zuerst mit einem Spezialprogramm (im Notfall geht das auch mit `COPY PROGRAMM.EXE/B+PARMS.DAT/B PROG.EXE`) ein kompletter Parameter-Satz hinten angefügt werden. Dies ist aber nur ein einziges Mal nötig, sozusagen als Teil des Kompiliervorgangs.

Daß Sie das Programm dadurch um Nicht-Maschinensprache-Daten verlängern, macht nichts aus. Die Daten werden zwar beim Start mit in den Speicher geladen, aber nicht ausgeführt, da keine Sprunganweisung auf sie zeigt.

Die einzige Schwierigkeit dieses Verfahrens ist, daß Anti-Viren-Programme, die beim Benutzer installiert sind, eventuell Alarm schlagen, wenn versucht wird, in ein EXE-File zu schreiben. Der Benutzer sollte also zumindest auf diese Eigenart aufmerksam gemacht werden.

Art der Speicherung

Prinzipiell gibt es vier Möglichkeiten, Daten in einer Datei abzuspeichern. Da wäre zuerst einmal die ISAM-Methode. Sie scheidet für Parameterdateien von vornherein aus, da die kleinstmögliche ISAM-Datei 64 KB umfaßt. Auch die RANDOM-Methode entfällt, denn sie erfordert eine konstante Satzlänge, und in einer Parameterdatei sollen ja verschiedenste Daten gespeichert werden. Es verbleiben noch die Möglichkeit einer Textdatei (mit OPEN FOR OUTPUT) und die einer BINARY-Datei. Die ASCII-Datei wäre für diese Aufgabe zwar ein recht flexibles Konzept, aber es ist in vielen Fällen nicht zweckmäßig, dem Benutzer die Möglichkeit zu geben, mit einem gewöhnlichen Editor die Daten manipulieren zu können.

Um verschiedene Datentypen in beliebiger Reihenfolge in einer BINARY-Datei abspeichern zu können, können Sie die GET- und PUT-Befehle benutzen. Mit ihnen lassen sich Variablen von fester Länge direkt schreiben und einlesen:

```
PUT #1, , a%
PUT #1, , b&
PUT #1, , c@
```

Diese Befehle schreiben in die Datei mit der Nummer 1 die angegebenen Variablen (jeweils an der aktuellen Position, die nach dem Öffnen 1 ist und dann automatisch erhöht wird). Später können Sie mit GET die Daten ebenso aus der Datei auslesen. Da es sich um Daten mit fester Länge handelt (*b&* benötigt zum Beispiel immer vier Bytes), entstehen keine Schwierigkeiten. Lediglich für die Ein- und Ausgabe von Strings mit variabler Länge benötigt man eine gesonderte Routine. Sie könnten zwar einfach mit PUT geschrieben werden, aber dann wüßte man beim Einlesen nicht mehr, wieviele Bytes zum String gehören, und käme mit dem Rest der Datei durcheinander:

```
SUB PutS (FileNummer AS INTEGER, Text AS STRING)

   x% = LEN(Text) ' PUT #FileNummer,,LEN(Text) ist nicht erlaubt, deshalb der Umweg
   PUT #FileNummer, , x%: PUT #FileNummer, , Text

END SUB
```

```
SUB GetS (FileNummer AS INTEGER, Text AS STRING)

   GET #FileNummer, , x%
   Text  = SPACE$(x%): GET #FileNummer, , Text ' Hierdurch werden x% Byte gelesen

END FUNCTION
```

Listing 10–2: GETPUTS.BAS

Diese Routinen stellen dem String eine INTEGER-Zahl voran, die seine Länge angibt. Sie können so problemlos beliebige Daten in beliebiger Reihenfolge in eine Datei schreiben oder aus einer Datei lesen, und deshalb dürfte die BINARY-Methode sich am besten für eine Parameterdatei eignen.

Datenblockgröße

Bei allen Überlegungen zur Dateigröße müssen Sie bedenken, daß Dateien auf Datenträgern niemals ihre wahre Größe belegen, sondern immer ein Vielfaches der Cluster-Größe. Diese beträgt bei Festplatten zumeist 2048 oder 4096 Byte; modernere CHKDSK-Versionen zeigen sie an („4096 Byte in jeder Zuordnungseinheit“). Mit der in Kapitel 22 vorgestellen Routine zum Ermitteln des freien Platzes auf einem Datenträger können Sie die Größe ebenfalls ermitteln.

Auch die kleinste Datei belegt also mindestens einen vollen Cluster.

10.2 Druckertreiber

Charakteristika

Für unser Beispielprogramm geht es als nächstes um die Speicherung von Druckertreibern. Die Information über einen Drucker besteht aus seiner Bezeichnung sowie einer Anzahl von zusätzlichen Informationen, nehmen wir an, etwa rund 1 KB pro Drucker. Der Benutzer muß die Möglichkeit haben, bestehende Druckertreiber zu verändern und eigene zu definieren; das Programm muß stets auf eine Liste aller gespeicherten Treiber zugreifen können.

Eine oder mehrere Dateien?

Eine prinzipielle Entscheidung muß vorweg getroffen werden: Sollen alle Druckertreiber in einer einzigen Datei gespeichert werden, oder sollte man für jeden Drucker eine eigene Datei anlegen? Für eine einzige Datei spricht der kleinere Verbrauch an Platz auf der Festplatte bzw. Diskette. Bei einzelnen Dateien könnte man andererseits darauf verzichten, alle auf der Platte zu installieren. Der Benutzer müßte nur diejenigen kopieren, die er auch benötigt. Es könnte allerdings schwierig sein, für die einzelnen Dateien sinnvolle Namen zu finden.

Gegen eine einzige Datei spricht wiederum folgendes: Was ist, wenn jemand drei eigene Druckertreiber definiert hat, und dann entweder von einem Freund dessen Treiber übernehmen will oder vom Hersteller (also Ihnen) eine neue Version des Programms bekommt? Dann müßte er entweder die neue Drucker-

datei übernehmen und damit seine eigenen Definitionen überschreiben, oder er verwendet ein speziellen Konvertierprogramm. Mit einzelnen Dateien ist so etwas wiederum kein Problem. Die Dateien könnten in sich so aufgebaut sein wie die Parameterdatei, die ich oben behandelt habe – BINARY mit GET und PUT. Das dürfte auch hier der effizienteste Weg sein.

Verzeichnis vorhandener Druckertreiber (Daten)

Um dem Benutzer die Auswahl eines Druckers aus einer Liste zu ermöglichen, benötigt man ein Verzeichnis, eine Tabelle, die Auskunft darüber gibt, welches Druckermodell in welcher Datei gespeichert ist. Diese Tabelle darf nicht statisch sein, denn das Programm soll flexibel reagieren, wenn der Benutzer „von Hand“ eine der Druckerdateien löscht oder eine zusätzliche von irgendwo kopiert. Das Verzeichnis anzulegen, kann allerdings etwas länger dauern, da jede einzelne Druckerdatei geöffnet werden muß, um aus ihr die zugehörige Druckerbezeichnung zu lesen. Deshalb wäre es praktisch, wenn eine solche Tabelle immer vorhanden wäre und nur bei Veränderungen am Bestand der vorhandenen Druckerdateien neu erstellt würde.

Das Problem kann wie folgt gelöst werden: Wir benötigen nicht nur die Druckerdateien selbst, sondern zwei zusätzliche Informationsdateien: Eine, die das sortierte Inhaltsverzeichnis (nur Dateinamen, Größe und Datum) des Druckerdateibestandes enthält, und eine zweite, eine Druckertabelle, in der jedem Dateinamen eine Druckerbezeichnung zugeordnet ist.

Wenn das Programm gestartet wird, prüft es, ob eine Druckertabelle auf der Platte ist. Wenn nicht, wird sie neu erzeugt. Ist eine vorhanden, dann wird nur ein Verzeichnis aller vorhandenen Druckerdateien erzeugt (ohne Daten *aus* ihnen zu lesen). Dieses wird – mit Größe und vielleicht auch Datum – sortiert. Dann wird mit Hilfe dieses Verzeichnisses und der Datei, die die Verzeichnisliste vom letzten Aufruf enthält, festgestellt, ob sich am Druckerbestand etwas verändert hat. Wenn ja, wird die Druckertabelle neu erzeugt.

Verzeichnisliste wie Druckerdatei können, da es sich bei beiden um relativ kleine Datenbestände handelt, die eine feste Länge haben, als RANDOM-Dateien gespeichert werden. Der Zugriff erfolgt mit GET und PUT und je einem selbstdefinierten Datentyp.

Für den relativ kleinen Aufgabenbereich „Druckertreiber“ mag diese Lösung etwas aufgeblasen scheinen. Aber denken Sie zum Beispiel an ein Datenverwaltungsprogramm, das Hunderte von Datenreihen in einzelnen Dateien speichert und zu jeder Datenreihe einen 60stelligen Titel besitzt. Da lohnt es sich durchaus, solche Methoden zu implementieren, wenn man dem Benutzer lange War-

tezeiten ersparen will. Man kann dann sogar so weit gehen, daß man, wenn sich am Datenbestand etwas geändert hat, nicht das ganze Verzeichnis neu erzeugt, sondern nur die Änderungen übernimmt, oder daß man, wenn das Programm selbst etwas am Bestand ändert (d.h. im Beispiel einen neuen Druckertreiber erstellt), die Verzeichnisdateien schnell aktualisiert.

Wartezeiten

Wo wir gerade beim Thema sind: Wenn Ihr Programm beim Start erst ein umfangreiches Verzeichnis erstellt und dafür einige Zeit braucht, können Sie u. U. die Bedienung angenehmer gestalten, wenn Sie das Programm ereignisgesteuert konzipieren und in der Prozedur, die das Verzeichnis erzeugt, einige DOEVENTS-Aufrufe einbauen. Dann hat der Benutzer die Möglichkeit, schon – zumindest begrenzt – mit dem Programm zu arbeiten, während „im Hintergrund“ noch das Verzeichnis erstellt wird. Er könnte dann alle Operationen schon ausführen, für die das Verzeichnis nicht unbedingt benötigt wird. Mehr zum Thema „Pseudo-Multitasking“ finden Sie im Kapitel 7.

10.3 Formularbeschreibungen

Nun geht es um die Verwaltung von Formularbeschreibungen. Dieses Problem ist schnell abgehandelt, denn es gleicht dem mit den Druckertreibern. Es handelt sich um relativ kleine Dateneinheiten mit einem kleinen Gesamtvolumen. Jede hat einen Namen, der sich vermutlich in den acht Zeichen des DOS-Namens nicht ausdrücken läßt, und – hier besonders wichtig – sie müssen problemlos zwischen verschiedenen Systemen austauschbar sein. In einer korrekten Formularbeschreibung steckt nämlich unter Umständen viel Meßarbeit, so daß es häufig erwünscht sein wird, eine Formularbeschreibung von jemandem zu kopieren, der sich bereits die Mühe gemacht hat.

Das Problem kann also ebenso gelöst werden wie das der Druckertreiber.

10.4 Formularbeschriftungen

Charakteristika

Bei dem Problem der Beschriftungen einzelner Formulare sind allerdings wieder andere Dinge zu bedenken. Im einfachsten Fall würde ein Formular im Rechner beschriftet, dann ausgedruckt und danach gelöscht. Die Beschriftung müßte nicht weiter gespeichert werden.

Ganz so einfach will ich es nicht machen. Man möchte sicherlich häufig eine alte Beschriftung nur mit kleiner Änderung nochmals drucken oder in alten Beschriftungen etwas nachsehen etc. Die Formularbeschriftungen sollen also gespeichert und aufbewahrt werden.

Hier muß man davon ausgehen, daß eine relativ große Anzahl von Beschriftungen mit einem ebensolchen Gesamtvolumen verwaltet werden muß. Einzelne Dateien zu verwenden, wie ich es bei den Formularbeschreibungen und Drukkertreibern tat, wäre hier deshalb ungünstig. Außerdem entfällt bei Formularbeschriftungen die Notwendigkeit, einzelne Beschriftungen von einem Rechner auf einen anderen zu kopieren.

Da jede Formularbeschriftung eindeutig einer existierenden Formularbeschreibung zugeordnet ist, könnte man die Beschriftungen entweder direkt zur Beschreibung des zugehörigen Formulars in die gleiche Datei packen oder zumindest pro existierender Formularbeschreibung eine Datei mit allen Beschriftungen dieses Formulars anlegen. Das letztgenannte wäre hier die praktikablere Lösung, da sie die Portabilität der Formularbeschreibungen selbst nicht einschränkt.

Auch nicht zu verachten ist allerdings die Möglichkeit, sämtliche Formularbeschriftungen in eine einzige Datei zu packen, da sich dann nicht gar so viele Dateien auf der Platte tummeln. Wie dem auch sei, das folgende gilt für beide Varianten.

Wie speichern?

ISAM eignet sich nicht für diese Aufgabe. Erstens sind 64 KB Minimum pro Datei hier nicht akzeptabel. Zweitens müssen für eine ISAM-Datei die Datenstrukturen (Anzahl und Länge der Felder im Forumlar) schon bei der *Erstellung* des Programms festliegen, weil eine entsprechende TYPE-Vereinbarung getroffen werden muß. Zwar könnte man eine Universal-Typvereinbarung benutzen, die für alle Formulare Anwendung finden kann:

```
TYPE FormularBeschriftungsTyp
   FormularName AS STRING * 25
   BeschriftungName AS STRING * 25
   Beschriftung(1 TO MaxFeld) AS STRING * MaxZeichenProFeld
END TYPE
```

Diese hat aber den Nachteil, daß auch für kurze Formulare und kurze Texteinträge immer das Maximum an Speicherplatz belegt würde. Jedes Formular benötigt so viel Speicherplatz, wie das längste Formular belegen darf. Deshalb ist es vom Platzaufwand her nicht vertretbar, die ISAM-Lösung zu wählen.

Eine zweite Möglichkeit, die man unter der Bedingung nutzen kann, daß eine Formularbeschriftung nicht mehr als 16 KB belegt, ist folgende: Eine Formularbeschriftung wird als ein String gespeichert. Die einzelnen Felder werden dabei durch ein Zeichen getrennt, das im Text nie vorkommt (zum Beispiel `CHR$(1)`). Dadurch ist nicht nur die Anzahl der Felder beliebig, sondern auch die Anzahl der Zeichen in einem Feld. Als erstes und zweites Feld könnte man den Namen der Beschriftung und – wenn man alle in eine Datei packt – den Namen des Formulars, zu der sie gehört, speichern.

Alle Beschreibungen könnten dann in einer sequentiellen Datei (mit OPEN FOR OUTPUT) als Strings hintereinander abgespeichert werden. Dabei gibt es natürlich auch Probleme:

- Beim Löschen einer Beschriftung muß der gesamte Inhalt der Datei in eine zweite Datei kopiert werden, wobei die zu löschende Zeile ausgelassen wird, oder man muß sich einen Trick einfallen lassen, mit dem eine Zeile als „gelöscht“ markiert werden kann (mit OPEN FOR BINARY ließe sich so etwas bewerkstelligen).
- Beim Suchen. Man kann entweder das binäre Suchen für Textdateien (vorgestellt in „Suchen“, Kapitel 20) benutzen, muß dann aber dafür sorgen, daß die Datei sortiert ist (bei der Erstellung einer neuen Beschriftung diese sofort einsortieren) oder aber jedesmal zum Suchen die ganze Datei Zeile für Zeile prüfen. Wenn man das binäre Suchen nicht benutzt, darf eine Beschriftung sogar bis zu 32 KB haben.

Welche dieser Möglichkeiten man wählt, hängt davon ab, wie man die Prioritäten setzt. Die Verwendung von ISAM verbraucht in dreierlei Hinsicht mehr Speicher. Die Datenfiles werden größer, das Programm selbst wird durch Hinzulinken der ISAM-Routinen deutlich länger, und während des Programmablaufs belegt ISAM für zusätzlich noch RAM-Speicherplatz.

Wenn Sie mit ASCII-Dateien arbeiten, wird nur soviel Speicher verbraucht, wie wirklich unbedingt nötig ist. Dafür zahlen Sie mit einem Verlust an Zeit, weil die Bearbeitung einer solchen Datei aus den schon genannten Gründen langsamer ist als der Umgang mit ISAM.

10.5 Text-Datenbank

Um noch ein weiteres Beispiel zur Datenspeicherung vorzustellen, nehme ich an, das Formular-Programm sollte Texte seiner Benutzeroberfläche nicht direkt enthalten, sondern aus einer Datei einlesen, so daß man nur die Textdatei auswechseln muß, damit das Programm in einer anderen Sprache bedient werden kann.

Ich gehe davon aus, daß es sich dabei um so viel Text handelt, daß es nicht möglich ist, alle Texte beim Start des Programms in ein Array einzulesen. Dieser Datenbestand, die Datei, aus der das Programm seine Texte liest, muß zwei Kriterien genügen: Erstens muß der Zugriff direkt erfolgen können, und zweitens muß er sehr schnell sein. Zweckmäßigerweise schreibt man eine Funktion, der man dann im Programm die Nummer des gewünschten Textes mitteilt, und die diesen Text dann aus der Datei liest und zurückgibt.

ISAM- und ASCII-Dateien scheiden für diese Aufgabenstellung aus, denn bei beiden kann kein direkter Zugriff auf einen bestimmten Datensatz erfolgen. Bei beiden ließe sich das allerdings mit einem Trick erreichen, indem man nämlich jedem Text eine laufende Nummer zuordnet, nach der man dann ja mit einem ISAM-SEEK-Befehl oder mit dem binären Suchen für Textdateien suchen kann. Ich will hier allerdings auf ein noch schnelleres und weniger platzaufwendiges Verfahren hinaus.

Eine Random-Access-Datei mit fester Satzlänge wäre die nächste Möglichkeit; der Nachteil hierbei ist, daß man sich auf eine maximale Textlänge festlegen muß und daß *jeder* Text dann diese Menge an Bytes belegt. Wenn sich die Textmenge in Grenzen hält, kann man diese Lösung wählen. Obwohl man, nachdem der Text eingelesen ist, noch überschüssige Leerzeichen am Ende abschneiden muß, ist sie die schnellste von allen vorgestellten Methoden.

Die eleganteste Lösung aber ist in etwa eine Kombination aus dem ASCII- und dem Random-Access-Verfahren. Sie erinnert an die Abhandlung der Parameter-Datei:

Der Text-Compiler

Die Datei mit den Texten enthält am Anfang für jeden gespeicherten Text einen 6-Byte-Datenblock, der aus einer LONG-Zahl für die Adresse und einer INTEGER-Zahl für die Länge des Strings besteht. Die Adresse beschreibt die Byte-Position in der Datei, an der der String beginnt. Nach diesem Index-Bereich, der, wie gesagt, pro Text 6 Bytes umfaßt, folgen alle Texte ohne Trennzeichen hintereinander.

Es wird also erst aus dem Indexbereich die Adresse des gesuchten Textes und danach aus dem Textbereich der Text selbst gelesen. Diese Methode ist etwas langsamer als die reine Random-Access-Methode, aber, wie Sie sich sicher denken können, wesentlich sparsamer mit dem Speicherplatz.

Die Routine, die den Text aus der Datei lesen soll, könnte dann so formuliert werden:

```
FUNCTION LiesText$(Nummer AS INTEGER)

   DIM IndexPosition AS INTEGER, Offset AS LONG, Laenge AS INTEGER
   SHARED TextFile AS INTEGER

   IndexPosition = (Nummer - 1) * 6 + 1 ' feststellen, wo der Indexeintrag steht
   GET #TextFile, IndexPosition, Offset: GET #TextFile, , Laenge

   x$ = SPACE$(Laenge): GET #TextFile, Offset, x$

   LiesText$ = x$

END FUNCTION
```

Listing 10–3: TEXTCOMF.BAS

Der Funktion wird die laufende Nummer des Textes, den sie einlesen soll, übergeben. Außerdem benötigt sie eine Variable namens *TextFile* im Hauptprogramm, die die Nummer der Text-Datei enthält. Diese muß vor dem ersten Funktionsaufruf bereits mit OPEN geöffnet sein.

Das größte Problem beim Umgang mit derartigen Textdatenbanken ist die *Herstellung* der Datei mit Index- und Datenbereich. Dazu finden Sie auf der Diskette neben der abgedruckten Routine das Programm TEXTCOM.BAS, das diese Aufgabe wahrnimmt.

10.6 Formen und Steuerelemente speichern

Häufig ist es notwendig, die Eigenschaften einer Form oder die Eigenschaften von Steuerelementen zu speichern. Wenn Sie z. B. (durch die Eigenschaft *BorderStyle*) dem Benutzer erlauben, die Formen Ihres Programms auf dem Bildschirm zu verschieben und ihre Größe zu verändern, so wäre es ja wünschenswert, daß der Bildschirm beim nächsten Programmstart nicht wieder nach „Standard“ aufgebaut ist, sondern die Benutzereinstellung beibehalten wurde.

Position und Größe von Formen

Diese Informationen zu speichern, ist an sich kein großes Problem; es handelt sich ja ausnahmslos um Zahlen, die ins INTEGER-Format passen. Leider können Eigenschaften nicht direkt mit GET und PUT verarbeitet werden; statt `PUT #1,,HauptForm.Left` muß es also heißen: `x%= HauptForm.Left: PUT #1,,x%`. Es bietet sich an, für solche Zwecke eine Prozedur zu formulieren:

```
SUB SchreibeGroesse (Datei AS INTEGER, Objekt AS FORM)

   DIM Byte AS STRING * 1
   Byte = CHR$(Objekt.Left): PUT #Datei, , Byte
   Byte = CHR$(Objekt.Top): PUT #Datei, , Byte
   Byte = CHR$(Objekt.Width): PUT #Datei, , Byte
   Byte = CHR$(Objekt.Height): PUT #Datei, , Byte

END SUB
```

Listing 10–4: (Auszug aus) FORMIO.BAS

Hier wird sogar zum Speichern nur ein Byte belegt (statt zwei für eine INTEGER-Zahl), da die Werte der entsprechenden Form-Eigenschaften ohnehin auf 255 begrenzt sind.

Das Lesen aus der Datei könnte analog funktionieren, wäre allerdings etwas langsam: Erstens würde die Form auf dem Schirm ihre neue Position und Größe in vier einzelnen Schritten annehmen, was u.U. etwas seltsam aussieht; zweitens werden hintereinander vier *Resize*-Ereignisse erzeugt. Schneller geht es so:

```
SUB LiesGroesse (Datei AS INTEGER, Objekt AS FORM)

   DIM FourByte AS STRING * 4
   GET #Datei, , FourByte
   Objekt.MOVE ASC(FourByte), ASC(MID$(FourByte, 2)), ASC(MID$(FourByte, 3)),  ↵
               ASC(MID$(FourByte, 4))

END SUB
```

Listing 10–5: (Auszug aus) FORMIO.BAS

Beide Routinen sind kompatibel, das heißt, was mit der einen geschrieben wird, kann mit der anderen gelesen werden.

Eigenschaften von Steuerelementen

Bei den Steuerelementen ist es nicht so einfach, eine allgemein verwendbare Prozedur zu schreiben. Verschiedene Steuerelemente haben verschiedene Eigenschaften, und außerdem möchten Sie evtl. nur einige wenige speichern. Die IF TYPEOF-Abfrage ist leider in der Anwendung etwas klobig und verursacht ziemlich umfangreiche Strukturen.

Eine Möglichkeit wäre, beim Speichern eines Steuerelements mit der Holzhammermethode vorzugehen und den Versuch zu unternehmen, alle Eigenschaften zu speichern, die es gibt. Die Eigenschaften, bei denen ein Fehler auftritt, sind offenbar für das Objekt nicht verfügbar.

Das mag in Einzelfällen ein praktikables Verfahren sein, insbesondere, weil es wirklich mit einer einzigen Prozedur alle Steuerelemente erschlägt; da das Speichern von Steuerelementen jedoch meist erfolgen wird, um die vom Benutzer veränderbaren Eigenschaften (bei Textfeldern *Text*, bei Kontroll- und Optionsfeldern sowie Bildlaufleisten *Value*) zu speichern, wäre evtl. auch eine solche Prozedur sinnvoll:

```
SUB SchreibWert (Datei AS INTEGER, Object AS CONTROL)

   DIM Byte AS STRING * 1, Intg AS INTEGER, Text AS STRING

   IF TYPEOF Object IS TextBox THEN
      Text = Object.Text: Intg = LEN(Text): PUT #Datei, , Intg: PUT #Datei, , Text
   ELSEIF TYPEOF Object IS OptionButton THEN
      Byte = CHR$(Object.Value + 128): PUT #Datei, , Byte
   ELSEIF TYPEOF Object IS CheckBox THEN
      Byte = CHR$(Object.Value + 128): PUT #Datei, , Byte
   ELSE
      Intg = Object.Value: PUT #Datei, , Intg
   END IF

END SUB
```

Listing 10–6: (Auszug aus) CTRLIO.BAS

Diese Routine schreibt nur die eine, veränderliche Eigenschaft in die angegebene Datei. (Das Gegenstück LiesWert befindet sich ebenfalls in der Datei auf der Diskette.)

Fehlerbehandlung 11

„Error trapping“ ist fast zwangsläufig Bestandteil jedes Programms. Wer möchte schon gerne, daß sein Programm sich plötzlich mit der Meldung `Eingabe über das Ende der Datei hinaus in Zeile 0 bei Adresse 2345:ABCD im Modul LOADFILE, Zum Fortfahren Taste drücken` oder etwas Vergleichbarem abmeldet? Schon jeder einfache BASIC-Interpreter bietet heutzutage Möglichkeiten des Error-Trapping. In VBDOS gehören zu den umfangreichen Fähigkeiten auf diesem Gebiet die Befehle ON ERROR, ON LOCAL ERROR, RESUME und ERROR, die Systemvariablen ERR und ERL und die Funktionen ERDEV, ERDEV$ und (neu hinzugekommen) ERROR$.

11.1 Die einfachste Lösung

Ein simples Error-Trapping erreicht man, indem man einfach den Befehl `ON ERROR GOTO Fehler` (oder ein anderes Zeilenlabel) an den Anfang des Programms setzt und irgendwo im Programm unter dem Zeilenlabel `Fehler` eine kleine Fehlerbehandlungsroutine schreibt. Zum Beispiel so:

```
ON ERROR GOTO Fehler

' hier ist eine Menge Programmcode
' (Das Label "Fehler" muß im Hauptprogramm, nicht in einem
' SUB stehen!)

Fehler:
   IF ERR = 7 THEN
      PRINT "Der Speicherplatz reicht nicht aus!"
   ELSE
      PRINT "Programmfehler Nr.";STR$(ERR);". Bitte rufen Sie"
      PRINT "uns an und teilen Sie uns die Umstände mit, unter"
      PRINT "denen der Fehler aufgetreten ist!"
   END IF
   SYSTEM
```

Listing 11–1: ERROR1.BAS

Diese Art der Fehlerbehandlung ist jedoch sicherlich nicht viel eleganter als die oben genannte Compiler-Meldung, selbst wenn man die Fehlermeldungen noch detaillierter gestaltet als in diesem Beispiel. Außerdem dürfen Formmodule keinen Code auf Modulebene enthalten, deshalb ist in ihnen ein globaler Error-Handler überhaupt nicht möglich.

11.2 Fortgeschrittene Methoden

Das zentrale Problem bei der Einrichtung flexibler Fehlerbehandlungsmethoden, die Fehler selbst korrigieren oder wenigstens dem Benutzer die Chance dazu geben, ist, daß Ihre Fehlerbehandlungsroutine immer „wissen“ muß, wo und warum der Fehler auftrat, damit sie dem Benutzer entsprechende Hinweise geben oder das Programm nach Bedarf fortsetzen kann. *Welcher* Fehler aufgetreten ist, läßt sich mit Hilfe von ERR und notfalls ERDEV und ERDEV$ recht gut herausfinden. *Wo* das passiert ist, ist in den meisten Programmen schwer zu lokalisieren – ERL gibt zwar Zeilennummern, aber keine Labels zurück, und außerdem gibt es dank moderner Kontrollstrukturen wie DO...LOOP und dank der Funktionen und Subroutinen nur noch wenig Grund, überhaupt Zeilennummern oder Zeilenlabels im Programm zu verwenden.

Zwei sinnvolle Lösungen bieten sich für dieses Problem an, wenn Sie nicht doch wieder vor jede Zeile eine Nummer setzen wollen, um mit ERL arbeiten zu können:

„Halblokale“ Fehlerbehandlung mit ON ERROR RESUME NEXT

Die erste Lösung wäre, die eigentliche Fehlerbehandlung gar nicht in der Fehlerbehandlungsroutine stattfinden zu lassen, sondern direkt im Programm, am Ort des Geschehens, unmittelbar dort, wo der Fehler auftritt. Dann muß die Fehlerbehandlungsroutine nur noch eine globale Variable setzen, aus der zu entnehmen ist, welcher Fehler auftrat, und das Programm kümmert sich um den Rest. Beispiel:

```
ON ERROR GOTO Fehler

DIM SHARED FehlerCode AS INTEGER

' hier ist eine Menge Programmcode, unter anderem:
DateiName$ = ""

DO
   IF DateiName$ = "" THEN LINE INPUT "Dateiname: ", DateiName$

   FehlerCode = 0
   OPEN DateiName$ FOR OUTPUT AS #1
   FOR a% = 1 TO AnzahlElemente
      PRINT #1, Element(a%)
      IF FehlerCode THEN EXIT FOR
   NEXT
   CLOSE 1
```

⇨

```
   IF FehlerCode THEN
   SELECT CASE FehlerCode
      CASE 61, 67: PRINT "Diskette voll. Auswechseln. Taste drücken.": SLEEP
      CASE 52, 54, 55, 64: PRINT "Ungültiger Dateiname.": DateiName$ =""
      CASE ELSE: PRINT "Programmfehler!" : SYSTEM
      END SELECT
   END IF
LOOP UNTIL FehlerCode = 0

' hier geht's weiter
' .....
END

Fehler:
   IF ERR = 7 OR ERR = 14 THEN
      PRINT "Der Speicherplatz reicht nicht aus!": SYSTEM
   ELSE
      FehlerCode = ERR: RESUME NEXT
  END IF
```

Listing 11–2: ERROR2.BAS

(Die im Programm benutzten Fehlernummern erheben keinen Anspruch auf Vollständigkeit, und auch der etwas rauhe Umgang mit dem Benutzer ist nicht zur Nachahmung empfohlen – es geht nur ums Prinzip!)

Dieses Programm handelt nur wirklich schwere Fehler gleich in der Fehlerroutine ab; bei allen anderen wird die Fehler-Variable gesetzt, und das Programm fährt fort. Die Abfrage der Variable und die Reaktion darauf kann dann an beliebiger Stelle im Programm erfolgen.

Wichtig ist allerdings bei einer solchen Vorgehensweise, daß man regelmäßig im Programm prüft, ob ein Fehler aufgetreten ist. Die Zeile `IF FehlerCode THEN EXIT FOR` hätte man zum Beispiel auch weglassen können. Aber dann hätte das Programm – angenommen, es gab schon beim Öffnen der Datei einen Fehler – unter Umständen sehr häufig versucht, in eine nicht geöffnete Datei zu schreiben, was wiederum genausoviele Aufrufe der Fehlerbehandlungsroutine zur Folge gehabt und entsprechend viel Zeit gekostet hätte.

Mit dem Befehl `ON ERROR RESUME NEXT` kann man sich sogar die Fehlerbehandlungsroutine und die globale Variable sparen. Wenn `ON ERROR RESUME NEXT` aktiv ist, wird nur die Systemvariable ERR auf den Fehlercode gesetzt, und das Programm läuft einfach weiter. Analog zum obigen Beispiel könnte man dann statt `FehlerCode` einfach `ERR` abfragen und auf 0 setzen. Allerdings ist es dann nicht ganz so bequem, gewisse Fehler eben doch ungeachtet des Ortes ihres Auftretens immer gleich zu behandeln, wie es im Beispiel mit den Fehlernummern 7 und 14 geschieht.

Lokale Fehlerbehandlung

Die zweite Lösung für unser Problem, die Fehler ordnungsgemäß zu behandeln, wäre die, daß man für verschiedene Programmabschnitte verschiedene Fehlerbehandlungsroutinen programmiert und vor dem Beginn eines jeden solchen Abschnitts mit ON ERROR GOTO die jeweilige Routine aktiviert.

Dieses Verfahren ist in Kombination mit SUBs und FUNCTIONs eine äußerst praktische Sache, da man mit ON LOCAL ERROR GOTO einer Prozedur einen eigenen, lokalen Error-Handler an die Seite stellen kann. Dies ist auch die einzige Möglichkeit, ein Error-Trapping in Formmodulen zu realisieren.

Wenn in einer Prozedur ein Fehler auftritt und eine globale Fehlerbehandlungsroutine aktiv ist, also eine, die im Modulcode steht und mit ON ERROR GOTO aktiviert wurde, dann wird diese Routine zur Fehlerbehandlung aufgerufen, und mit RESUME und RESUME NEXT kann zu dem Befehl, der den Fehler verursachte, oder seinem Nachfolger zurückgesprungen werden – egal, ob diese Befehle in einer Prozedur oder dem Modulcode stehen.

Hat eine Prozedur ihren eigenen, lokalen Error-Handler, dann sorgt ein Fehler innerhalb der Prozedur dafür, daß dieser aufgerufen wird. Erst wenn innerhalb des lokalen Error-Handlers wieder ein Fehler auftritt, wird die übergeordnete Fehlerbehandlungsroutine, also in diesem Fall der globale Error-Handler, aufgerufen. Das kann man sich zunutze machen, wenn einige Fehler in der Prozedur direkt abgefangen werden sollen, andere aber an den übergeordneten Error-Handler weiterzureichen sind.

Auch jede Prozedur, die von einer Routine mit lokalem Error-Handler aus aufgerufen wird und selbst keinen Error-Handler hat, verzweigt bei einem Fehler in diesen lokalen Error-Handler. Der lokale Error-Handler kann jedoch mit RESUME und RESUME NEXT nur an eine Stelle in seiner eigenen Prozedur springen, und nicht, wie globale im Hauptprogramm, an jede beliebige Stelle. Betrachten Sie das folgende Beispiel:

```
ON ERROR GOTO GlobalFehler

' tu was:
RoutineEins
RoutineZwei
END

GlobalFehler:
   PRINT "Es ist der Fehler »"; ERROR$;"« aufgetreten, aber keine Sorge, ich bin
   PRINT "unermüdlich!"
   RESUME NEXT
```

⇨

```
SUB RoutineEins

   ON LOCAL ERROR GOTO EinsFehler
   RoutineZwei
   EXIT SUB

EinsFehler:
   PRINT "Fehler »"; ERROR$; "«! Soll ich weitermachen?"
   IF INPUT$(1) = "J" THEN RESUME NEXT ELSE ERROR ERR

END SUB
```

```
SUB RoutineZwei

   ' hier gibt's keinen lokalen Handler
   ' das müßte einen Fehler geben:
   LOCATE 0, 0
   ' das ist ok:
   PRINT "RoutineZwei meldet sich zum Dienst!"

END SUB
```

Listing 11–3: ERROR3.BAS

In diesem Programm wird die RoutineZwei zweimal aufgerufen, und zwar einmal über den Umweg RoutineEins und das andere Mal direkt aus dem Hauptprogramm.

Zunächst wird im Hauptprogramm der globale Error-Handler aktiviert. Dann wird die RoutineEins aufgerufen, die einen eigenen, lokalen Error-Handler aktiviert und dann die RoutineZwei startet. Diese produziert einen Fehler, woraufhin die Fehlerbehandlungsroutine aus RoutineEins fragt, ob weitergearbeitet werden soll. Wenn Sie „J“ eingeben, führt sie einen RESUME NEXT-Befehl aus. Weil ein lokaler Error-Handler aber nicht per RESUME NEXT in eine fremde Prozedur springen kann, wird das Programm nicht beim PRINT-Befehl in RoutineZwei, sondern beim EXIT SUB-Befehl in RoutineEins fortgesetzt, weil dieser der erste Befehl nach dem Aufruf der Prozedur ist, in deren Verlauf der Fehler auftrat.

RoutineEins wird verlassen, und damit ist auch automatisch ihr lokaler Error-Handler nicht mehr aktiv. Gültig ist jetzt wieder der globale Error-Handler. RoutineZwei wird nochmals aufgerufen, diesmal direkt vom Hauptprogramm. Wieder verursacht sie den Fehler, und wieder wird in die Fehlerbehandlungsroutine verzweigt – diesmal allerdings in die globale. Diese führt, wie gehabt, ein RESUME NEXT aus; weil sie als globale Routine aber viel mächtiger ist als ihre lokalen Kolleginnen, kann sie die Ausführung wirklich direkt nach dem fehlerverursachenden Befehl fortführen – und endlich „meldet sich Routine-

Zwei zum Dienst“.Wenn Sie im lokalen Handler der `RoutineEins` übrigens nicht mit „J“ geantwortet hätten, wäre folgendes passiert: Mit dem `ERROR ERR`-Befehl wäre der Fehler, wegen dessen die Routine aufgerufen wurde, erneut erzeugt worden. Das wäre dann aber ein Fehler in der Fehlerbehandlungsroutine gewesen und hätte dafür gesorgt, daß die übergeordnete Fehlerbehandlungsroutine aufgerufen worden wäre. Diese – in unserem Falle ist es die globale – hätte brav ihr `RESUME NEXT` ausgeführt. Dadurch wäre `RoutineEins` hinter dem `ERROR ERR`-Befehl fortgesetzt worden, und alles weitere wäre wie oben abgelaufen.

In diesem Beispiel ist es recht wirkungslos, aber in der Praxis kann man den `ERROR ERR`-Befehl (oder `ON LOCAL ERROR GOTO 0`, das hat dieselbe Wirkung) gut benutzen, um bei schweren Fehlern, die die Prozedur nicht selbst regeln kann, die übergeordnete Fehlerbehandlungsroutine aufzurufen. Das kann natürlich auch eine lokale Routine sein, da Sie aus einer Prozedur mit eigenem Error-Handler auch eine weitere mit eigenem Error-Handler aufrufen können, sich also die verschiedenen Error-Handler sozusagen auf dem Stack stapeln und immer die zuletzt aktivierte benutzt wird.

Lokale Fehlerbehandlung kann auch mit dem Befehl `ON LOCAL ERROR RESUME NEXT` etabliert werden. Lokale Fehlerbehandlungsroutinen können mit `ON LOCAL ERROR GOTO 0` abgeschaltet werden; außerdem werden sie automatisch ausgeschaltet, wenn die Prozedur mit `EXIT SUB/FUNCTION` oder `END SUB/FUNCTION` verlassen wird.

Fehlerbehandlung bei mehreren Modulen

Wenn Ihr Programm aus mehreren Modulen – zum Beispiel aus einem Form- und einem Codemodul – besteht, also mehrere einzelnen .BAS- und .FRM-Dateien umfaßt, ist die Mächtigkeit eines globalen Error-Handlers eingeschränkt. Ein globaler Error-Handler aus einem Modul bleibt zwar, solange er nicht abgestellt wird, auch aktiv, während Programmteile aus anderen Modulen ausgeführt werden; er kann aber mit `RESUME` und `RESUME NEXT` nur auf oder hinter Befehle zurückspringen, die in *seinem* Modul liegen. Das hat Konsequenzen analog zu denen der oben verdeutlichten Schwäche eines lokalen Error-Handlers.

Bei mehreren Codemodulen kann jedes von ihnen einen globalen Error-Handler auf Hauptprogramm-Ebene enthalten. Achten Sie, wenn Sie so etwas planen, jedoch darauf, daß dieser Handler dann auch von den Prozeduren aus diesem Modul mit `ON ERROR GOTO` aktiviert wird. Wenn Sie mit nur einem Modul arbeiten, können Sie dessen `ON ERROR GOTO` einfach ins Hauptprogramm desselben schreiben, weil es dort bestimmt ausgeführt wird. Ein ON ERROR GOTO im

Hauptprogramm eines Zweit- oder Drittmodules würde jedoch nicht ausgeführt und muß deshalb in den Prozeduren dieser Module stehen.

Es ist sehr empfehlenswert, daß Sie alle Routinen, die überhaupt keine eigene Fehlerbehandlung benötigen (bei denen ein übergeordneter Error-Handler ausreicht) in einem Modul zusammenfassen, das sie dann ohne die /X- oder /E-Option kompilieren können, um Speicherplatz und Zeit zu sparen.

11.3 Resumée

Lokale Error-Handler sind praktisch, wenn eine Prozedur Fehler beheben soll bzw. kann, die ihre Ursache in ihr selbst haben. Die Selbständigkeit der Prozeduren wird dadurch wesentlich erhöht.

In Formmodulen sind lokale Error-Handler die einzige sinnvolle Möglichkeit, Fehler abzufangen.

In den meisten Fällen und vor allem bei unüberschaubar großen Programmen bewährt sich eine Kombination aus lokalen Error-Handlern und der Methode mit globalen Fehlervariablen, wie sie zum Beispiel auch von den Font- und Präsentationsgrafik-Toolboxen benutzt wird.

Datenbankprogrammierung mit ISAM 12

12.1 Was ist ISAM? Wozu ISAM?

ISAM ist der Name für ein Konzept zur Datenbankverwaltung. Die vier Buchstaben stehen für „Index Sequential Access Method“, was soviel bedeutet wie „Indexbasierte, sequentielle Zugriffsmethode“. Die professionelle Version von VBDOS enthält eine Anzahl von Befehlen, die es ermöglichen, Daten mit ISAM abzulegen und zu verwalten. Die ISAM-Befehle nehmen dem Programmierer viel Arbeit ab; größere Datenbestände sind damit leicht zu pflegen.

Gleich vorweg jedoch eine gravierende Einschränkung: Die kleinstmögliche Länge einer ISAM-Datei beträgt 64 KB.

ISAM setzt etwa da an, wo der BASIC-Programmierer bisher vielleicht damit liebäugelte, auf dBASE oder eine andere Datenbanksprache umzusatteln. ISAM ist eine „eingebaute Datenbanksprache“. Die Verwendung von ISAM geht auf Kosten der Flexibilität, denn man kann nicht mehr direkt auf die Datenbankdateien zugreifen. Andererseits muß man sich aber auch nicht mehr darum kümmern, ob die Datenbank kompakt genug ist, ob neue Datensätze korrekt einsortiert werden, ob gesuchte Datensätze schnell genug gefunden werden usw.

12.2 Das ISAM-Konzept

ISAM-Funktionsweise

Der Name verrät bereits die zwei wichtigsten Charakteristika: ISAM ist indexbasiert und sequentiell.

„Indexbasiert“ bedeutet, daß die Daten grundsätzlich in der Reihenfolge abgespeichert werden, in der man sie in die Datei aufnimmt. Die ISAM-Datenbank selbst wird niemals sortiert, dafür lassen sich aber beliebig viele Indexlisten anlegen, in denen die Daten nach einem oder mehreren Schlüssel-Inhalten sortiert abrufbar sind. Anders ausgedrückt: Es findet eine „Pseudo-Sortierung“ statt; die Daten sind verfügbar, *als ob* sie sortiert wären. Wie die Daten „pseudosortiert“ sind, wird in einer Indexliste (kurz: einem Index) festgelegt.

Das zweite Schlagwort „sequentiell“ deutet an, daß die Datensätze nur einer nach dem anderen, vom ersten bis zum letzten, gelesen werden können. Das klingt überholt und altmodisch, da „Random Access“ ja längst üblich ist. Man muß aber bedenken, daß sich dieses sequentielle Lesen auf die durch einen Index festgelegte Reihenfolge der Daten bezieht und nicht auf die ursprüngliche Eingabereihenfolge. Die durch den Index determinierte Reihenfolge kann sich

innerhalb von Sekundenbruchteilen völlig verändern, indem einfach auf einen anderen Index umgeschaltet wird. Außerdem gibt es Suchbefehle, die schnell zu einem bestimmten Datensatz springen.

Begriffsbestimmungen

Die eigentlichen (unsortierten) Daten bilden zusammen mit der Typenbeschreibung der Daten eine **Tabelle**, wobei die Typenbeschreibung deren Kopf bildet. Zu einer Tabelle können verschiedene Indizes existieren. Eine Adreßdatenbank zum Beispiel könnte einen Nachnamens- und einen Ortsindex enthalten. Die Datentabelle mit allen zu ihr gehörigen Indizes nenne ich eine **Datenbank**. Besteht einmal eine Datenbank mit einer Anzahl von Indizes, so kümmert sich ISAM automatisch beim Löschen oder Hinzufügen eines Datensatzes darum, daß alle Indizes auf den neuesten Stand gebracht werden. Außerdem ist es ein Leichtes, mit ISAM einen neuen Index zu erstellen.

Ein ISAM-**Datenfile** ist eine Datei auf der Festplatte, deren Namen auch beim OPEN-Befehl angegeben wird. Ein solches Datenfile kann beliebig viele Datenbanken enthalten. Jede Datenbank besteht aus einer Anzahl von **Datensätzen** und zusätzlich bis zu 500 **Indizes**. Jede Datenbank und jeder Index hat einen Namen, über den beide aktiviert werden. Indizes werden hier synonym auch als Indexlisten bezeichnet und bestimmen die **Sortierfolge**.

12.3 ISAM-Datenfiles auf der Platte

Eine ISAM-Datei auf dem Massenspeicher kann beliebig viele ISAM-Datenbanken enthalten und bis zu 128 MB umfassen. Aufgrund seiner Komplexität rechnet ISAM jedoch auch in anderen Speicherdimensionen, als Sie es vielleicht gewohnt sind: Die kleinstmögliche ISAM-Datei (eine Datei mit einer Datenbank, einem Datensatz und überhaupt keinem Index), belegt schon 64 KB.

Davon sind allerdings 25 KB noch leer und zur Aufnahme von weiteren Datensätzen oder Indizes geeignet. Eine ISAM-Datei wächst in 32 Kilobyte-Schritten, anstatt sich bei jeder Änderung ein wenig zu vergrößern. Ein Datenfile benötigt etwa 3 KB an Verwaltungsinformationen, jede Datenbank darin 4 KB (zuzüglich der in ihr gespeicherten Daten), jeder zusätzliche Index mindestens 2 KB.

Auf der anderen Seite ist ISAM – ausgelegt auf gewaltige Datenmengen – auch sparsam. Beim Abspeichern von Strings mit fester Länge (andere Strings lassen sich mit ISAM nicht verarbeiten) belegt ISAM nur so viel Speicherplatz, wie wirklich Zeichen darin enthalten sind, und nicht die ganze vereinbarte Länge.

12.4 Die Schnittstelle zwischen ISAM und BASIC

Da der Programmierer den Aufbau einer ISAM-Datei nicht kennt, benötigt er eine Schnittstelle zu ISAM. Diese Schnittstelle wird durch eine Anzahl von neuen und erweiterten Befehlen zur Verfügung gestellt.

Als Voraussetzung für den Zugriff auf ISAM-Datenbanken müssen Sie einen eigenen Datentyp und eine Variable definieren, die einen Datensatz aus der ISAM-Datenbank aufnimmt, zum Beispiel:

```
TYPE AdressenTyp
   Vorname AS STRING * 30
   Nachname AS STRING * 30
   Strasse AS STRING * 30
   PLZOrt AS STRING * 30
END TYPE
DIM Adresse AS AdressenTyp
```

Der neue Datentyp darf alle Datentypen mit Ausnahme des SINGLE-Typs enthalten. Arrays, weitere selbstdefinierte Typen und Strings mit einer Länge von über 255 Zeichen dürfen zwar im `AdressenTyp` enthalten sein, können aber nicht indiziert werden (mehr zum Thema Indizes später in diesem Kapitel).

Die Tatsache übrigens, daß der Typ auch Arrays enthalten darf, eröffnet die erfreuliche Möglichkeit, Grafiken in einer ISAM-Datenbank abzuspeichern, die mit dem GET-Befehl vom Bildschirm in ein Array eingelesen werden können. So lassen sich Datenbanken erstellen, denen viele Verkäufer schon das Modewort „Multimedia“ zuordnen würden.

Mit dem OPEN-Befehl können Sie nun eine Datenbank öffnen. Beachten Sie, daß bei ISAM jede *Datenbank* geöffnet werden muß und eine eigene Dateinummer erhält; es kann also ein und dieselbe ISAM-Datei unter verschiedenen Dateinummern (für verschiedene Datenbanken) zugleich geöffnet sein.

```
OPEN "ADRESS.ISM" FOR ISAM AdressenTyp "Adressen" AS #1
```

Hinter dem Kennwort „FOR ISAM“ folgt zunächst der Name des Datentyps, über den der Zugriff stattfinden soll. Danach muß der Name der Datenbank, die innerhalb des angegebenen Files benutzt wird, genannt werden. Schließlich folgt die Dateinummer, unter der die Datenbank geöffnet wird. Der OPEN FOR ISAM-Befehl ist vergleichbar mit einem OPEN FOR APPEND: Wenn die Datei bereits existiert, wird auf sie zugegriffen; existiert sie noch nicht, wird sie neu erstellt.

Nach einem OPEN-Befehl ist zunächst der sogenannte Null-Index aktiv. Auf die Daten wird in der Reihenfolge zugegriffen, in der sie gespeichert wurden.

Das gilt allerdings nur, solange Sie keine Daten löschen, denn die dadurch entstehenden Lücken füllt ISAM bei nächster Gelegenheit mit neu hinzukommenden Daten. Dann kann man nicht mehr sagen, der Null-Index enthalte die Daten in der Reihenfolge, in der sie aufgenommen wurden.

Die aktuelle Position

Ein Datenaustausch zwischen BASIC-Programm und ISAM-Datenbank ist immer nur an der aktuellen Position möglich. Nach einem OPEN-Befehl zeigt der Dateizeiger auf den ersten Datensatz in der Tabelle, Position 1 ist also die aktuelle Position. Zum Verschieben der aktuellen Position innerhalb der Datei gibt es zwei wichtige Gruppen von Befehlen: MOVE- und SEEK-Befehle. Die MOVE-Befehle verschieben den Dateizeiger unabhängig vom Inhalt der Datensätze. `MOVEFIRST` springt zum ersten, `MOVELAST` zum letzten, `MOVENEXT` zum nächsten und `MOVEPREVIOUS` zum vorherigen Datensatz. Den vier Befehlen muß dabei jeweils die Dateinummer der geöffneten Datenbank folgen.

Die SEEK-Befehle ermöglichen es, nach einem Datensatz zu suchen, der eine bestimmte Bedingung erfüllt. Ich komme darauf zurück, nachdem ich die Indizes behandelt habe.

Erstellen und Benutzen eines neuen Index

Der Befehl, mit dem man einen neuen Index anlegt, heißt `CREATEINDEX`. Hierbei muß zunächst die Dateinummer der Datenbank angegeben werden, zu der ein Index erstellt werden soll. Danach folgt der Name des neuen Index und dann ein numerisches Argument („universell“), das bestimmt, ob unter den Feldinhalten, die zu sortieren sind, Dubletten vorkommen dürfen oder nicht. Schließlich folgt der Name des Feldes, nach dem sortiert werden soll. Dieser Name muß in der TYPE...END TYPE-Struktur schon aufgetreten sein und entweder für einen numerischen Wert oder für einen String mit weniger als 256 Zeichen stehen.

Wenn „universell“ einen Wert ungleich Null (TRUE) annimmt, dürfen keine zwei Datensätze aus der Datenbank im angegebenen Feld denselben Inhalt haben. Ist „universell“ dagegen Null (FALSE), kommt es darauf nicht an. Um einen Ortsindex für die Adressendatei anzulegen, könnte man also schreiben:

```
CREATEINDEX 1, "Ortsindex", FALSE, "PLZOrt"
```

„Universell“ muß Null (FALSE) sein, denn es können ja durchaus mehrere Adressen im gleichen Ort liegen. Es lassen sich auch Indizes erstellen, die die Datensätze nach mehr als nur nach einem Feld sortieren. Dazu fügt man einfach weitere Felder hinten an:

```
CREATEINDEX 1, "NameVorname", TRUE, Nachname, Vorname
```

Hier wird primär nach dem Familiennamen sortiert, bei gleichen Familiennamen aber noch der Vorname herangezogen. In diesem Beispiel kann „universell“ schon eher auf TRUE gesetzt werden, denn es ist unwahrscheinlich, daß in derselben Datenbank zweimal dieselbe Kombination aus Vor- und Nachnamen auftritt (da es dennoch *möglich* ist, sollte man in der Praxis besser darauf verzichten). Solche kombinierten Indizes sind dadurch beschränkt, daß die Summe der Längen der angegebenen Felder 255 nicht übersteigen darf (INTEGER-Zahlen haben immer die Länge 2, LONG-Zahlen 4, DOUBLE- und CURRENCY-Zahlen 8).

Einen bestehenden Index wählt man mit dem Befehl `SETINDEX` zum aktuellen Index, wobei man Dateinummer und Indexnamen angeben muß:

```
SETINDEX 1, "Ortsindex"
```

Nachdem ein `SETINDEX`-Befehl ausgeführt wurde, zeigt der Dateizeiger auf den Datensatz, der nach der neu festgelegten Sortierung der erste ist.

Suchen nach einem bestimmten Datensatz

Mit den bereits erwähnten SEEK-Befehlen läßt sich auf der Grundlage eines Index leicht nach bestimmten Datensätzen suchen. Man kann dabei jedoch nur in den Feldern suchen, aus denen der aktuelle Index gebildet ist; es wäre also nicht möglich, nach einem Namen zu suchen, wenn der „Ortsindex“ der Adreßdatei aktiv ist. `SEEKGE` (*greater or equal*) sucht den ersten Datensatz, dessen indizierte Felder größer oder gleich einem angegebenen Inhalt sind, `SEEKGT` (*greater than*) sucht nur größere Feldinhalte und `SEEKEQ` (*equal*) sucht den ersten Datensatz, dessen indizierte Felder mit den angegebenen übereinstimmen. Mit „größer“ ist hier gemeint, daß ein String weiter hinten im Alphabet einsortiert wird als ein anderer (siehe dazu „Wie ISAM Strings sortiert“ später in diesem Kapitel). Hinter einem SEEK-Befehl wird zuerst eine Dateinummer und dann mindestens ein Schlüsselwert angegeben, maximal so viele, wie Felder indiziert wurden.

Bei einem einfachen Index wie unserem „Ortsindex“ muß und darf nur ein Wert angegeben werden.

```
SEEKEQ 1, "53111 Bonn"
```

würde also – ausgehend vom aktuellen Index – den ersten Datensatz suchen, dessen Ortsangabe exakt „53111 Bonn“ lautet (Groß- und Kleinschreibung werden allerdings ignoriert). Wenn kein solcher Datensatz vorhanden ist, wird der Dateizeiger hinter den letzten Datensatz gesetzt (EOF wird TRUE). In einem Index wie „NameVorname“, in dem mehrere Felder kombiniert wurden, kann

auch gesucht werden, wenn im SEEK-Befehl weniger Schlüsselwerte als im Index benutzt angegeben werden (im Beispiel müßte man diesen Index vorher allerdings mit `SETINDEX 1, "NameVorname"` aktivieren):

```
SEEKGE 1, "Müller-Hinterfeld"
```

Hier würde nach dem nächsten Datensatz gesucht, dessen Nachnamens-Eintrag größer oder gleich „Müller-Hinterfeld“ ist, unabhängig vom Vornamenseintrag. Dasselbe gilt für `SEEKGT`: Man darf Suchangaben weglassen. Nicht jedoch bei `SEEKEQ`; dieser Befehl braucht eine vollständige Angabe, sonst findet er nichts:

```
SEEKEQ 1, "Müller-Hinterfeld", "Claudia"
```

Zu beachten ist bei der Verwendung der SEEK-Befehle, daß sie stets vom ersten Datensatz und nicht etwa von der aktuellen Position des Dateizeigers ausgehend suchen. Der Befehl `SEEKGE 1, "Meier"` findet also immer den ersten Meier, auch wenn fünf in der Datenbank stehen und der Befehl fünfmal hintereinander ausgeführt wird.

Wie ISAM Strings sortiert

Sie unken schon, ISAM gehöre bestimmt auch zu denen, die Gerber vor Gärtner sortieren? Weit gefehlt! ISAM ist der internationalen Umlaut-Sortierweisen mächtig und ignoriert dabei sogar korrekt die Groß- und Kleinschreibung. Leerzeichen am Stringende werden abgeschnitten. Und selbst das „ß“ wird dort einsortiert, wo es hingehört: bei „ss“. Es gibt vier nationale Spezial-Sortierfolgen (siehe Anhang D); beim Installieren wird die entsprechende ausgewählt. Nachträgliche Änderungen sind nur durch erneute Installation möglich. Standard ist dabei die Tafel I, die für Englisch, Französisch, Italienisch, Portugiesisch und Deutsch gilt.

Damit Sie die ISAM-Sortierweise auch in eigenen Programmen nachempfinden können, gibt es eine ISAM-Funktion namens `TEXTCOMP`, die als Argumente zwei Strings hat; sie gibt –1 zurück, wenn der erste String kleiner als der zweite ist, 1 beim umgekehrten Verhältnis und 0, wenn beide Strings identisch sind. Dabei werden dieselben Maßstäbe angelegt, nach denen ISAM selbst sortiert.

`TEXTCOMP` berücksichtigt bei seiner Prüfung auf Gleichheit keine Leerzeichen am Ende der Strings. `TEXTCOMP` vergleicht generell nur die ersten 255 Zeichen eines Strings; auf der beiliegenden Diskette findet sich jedoch eine Routine namens LONGCOMP.BAS, die Sie auch diese Hürde überwinden läßt, sollte es denn einmal notwendig sein.

Zugriff auf die Daten

Der eigentliche Zugriff auf die Datenbank erfolgt stets an der Stelle, an der sich der Dateizeiger befindet, und zwar mittels der Befehle RETRIEVE (Lesen eines Datensatzes), UPDATE (Überschreiben), DELETE (Löschen) und INSERT (Einfügen). DELETE benötigt nur die Angabe einer Dateinummer und löscht dann den Datensatz, auf den der Dateizeiger weist. Die anderen drei Befehle werden jeweils mit Dateinummer und einer Variable, die den Datentyp hat, mit dem die Datenbank geöffnet wurde, aufgerufen.

```
UPDATE 1, Adresse
```

würde also zum Beispiel in unserer Datenbank den aktuellen Datensatz durch den Inhalt der Variable „Adresse“ überschreiben.

Transaktionen

Bei ISAM lassen sich Anweisungen, die die Datenbank verändern, in Gruppen zusammenfassen. Das hat die angenehme Folge, daß man eine so gebündelte Gruppe von Befehlen, die längst ausgeführt sind, ganz oder teilweise wieder zurücknehmen kann, solange sie nicht endgültig bestätigt wird. Diese Befehlsgruppen werden **Transaktionen** genannt.

Im Zusammenhang mit Transaktionen sind die Befehle BEGINTRANS, COMMITTRANS und ROLLBACK sowie die Funktion SAVEPOINT von Bedeutung.

BEGINTRANS hat keine Argumente und startet eine Transaktion. Alle Manipulationen an offenen ISAM-Datenbanken werden von der Ausführung dieses Befehls an „mitgeschnitten“, also Befehl für Befehl im Speicher notiert. Die Befehle werden trotzdem sofort ausgeführt und nicht etwa zurückbehalten, bis ein COMMITTRANS folgt. Transaktionen können nicht verschachtelt werden. Während eine Transaktion läuft, darf kein zweites BEGINTRANS auftreten.

COMMITTRANS beendet eine Transaktion. Die Liste der „gemerkten“ Datenveränderungen wird dabei gelöscht, die Änderungen können dann nicht mehr zurückgenommen werden. Einige ISAM-Befehle (wie CLOSE, DELETETABLE und DELETEINDEX) führen automatisch ein COMMITTRANS aus. Die meisten Fehler, die im Zusammenhang mit ISAM-Zugriffen auftreten, tun dies ebenfalls.

Die Funktion SAVEPOINT setzt innerhalb einer laufenden Transaktion eine unterteilende Markierung. Es können beliebig viele Markierungen gesetzt werden. SAVEPOINT gibt als Funktionswert die laufende Nummer der Markierung zurück. Anhand dieser Nummer kann später genau der Zustand wiederhergestellt werden, in dem die Daten sich zum Zeitpunkt des Funktionsaufrufs befanden.

Mittels des ROLLBACK-Befehls kann man alle Befehle, die seit BEGINTRANS ausgeführt wurden, rückgängig machen (oder einen Teil davon). Dabei wird jeder ausgeführte Befehl einzeln aus der Transaktionsmitschrift gelesen und annulliert. ROLLBACK ALL macht alle Befehle der laufenden Transaktion rückgängig. ROLLBACK ohne Argument geht zurück bis zur letzten SAVEPOINT-Markierung; gibt es keine, ist die Wirkung mit ROLLBACK ALL identisch. Wird ROLLBACK von einem numerischen Argument gefolgt, dann werden alle seit der angegebenen SAVEPOINT-Markierung ausgeführten Befehle aufgehoben.

Weitere ISAM-Befehle und -Funktionen

Es existiert eine Funktion namens GETINDEX$, die den Namen des aktuellen Index zurückgibt. Der Befehl DELETETABLE löscht eine ganze Datenbank aus einer ISAM-Datei. DELETEINDEX löscht einen einzelnen Index. Die Funktion BOF ist TRUE, wenn der Dateizeiger vor den ersten Datensatz zeigt (also auf die Position vor der, an die mit MOVEFIRST gesprungen wird). Mit EOF verhält es sich ähnlich, sie ist TRUE, wenn der Dateizeiger auf die Position hinter dem letzten Datensatz zeigt. LOF gibt zurück, wie viele Datensätze in einer ISAM-Datenbank enthalten sind. Die Funktion FILEATTR wurde ergänzt, so daß sie auch ISAM-Dateien verarbeitet. Der Befehl CHECKPOINT sorgt dafür, daß alle Daten, die noch im Buffer stehen, in die Datei geschrieben werden (zum Beispiel als Vorsorge für ein unvorhergesehenes Abschalten des Systems).

Programmbeispiel

„Oh nein, nicht schon wieder ADRESSEN!“ höre ich die Hälfte aller Leser vor meinem geistigen Ohr klagen. Aber zu meiner Verteidigung muß ich vorbringen, daß es hier nicht darum geht, ein interessantes neues Programm zu schöpfen, sondern darum, in einem möglichst kurzen Programm möglichst alle ISAM-Befehle unterzubringen und im Kontext zu verdeutlichen. Und was eignet sich dazu besser als eine Adreßverwaltung, die so oder ähnlich (es müssen ja nicht Adressen sein) schon jeder Programmierer einmal absolviert hat?

Mit dem Wissen, daß es sich hier um ein an didaktischen Gesichtspunkten orientiertes Programm handelt, hat der Leser, so hoffe ich, auch Verständnis für die schlichte Benutzerführung. Ich habe bewußt darauf verzichtet, die Methoden der ereignisgesteuerten Programmierung hier einzusetzen, um das Wesentliche – die ISAM-Befehle – zu betonen.

Das Beispiel ist so reichlich dokumentiert, daß es sich gut von vorne bis hinten durchlesen läßt – und genau das empfehle ich Ihnen.

```
' ISAMDEMO.BAS   Adreßverwaltung zur Demonstration der ISAM-Fähigkeiten

CONST Datenbank = "Adressen"              ' Datenbankname in der ISAM-Datei
CONST Datenfile = "ISAMDEMO.ISM"          ' Name der ISAM-Datei
CONST Ausfuehrlich = 1, Kurz = 2          ' Anzeigemodi für AnzeigeDatensatz

TYPE Datentyp                             ' Datentyp für ISAM-Zugriff
   Name AS STRING * 30
   Vorname AS STRING * 30
   Strasse AS STRING * 40
   PLZ AS STRING * 6
   Ort AS STRING * 30
   Telefon AS STRING * 30
END TYPE
DIM SHARED Zugriff AS Datentyp            ' Zugriffsvariable
DIM SHARED Leer AS Datentyp               ' Leere Adresse

Leer.Name = "": Leer.Vorname = ""         ' Ohne diese Befehle enthielte die "leere
Leer.Strasse = "": Leer.PLZ = ""          ' Adresse" lauter CHR$(0)-Zeichen, und
Leer.Ort = "": Leer.Telefon = ""          ' RTRIM würde nicht funktionieren. Hier-
                                          ' durch wird die "leere Adresse" mit
                                          ' CHR$(32), also Leerzeichen, gefüllt.
DatenbankOeffnen                          ' Eingangsmeldung anzeigen & Daten-
                                          ' bank als Datei #1 öffnen
HauptMenue                                ' Hauptmenü
PRINT "Sollen die Änderungen, die Sie gemacht haben,"
PRINT "gespeichert werden (J/N)? ";
DO
   DO: a$ = UCASE$(INKEY$): LOOP UNTIL LEN(a$)
LOOP UNTIL a$ = "N" OR a$ = "J"
PRINT a$
IF a$ = "N" THEN ROLLBACK ALL             ' Wenn Änderungen rückgängig gemacht wer-
                                          ' den müssen: ROLLBACK ALL stellt den
                                          ' Status zum Zeitpunkt des BEGINTRANS-
                                          ' Befehls wieder her
COMMITTRANS: CLOSE                        ' COMMITTRANS wäre eigentlich gar nicht
                                          ' notwendig, da CLOSE es automatisch aus-
                                          ' führt. Nach COMMITTRANS kann mit
                                          ' ROLLBACK nichts mehr zurückgenommen
                                          ' werden.
SYSTEM                                    ' Programmende
```

```
' Diese Prozedur zeigt den Datensatz, der in der globalen Variablen Zugriff
' gespeichert ist, entweder ausführlich (vierzeilig) oder kurz (einzeilig) an. Die
' Funktion RTRIM$ zum Entfernen der überflüssigen Leerstellen wird nur da benutzt,
' wo in derselben Zeile noch weitere Zeichen ausgegeben werden.
SUB AnzeigeDatensatz (Modus AS INTEGER)

   IF Modus = Ausfuehrlich THEN
      PRINT
      PRINT SPC(5); RTRIM$(Zugriff.Vorname); " "; Zugriff.Name
```

```
      PRINT SPC(5); Zugriff.Strasse
      PRINT SPC(5); RTRIM$(Zugriff.PLZ); " "; Zugriff.Ort
      PRINT SPC(5); "Telefon "; Zugriff.Telefon
   ELSE
      PRINT RTRIM$(Zugriff.Vorname); " "; RTRIM$(Zugriff.Name); ", ";
      PRINT RTRIM$(Zugriff.Strasse); ", ";
      PRINT RTRIM$(Zugriff.PLZ); " "; RTRIM$(Zugriff.Ort); ", ";
      PRINT "Telefon "; RTRIM$(Zugriff.Telefon); "."
   END IF
END SUB
```

```
' Diese Prozedur öffnet die ISAM-Datenbank (gemäß den Konstanten im Modulcode).
' Außerdem werden die beiden benötigten Indexlisten erstellt, wenn die Datenbank
' noch nicht existiert.
SUB DatenbankOeffnen

   PRINT
   PRINT "------------------------"
   PRINT "---   ISAMDEMO.BAS   ---"
   PRINT "---  Adreßdatenbank  ---"
   PRINT "------------------------"
   PRINT
   ON LOCAL ERROR RESUME NEXT: OPEN Datenfile FOR ISAM Datentyp Datenbank AS #1
   IF ERR = 0 THEN
      ' Wenn die Datenbank schon existiert, gibt das hier
      ' Fehler. Aber das spielt ja keine Rolle.
      CREATEINDEX #1, "NameVorname", 1, "Name", "Vorname"
      CREATEINDEX #1, "PLZOrt", 0, "PLZ", "Ort"
   ELSE
      'Programm abbrechen.
      IF ERR = 73 THEN
         PRINT "ISAM-Routinen nicht verfügbar!"
      ELSE
         PRINT "Die Datei "; Datenfile; " kann nicht geöffnet werden."
      END IF
      SYSTEM
   END IF

   ' Hier fängt die "Transaktion" an. BEGINTRANS wird benötigt, um den ROLLBACK-
   ' Befehl benutzen zu können. Alle Operationen, die mit dem Programm durchgeführt
   ' werden, werden als eine einzige Transaktion verstanden.
   BEGINTRANS

END SUB
```

```
' Diese Prozedur erstellt mittels der Prozedur AnzeigeDatensatz eine Liste aller
' Adressen (alphabetisch sortiert) oder eine Liste eines beliebigen Postleitzahl-
' gebietes (nach PLZ sortiert).
SUB Datenliste

   DIM VonPLZ AS STRING, BisPLZ AS STRING
   PRINT:PRINT "Datenliste für alle Datensätze oder bestimmtes PLZ-Gebiet (A/P)? ";
```

```
   DO
      DO: a$ = UCASE$(INKEY$): LOOP UNTIL LEN(a$)
   LOOP UNTIL a$ = "A" OR a$ = "P"
   PRINT a$: PRINT

   SELECT CASE a$
   CASE "A"                              ' LISTE NACH NAMEN SORTIERT:
      SETINDEX 1, "NameVorname"          ' Indexliste mit Sortierung nach Namen
                                         ' anwählen
      MOVEFIRST 1                        ' Zeiger auf ersten Datensatz nach neuer
                                         ' Sortierung setzen (eigentl. hier über-
                                         ' flüssig, weil von SETINDEX automatisch
                                         ' ausgeführt)
      DO UNTIL EOF(1)                    ' Solange, bis Zeiger hinter den letzten
                                         ' Datensatz zeigt:
         RETRIEVE 1, Zugriff             '    Datensatz, auf den Zeiger zeigt, in
                                         '     Zugriff  lesen,...
         AnzeigeDatensatz Kurz           '    ...anzeigen...
         MOVENEXT 1                      '    und Zeiger weiterstellen.
      LOOP
   CASE "P"                              ' LISTE NACH PLZ sortiert:
      PRINT "Von PLZ (ganze PLZ oder Anfangsziffern; 0 = von Anfang): ";
      LINE INPUT VonPLZ
      PRINT "Bis PLZ (ganze PLZ oder Anfangsziffern; 0 = bis Ende):   ";
      LINE INPUT BisPLZ
      SETINDEX 1, "PLZOrt"               ' Index für PLZ-Sortierung setzen
      IF VonPLZ = "0" THEN               ' Wenn "Von Anfang" gewählt, dann
         MOVEFIRST 1                     ' Zeiger auf erstes Element setzen,
      ELSE                               ' sonst auf das erste Element, das
         SEEKGE 1, VonPLZ                ' >= der Anfangs-PLZ ist
      END IF
      DO UNTIL EOF(1)                    ' Solange, bis Zeiger hinter letzten
                                         ' Datensatz zeigt:
       RETRIEVE 1, Zugriff               ' Datensatz, auf den Zeiger zeigt,  in
                                         ' "Zugriff  einlesen
       IF BisPLZ <> "0" THEN             ' wenn nicht "bis Ende" gewählt wurde:
         IF TEXTCOMP(LEFT$(Zugriff.PLZ, LEN(BisPLZ)), BisPLZ) = 1 THEN EXIT DO
       END IF                            ' prüfen, ob die PLZ des Datensat-
                                         ' zes schon außerhalb d. Bereiches liegt;
                                         ' wenn ja, Schleife verlassen.
       AnzeigeDatensatz Kurz             ' Datensatz anzeigen
       MOVENEXT 1                        ' Zeiger weiterstellen
      LOOP
   END SELECT
END SUB
```

```
' Diese Prozedur läßt den Benutzer   unkomfortabel zwar, aber ausreichend - einen
' Datensatz eingeben. In der Variablen Vorgabe steht, was eingesetzt wird, wenn der
' Benutzer einfach ENTER drückt. Die Eingaben werden in die Variable NeuerSatz
' geschrieben. Wenn NeuerSatz schon Daten enthält, werden diese Elemente nicht mehr
' abgefragt (eine Tatsache, die in diesem Demo-Programm gar nicht genutzt wird).
SUB DatensatzEingabe (Vorgabe AS Datentyp, NeuerSatz AS Datentyp)
```

```
   PRINT
   DO UNTIL LEN(RTRIM$(NeuerSatz.Name))
      IF LEN(RTRIM$(Vorgabe.Name)) THEN PRINT "(Vorgabe)      "; Vorgabe.Name
      LINE INPUT "Name:        "; temp$: IF temp$ = "" THEN temp$ = Vorgabe.Name
      NeuerSatz.Name = temp$
   LOOP

   DO UNTIL LEN(RTRIM$(NeuerSatz.Vorname))
      IF LEN(RTRIM$(Vorgabe.Vorname)) THEN PRINT "(Vorgabe)      "; Vorgabe.Vorname
      LINE INPUT "Vorname:     "; temp$: IF temp$ = "" THEN temp$ = Vorgabe.Vorname
      NeuerSatz.Vorname = temp$
   LOOP

   ' ebenso für Strasse, PLZ, Ort, Telefon; auf Abdruck habe ich verzichtet

END SUB
```

```
' Diese Prozedur zeigt das Hauptmenü an und verarbeitet die Auswahlen des
' Benutzers, so lange, bis er "Programmende" wählt.
SUB HauptMenue

   DIM Vorgabe AS Datentyp
   DO
      PRINT
      PRINT "---------------------------------"
      PRINT "Datenbank enthält"; LOF(1); "Sätze"
      PRINT "1. Daten anfügen": PRINT "2. Daten löschen"
      PRINT "3. Daten ändern": PRINT "4. Einzelnen Datensatz anzeigen"
      PRINT "5. Datenliste anzeigen": PRINT "6. Undo"
      PRINT "9. Programmende"
      LOCATE , , 1                        ' Cursor einschalten
      PRINT "Ihre Wahl> ";
      DO: a$ = INPUT$(1): LOOP UNTIL LEN(a$): PRINT a$
      PRINT "---------------------------------"

      SELECT CASE VAL(a$)
      CASE 1
         PRINT: Zugriff = Leer
         DatensatzEingabe Leer, Zugriff   ' Neuen Datensatz eingeben
         x% = SAVEPOINT                   ' SAVEPOINT-Markierung setzen
         INSERT 1, Zugriff                ' Neuen Datensatz in Datenbank aufnehmen
      CASE 2
         SucheDatensatz "Welchen Datensatz löschen?"
         IF NOT EOF(1) THEN               ' Wenn der Zeiger nicht hinter letztem
                                          ' Datensatz steht,
            x% = SAVEPOINT                ' SAVEPOINT-Markierung setzen
            DELETE #1                     ' Datensatz, auf den Zeiger zeigt, löschen
            PRINT "* 1 Datensatz gelöscht"
         ELSE
            PRINT "* Kein Datensatz gelöscht"
         END IF
```

```
      CASE 3
         SucheDatensatz "Welchen Datensatz möchten Sie ändern?"
         IF NOT EOF(1) THEN              ' Wenn der Zeiger nicht hinter den letzten
                                         ' Datensatz zeigt,
            Vorgabe = Zugriff            ' Zu ändernde Variable als Vorgabe f. Neu-
            Zugriff = Leer               ' eingabe:
            DatensatzEingabe Vorgabe, Zugriff
            x% = SAVEPOINT               ' SAVEPOINT-Markierung setzen
            UPDATE 1, Zugriff            ' Datensatz, auf den Zeiger zeigt, durch
                                         ' "Zugriff" ersetzen
         END IF
      CASE 4
         SucheDatensatz "Welcher Datensatz soll angezeigt werden?"
         ' SucheDatensatz liest den Datensatz selbst in "Zugriff" ein, also
         ' muß nur noch die Anzeige aufgerufen werden:
         IF NOT EOF(1) THEN AnzeigeDatensatz Ausfuehrlich
      CASE 5
         Datenliste                       ' Die Prozedur erledigt alles
      CASE 6
         ROLLBACK                         ' Zustand der Datenbank zum Zeitpunkt der
                                          ' letzten SAVEPOINT-Markierung herstellen
      CASE 9
         EXIT DO                          ' Menü verlassen
      END SELECT
   LOOP
END SUB
```

```
' Diese Prozedur wird von allen anderen dazu benutzt, einen bestimmten Datensatz
' ausfindig zu machen. Wenn der Benutzer mit Hilfe dieser Prozedur einen Datensatz
' als "richtigen" identifiziert, ist dieser Datensatz beim Verlassen der Prozedur
' in der Variable Zugriff gespeichert, und außerdem zeigt der Dateizeiger auf diesen
' Datensatz in der ISAM-Datei. Verläuft diese Prozedur jedoch erfolglos (Abbruch
' oder kein Datensatz gefunden), dann zeigt der Datenzeiger hinter den letzten
' Datensatz, EOF(1) hat dann den Funktionswert  1, und daran erkennen alle anderen
' Prozeduren, daß SucheDatensatz nicht erfolgreich war.
SUB SucheDatensatz (Titel AS STRING)
   DIM NachName AS STRING, Postleitzahl AS STRING

   PRINT: PRINT Titel
   PRINT "1. Suchen nach Namen": PRINT "2. Suchen nach Postleitzahl"
   PRINT "Ihre Wahl> ";
   DO: DO: a$ = INKEY$: LOOP UNTIL LEN(a$): LOOP UNTIL a$ = "1" OR a$ = "2"
   PRINT a$
   PRINT "- (j/n/a) bedeutet (ja/nein/abbruch) -"
   SELECT CASE VAL(a$)
   CASE 1                                ' SUCHEN NACH NAMEN
      SETINDEX 1, "NameVorname"          ' Namensindex aktivieren
      PRINT "Nachname (oder Anfangsbuchstaben dessen): ";
      LINE INPUT NachName
      SEEKGE 1, NachName                 ' Zeiger auf ersten Datensatz setzen, des-
                                         ' sen Nachname ≥ dem eingegebenen Namen ist
```

```
      DO UNTIL EOF(1)                    ' Solange bis der Datenzeiger hinter dem
                                         ' letzten Datensatz steht:
         RETRIEVE 1, Zugriff             ' Datensatz, auf den Datenzeiger zeigt,
                                         ' in Zugriff einlesen
         IF TEXTCOMP(LEFT$(Zugriff.Name, LEN(NachName)), NachName) THEN
                                         ' Wenn der eingelesene Name schon zu weit
                                         ' hinten im Alphabet steht (die ersten
                                         ' Buchstaben nicht mehr mit den eingege-
                                         ' benen übereinstimmen), wird abgebrochen.
            EXIT DO
         END IF
         PRINT RTRIM$(Zugriff.Vorname); " "; RTRIM$(Zugriff.Name);
         PRINT TAB(60); "OK (j/n/a)? ";  ' Namen ausgeben und fragen, ob er der
                                         ' gewünschte ist.
         DO: DO: a$ = UCASE$(INKEY$): LOOP UNTIL LEN(a$)
         LOOP UNTIL INSTR("JNA", a$)
         PRINT a$
         IF a$ = "J" OR a$ = "A" THEN
            EXIT DO                      ' Bei JA und ABBRUCH: Suche beenden.
         END IF
         MOVENEXT 1                      ' Sonst Zeiger weiterstellen, weitersuchen
      LOOP
      IF a$ = "J" THEN
         EXIT SUB                        ' Bei JA: Prozedur verlassen
      ELSE
         MOVELAST 1: MOVENEXT 1          ' Bei ABBRUCH oder wenn keine Datensätze
                                         ' mehr gefunden wurden, Zeiger hinter
                                         ' den letzten Datensatz stellen.
         IF NOT a$ = "A" THEN
            PRINT "* Keine weiteren Datensätze gefunden."
         END IF
      END IF
   CASE 2
      SETINDEX 1, "PLZOrt"               ' SUCHEN NACH PLZ (Prinzip wie oben)
      PRINT "Postleitzahl (oder erste Ziffern davon): "
      LINE INPUT Postleitzahl
      ' funktioniert genauso wie die Suche nach Namen - hier nicht mehr abgedruckt
   END SELECT

END SUB
```

Listing 12–1: ISAMDEMO.BAS

12.5 ISAM und VBDOS

Damit ISAM von Programmen aus benutzt werden kann, die in der Entwicklungsumgebung VBDOS laufen, muß vor dem Aufruf von VBDOS ein speicherresidentes Programm geladen werden, das die ISAM-Routinen zur Verfügung stellt.

Zwei Programme werden hierzu mitgeliefert: PROISAM.EXE und PROISAMD.EXE. Das kürzere PROISAM.EXE enthält alle Routinen bis auf die zum Erstellen und Löschen von ganzen Datenbanken und die zum Erstellen und Löschen von Indizes. Da ein fertiges Programm zumeist nur die Datensätze manipuliert und nicht die Datenstruktur, reicht PROISAM.EXE für die meisten Fälle aus. Zur Manipulation an Datenbanken und Indizes muß man PROISAMD.EXE laden, das sämtliche ISAM-Routinen enthält.

Die beiden speicherresidenten Programme benutzen drei Switches, mit denen man ihren Speicherverbrauch kontrollieren kann. Der Aufruf von PROISAM.EXE lautet (PROISAMD entsprechend):

```
PROISAM [/Ib:puffer] [/Ie:emsrest] [/Ii:indexanzahl]
PROISAM /D
```

Der Switch /D entfernt das bereits geladene Programm wieder aus dem Speicher. Das ist nur möglich, wenn nach ihm kein weiteres speicherresidentes Programm geladen wurde.

Die anderen Switches (die genauso auch beim Compiler BC verwendet werden – siehe Kapitel 6) haben folgende Funktion:

puffer ist die Anzahl der Zwischenspeicher, die für ISAM-Operationen benutzt werden können. Jeder Puffer benötigt 2 KB Speicher im EMS oder, wenn das EMS schon belegt ist, im normalen Speicher. Für PROISAM werden standardmäßig 6, für PROISAMD 9 Puffer belegt. Je größer Sie diesen Wert wählen, desto schneller wird das ISAM-System arbeiten. Aufwendige Operationen können sogar unter Umständen mit den oben genannten Standardwerten nicht ausgeführt werden. Um ganz exakt errechnen zu können, wieviele Puffer Ihr Programm maximal benötigt, benutzen Sie die folgende Formel: *puffer* = 1 + maximale Anzahl gleichzeitig geöffneter Datenbanken + Anzahl der Indizes, die das Programm benutzt (Null-Index nicht eingerechnet). Addieren Sie weitere 4, wenn Sie `INSERT`- oder `UPDATE`-Befehle benutzen, und noch einmal 8, wenn Sie den `CREATEINDEX`-Befehl benutzen. Wie gesagt, kann es der Geschwindigkeit nie schaden, zu viele Puffer zu haben – höchstens dem Speicherplatz. Maximal 512 Puffer sind erlaubt. Egal, wieviele Puffer Sie hier angeben, wenn nach der Belegung des Pufferspeichers und eines für ISAM benötigten Datenspeichers von 5 KB für PROISAM bzw. bis zu 16 KB für PROISAMD noch EMS frei sein sollte, wird der ganze EMS-Rest mit ISAM-Buffern aufgefüllt. Das geschieht so lange, bis das EMS voll oder das Limit von insgesamt 1,2 MB für ISAM erreicht ist.

Um ISAM von dieser Speicherbelegung abzuhalten, können Sie einen *emsrest* angeben. *emsrest* legt fest, wieviel EMS (in KB) ISAM nicht belegen darf. Üblicherweise, wenn Sie den /Ie-Switch weglassen, darf ISAM das EMS

unbegrenzt benutzen (es benötigt allerdings nie mehr als 1,2 MB). Geben Sie als *emsrest* –1 an, dann benutzt ISAM das EMS überhaupt nicht.

indexanzahl können Sie im Normalfall weglassen, da die Standard-Indexanzahl von 28 zumeist ausreicht. Wollen Sie in ihrem Programm mehr als 28 Indizes benutzen, dann müssen Sie die maximale Anzahl hier angeben. Mehr als 500 Indizes sind allerdings selbst mit diesem Switch nicht möglich.

12.6 ISAM in kompilierten Programmen

In VBDOS sind die Routinen zur ISAM-Unterstützung fest im Runtime-Modul bzw. der VBDCL10-Library integriert. Wenn Sie EXE-Programme erstellen, die ISAM-Funktionen benutzen, werden die ISAM-Routinen beim Kompilieren mit /O in das EXE-Programm eingebunden, beim Kompilieren ohne /O werden sie aus dem Runtime-Modul gelesen.

Die speicherresidenten Programme PROISAM.EXE und PROISAMD.EXE können nicht verwendet werden, um Ihren eigenen Programmen zu ISAM-Unterstützung zu verhelfen. In diesem Punkt weicht VBDOS vom BASIC PDS ab, bei dem eigene Programme die PROISAM-TSRs nutzen konnten.

Wenn Sie eigene Runtime-Module erstellen, können Sie verhindern, daß die ISAM-Routinen in das Runtime-Modul eingebunden werden (vgl. Kapitel 14).

12.7 ISAM – Programmierdetails und Vorsichtsmaßnahmen

Anzahl gleichzeitig geöffneter ISAM-Dateien und -Datenbanken

Sie können mehrere ISAM-Dateien zugleich öffnen. Das wird aber in den meisten Fällen nicht notwendig sein, da schon eine einzelne ISAM-Datei beliebig viele Datenbanken mit einer Gesamtgröße von bis zu 128 MB beherbergen kann. Wenn Sie dennoch mehrere ISAM-Dateien gleichzeitig benutzen, beachten Sie: Sie können maximal vier ISAM-Dateien gleichzeitig geöffnet haben, und die Anzahl der gleichzeitig geöffneten Datenbanken ist $1+3\cdot(5-n)$, wobei n die Anzahl der verschiedenen ISAM-Dateien ist.

Probleme mit den speicherresidenten ISAM-Routinen

Wie nur zu oft mit speicherresidenten Programmen, kann es auch mit den Programmen PROISAM.EXE und PROISAMD.EXE zu Schwierigkeiten kommen. So kann es – je nach Konfiguration – zum Beispiel passieren, daß beim ersten OPEN-Befehl das ganze Programm abstürzt, obwohl PROISAM.EXE geladen

wurde. Abhilfe kann man häufig schaffen, indem man sicherstellt, daß PROISAM.EXE als erstes speicherresidentes Programm (also noch vor dem Tastaturtreiber KEYB.COM oder KEYBGR.EXE o. ä.) geladen wird. Dann allerdings kann PROISAM.EXE nicht mehr mit dem Switch /D aus dem Speicher entfernt werden.

Wenn Sie mit WINDOWS arbeiten, müssen Sie WINDOWS starten, bevor Sie PROISAM.EXE oder PROISAMD.EXE laden. Rufen Sie die „DOS-Eingabeaufforderung“ aus der Hauptgruppe auf, und starten Sie dann unter DOS PROISAM oder PROISAMD und VBDOS.

Datentyp SINGLE

ISAM unterstützt den Datentyp SINGLE nicht. Sie haben zwei Möglichkeiten, wenn Sie trotzdem Fließkommazahlen vom Typ SINGLE in einer ISAM-Datenbank speichern wollen: Die eine ist, Felder vom Typ DOUBLE zu spezifizieren und dort die SINGLE-Variablen einzutragen, eventuell nach einer Typumwandlung mit CDBL. Der Nachteil dabei ist, daß für Ihre SINGLE-Variable eigentlich nur 4 Bytes benötigt werden, DOUBLE aber 8 braucht, so daß 4 Bytes pro „verkleideter“ SINGLE-Zahl verschwendet werden. Der Vorteil dieser Lösung ist, daß die entstehenden DOUBLE-Felder in Indizes verwendet werden können. Die zweite Möglichkeit ist die, einen String der Länge 4 als Feld zu definieren und die SINGLE-Zahlen mit der Funktion MKS$ in 4-Byte-Strings umzuwandeln, bevor sie in die Datei geschrieben werden. Diese Version verschwendet zwar keinen Speicherplatz, allerdings können Sie den String, der die umgewandelte SINGLE-Zahl enthält, nicht in einem Index benutzen, weil die komprimierte 4-Byte-Darstellung nicht sortiert werden kann.

Interne Datenbanken

ISAM erfordert eine große Menge an Verwaltungsaufwand. Ein großer Teil der Verwaltungsinformationen einer ISAM-Datei ist in der ISAM-Datei selbst in Form gewöhnlicher Datenbanken gespeichert. Diese internen Datenbanken, von denen weder Benutzer noch Programmierer Kenntnis haben müssen, heißen zum Beispiel MSYSOBJECTS, MSYSRELATIONSHIPS, MSYSTEMPINDEXES, MSYSINDEXES und MSYSCOLUMNS. Struktur und Verwendungsweise dieser internen Datenbanken sind nicht dokumentiert, und gewiß ist es auch nicht vorgesehen, diese zu manipulieren. Man kann die Struktur einer dieser Datenbanken jedoch durch einen Trick mit Hilfe des Programms ISAMIO ermitteln: Man exportiert eine der o. g. internen Datenbanken in eine

ASCII-Datei (Befehl ISAMIO /E *asciidatei isamdatei datenbankname satzbeschreibung*). Damit wird nicht nur der Inhalt der internen Datenbank zugänglich, sondern es wird auch, was viel interessanter ist, eine Satzbeschreibung aller Felder in der angegebenen Datenbank ausgegeben, mit Typen- und gegebenenfalls Längenangabe. Diese Satzbeschreibung kann nun genutzt werden, um auf die Daten zuzugreifen oder sie zu manipulieren – wobei Sie natürlich stets den Verlust sämtlicher Daten in der ISAM-Datei riskieren, aber die Chance haben, der Bedeutung der internen Datenbanken auf die Spur zu kommen. Durch genauere Kenntnis der internen Datenbanken wären Sie beispielsweise in der Lage, ein Programm zu schreiben, das die Struktur einer ISAM-Datei feststellt u.ä.

Die Programmierung eines Systems, das mit verschiedensten ISAM-Dateien arbeiten kann, ist mit BASIC allerdings nicht möglich (es sei denn, Sie begeben sich auf die Byte-Ebene hinab und operieren mit OPEN FOR BINARY), denn die Art der ISAM-Datei, die benutzt werden soll, muß ja beim Kompilieren schon feststehen, da dort die TYPE...END TYPE-Definitionen festgelegt sind.

12.8 ISAM-Utilities

Vier wichtige Hilfsprogramme werden mit dem Compiler geliefert. Erfreulich ist, daß Sie die Erlaubnis haben, diese Programme an die Benutzer von ISAM-Programmen, die Sie geschrieben haben, weiterzugeben.

ISAMCVT

Mit Hilfe von ISAMCVT lassen sich drei Dateiformate ins ISAM-Format konvertieren: Btrieve, MS/ISAM (eine frühere ISAM-Version, die mit dem IBM BASIC-Compiler 2.0 ausgeliefert wurde) und das dBase-Format. Die Syntax ist wie folgt:

```
ISAMCVT /modus dateiname datenbank isamdateiname [satzbeschreibung]
```

modus ist entweder M (für MS/ISAM), D (für dBase) oder B (für Btrieve). Beim Konvertieren von Btrieve-Dateien muß hinter *isamdateiname* noch eine *satzbeschreibungsdatei* angegeben werden.

dateiname ist der Name der zu konvertierenden Datei, *isamdateiname* der Name der ISAM-Datei, in die konvertiert werden soll, und *datenbank* der Name der Datenbank innerhalb *isamdateiname*, die die Ergebnisse aufnimmt.

Für die Konvertierung von Btrieve-Dateien muß das speicherresidente Btrieve-Programm geladen sein.

satzbeschreibung ist der Name einer Datei, die Sie mit einem beliebigen Texteditor erstellen können und die alle Felder aus der Btrieve-Datei mit Typ- und Längenangabe enthalten muß, die konvertiert werden sollen. Dabei wird für jedes Feld eine neue Zeile angefangen, und jede Zeile hat die Form *datentyp, länge, feldname*. *datentyp* ist dabei der Datentyp des Feldes; erlaubt sind String, Logical, Integer, Long, Single, Double, SMBF und DMBF (die letzten beiden sind Kürzel für Single- und Double-Microsoft-Binary-Format und werden genauso konvertiert wie Single und Double). *länge* ist die Länge des Feldes in Btrieve; in BASIC spielt sie nur für Strings eine Rolle, da Logical und Integer in INTEGER-, Long in LONG- und Single und Double in DOUBLE-Typen konvertiert werden und diese Typen in BASIC ja eine festgelegte Länge haben. Sie muß trotzdem angegeben werden, weil die Typen in Btrieve variable Längen haben können.

Eine *satzbeschreibung* könnte so aussehen (rechts daneben die dazu passende TYPE-Definition in einem BASIC-Programm):

```
String, 30, Name          Name AS STRING * 30
String, 30, Vorname       Vorname AS STRING * 30
Single, 10, Gehalt        Gehalt AS DOUBLE
Logical, 1, Praemie       Praemie AS INTEGER
```

Um Indizes zu konvertieren, die in Btrieve in einer eigenen Datei stehen, starten Sie nach der Konvertierung der Daten ISAMCVT ein zweites Mal mit denselben Parametern bis auf *dateiname,* für den Sie diesmal den Namen der Indexdatei angeben.

Für die dBase-Konvertierung brauchen Sie keine zusätzlichen Satzbeschreibungsangaben zu machen; sie geschieht automatisch.

Aus der folgenden Tabelle können Sie entnehmen, welche Datentypen Sie verwenden müssen, um eine konvertierte dBase-Datei mit ISAM bearbeiten zu können.

dBASE-Datentyp	*BASIC-Datentyp*
Textfeld mit Länge n oder Memo	STRING * n
Logisch	INTEGER
Zahl ohne Dezimalstellen	INTEGER oder LONG (je nach Bereich)
Zahl mit Dezimalstellen oder Datum	DOUBLE (Datum wird zu Zeitcode, vgl. Kapitel 8)

Für Indizes gilt das, was auch schon zu Btrieve-Dateien gesagt wurde.

MS/ISAM ist, wie gesagt, eine alte und wenig verbreitete Form des ISAM-Systems, sozusagen ein entfernter Vorläufer des heutigen ISAM. Für die Kon-

vertierung von Dateien dieser Bauart sind keine besonderen Regeln zu beachten. Das speicherresidente MS/ISAM-Programm muß geladen sein. SINGLE-Werte werden bei der Konvertierung zu DOUBLE-Werten; DOUBLE, INTEGER, LONG und STRING behalten ihren Typ.

ISAMIO

ISAMIO dient zum Konvertieren eines ISAM-Datenfiles in ein ASCII-File und umgekehrt. ISAMIO wird wie folgt aufgerufen:

```
ISAMIO /modus asciifile isamfile datenbank beschreibung
```

Als *modus* muß entweder I (für Import) oder E (für Export) angegeben werden. *asciifile* ist der Name der ASCII-Datei, *isamfile* der Name der ISAM-Datei, die benutzt werden soll. *datenbank* ist der Name der Datenbank innerhalb der ISAM-Datei, die betroffen ist. *beschreibung* ist der Name einer Satzbeschreibungsdatei. Mit *asciifile* ist die Datei gemeint, in die die Datensätze exportiert bzw. aus der die Datensätze importiert werden sollen.

Beim Import wird aus *asciifile* gelesen und in *isamfile* geschrieben. Wenn die angegebene *datenbank* noch nicht existiert, wird sie neu erzeugt. Die Datei *beschreibung* muß angegeben werden; in ihr sind die einzelnen Elemente (Spalten) der Tabelle in der folgenden Form aufgelistet:

```
typ, größe, spaltenbezeichnung
```

typ beschreibt den Datentyp. INTEGER, LONG, CURRENCY, REAL, VS und BINARY sind erlaubt. Benutzen Sie REAL für DOUBLE-Typen. BINARY ist für selbstdefinierte Typen, Arrays und Strings mit einer Länge von mehr als 255 Zeichen. VS steht für alle anderen Strings. *größe* wird nur bei Strings angegeben und ist die zugehörige Länge. *spaltenbezeichnung* ist die Bezeichnung, die die Spalte später im ISAM-File haben wird. Ein Beispiel:

BASIC-Typvereinbarung	*zugehörige Satzbeschreibungsdatei*
`TYPE ZugriffType`	`LONG,,KundenNummer`
`  KundenNummer AS LONG`	`INTEGER,,BestellMenge`
`  BestellMenge AS INTEGER`	`REAL,,ArtikelNummer`
`  ArtikelNummer AS DOUBLE`	`CURRENCY,,Preis`
`  Preis AS CURRENCY`	`VS,50,Kommentar`
`  Kommentar AS STRING * 50`	`BINARY,,KundenInfo`
`  KundenInfo(0 TO 9) AS INTEGER`	
`END TYPE`	

Um Daten aus einem ASCII-File in eine ISAM-Datenbank zu importieren, schreibt man also:

```
ISAMIO /I asciifile isamfile datenbank beschreibung
```

wobei *beschreibung* der Name der o.g. Satzbeschreibungsdatei ist. Außerdem können hinter dem Namen der Satzbeschreibungsdatei noch die Switches /A, /C oder /F angegeben werden. /A bedeutet, daß die importierten Daten an die Datenbank *angehängt* werden sollen, anstatt sie zu überschreiben. /C bedeutet, daß die Spaltenbezeichnungen aus der Satzbeschreibungsdatei ignoriert werden sollen, und daß die korrekten Spaltenbezeichnungen in der ersten Zeile der zu importierenden Datei stehen. /F schließlich bedeutet, daß die zu importierenden Daten nicht im Standard-ASCII-Format vorliegen, sondern als Felder mit fester Länge. Wenn Sie /F angeben, muß die Satzbeschreibungsdatei zusätzlich am Anfang jeder ihrer Zeilen die Länge des betr. Feldes in der ASCII-Datei nennen.

Ein Beispiel hierzu:

Die Standard-ASCII-Datei

```
30,"Meier",128
123,"Schmitt",111
28,"Huber",912
1,"Allertz",18
```

könnte ohne /F mit der Satzbeschreibungsdatei

```
INTEGER,,Kennzahl
VS,20,Name
INTEGER,,Personalnummer
```

verarbeitet werden. Hat die ASCII-Datei jedoch eine feste Feldlänge (und wird nicht durch Kommata getrennt), so wie diese...

```
  30Meier          128
 123Schmitt        111
  28Huber          912
   1Allertz         18
```

...dann muß sie mit /F konvertiert werden, und zwar zum Beispiel mit der Satzbeschreibungsdatei

```
5,INTEGER,,Kennzahl
12,VS,20,Name
5,INTEGER,,Personalnummer
```

Das Importieren von selbstdefinierten Typen und Arrays ist nicht problemlos. Sie müssen nämlich genau in der Form vorliegen, in der sie im Speicher gehalten werden. Ein Array von 4·4 LONG-Zahlen müßte also als 64-Byte-String (eine LONG-Zahl braucht 4 Bytes Speicherplatz) in der Datei stehen. Ein sol-

cher String ließe sich zwar mit MKL$ erzeugen, doch ist es nicht unwahrscheinlich, daß er Anführungszeichen oder Dateiende-Kennzeichen enthielte, die ISAMIO ziemlich durcheinanderbringen würden.

Der Export von Daten aus einer ISAM-Datei in ein ASCII-File ist wesentlich unproblematischer. Der Befehl lautet einfach:

```
ISAMIO /E asciifile isamfile datenbank
```

wobei es erlaubt ist, noch eine Satzbeschreibungsdatei anzugeben, die dann anhand der der exportierten Daten neu erzeugt wird.

Beim Exportieren sind ebenfalls die zusätzlichen Switches /C und /F zulässig. /C schreibt die Spaltenbezeichnungen in die erste Zeile der erzeugten ASCII-Datei, und /F exportiert die Felder nicht mit Kommata und Anführungszeichen, sondern mit fester Länge (siehe bei Import).

Arrays und selbstdefinierte Typen werden einfach als „Variable text“ exportiert, das heißt, daß ein Array von 10 INTEGER-Zahlen als 20-Byte-Codestring ausgegeben wird und nicht als 10 Zahlen.

ISAMREPR

Das Programm ISAMREPR ist in der Lage, eine durch Diskettenfehler, „Abstürze“ der durch versehentliches Abschalten des Computers unbrauchbar gewordene ISAM-Datei wieder benutzbar zu machen. Wieviel von den ursprünglich enthaltenen Daten dabei erhalten bleibt, hängt von der Schwere des Fehlers ab. Was ISAMREPR nicht aufgrund redundanter Information wiederherstellen kann, das wird gelöscht – alles unter der Prämisse „Hauptsache, nachher geht's wieder“. ISAMREPR wird mit dem Namen der defekten ISAM-Datei als Argument aufgerufen und kennt keinerlei Switches.

ISAMREPR sollte nur auf eine Datei losgelassen werden, wenn der Fehler *Datenbank inkonsistent* beim Zugriff auf eine ISAM-Datenbank auftrat. Wenn ISAMREPR Daten aus der Datenbank löscht, sollte sie danach mit ISAMPACK kompaktiert werden.

ISAMPACK

Wenn Daten aus einer Datenbank gelöscht werden, so werden diese – ähnlich wie beim Löschen von Dateien auf der Festplatte – nicht wirklich gelöscht, sondern nur als überschreibbar markiert. Das heißt, daß neue Datensätze, die später hinzugefügt werden, den Platz einnehmen können, den zuvor der gelöschte Da-

tensatz belegte. Werden aber keine neuen Datensätze mehr hinzugefügt, so bleibt der Platz auf immer benutzt, aber unbelegt.

ISAMPACK *löscht* alle als überschreibbar markierten Datensätze, so daß der Platz, den sie belegten, wirklich frei wird. Allerdings wächst eine ISAM-Datei ja, wie bekannt, nur in 32 KB-Schritten, und beim Schrumpfen verhält es sich ebenso. Also verkleinert ISAMPACK die Datei nur dann, wenn insgesamt mindestens 32 KB als „überschreibbar“ markiert sind. Selbst dann, wenn ISAMPACK die Datei nicht verkleinert, so wird sie doch intern kompaktiert, was den künftigen Zugriff auf die Datei beschleunigen kann. ISAMPACK zeigt außerdem eine umfangreiche Liste aller Datenbanken in der Datei an.

Die Aufrufsyntax von ISAMPACK lautet:

```
ISAMPACK isamdatei [neuedatei]
```

Wenn Sie keine *neuedatei* eingeben, erhält die alte Datei die Extension .BAK, und das Ergebnis der Kompaktierung bekommt denselben Namen wie *isamdatei*.

Toolboxen 13

13.1 Finanzmathematik

Die Toolbox enthält insgesamt 13 Funktionen aus der Zins- und Investitionsrechnung. Dazu zählen Abschreibung, Endwert, interner Zinsfuß, Kapitalwert, Zinssatz und andere. Die Beispiele zu den meisten Funktionen im Referenzteil geben auch „Finanz-Laien“ nützliche Hinweise zur Anwendung der Toolbox.

Die genaue Beschreibung aller Finanzfunktionen finden Sie im Toolbox-Referenzteil.

Die Finance-Routinen sind nicht in BASIC programmiert. Die Quelltexte werden nicht mitgeliefert, sondern nur die fertigen Libraries. Die Quick Library für VBDOS heißt FINANCE.QLB, zum Kompilieren gibt es FINANCE.LIB (für die Emulator-Library) oder FINANCEA.LIB (Alternate Math-Library).

Die Anwendung der Finance-Routinen erfordert außerdem die Einbindung der Include-Datei FINANCE.BI, die die DECLARE-Anweisungen für die verwendeten Funktionen enthält.

13.2 Matrizenmathematik

Berechnungen mit Matrizen sind elementarer Bestandteil vieler mathematischer Operationen. Das Lösen von linearen Gleichungssystemen oder die Ausführung von Basistransformationen sind nur zwei Beispiele dafür. Die Matrizenmathematik-Toolbox enthält Routinen zur Addition, Subtraktion und Multiplikation zweier Matrizen, zum Ermitteln der Determinante, zum Invertieren einer quadratischen Matrix und zum Lösen linearer Gleichungssysteme.

Die Matrizenmathematik-Toolbox ist im Source-File MATH.BAS gespeichert. Um sie zu benutzen, müssen Sie in Ihrem Programm die Include-Datei MATH.BI angeben und die adäquate Library benutzen. Die Libraries heißen MATH.QLB (für VBDOS), MATH.LIB (zum Kompilieren mit Emulator-Library) und MATHA.LIB (zum Kompilieren mit Alternate Math-Library).

Da bei der Matrizenmathematik keine Assembler-Routinen im Spiel sind, können Sie allerdings auch auf sämtliche Libraries verzichten und die Routinen, die Sie aus der Matrizen-Toolbox benötigen, direkt in Ihr Programm kopieren. Achten Sie dabei darauf, daß MATH.BAS zusätzliche, undokumentierte Routinen enthält, die intern Verwendung finden und die Sie eventuell mitkopieren müssen.

Wenn Sie Änderungen an den Routinen vornehmen, können Sie die neuen Libraries gleich aus VBDOS heraus erstellen.

Die Toolbox besitzt für alle Funktionen mehrere Implementationen, eine für jeden Datentyp. Dieser Datentyp wird in Form eines Kennbuchstabens am Ende des Funktionsnamens angebracht; die Argumente müssen dann den entsprechenden Typ haben. Es gilt: I = INTEGER, L = LONG, C = CURRENCY, S = SINGLE, D = DOUBLE. Ausnahmen sind nur die Funktionen zum Invertieren einer Matrix und zum Lösen eines linearen Gleichungssystems. Dort existieren lediglich die Versionen C, S und D.

Die Funktionen der Toolbox (alle Routinen sind als Funktionen implementiert) geben als Funktionswert einen Code zurück, aus dem man entnehmen kann, ob die Operation erfolgreich war. 0 bedeutet OK, andere Werte indizieren – je nach benutzter Funktion – einen Fehler (genauere Informationen im Referenzteil).

Matrizen werden als zweidimensionale Arrays an die Funktionen übergeben.

Anwendungsbeispiele

Im folgenden finden Sie ein kurzes Beispielprogramm abgedruckt, das die Routine `MatSEqnS` (ein Kürzel für Matrix-Solve-Equation-SINGLE) aus der Matrizen-Toolbox benutzt, um ein vom Benutzer eingegebenes lineares Gleichungssystem zu lösen.

```
REM $INCLUDE: 'MATH.BI'

DIM Variable AS INTEGER, Eingabe AS STRING, Zahl AS SINGLE
DIM XPos AS INTEGER, YPos AS INTEGER, ReturnCode AS INTEGER

CLS
DO
   PRINT "Lösen linearer Gleichungssysteme"
   INPUT "Wieviele Variablen (2-7)? ", Variable
LOOP UNTIL Variable > 1 AND Variable < 8

DIM Matrix(1 TO Variable, 1 TO Variable) AS SINGLE
DIM Resultat(1 TO Variable) AS SINGLE

' Ausgabe der Variablenbezeichnungen
FOR i% = 1 TO Variable
   FOR j% = 1 TO Variable
      LOCATE i% * 2 + 2, j% * 10 - 2: PRINT CHR$(96 + j%);
      IF j% < Variable THEN PRINT " + " ELSE PRINT " = "
   NEXT
NEXT
```

⇨

```
PRINT
PRINT "Bitte geben Sie nacheinander die Koeffizienten ein."
PRINT "Geben Sie QUIT ein, um abzubrechen; drücken Sie"
PRINT "F1, um die letzte Eingabe zu wiederholen."

' Taste F1 umdefinieren, damit INPUT auf sie reagiert (CHR$(27) löscht bisherige
' Eingabe, CHR$(255) ist Erkennungsmerkmal, CHR$(13) simuliert Drücken von ENTER)
KEY 1, CHR$(27) + CHR$(255) + CHR$(13)

' Koeffizienten einlesen (Koeffizienten in Matrix(), rechte Seite der Gleichung in
' Resultat())
i% = 0: j% = 1
DO
   j% = 1: i% = i% + 1
   DO
      YPos = i% * 2 + 2: XPos = j% * 10 - 8
      LOCATE YPos, XPos: PRINT SPACE$(6)
      LOCATE YPos, XPos: LINE INPUT Eingabe
      IF UCASE$(Eingabe) = "QUIT" THEN
         CLS : END
      ELSEIF Eingabe = CHR$(255) THEN
         IF j% = 1 AND i% = 1 THEN
            CLS : RUN
         ELSE
            j% = j% - 1: IF j% = 0 THEN i% = i% - 1: j% = Variable + 1
         END IF
      ELSE
         Wert = VAL(Eingabe)
         IF j% > Variable THEN
            Resultat(i%) = Wert
         ELSE
            Matrix(i%, j%) = Wert
         END IF
         Eingabe = LTRIM$(STR$(Wert))
         IF LEN(Eingabe) < 6 THEN
            ' Eingabe formatiert wieder ausgeben
            LOCATE YPos, XPos
            PRINT SPACE$(6 - LEN(Eingabe)); Eingabe
         END IF
         j% = j% + 1
      END IF
   LOOP UNTIL j% > Variable + 1
LOOP UNTIL i% = Variable

PRINT
PRINT "Danke. Die Lösung wird berechnet..."; TAB(80); ""
PRINT TAB(80); "": PRINT TAB(80); ""

' Der große Augenblick: Matrizen-Toolbox wird aufgerufen
ReturnCode = MatSEqnS(Matrix(), Resultat())
```

```
SELECT CASE ReturnCode
CASE -1  ' Null-Determinante
   PRINT "System hat keine Lösung!"
CASE 0   ' Alles o.k.
   PRINT "Die Lösungen sind: ";
   FOR i% = 1 TO Variable
      PRINT CHR$(i% + 96); " ="; STR$(Resultat(i%)); "    ";
   NEXT
CASE ELSE 'irgendein BASIC-Fehler
   PRINT "Fehler"; ERROR$(ReturnCode): PRINT "Lösung nicht möglich"
END SELECT

END
```

Listing 13–1: LINGL.BAS

Ein Beispiel für die Ausgabe dieses Programmes:

```
Wieviele Variablen (2-7)? 3
      3a +     15b +      8c =  -66.1
    6.3a +    -32b +   -1.8c = 253.61
     27a +   -1.8b +     12c =  194.7
Danke. Die Lösung wird berechnet...
Die Lösungen sind: a = 5.5    b =-7    c = 2.8
```

Noch ein Beispiel zur Matrizen-Toolbox. Sie enthält keine Routine zur Division einer Matrix durch eine andere. Die Division einer Matrix A durch eine Matrix B kann jedoch ausgedrückt werden als Multiplikation der Matrix A mit der Inversen der Matrix B. Hier sehen Sie eine Divisionsroutine, die als Ergänzung zur Matrizen-Toolbox benutzt werden kann:

```
' Es können nur quadratische Matrizen invertiert werden. Bei der
' Multiplikation einer k·m-Matrix mit einer m·n-Matrix entsteht eine k·n-
' Matrix. Aus diesen beiden Voraussetzungen folgt, daß
' - 1. Matrix2 quadratisch sein muß (n·n)
' - 2. Matrix1 und Matrix3 die Dimension k·n haben müssen.
' (In Matrix 3 wird das Ergebnis der Division Matrix 1 durch Matrix 2 geschrieben.)
FUNCTION MatDivS (Matrix1() AS SINGLE, Matrix2() AS SINGLE,
                  Matrix3() AS SINGLE) AS INTEGER

   DIM FehlerCode AS INTEGER
   FehlerCode = MatInvS(Matrix2())
   IF FehlerCode = 0 THEN
      FehlerCode = MatMultS(Matrix1(), Matrix2(), Matrix3())
   END IF
   MatDivS% = FehlerCode

END FUNCTION
```

Listing 13–2: (Auszug aus) MATDIV.BAS

(Routinen für andere Datentypen funktionieren analog hierzu.)

Dies soll zur Demonstration der Leistungsfähigkeit genügen; die genaue Beschreibung der einzelnen Matrix-Funktionen entnehmen Sie bitte dem Referenzteil zur Matrizenmathematik-Toolbox.

13.3 Die Font-Toolbox

Üblicherweise steht im Grafikmodus nur eine einzige Schriftart zur Verfügung, die für sämtliche PRINT-Befehle benutzt wird. Das ist der Standard-Font für den jeweiligen Grafikmodus, und er hat zwei große Nachteile: Erstens läßt er sich nicht beliebig, sondern nur zeilen- und spaltenweise positionieren (es sei denn, man arbeitet aufwendig mit GET und PUT für Grafik), und zweitens kann man seine Größe nicht verändern. Außerdem benutzen professionelle Systeme (etwa WINDOWS oder der Apple Macintosh) längst proportionale Schriftarten*.

Die Font-Toolbox ermöglicht die Verwendung verschiedener Schriftarten in verschiedenen Größen und stellt Funktionen zur Verfügung, die den Umgang damit vereinfachen. Die Schriftarten müssen – bis auf eine fest eingebaute – aus Schriftartdateien gelesen werden.

Die Font-Routinen sind größtenteils in BASIC programmiert (FONT.BAS). Einige Unterfunktionen wurden allerdings auch in Assembler erstellt und befinden sich – schon assembliert – in der Datei FONTASM.OBJ (der Assembler-Quelltext wird als FONTASM.ASM mitgeliefert).

Die Quick Library FONT.QLB, die die beiden oben genannten Dateien enthält, dient zur Verwendung der Font-Toolbox in VBDOS; da alle Teile von FONT.QLB auch in der Präsentationsgrafik-Library CHART.QLB enthalten sind, muß die Font-Toolbox nicht extra geladen werden, wenn man mit den Chart-Routinen arbeitet. Dasselbe gilt für die Libraries: FONT.LIB ist in CHART.LIB eingeschlossen. FONTA.LIB dient zur Verwendung mit der Alternate Math-Library und ist auch in CHARTA.LIB enthalten.

Jedes Programm, in dem die Font-Routinen benutzt werden sollen, muß die Include-Datei FONT.BI enthalten.

Zur Toolbox gehören auch sämtliche FON-Dateien. Mitgeliefert werden Helvetica, Times Roman und Courier. Für jede dieser Schriftarten gibt es drei Dateien mit den Zusatzbezeichnungen A, B und E. Insgesamt stehen also zur Verfü-

* Proportionale Schriftarten sind solche, bei denen die Buchstaben verschieden breit sind, ein w zum Beispiel wesentlich mehr Platz belegt als ein i.

gung: HELVA, HELVB, HELVE, TMSRA, TMSRB, TMSRE, COURA, COURB und COURE. Die Bedeutung der Buchstabenzusätze wird im Abschnitt „Die Schriftart-Dateien“ erläutert.

Wenn Sie Änderungen an den Font-Routinen vornehmen, müssen Sie alle Font-Libraries und die Quick Library der Präsentationsgrafik-Toolbox neu erstellen. Die Font-Quick Library wird so produziert:

```
BC FONT /X;
LINK /Q FONT+FONTASM,FONT.QLB,,VBDOSQLB;
```

Dabei entsteht die neue Quick Library FONT.QLB mit den geänderten Routinen.

Die FONT.LIB-Library ändern Sie so:

```
BC FONT /X;
LIB FONT.LIB -+FONT;
```

(Für FONTA.LIB kompilieren Sie mit /FPa.)

Alle Font-Libraries sind ohne den /G2- oder /G3-Switch erstellt. Wenn Sie Programme erzeugen, die nur auf 286er- bzw. 386er-Rechnern laufen, können Sie die Libraries unter Verwendung dieser Switches neu erstellen, um etwas Platz und Zeit einzusparen.

Wenn Sie mit den mitgelieferten Schriftarten (*.FON) arbeiten wollen, müssen diese Dateien zur Laufzeit des Programmes vorhanden sein. Es gibt keine Möglichkeit, FON-Dateien fest in ein EXE-File einzubauen (siehe dazu aber auch `RegisterMemFont` im Font-Referenzteil).

Die FON-Dateien sind gewöhnliche WINDOWS-Bitmapfonts*; jede andere Bitmap-Fontdatei von WINDOWS kann ebenfalls benutzt werden. Es ist nicht möglich, TrueType-Schriftartendateien zu verwenden.

Anwendung

Die wesentlichen Elemente und die Vorgehensweise bei der Verwendung der Schriftarten, die diese Toolbox zur Verfügung stellt, lassen sich am besten an den Schriftarten selbst erklären. Eine Schriftart – der Kürze halber hier auch Font genannt – kann verschiedene Stati haben. Der niedrigste Status wäre „ist auf der Diskette/Festplatte vorhanden“. Dieser Status äußert sich in der Toolbox überhaupt nicht; der Font belegt nur auf dem Datenträger einen gewissen Speicherplatz und kann nicht benutzt werden. Es können – selbstversätndlich – be-

* Im Gegensatz zu einem Vektorfont werden die Buchstaben eines Bitmapfonts als kleine Bilder gespeichert, die eine feste Anzahl von Pixeln benötigen und deshalb nicht vergrößert oder verkleinert werden können.

liebig viele Fonts vorhanden sein. Der nächsthöhere Status ist „registriert“. Über einen registrierten Font können nähere Informationen eingeholt werden, oder er kann gleich geladen werden. Das Registrieren eines vorhandenen Fonts geschieht durch eine dafür vorgesehene Prozedur der Toolbox (`RegisterFont`). Sie können beliebig viele Fonts registrieren lassen. Jeder registrierte Font belegt etwa 200 Bytes im Speicher Ihres Programms. FON-Dateien, die meist mehrere Fonts enthalten, können nur komplett registriert werden (keine einzelnen Fonts aus einer Datei).

Ein registrierter Font kann jedoch noch immer nicht benutzt werden. Dazu müssen Sie ihn zunächst – wieder mit einer speziellen Prozedur – in den nächsthöheren Status versetzen: Er muß geladen werden. Dabei werden sämtliche Font-Informationen aus der Datei in den Speicher gelesen, und deshalb benötigt ein geladener Font nicht eben wenig Speicherplatz (mehrere KB, je nach Komplexität). Sie können beliebig viele Fonts laden, solange noch Speicher frei ist.

Schließlich, bevor das erste Zeichen durch die Textausgabefunktion `OutGText` am Bildschirm erscheint, müssen Sie den Font noch aktivieren. Da mehrere Fonts geladen sein können, muß der Toolbox gesondert mitgeteilt werden, welcher davon zu benutzen ist. Auch dafür gibt es eine Prozedur.

Die Toolbox verwaltet also zwei Bereiche im Speicher Ihres Programms: Einen mit der Liste aller registrierten Fonts, die sich aber de facto noch auf der Platte oder Diskette befinden, und einen mit den Daten der geladenen Fonts.

Die Schriftart-Dateien

Wie eingangs erwähnt, wird eine Anzahl von Schriftart-Dateien mitgeliefert. Die HELV- und TMSR-Dateien enthalten je 6 proportionale, die COUR-Dateien je 3 nichtproportionale Schriftarten. Die Varianten A, B und E einer jeden Datei stehen für verschiedene Bildschirmauflösungen.

Dies ist notwendig, da es sich um Bitmap-Schriftarten handelt, die beispielsweise bei einer Auflösung von 720 x 350 Bildschirmpunkten anders aussehen als bei 640 x 480 Punkten; es ergeben sich Verzerrungen. Sie können aus der Font-Tabelle entnehmen oder noch besser ausprobieren, welche Gruppen sich für welche Bildschirm-Modi eignen. Relativ unverzerrte Ergebnisse erhalten Sie, wenn Sie sich an diese Verteilung halten:

Gruppe	*für SCREEN-Modi*
A	2, 3, 4, 8
B	1, 7, 9, 10, 13
E	11 und 12

Die interne Schriftart

Damit im schlimmsten Fall – wenn aus irgendeinem Grunde keine Schriftartdateien registriert und geladen werden können – trotzdem mindestens eine Schriftart verfügbar ist, sind die Daten für einen simplen, nichtproportionalen 8 x 8-Font fest in der Toolbox enthalten. Inkonsequenterweise befinden sich die Routine dafür und die Font-Daten nicht in der Datei FONTASM.OBJ, sondern in CHARTASM.OBJ, die zur Präsentationsgrafik-Toolbox gehört. Wenn Sie mit der Präsentationsgrafik-Toolbox arbeiten, verfügen Sie automatisch über die interne Schriftart. Ansonsten müssen Sie in Ihr Programm die Zeile

```
DECLARE SUB DefaultFont (SEG Segment%, SEG Offset%)
```

einfügen und die Datei CHARTASM.OBJ in die Quick Library einbauen (schreiben Sie in die LINK-Zeile weiter oben im Kapitel einfach hinter FONTASM noch +CHARTASM dazu). Außerdem müssen Sie CHARTASM.OBJ entweder bei jedem separaten Kompiliervorgang einzeln angeben (zum Beispiel `LINK PROGRAMM+CHARTASM,,,FONT;`), oder Sie verwenden gleich die CHART- anstelle der FONT-Library.

Die interne Schriftart muß mit einer speziellen Prozedur registriert und wie alle anderen geladen werden, bevor sie benutzt werden kann. Details darüber finden Sie im Font-Referenzteil unter `RegisterMemFont` und `SetGCharSet`.

Anwendungsbeispiel

Die einfachsten Font-Funktionen demonstriert dieses kleine Beispiel, das dazu geeignet ist, bestimmte Font-Dateien auf dem Bildschirm anzuzeigen (SCREEN 12 kann ohne Schwierigkeiten in einen anderen Modus geändert werden).

Das Programm benutzt nur die Standard-Routinen zum Registrieren und Laden von Schriftart und zur Textausgabe. Darüberhinaus bietet die Toolbox noch die Möglichkeit, umfangreiche Informationen über die gerade benutzte Schriftart einzuholen. Ich benutze die Routine GetFontInfo im Beispiel nur, um die Schrifthöhe festzustellen.

Weitere Routinen stellen die Textfarbe ein und bestimmen die Richtung, in der der Text ausgegeben wird. Die Details finden Sie im Referenzteil.

```
REM $INCLUDE: 'FONT.BI'
DIM Beschreibung AS FontInfo, FontDatei AS STRING,
DIM FontAnzahl AS INTEGER, PixelZeile AS SINGLE

DATA TMSRA, HELVB, COURC, TMSRE, *
SCREEN 12
PixelZeile = 5
```

```
DO
   ' Fontdateinamen aus DATA-Zeilen lesen
   READ FontDatei
   IF FontDatei = "*" THEN EXIT DO
   FontDatei = FontDatei + ".FON"

   ' eventuell noch registrierte Fonts löschen
   UnRegisterFonts: SetMaxFonts 6, 6

   ' Neue Datei registrieren
   FontAnzahl = RegisterFonts(FontDatei)

   FOR a% = 1 TO FontAnzahl
      ' Font Nummer a% laden (alle anderen geladenen Fonts werden gelöscht)
      x% = LoadFont("n" + STR$(a%))

      ' SelectFont ist nicht erforderlich, denn es ist ja nur ein Font geladen
      ' Text ausgeben
      x% = OutGText(0, PixelZeile, "Datei "+FontDatei+", Schriftart Nr."+STR$(a%))

      ' Beschreibung zu aktuellem (=geladenem) Font holen und Zeilenzähler erhöhen
      GetFontInfo Beschreibung
      PixelZeile = PixelZeile + 5 + Beschreibung.PixHeight
   NEXT
LOOP
```

Listing 13–3: FONTDEMO.BAS

Die Ausgabe dieses Programms sieht etwa so aus:

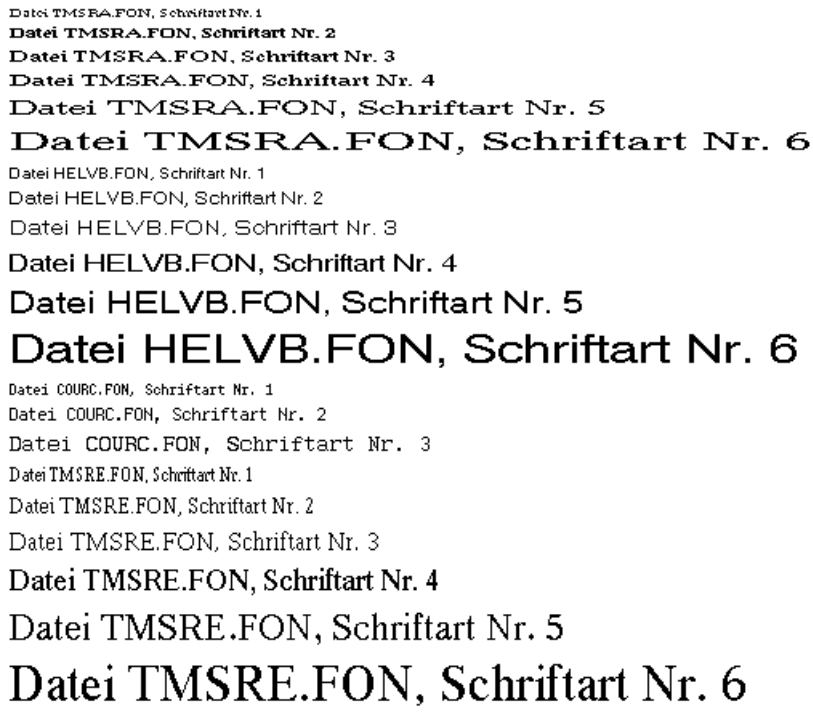


Abbildung 13–1: Schriftarten der Font-Toolbox

Fehlermeldungen der Font-Toolbox

Die Font-Toolbox hat eine eigene globale Fehlervariable namens `FontErr`. Mittels dieser kann festgestellt werden, ob die Font-Routinen ordnungsgemäß gearbeitet haben. Die Aufschlüsselung der möglichen Fehlermeldungen finden Sie im Anhang C.

Programmfehler in der Font-Toolbox

Die Font-Toolbox enthält einen Fehler, der sich dahingehend auswirkt, daß der „t"-Parameter der `LoadFont`-Funktion nicht funktioniert und seine Verwendung das Laden von Schriftarten in den meisten Fällen scheitern läßt. Die Beschreibung zu `LoadFont` im Referenzteil geht davon aus, daß Sie gegebenenfalls die im folgenden genannte Korrektur durchgeführt haben. Erzeugen Sie nach der Änderung die Font-Libraries neu. Wenn die Funktion `flGetNextSpec` in der Datei FONT.BAS bei Ihnen ab Zeile 43 auch so aussieht...

```
' Scan for font title until blank or end of string:
StartPos% = ChPos%
DO UNTIL ChPos% > SpecLen%
   Char$ = MID$(SpecTxt$, ChPos%, 1)
   ChPos% = ChPos% + 1
LOOP
```

... dann müssen Sie eine Zeile einfügen, so daß es danach heißt:

```
' Scan for font title until blank or end of string:
StartPos% = ChPos%
DO UNTIL ChPos% > SpecLen%
   Char$ = MID$(SpecTxt$, ChPos%, 1)
   IF Char$ = "/" THEN EXIT DO
   ChPos% = ChPos% + 1
LOOP
```

13.4 Präsentationsgrafik

Die Grafik-Toolbox dient dazu, typische Geschäftsgrafiken einfach und schnell zu erstellen. Mit Hilfe der Toolbox können Sie mühelos in ein bestehendes Programm einige simple Grafikfunktionen einbauen. Sie eignet sich wunderbar, um einem gewöhnlichen Programm durch ein paar Grafiktricks „das gewisse Etwas" zu verleihen. In Programmen, deren wesentlicher Bestandteil das Erstellen von Grafiken ist, ist die Anwendung dieser Toolbox schon schwieriger,

denn je mehr man von den Standard-Einstellungen abweicht und individuelle Anforderungen hat, desto weniger Arbeit kann die Toolbox übernehmen.

Wenn man mit der Standardgrafik zufrieden ist, kann man mit vier, fünf Zeilen eine wirklich vorzeigbare Grafik auf den Bildschirm zaubern, und ein bißchen Benutzeroberfläche reicht aus, um ein eindrucksvolles Grafikpaket (nämlich die CHRTDEMO-Programme von Microsoft, die im VBDOS-Lieferumfang enthalten sind) zu erstellen.

Die Toolbox besteht aus einigen Routinen, die Grafik darstellen, einigen Hilfsroutinen, die die vorhergehenden Berechnungen erleichtern und schließlich noch ein paar Prozeduren, die zusätzliche Texte an die Achsen schreiben oder bei der Berechnung von Musterpaletten helfen.

Die meisten Präsentationsgrafik-Routinen sind in BASIC programmiert. Die Quellcodes werden in der Datei CHART.BAS mitgeliefert. Außerdem gehören die Daten der internen Schriftart dazu, die sich in der Datei CHARTASM.OBJ befinden (der Assembler-Quelltext heißt CHARTASM.ASM).

Die Präsentationsgrafik läuft nicht ohne die Font-Toolbox (FONT.BAS und FONTASM.OBJ); daher ist die Font-Toolbox komplett in den CHART-Libraries enthalten.

Beim Installieren wird eine Quick Library namens CHART.QLB erzeugt, die alle vier oben genannten Dateien enthält und die Benutzung der Präsentationsgrafik-Toolbox innerhalb von VBDOS ermöglicht.

Außerdem werden die Libraries CHART.LIB und CHARTA.LIB für Emulator- bzw. Alternate Math-Library zur Verfügung gestellt.

Jedes Programm, das die Präsentationsgrafik-Toolbox benutzen will, muß die Dateien CHART.BI und FONT.BI als Include-Dateien laden.

Wenn Sie Änderungen an den BASIC-Chart-Routinen vornehmen, müssen Sie sämtliche Chart-Libraries neu erstellen. Für die Quick Library geht das so:

```
BC CHART /X;
BC FONT /X;
LINK /Q CHART+CHARTASM+FONT+FONTASM,CHART.QLB,,VBDOSQLB;
```

Dann entsteht eine neue Quick Library namens CHART.QLB mit den geänderten Routinen.

Die CHART.LIB-Library ändern Sie so:

```
BC CHART /X;
BC FONT /X;
LIB CHART.LIB -+CHART-+FONT;
```

Alle Chart-Libraries sind ohne den /G2- oder /G3-Switch erstellt. Wenn Sie Programme erzeugen, die nur auf 286er- bzw. 386er-Rechnern laufen, können Sie die Libraries unter Verwendung dieser Switches erstellen, um etwas Platz und Zeit einzusparen.

Allgemeine Hinweise

Die Toolbox kennt vier Arten von Grafiken: Balken- (horizontal und vertikal), Linien-, Kuchen- und Punktgrafiken.

Balken-, Linien- und Kuchengrafiken haben gemeinsam, daß sie eine gewisse Anzahl von Elementen grafisch darstellen, wobei jedem Element eine Bezeichnung und ein Wert zugeordnet ist. Bei Punktgrafiken, die hauptsächlich für statistische Zwecke benutzt werden, tritt an die Stelle einer Bezeichnung ein zweiter Wert.

Eine Variable vom Typ `ChartEnvironment` spielt die zentrale Rolle in der Grafik-Toolbox. Sie bestimmt – neben den darzustellenden Werten natürlich – das Aussehen der Grafik in allen Details. Zum Glück existieren Routinen in der Toolbox, die diese umfangreiche Variable (ein Standard-`ChartEnvironment` verbraucht immerhin 596 Bytes!) automatisch auf sinnvolle Werte einstellen, so daß man sich – wenn man damit zufrieden ist – um nichts mehr kümmern muß.

Anwendungsbeispiele

Das folgende Programm...

```
REM $INCLUDE: 'FONT.BI'
REM $INCLUDE: 'CHART.BI'
DIM Wert(1 TO 5) AS SINGLE, Label(1 TO 5)  AS STRING, GrafikArt AS ChartEnvironment

Wert(1) = 7: Label(1) = "Peter":     Wert(2) = 10.5: Label(2) = "Egon"
Wert(3) = 12: Label(3) = "Claudia":  Wert(4) = 8.9: Label(4) = "Herbert"
Wert(5) = 3: Label(5) = "Bernd"

ChartScreen 11
DefaultChart GrafikArt, cColumn, cPlain
Chart GrafikArt, Label(), Wert(), 5

DO WHILE LEN(INKEY$) = 0: LOOP
```

Listing 13–4: GRDEMO1.BAS

...reicht schon aus, um die Grafik auf der folgenden Seite zu erzeugen:

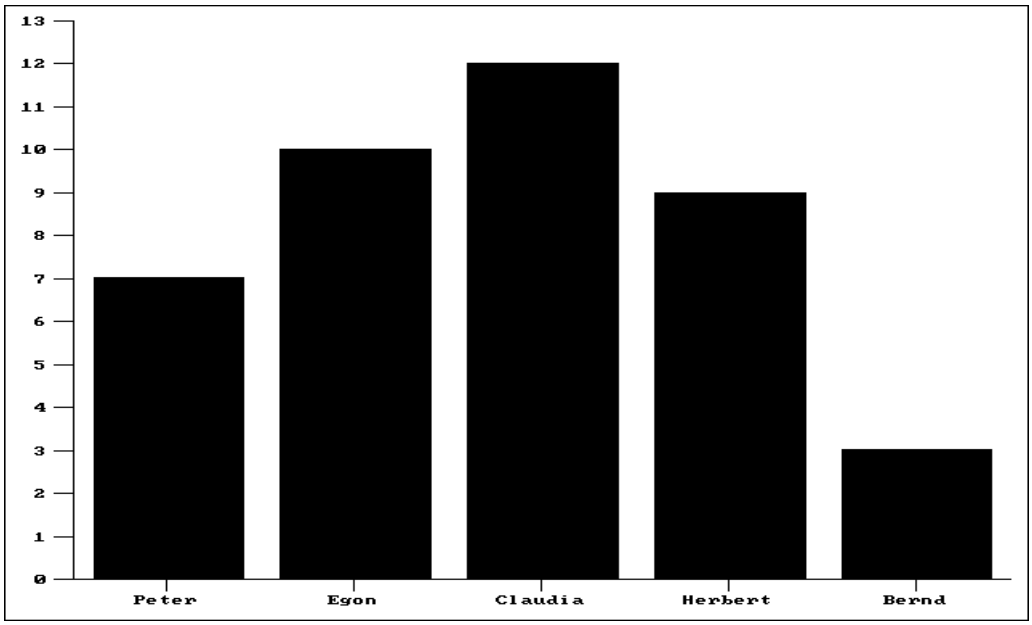


Abbildung 13–2: Einfache Balkengrafik mit der Chart-Toolbox

Die im Programm benutzten Konstanten `cColumn` und `cPlain` sind, ebenso wie eine ganze Reihe weiterer Konstanten, in CHART.BI definiert.

Was geht dabei vor? Zunächst muß mit der Routine `ChartScreen` ein Grafikmodus eingeschaltet werden. `ChartScreen` arbeitet im Prinzip genau wie der eingebaute SCREEN-Befehl. Allerdings funktioniert die Toolbox nicht, wenn man den SCREEN-Befehl benutzt, weil `ChartScreen` zugleich noch einige globale Variablen setzt, zum Beispiel für die Anzahl der Pixel in x- und y-Richtung. Benutzen Sie einfach immer `ChartScreen` anstatt SCREEN, wenn Sie mit der Grafik-Toolbox arbeiten. In unserem Beispiel wird mit `ChartScreen 11` ein hochauflösender VGA-Modus ausgewählt, aber selbst der Modus 1, der nur etwa ein Fünftel der Auflösung des Modus 11 bietet, hätte zu einer ähnlichen Grafik geführt.

Danach ruft unser Beispielprogramm die Routine `DefaultChart` auf, die dafür zuständig ist, die Standardwerte für die gewünschte Grafik in die Beschreibungsvariable *GrafikArt* zu schreiben. Dieser Routine werden der gewünschte Grafiktyp (im Beispiel `cColumn` für vertikale Balken) und eine von zwei Varianten des Typs (`cPlain` heißt „Balken nebeneinander“, `cStacked` für „Balken aufeinander“ wäre die andere Möglichkeit gewesen) übergeben. Sie sehen, daß zur Grafik-Toolbox eine ganze Anzahl von Konstanten wie `cPlain` und `cColumn` gehören. Sie werden alle in CHART.BI definiert und beginnen alle mit einem C. Auch die Konstanten der Font-Toolbox haben ein C als ersten Buchstaben.

Schließlich wird die `Chart`-Routine aufgerufen, die für Balken- und Liniengrafiken zuständig ist. Ihr werden *GrafikArt*, die Daten und ihre Anzahl übergeben, und sie zeichnet dann entsprechend den Anweisungen in *GrafikArt* ein Bild auf den Schirm.

Neben der einfachen `Chart`-Routine gibt es noch eine Prozedur, mit der man beliebig viele Datenreihen gleichzeitig zeichnen kann, zwei Prozeduren für Punktgrafik und `ChartPie` für Kuchengrafik. Alle funktionieren nach dem hier gezeigten Prinzip und sind im Referenzteil genau beschrieben.

Die ChartEnvironment-Variable und ihre Subtypen

Der Kern der Toolbox ist, wie schon erwähnt, die Variable, die ich im Beispiel *GrafikArt* genannt habe. Mit ihr kann man das Aussehen der gesamten Grafik bestimmen. Sie ist vom Typ `ChartEnvironment`, der in eine Vielzahl von Subtypen zerlegt ist. Alle sind in CHART.BI definiert.

Ich will mich hier jedoch darauf beschränken, ein Beispiel für die möglichen Manipulationen zu zeigen; falls Sie größere Manipulationen im Schilde führen, muß ich Sie auf das Studium der gutdokumentierten Quelldatei CHART.BAS oder der Microsoft-Dokumentation verweisen.

Durch einige geringfügige Manipulationen an der *GrafikArt*-Variablen ist es leicht möglich, auch eine Grafik wie die folgende zu erzeugen, die schon recht anspruchsvoll aussieht:

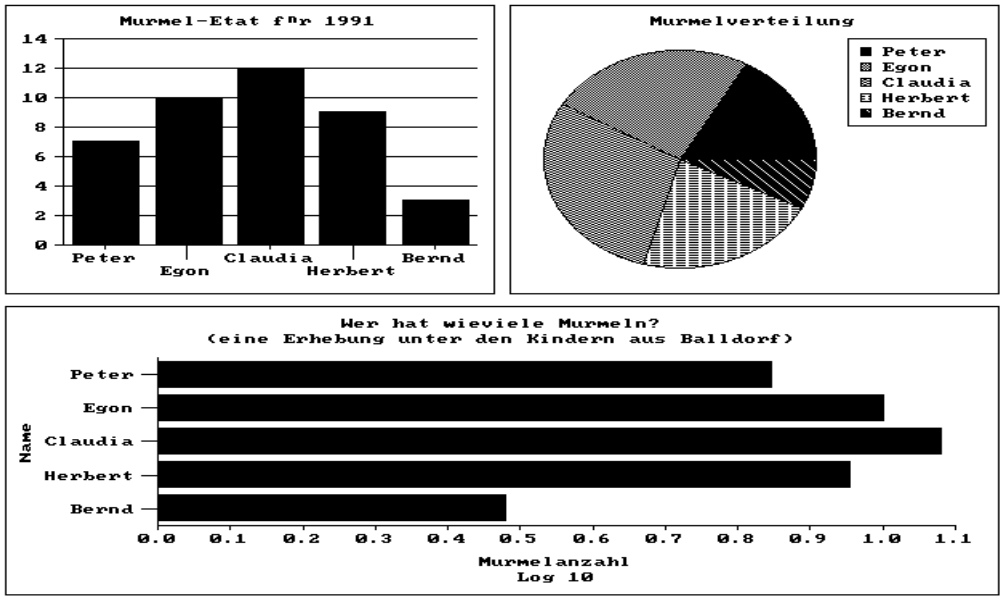


Abbildung 13–3: Auch kompliziertere Grafiken sind mit der Chart-Toolbox kein Problem

Das Programm dazu baut auf unserem ersten Beispielprogramm auf; die Zahlen mußten etwas geändert werden, da Kinder ungern mit halben Murmeln spielen. Das Prinzip der Grafikerzeugung ist das gleiche wie im ersten Beispiel. Lediglich werden hier die von der `DefaultChart`-Routine erstellten Werte ein wenig manipuliert, bevor die Grafik ausgegeben werden darf.

```
REM $INCLUDE: 'FONT.BI'
REM $INCLUDE: 'CHART.BI'
DIM Wert(1 TO 5) AS SINGLE, Label(1 TO 5)  AS STRING
DIM Ausruecken(1 TO 5) AS INTEGER, GrafikArt AS ChartEnvironment

Wert(1) = 7: Label(1) = "Peter"    : Wert(2) = 10: Label(2) = "Egon"
Wert(3) = 12: Label(3) = "Claudia" : Wert(4) = 9: Label(4) = "Herbert"
Wert(5) = 3: Label(5) = "Bernd"

ChartScreen 11

DefaultChart GrafikArt, cBar, cPlain
GrafikArt.ChartWindow.x1 = 10   : GrafikArt.ChartWindow.y1 = 245
GrafikArt.ChartWindow.x2 = 630  : GrafikArt.ChartWindow.y2 = 470
GrafikArt.MainTitle.Title = "Wer hat wieviele Murmeln?"
GrafikArt.SubTitle.Title = "(eine Erhebung unter den Kindern aus Balldorf)"
GrafikArt.XAxis.RangeType = cLog
GrafikArt.XAxis.AxisTitle.Title = "Murmelanzahl"
GrafikArt.YAxis.AxisTitle.Title = "Name"
Chart GrafikArt, Label(), Wert(), 5

DefaultChart GrafikArt, cColumn, cPlain
GrafikArt.ChartWindow.x1 = 10  : GrafikArt.ChartWindow.y1 = 10
GrafikArt.ChartWindow.x2 = 315 : GrafikArt.ChartWindow.y2 = 235
GrafikArt.MainTitle.Title = "Murmel-Etat für 1991"
GrafikArt.YAxis.Grid = cYes
Chart GrafikArt, Label(), Wert(), 5

DefaultChart GrafikArt, cPie, cNoPercent
GrafikArt.ChartWindow.x1 = 325 : GrafikArt.ChartWindow.y1 = 10
GrafikArt.ChartWindow.x2 = 630 : GrafikArt.ChartWindow.y2 = 235
GrafikArt.MainTitle.Title = "Murmelverteilung"
GrafikArt.Legend.LegendWindow.Border = cYes
ChartPie GrafikArt, Label(), Wert(), Ausruecken(), 5

DO WHILE LEN(INKEY$) = 0: LOOP
```

Listing 13–5: GRDEMO2.BAS

Wenn Sie sich die Grafik etwas genauer anschauen, werden Sie allerdings etwas finden, das Ihr europäisches Gemüt betrübt. Das ü in der Überschrift der ersten Grafik ist verstümmelt worden. Das liegt an einem Fehler in der Grafik-Toolbox. Dieser Fehler tritt nur bei der internen Schriftart auf, die benutzt wird,

wenn Sie keine Schriftart von der Diskette laden lassen. Wie Sie den Umlauten zu Geltung verhelfen, steht im Abschnitt über `SetGCharSet` im Referenzteil.

Sie können das Problem auch umgehen, indem Sie eine Schriftart laden, bevor Sie die Grafik-Routinen benutzen. Wenn nämlich Schriftarten geladen sind, benutzen die Grafikroutinen nicht den internen Schrifttyp, sondern einen der geladenen (üblicherweise den ersten, das kann aber durch die diversen Font-Variablen im ChartEnvironment-Typ verändert werden).

Die Analyze-Routinen

Sie haben im obenstehenden Beispiel schon beobachten können, daß ich mir von `DefaultChart` die Grafikdaten nach Standard berechnen ließ und dann nur einige Daten zusätzlich in die *ChartEnvironment*-Variable eintrug.

Um das in größerem Stil zu tun, müssen Sie sich allerdings mehr „vorrechnen“ lassen, als `DefaultChart` das macht. Dazu dienen die Analyze-Routinen. Zu jeder Grafikfunktion (wie `ChartPie` im Beispiel) gibt es eine zugehörige Analyze-Prozedur (in diesem Falle `AnalyzePie`), die dieselben Argumente hat wie ihre verwandte Chart-Prozedur und auch dieselben Berechnungen durchführt, aber keine Grafik zeichnet.

Wenn Sie zum Beispiel die Legende Ihrer Kuchengrafik um 10 Pixel weiter rechts gezeichnet haben wollen, als die `ChartPie`-Routine das normalerweise tut, können Sie zuerst `AnalyzePie` aufrufen, damit die Größe der Legende berechnet wird. Dann setzen Sie *GrafikArt.Legend.AutoSize* auf `cNo` und führen Ihre Manipulationen am Legenden-Fenster *GrafikArt.Legend.Legendwindow* durch. Ohne die Analyze-Routine hätten Sie es sich nicht erlauben können, mit `AutoSize = cNo` die Neuberechnung der Legendenposition und -größe zu unterbinden, weil Sie die benötigte Größe nicht kannten.

Die Farb- und Musterpalette

Die Grafik-Toolbox verwaltet eine interne Farb- und Musterpalette, die aus sechzehn Einträgen besteht. Jedem Paletteneintrag sind eine Farbe, ein Linientyp, ein Füllmuster, ein Zeichen für Punkte in Linien- und Punktgrafiken und ein Linientyp für Rahmen zugeordnet.

Was die Palette genau enthält, hängt vom Bildschirmmodus ab, den Sie mit der Prozedur `ChartScreen` einstellen. In einem Modus, in dem es sechzehn Farben gibt, wird zum Beispiel jedem Paletteneintrag eine unterschiedliche Farbe zugeordnet. Das Füllmuster ist dann bei jedem Paletteneintrag eine einfache Fläche in der jeweiligen Farbe. Bei einem anderen Modus, der nur zwei Farben kennt,

ist zum Beispiel nur dem Paletteneintrag 0 die Farbe 0, allen anderen die Farbe 1 zugeordnet. Dafür unterscheiden die Einträge sich stärker im Füllmuster. Es gibt hier Schraffierungen und echte Muster, nicht nur Flächen. Auch die Linientypen sind bei einem zweifarbigen Modus unterscheidbar, während bei einem sechzehnfarbigen Modus alle Linientypen durchgezogen sind, da die Farbe hier ja den Unterschied macht. In einem vierfarbigen Modus gäbe es von jeder Farbe eine durchgezogene und drei verschieden gestrichelte Linien, so daß sich wieder 16 Kombinationen ergeben.

Die Farbe des Paletteneintrags Nr. 0 ist immer schwarz, die des ersten immer weiß, und alle anderen werden nach Verfügbarkeit zugeordnet.

Die Punktzeichen und die Linientypen für Rahmen sind von der Farbverfügbarkeit unabhängig.

Im Normalfall brauchen Sie sich um die Palette nicht zu kümmern. Die Chart-Routinen sorgen selbst dafür, daß verschiedene Kurven, verschiedene Balken oder Kuchenstücke gut voneinander unterscheidbar sind.

Sollten Sie dennoch einmal die Palette speziell anpassen wollen, stehen Ihnen dafür Funktionen zur Verfügung, die die Palette auslesen bzw. neu eintragen (`GetPaletteDef` und `SetPaletteDef`). Nähere Erläuterungen finden Sie im Referenzteil.

13.5 Mausroutinen

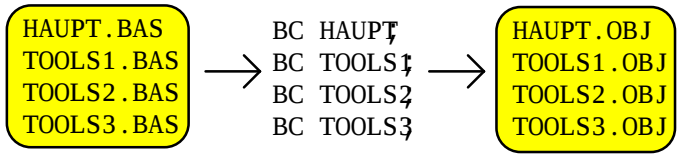
Die Prozeduren und Funktionen zur Mausunterstützung stellen mehr oder weniger nur eine Übertragung der Maustreiberfunktionen auf geeignete BASIC-Äquivalente dar. Sie sind nur im Referenzteil beschrieben. Wenn Ihnen die vorhandenen Routinen zur Mauskontrolle nicht ausreichen, können Sie leicht mit Hilfe von Interrupts oder der `MouseDriver`-Routine den Maustreiber direkt ansprechen. Sie müssen dazu lediglich die entsprechenden Maus-Interrupt-Funktionsnummern kennen (theoretisch von Maustreiber zu Maustreiber unterschiedlich; es gibt jedoch einen Standard von Microsoft, an den sich fast alle halten; zu finden in technischen PC-Handbüchern oder im „Microsoft Mouse Programmer's Guide“).

Sie werden die Mausroutinen im Textmodus selten einsetzen, denn dort bieten die formbasierten Anwendungsprogramme eine sehr einfache Möglichkeit der Mausunterstützung. Interessant werden die Routinen erst im Grafikmodus, weil dort keine Formen angezeigt werden können und Sie auf eigene Konstrukte angewiesen sind.

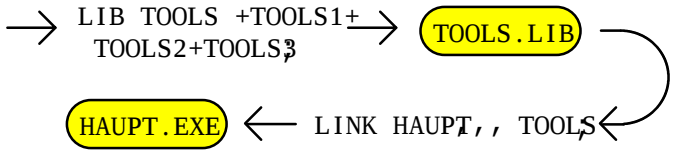
Individuelle Runtime-Module 14

14.1 Wozu Runtime-Module?

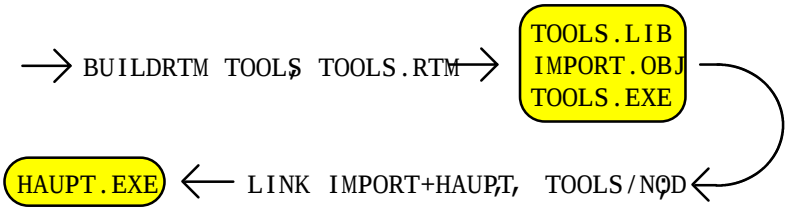
Vielleicht erinnern Sie sich noch an dieses Beispiel aus Kapitel 6:



Bis hierher ändere ich nichts. Die TOOLS-Dateien enthalten Hilfsroutinen und HAUPT.OBJ ist die Startdatei. Im Original ging es jetzt so weiter:



Dabei wird ein einziges EXE-File erzeugt, in dem alle TOOLS-Routinen fest eingebunden sind. Mit BUILDRTM gibt es nun die folgende Alternative:



Ein ziemlich ähnlicher Prozeß. Ich habe BUILDRTM statt LIB benutzt, dabei ist genau wie vorher ein TOOLS.LIB entstanden, zusätzlich aber auch noch IMPORT.OBJ und TOOLS.EXE. Außerdem kam zusätzlich eine Datei namens TOOLS.RTM ins Spiel, deren Herkunft noch ungeklärt ist. (Es handelt sich um eine „Exportliste“, und wir werden später mehr davon hören.) Was Sie im Diagramm nicht sehen können, sind folgende Details:

- TOOLS.LIB ist wesentlich kleiner als zuvor.
- HAUPT.EXE ist ebenfalls geschrumpft.
- HAUPT.EXE funktioniert nicht, wenn es TOOLS.EXE nicht finden kann.

Wenn man von den Unterschieden bei der Herstellung des Programms absieht, ist die einzige Veränderung, daß die TOOLS-Routinen vorher schon beim Linken fest in das Programm eingebunden und ein Teil dessen wurden, während das Einbinden der TOOLS-Routinen nun nicht schon beim Linken, sondern *erst zur Laufzeit* passiert. HAUPT.EXE enthält also nicht mehr die TOOLS-Routinen, sondern nur die Information „Hole dir die TOOLS-Routinen aus TOOLS.EXE“. Deshalb wird TOOLS.EXE das *Runtime-Modul* (oder Laufzeit-

modul) von HAUPT.EXE genannt. TOOLS.EXE allein ist nicht lauffähig. Bei seinem Aufruf erscheint nur eine Meldung auf dem Bildschirm.

Jedes Runtime-Modul besitzt eine zugehörige Library (in unserem Fall TOOLS.LIB), die Verweise auf das Modul enthält; bei der Erstellung von eigenen Runtime-Modulen wie TOOLS.EXE wird zusätzlich noch eine Datei namens IMPORT.OBJ erzeugt, die unbedingt beim Linken mit angegeben werden muß. Außerdem muß beim Linken mit eigenen Runtime-Libraries immer der Switch /NOD angegeben werden.

Der Vorteil dieses Verfahrens ist, daß sich mehrere Programme das gleiche Runtime-Modul teilen können. Haben Sie drei Programme auf der Platte, die alle die TOOLS-Routinen benutzen, müssen Sie, wenn Sie ohne Runtime-Modul arbeiten, dieselben Routinen in alle drei Programme einbinden, so daß sie sich nachher wirklich dreimal auf der Festplatte befinden. Verwenden Sie stattdessen das Runtime-Modul TOOLS.EXE, können alle drei Programme die TOOLS-Routinen daraus aufrufen, und der Platz wird nur einmal belegt.

Ein Programm kann nur ein einziges Runtime-Modul benutzen. Beim Linken wird in das Programm der Name des verwendeten Runtime-Moduls geschrieben, so daß das lauffähige Programm später das richtige Runtime-Modul findet. Außerdem wird auch das Datum des Runtime-Moduls in Ihr Programm eingebunden. Daher reicht es nicht aus, dem Programm irgendein Runtime-Modul mit dem richtigen Namen hinzustellen, sondern es muß genau das sein, das gleichzeitig mit der Runtime-Verweislibrary (hier TOOLS.LIB) erstellt wurde.

14.2 Standard-Runtime-Module

Ähnlich wie ich im obigen Beispiel geht VBDOS auch selbst vor: Um ein kompiliertes BASIC-Programm laufen zu lassen, wird eine große Zahl von Hilfsroutinen benötigt. Diese Routinen rufen Sie nie direkt auf, und Sie werden auch ihre Namen nicht kennen; die meisten BASIC-Befehle, die Sie verwenden, werden jedoch intern in einen Aufruf irgendeiner dieser Funktionen verwandelt.

Auch für diese Routinen können Sie wählen, ob sie direkt in das EXE-File eingebunden oder in einem Runtime-Modul zur Verfügung gestellt werden sollen.

Bei der Installation von VBDOS werden die Runtime-Module VBDRT-10E.EXE (für die Emulator-Library) und VBDRT10A.LIB (Alternate Math-Library) auf die Festplatte kopiert. In der Standard-Version von VBDOS heißt das Runtime-Modul einheitlich VBDRT10.EXE.

Unter gleichem Namen werden die Runtime-Referenzlibraries kopiert (sie heißen .LIB).

Wenn Sie EXE-Programme mit dem Compiler-Switch /O bzw. der VBDOS-Option „Eigenständige EXE-Datei“ erstellen, wird beim Linken keine der Runtime-Referenzlibraries, sondern VBDCL10A.LIB bzw. VBDCL10E.LIB verwendet. Diese Libraries enthalten keine Verweise auf das Runtime-Modul, sondern direkt alle benötigten Funktionen. Dadurch wird Ihr Programm um einiges länger, läuft dafür aber später unabhängig von einem Runtime-Modul. Verzichten Sie stattdessen auf /O bzw. wählen „EXE benötigt Laufzeitmodul“ in VBDOS, werden beim Linken automatisch die o. g. Libraries eingebunden, und Ihr Programm benötigt später ein Runtime-Modul zur Ausführung.

14.3 Erstellen eines Runtime-Moduls mit BUILDRTM

Um ein eigenes Runtime-Modul (kurz ab jetzt: RTM) herzustellen, müssen Sie zunächst sämtliche BAS-Dateien kompilieren, die im RTM eingeschlossen werden sollen. Achten Sie dabei unbedingt darauf, daß alle mit denselben Switches für Mathematik-Library (/FPa oder /FPi) kompiliert werden. Auch alle Programme, die später das neue RTM benutzen sollen, müssen mit der entsprechenden Option kompiliert werden.

Die Exportliste

Wenn alle Programme kompiliert sind, müssen Sie – mit einem beliebigen Texteditor oder mit VBDOS als Dokument – eine sogenannte Exportliste anlegen. Diese hat die folgende Form:

```
#OBJECTS
dateiname
...
#EXPORTS
routinenname
...
#LIBRARIES
dateiname
...
```

Unter dem Schlüsselwort „#OBJECTS“ werden die Namen aller OBJ-Dateien aufgezählt, die in das RTM aufgenommen werden sollen. Unter „#EXPORTS“ müssen Sie alle die Namen aller SUBs und FUNCTIONs aufzählen, die Sie aus den angegebenen OBJ-Dateien benutzen wollen. Geben Sie den bloßen Namen an, ohne SUB oder FUNCTION, ohne Parameter und bei Funktionen auch ohne den Typbezeichner. Wenn die Routinen, die Sie ins RTM einbinden wollen,

ihrerseits Routinen aus Libraries benutzen (zum Beispiel aus der Font-Toolbox), müssen Sie diese Libraries unter dem Schlüsselwort „#LIBRARIES“ aufführen, damit BUILDRTM die benötigten Teile daraus in das RTM inkorporieren kann. Das ist notwendig, weil Runtime-Module in sich konsistent sein müssen, das heißt, keine Routine in einem RTM darf eine andere Routine außerhalb aufrufen. Damit wäre die Exportliste fertig. Sie können übrigens beliebig viele Leerzeilen und Kommentarzeilen (letztere beginnen immer mit #) in die Liste einstreuen.

Bevor Sie BUILDRTM aufrufen, sollten Sie sicherstellen, daß im aktuellen oder in Ihrem LIB-Directory die Libraries VBDA10.LIB, VBDC10.LIB, VBDE10.LIB und VBDL10.LIB vorhanden sind, die BUILDRTM zum Erstellen des RTMs braucht.

Der Aufruf von BUILDRTM

`BUILDRTM` *`switches rtm exportliste`*

Als *switches* sind /FPa (Alternate Math-Library) und /FPi (Emulator-Library) möglich, außerdem /FPi87, den Sie verwenden sollten, wenn Sie sicher sind, daß das fertige RTM nur auf Rechnern zum Einsatz kommt, die mit einem Coprozessor ausgestattet sind. Wenn Sie weder /FPa noch /FPi angeben, wird /FPi angenommen.

Der Switch /H zeigt eine Hilfsseite zu BUILDRTM an, verhindert aber jede andere Tätigkeit. /MAP als Zusatz sorgt dafür, daß unter dem Namen *rtm*.MAP eine komplette Liste für das erzeugte RTM ausgegeben wird. Da diese Liste auch alle BASIC-internen Funktionen enthält, ist sie recht unübersichtlich. Praktischer ist es, wenn Sie sich zur späteren Information einfach die Exportliste zu jedem RTM, das Sie erstellen, aufheben.

rtm ist der Name für das RTM und die Runtime-Library. Sie dürfen keine Extension angeben, da BUILDRTM selbständig .LIB (für die Runtime-Library) und .EXE (für das RTM selbst) anhängt.

exportliste ist der Name der Exportliste, die die Informationen über das zu erzeugende RTM enthält.

BUILDRTM erstellt daraufhin – wenn zwischendurch kein Fehler auftritt – drei Dateien: Die Runtime-Library *rtm*.LIB, das RTM *rtm*.EXE und eine dritte Datei namens IMPORT.OBJ (die Sie am besten sofort von Hand umbenennen, damit Sie nicht durcheinanderkommen, wenn Sie mit verschiedenen RTMs arbeiten – jedes RTM braucht ein anderes IMPORT.OBJ). Wenn Sie in Zukunft Programme linken, die mit dem RTM laufen sollen, müssen Sie die Datei IMPORT.OBJ

(immer als erstes) und *rtm*.LIB (im Library-Feld) sowie den Switch /NOD angeben, wie in diesem Beispiel:

```
LINK IMPORT+HAUPT.OBJ,HAUPT.EXE,,TOOLS/NOD;
```

Bei einem LINK-Vorgang wie diesem hier mit eigenem RTM spielt es übrigens keine Rolle, ob das Programm HAUPT.BAS mit oder ohne Switch /O kompiliert wurde. /O entscheidet üblicherweise darüber, ob die BASIC-Hilfsroutinen aus einem RTM genommen oder fest in das Programm eingebaut werden sollen; haben Sie aber ein eigenes RTM (in das BUILDRTM stets sämtliche BASIC-Hilfsroutinen inkorporiert), dann ist selbstverständlich, daß die Routinen aus dem RTM geholt werden.

14.4 Technische Details

Runtime-Module und Verzicht-Files

Im Kapitel 23 ist eine Anzahl von Verzicht-Files aufgeführt, die die Größe eines Programmes verringern, indem Sie die Aufnahme bestimmter BASIC-Routinen verhindern. Sie können diese Verzicht-Files auch in den #OBJECTS-Teil der Exportliste aufnehmen und so Ihr Runtime-Modul verkleinern. Dann können die Befehle, die Sie dadurch entfernen, von keinem Programm, das das RTM benutzt, verwendet werden.

Insbesondere sind hier zwei Verzicht-Files zu erwähnen, die nur für die Verwendung mit Runtime-Modulen vorgesehen sind:

- NOISAM.OBJ entfernt die ISAM-Unterstützung im RTM (dadurch wird das RTM um etwa 80 KB kürzer); dann kann aber mit diesem RTM kein Programm kompiliert werden, das ISAM-Routinen aufruft, auch dann nicht, wenn vorher PROISAMD.EXE geladen wird.
- NOFORMS.OBJ entfernt die Unterstützung von Formen und ereignisgesteuerter Programmierung (auch die Befehle MSGBOX und INPUTBOX können dann nicht mehr verwendet werden). Die Verwendung von NOFORMS.OBJ verkleinert das RTM um etwa 235 KB.

Toolboxen in Runtime-Modulen

Wenn Sie ein RTM erstellen möchten, das alle Routinen einer oder mehrerer Toolboxen (z.B. Chart, Font, Fincance) enthält, müssen Sie dazu nur den Namen der LIB-Datei in die Exportliste eintragen. Achten Sie darauf, daß Sie für die gewählten Einstellungen (/FPi bzw. /FPa) die richtige Library benutzen.

Es gibt zwei Möglichkeiten, eine Toolbox (oder auch jede andere Library) in ein RTM einzubauen:

- Wenn Sie eine Library unter „#OBJECTS“ eintragen, wird der gesamte Inhalt der Library ins RTM übernommen. Alle Library-Routinen, die Sie von außerhalb aufrufen möchten, müssen namentlich unter „#EXPORTS“ erwähnt werden.
- Sie können den Library-Namen auch statt unter „#OBJECTS“ unter „#LIBRARIES“ in der Exportliste anführen und darauf verzichten, Eintragungen unter „#EXPORTS“ vorzunehmen. Dann stehen die entsprechenden Toolbox-Routinen jedoch nur den SUBs und FUNCTIONs zur Verfügung, die sich ebenfalls im RTM befinden. Von außerhalb können Sie dann keine der Routinen benutzen. Bei dieser Methode wird u. U. nicht die gesamte Library in das RTM übertragen, sondern nur die Teile daraus, die wirklich benötigt werden.

Inkompatible Runtime-Module

Wenn bei dem Versuch, ein Programm zu starten, das mit einem selbstgebastelten RTM arbeitet, die Meldung *Laufzeitmodul nicht kompatibel* erscheint, haben Sie wahrscheinlich einen der folgenden Fehler gemacht:

- Das Programm wurde mit /FPa kompiliert, das Runtime-Modul aber mit /FPi (oder umgekehrt).
- Das Programm hat einen COMMON-Block, der nicht mit dem verträglich ist, den die Routinen im RTM haben.
- Sie haben versehentlich beim Linken ein falsches IMPORT.OBJ benutzt, also nicht das, welches beim Erstellen des verwendeten RTMs erzeugt wurde.
- Sie haben das Runtime-Modul inzwischen neu erstellt (evtl. sogar, ohne etwas verändert zu haben), das Programm aber nicht neu gelinkt. Dadurch schlägt die Datumsprüfung fehl, die VBDOS vornimmt.
- Sie haben beim Linken IMPORT.OBJ nicht als erste OBJ-Datei genannt.

Fließkommazahlen und der Coprozessor 15

15.1 Details über Fließkommazahlen

Solange Ihr Programm nur Ganzzahlen (INTEGER und LONG) einsetzt und auch keine Befehle verwendet, die intern mit Fließkommazahlen arbeiten, können alle Rechenoperationen in Ihrem Programm vom Prozessor direkt ausgeführt werden. Der Compiler BC kann also solche BASIC-Befehle wie `Zahl& = Laenge& + Breite&` sofort in Maschinencode umwandeln, ohne eine der Routinen aus der Library aufrufen zu müssen (der Umgang mit LONG-Zahlen ist daher auch der Bereich, in dem der Einsatz des /G3-Switches enorme Vorteile bringt; BC nutzt dann nämlich spezielle Befehle des 386er Prozessors zur Behandlung von 32-Bit-Ganzzahlen).

Probleme gibt es erst bei der Verwendung von Fließkommazahlen, also SINGLE und DOUBLE. CURRENCY als Festkommadatentyp fällt ein wenig aus der Rolle (mehr dazu später). Darüber hinaus arbeiten viele Befehle intern mit Fließkommazahlen, ohne daß man es vermuten würde. Dazu zählen zum Beispiel SOUND, FORMAT$, die Verwendung der Zeitcode-Funktionen oder auch der Zufallsgenerator (RND); auch der Divisionsoperator / sorgt automatisch dafür, daß seine Argumente zunächst in Fließkommazahlen umgewandelt werden. (Verwenden Sie den Ganzzahl-Divisionsoperator \, um das zu verhindern.)

Die Befehle, die die Fließkommaeinheit benötigen, sind im Diskettenreferenzteil unter der Rubrik „Kompatibel“ mit „Mathe +“ markiert.

Für Rechenoperationen mit Fließkommazahlen besitzt der Prozessor – vom 486er und dem Pentium einmal abgesehen – keine eingebauten Befehle. BC baut an diesen Stellen einen Funktionsaufruf an eine Routine aus der Mathematik-Library ein, die ein Teil der VBDCL10-Library bzw. des Runtime-Moduls ist. Eine Ausnahme bilden die CURRENCY-Zahlen, die intern wie eine Ganzzahl gehandhabt werden. Hier können die elementaren Rechenoperationen zwar direkt in Prozessorbefehle übersetzt werden; immer, wenn eine CURRENCY-Zahl allerdings in einen anderen Datentyp umgewandelt (z. B. `a% = b@`), dividiert oder mit PRINT ausgegeben wird, schlägt auch hier die Mathematik-Library zu.

Da Aufrufe an die Mathematik-Library immer langsamer sind als Befehle, die der Prozessor direkt verarbeitet, sollten Sie versuchen, stets mit Ganzzahlen zu arbeiten und auch den CURRENCY-Datentyp zu meiden, wo es möglich ist. Ein Programm, das mit DM-Beträgen arbeitet, kann sehr leicht so geschrieben werden, daß intern alle Summen in einer LONG-Zahl als Pfennigbeträge ge-

speichert werden. Dann liegt die höchste darstellbare Zahl etwa in der Größenordnung von 42 Millionen DM, was häufig ausreicht, und Sie können auf Fließkommaarithmetik verzichten.

Genauigkeit bei der Fließkomma-Arithemtik

„Jetzt habe ich hier so einen teuren Computer stehen, und mein alter Taschenrechner rechnet trotzdem besser!“

Ein Programm wie das folgende hat vielleicht auch schon einmal Sie zu einem solchen Ausruf verleitet:

```
a = .05: b = 10: c = a + b
PRINT c - b
```

Anstatt anständig .05 wird nämlich die Zahl .05000019 angezeigt, und solche Phänomene können ein Programm ganz schön durcheinanderbringen. Wird in der nächsten Zeile nämlich geprüft „`IF c - b = .05 THEN...`“, passiert gar nichts.

Woher kommen diese Fehler? Selbst bei DOUBLE-Zahlen, die immerhin 8 Byte Speicher belegen, treten Rundungsfehler dieser Art auf.

Es gibt zwei Erklärungen hierfür. Die erste erschließt sich durch ein wenig Nachdenken: Mit 8 Byte kann man maximal 256^8 verschiedene Zustände (Zahlen) darstellen. Das sind etwa $1{,}8 \cdot 10^{19}$ Kombinationen. Eine DOUBLE-Zahl hat aber einen Wertebereich von $\pm 1{,}8 \cdot 10^{308}$. Das heißt nichts anderes, als daß jede DOUBLE-Zahl in Wirklichkeit sage und schreibe einen Zahlenbereich von etwa $2 \cdot 10^{298}$ repräsentiert, oder, anders ausgedrückt: Ich könnte etwa $2 \cdot 10^{298}$ verschiedene ganze Zahlen samt beliebig vieler dazwischenliegender Brüche nennen, die für VBDOS alle „gleich“ sind. (Und da spricht noch jemand von Genauigkeit?)

Die zweite Erklärung führt uns ein wenig in die Tiefen der Fließkommakodierung, und es bleibt Ihnen überlassen, ob Sie sich mit dieser „grauen Theorie“ beschäftigen möchten. Wenn nicht, lesen Sie auf Seite 244 weiter, ab der es wieder um die VBDOS-Praxis geht.

Das IEEE-Format

Irgendwie, das hatten wir festgestellt, müssen rund $2 \cdot 10^{298}$ Zahlen auf einem DOUBLE-Wert abgebildet werden. Wenn diese Aufteilung nun linear vorgenommen würde, hätte das fatale Folgen. Es würde nämlich bedeuten, daß die Werte 0 bis $2 \cdot 10^{298}$ alle als „gleich“ behandelt würden, ebenso die Werte

$2 \cdot 10^{298}+1$ bis $4 \cdot 10^{298}$ usw. – vernünftige Berechnungen wären hier natürlich nicht mehr möglich.

Das IEEE-Format, mit dem auch VBDOS arbeitet, macht sich stattdessen zu Nutze, daß eine Abweichung von einer Million nicht so schwerwiegend ist, wenn man mit Zahlen arbeitet, die ohnehin in Größenordnungen von mehreren Fantastilliarden (oder sagen wir, von $x \cdot 10^{200}$) liegen. Die verschiedenen Werte, die eine DOUBLE-Zahl annehmen kann, liegen also im Bereich um ±1 sehr eng beieinander, und VBDOS unterscheidet sehr wohl zwischen 0,00001 und 0,000011; die Werte 1.000.000.000,00001 und 1.000.000.000,000011 werden aber als „gleich“ angesehen, obwohl sie sich um den gleichen Betrag unterscheiden.

Die Speicherung einer Fließkommazahl wird durchgeführt, indem jede Zahl zunächst in eine Binärdarstellung umgewandelt wird. Dabei haben die Bits links vom „Dezimalkomma“ (das hier ja eher ein „Binärkomma“ ist) die üblichen Werte 1, 2, 4 usw., während vom Komma nach rechts 1/2, 1/4, 1/8 usw. gezählt wird. Und hier sind wir auch schon den meisten Rundungsfehlern auf der Spur: Die meisten Zahlen *lassen* sich nämlich gar nicht exakt umwandeln; es tritt dasselbe Phänomen auf, das Sie auch daran hindert, die Dezimalzahl 1/3 zu Papier zu bringen.

Die Zahl 2,5 ist leicht umzuwandeln: Sie heißt binär 10,1 (1·2+0·1+1·1/2). Aber schon bei 2,1 (um nur ein Beispiel zu nennen) gibt es Schwierigkeiten:

2	*1*	*,*	*1/2*	*1/4*	*1/8*	*1/16*	*1/32*	*1/64*	*1/ 128*	*1/ 256*	*1/ 512*	*1/ 1024*	*1/ 2048*	*1/ 4096*	*1/ 8192*
1	0	,	0	0	0	1	1	0	0	1	1	0	0	1	1

Selbst diese Annäherung (die grob geschätzt erst den Wert 2,0997 erreicht) braucht schon 15 Bit, und den genauen Wert wird man – ebenso, wie man 1/3 nicht durch eine 0,33333 mit endlos vielen Dreien darstellen kann – nie erreichen.

Wenn die Binärdarstellung der Zahl festliegt, wird sie so lange nach links oder rechts verschoben, bis das Bit mit dem Wert 1 auf 1 gesetzt ist und alle Bits links davon 0 sind; in unserem Beispiel wäre eine Verschiebung um eine Position nach rechts notwendig:

1	*,*	*1/2*	*1/4*	*1/8*	*1/16*	*1/32*	*1/64*	*1/ 128*	*1/ 256*	*1/ 512*	*1/ 1024*	*1/ 2048*	*1/ 4096*	*1/ 8192*	*1/ 16384*
1	,	0	0	0	0	1	1	0	0	1	1	0	0	1	1

Das Bit mit dem Wert 1 muß nicht gespeichert werden, da es ja nach der Verschiebung *immer* 1 ist; die Verschiebung – der Exponent – wird allerdings gespeichert, und zwar als negative Zahl für Rechts- und als positive für Linksverschiebung. Nun haben wir einen Exponenten (in unserem Beispiel –1) und eine Mantisse (in unserem Beispiel 00001100110011) erhalten, die gespeichert werden. Für SINGLE- und DOUBLE-Datentypen stehen in der Emulator-Library folgende Bitbreiten zur Verfügung:

Datentyp	*Anzahl Bits für x (Mantisse)*	*Anzahl Bits für y (Exponent)*
SINGLE	23	7
DOUBLE	52	10

Zwei Bits werden jeweils benötigt, um ein negatives Vorzeichen der Mantisse oder des Exponenten zu kennzeichnen; so ergibt sich der Speicherbedarf von 4 Bytes (32 Bit) für SINGLE- und 8 Bytes für DOUBLE-Zahlen.

Falls der Exponent bei der Kodierung einer Fließkommazahl mehr als 7 (bzw. 10) Bits benötigt, erzeugt VBDOS einen „Überlauf"-Fehler. Reichen die Bits für die Mantisse nicht aus, werden einfach die ersten 23 (bzw. 52) genommen, und der Rest wird ignoriert.

MKS$ und MKD$ entschlüsseln

Mit diesem Wissen können Sie jetzt nicht nur Rundungsfehler vorhersagen und vermeiden, sondern auch die Funktionen MKS$ und MKD$ „verstehen". MKD$ zum Beispiel wandelt eine DOUBLE-Zahl in einen 8 Byte langen String um; diesen String können Sie mittels der ASC-Funktion in acht Werte zwischen 0 und 255 zerlegen. Daraus ergeben sich – binär hintereinandergeschrieben – 64 Bit, von denen die ersten 52 für die Mantisse, das 53. für das Vorzeichen der Mantisse (1 = negativ), die folgenden 10 für den Exponenten und das 64. schließlich für das Vorzeichen des Exponenten steht.

15.2 Zwei Mathematik-Libraries

Wenn Sie trotz der Geschwindigkeitsnachteile und Rundungsfehler nicht um die Verwendung von SINGLE- und DOUBLE-Datentypen herumkommen, haben Sie in der professionellen Ausgabe von VBDOS die Wahl zwischen zwei verschiedenen Mathematik-Libraries. Zur Verfügung stehen

- die „Emulator"-Library, die Fließkomma-Rechenbefehle so aufbereitet, daß ein eventuell vorhandener Coprozessor bzw. die interne Fließkommaeinheit des 486 und Pentium sie übernehmen kann, und

- die „Alternate Math“-Library, die so arbeitet, daß ein eventuell vorhandener Coprozessor überhaupt nicht angesprochen wird.

Beim Kompilieren ist die Verwendung der Emulator-Library die Standardeinstellung. Die Standardversion von VBDOS enthält nur die Emulator-Library.

Die VBDOS-Libraries und die mitgelieferten Toolboxen gibt es jeweils in zwei Versionen: Eine für die Emulator- und eine für die Alternate Math-Bibliothek. Man erkennt den Unterschied meist an einem „E“ oder „A“ im Namen:

Standardversion (nur Emulator)	*Profi-Version Emulator*	*Alternate Math*
VBDCL10.LIB	VBDCL10E.LIB	VBDCL10A.LIB
VBDRT10.LIB	VBDRT10E.LIB	VBDRT10A.LIB
VBDRT10.EXE	VBDRT10E.EXE	VBDRT10A.EXE
CMNDLG.LIB	CMNDLG.LIB	CMNDLGA.LIB
–	FINANCE.LIB	FINANCEA.LIB
–	CHART.LIB	CHARTA.LIB
–	FONT.LIB	FONTA.LIB

Die beiden Libraries dürfen nicht vermischt werden; es ist zum Beispiel nicht möglich, beim Linken eines Programms, das ohne /FPa kompiliert wurde, die Library CHARTA.LIB zu verwenden.

Quick Libraries sind nur mit der Emulator-Library zu erstellen. VBDOS zeigt den Fehler „Format unzulässig“ an, wenn Sie versuchen, eine Quick Library zu laden, die mit /FPa kompiliert wurde.

Die Emulator-Library

Die Emulator-Library verwendet intern für alle Fließkomma-Befehle ein Format, das ein Coprozessor verstehen kann, und stellt gleichzeitig eigene Rechenroutinen zur Verfügung. Ist auf dem Rechner, auf dem das Programm später läuft, kein Coprozessor vorhanden, dann emuliert die Library ihn softwaremäßig (daher ihr Name). Das hat zur Folge, daß alle Berechnungen – ob ein Coprozessor vorhanden ist oder nicht – mit der für Coprozessoren typischen hohen Präzision durchgeführt werden: Zwischenergebnisse bei komplexen Ausdrükken werden mit einer Genauigkeit von 64 Bit für die Mantisse berechnet, und erst bei der Speicherung des Endergebnisses in einer Variablen muß auf 23 bzw. 52 Bit (SINGLE bzw. DOUBLE) gerundet werden.

Wenn Sie sicher wissen, daß das Zielsystem einen Coprozessor besitzt, können Sie durch Linken mit der Library 87.LIB verhindern, daß die Emulationsrouti-

nen überhaupt eingebunden werden. Ihr EXE-Programm wird dann um etwa 9 KB kürzer (es sei denn, Sie benutzen ein Runtime-Modul).

Der Nachteil der Emulator-Library ist, daß sie auf einem System ohne Coprozessor etwas langsamer ist als die Alternate Math-Library. Bei ihrer Verwendung entstehen grundsätzlich längere Programme als mit der Alternate Math-Version.

Die Alternate Math-Library

Die Alternate Math-Library bürdet die gesamte Rechenlast stets dem Hauptprozessor auf, egal, ob nun ein Coprozessor vorhanden ist oder nicht.

Auf Systemen ohne Coprozessor schneiden Programme, die mit der Alternate-Library erstellt wurden, sowohl von der Größe als auch von der Ausführungsgeschwindigkeit her besser ab als ihre Emulator-Pendants. Diesen Vorteil bezahlt man jedoch mit einem Verlust an Rechengenauigkeit. Die Alternate-Library verwendet für Zwischenergebnisse nur die Standard-Mantisse von 23 Bits für SINGLE- und 52 Bits für DOUBLE-Berechnungen. Das macht sich zwar bei einfachen Berechnungen, die wenig Zwischenergebnisse erfordern, nicht bemerkbar: Die Anweisung `X! = Y! * Z!` werden beide Libraries gleich ausführen. Bei komplizierteren Ausdrücken wie `X! = (Y! * Z!) + 3 * (A! + B!)` kommt es aber vor, daß die Ergebnisse der Alternate-Library etwas von den korrekten abweichen. Außerdem sind die von der Alternate-Library berechneten Werte transzendenter Funktionen zuweilen in der letzten Dezimalstelle falsch.

Die Verwendung der Alternate Math-Library schließt die Nutzung des CURRENCY-Datentyps aus.

Fazit

In den meisten Fällen wird die Genauigkeit der Alternate-Library ausreichend sein. Sie sollten sie dann verwenden, wenn Ihr Programm entweder hauptsächlich auf Rechnern ohne Coprozessor eingesetzt wird oder ohnehin nur wenig Fließkomma-Rechenoperationen durchführt.

Für besonders rechenintensive Probleme, für Anwendungen, bei denen es wahrscheinlich ist, daß das Gros der Betreiber einen Coprozessor oder einen 486- oder Pentium-Rechner benutzt, dürfte die Emulator-Library die bessere Wahl sein.

Der Microsoft Profiler 16

16.1 Sinn und Zweck des Profilers

Der Profiler ist ein Programm, das andere Programme ablaufen läßt und dabei Messungen anstellt. Sie können mit dem Profiler feststellen,

- ob und wie oft bestimmte Prozeduren, Funktionen oder Zeilen ausgeführt werden und
- wieviel Zeit durchschnittlich für die Ausführung einer Prozedur, Funktion oder Zeile benötigt wird.

Dadurch ermöglicht Ihnen der Profiler

- empirische Leistungsvergleiche zwischen verschiedenen Algorithmen,
- das Ermitteln „toter Programmbereiche“, die nie ausgeführt werden und
- die Programmoptimierung aufgrund von Ausführungshäufigkeiten.

Der Vorteil des Profilers gegenüber Messungen von Hand in der VBDOS-Umgebung ist, daß Sie auch für umfangreiche Messungen keinerlei Programmänderungen vornehmen müssen. Meßergebnisse können so auch nicht durch zusätzlichen Code, der zum Messen eingebaut wird, verzerrt werden.

Sie können die Arbeit des Profilers auf bestimmte Bereiche Ihres Programms beschränken; wenn Sie zum Beispiel zwei Programmversionen vergleichen wollen, in denen Sie nur die Prozedur SORT geändert haben (um zu testen, welche besser ist), ist es möglich, den Profiler auf diese eine Prozedur anzusetzen.

Der folgende Abschnitt erläutert die Funktionsweise des Profilers; vielleicht möchten Sie sich allerdings die Details ersparen und gleich zum Abschnitt 3 weiterblättern, der sich mit der einfachen Anwendung beschäftigt.

16.2 Ein Profiler – vier Programme

Der Profiler ist ein recht kompliziertes System aus drei Programmen zuzüglich einer „Shell“, einer Benutzeroberfläche, die die Bedienung vereinfachen soll.

Das eigentliche Profiler-Programm, PROFILER.EXE, liest eine .PBI-Datei, in der die Informationen über Art und Umfang der Messungen stehen müssen, startet das zu messende Programm und erzeugt nach dessen Ende eine .PBO-Datei, die die Ergebnisse enthält.

Da die .PBI- und .PBO-Dateien jedoch verschlüsselt und mit gewöhnlichen Editoren nicht zu bearbeiten sind, benötigt man zu ihrer Verarbeitung das Programm PREP.EXE. Dieses kann eine .PBI-Datei erzeugen und eine .PBO-Datei lesen. Die Ergebnisse werden dabei in eine Tabelle kopiert (.PBT), die bei jedem Lauf neu erzeugt werden kann, die es aber auch ermöglicht, Ergebnisse aus mehreren Profiler-Läufen zu sammeln.

Da PREP.EXE über die .PBI-Datei den Profiler steuert, muß man PREP.EXE und nicht PROFILER.EXE mitteilen, welche Art Messung man wünscht. Das geht – in bescheidenem Umfang – bei PREP über die Befehlszeile; besser ist es aber, mit einer .PCF-Datei zu arbeiten, in der man nun (endlich) mit einem gewöhnlichen Texteditor seine Profiler-Wünsche bekannt gibt.

Schließlich ist die .PBT-Datei, die nach einem Profiler-Lauf von PREP erzeugt wird, auch noch binär und muß erst von einem dritten Hilfsprogramm, PLIST.EXE, in ein verständliches Format umgewandelt werden. PLIST kann man noch Sortier-Optionen für die Ausgabe mitteilen, und es schließt auch Daten aus den .BAS- und .FRM-Dateien mit in die Liste ein.

Das vierte Programm, von dem in der Überschrift die Rede ist, ist „VBDPROF.EXE“ und wird im nächsten Abschnitt beschrieben.

Ein vollständiger Profiler-Ablauf gestaltet sich also wie folgt:

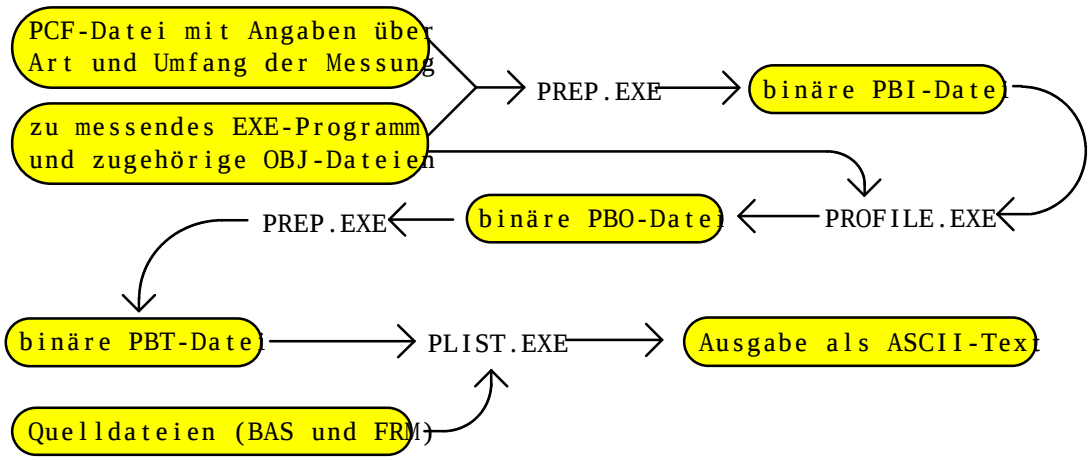


Abbildung 16–1: Der Profiler und seine Hilfsprogramme

Zum besseren Verständnis hier kurz die Klartexte der Dateikürzel:

Kürzel	*Bedeutung*
PCF	Profiler Command File
PBI	Profiler Binary Input
PBO	Profiler Binary Output
PBT	Profiler Binary Table

Voraussetzungen für die Verwendung des Profilers

Sie können den Profiler nur dann sinnvoll auf Ihre Programme anwenden, wenn diese mit der Option /Zi kompiliert und mit /CO, aber ohne /EX gelinkt wurden. Da VBDOS automatisch immer den Switch /EX beim Linken verwendet, müssen Sie Programme, die mit dem Profiler bearbeitet werden sollen, von Hand kompilieren und linken.

Weitere Informationen zu diesem Verfahren finden Sie in Kapitel 6.

Da mit /Zi und /CO erstellte Programme oft deutlich länger sind als üblich, sollten Sie diese Programmversionen nur ausnahmsweise zur Verwendung mit dem Profiler (und evtl. CodeView) erzeugen.

Mögliche Befehle in der PCF-Datei

Die PCF-Datei ist der wichtigste Bestandteil des Profiler-Laufes, weil Sie mit ihr bestimmen, welche Abschnitte des Programms auf welche Weise gemessen werden sollen.

Als erste Zeile in der PCF-Datei geben Sie den Profiler-Modus an. Dieser besteht aus zwei Worten. Er beginnt stets mit LPROFILE (für die zeilenweise Erfassung der Daten) oder FPROFILE (für die Erfassung pro Prozedur und Funktion). Danach folgt ein Leerzeichen, dann COUNT, COVERAGE, TIME oder SAMPLE. COUNT zählt, wie oft eine Zeile oder Prozedur aufgerufen wurde. COVERAGE stellt nur fest, ob eine Zeile oder Prozedur aufgerufen wurde oder nicht, und ist deshalb wesentlich schneller als COUNT. TIME mißt die Ausführungszeit und Aufrufhäufigkeit jeder Zeile oder Prozedur. SAMPLE ist eine Stichprobentechnik: in regelmäßigen Abständen (bis zu 10.000mal pro Sekunde) prüft der Profiler, welche Zeile oder Prozedur gerade ausgeführt wird. Dadurch kann ein schneller Überblick über die Aktivität des Programms gewonnen werden, ohne die (oft recht langwierige) Zeitmessung durchzuführen.

Hinweis: Bei meinen Experimenten habe ich festgestellt, daß die Ergebnisse der SAMPLE-Funktion nicht immer korrekt waren. Es kann jedoch sein, daß sie auf anderen Rechnern einwandfrei funktioniert. Versuchen Sie es ruhig einmal.

Nach diesem einleitenden Modus-Befehl folgen Befehle, die steuern, welche Teile des Programms einer Messung unterzogen werden sollen.

Der Profiler geht dabei von folgender Struktur aus: Ein EXE-Programm besteht aus mehreren OBJ-Dateien. Jede OBJ-Datei kann aus mehreren Quelldateien bestehen, jede Quelldatei aus mehreren Prozeduren oder Funktionen und jede

Prozedur oder Funktion aus mehreren Zeilen. (In VBDOS besteht allerdings jede OBJ-Datei, von Include-Dateien abgesehen, aus genau einer Quelldatei.)

Sie können nun mit den Befehlen PROGRAM, OBJECT, FILE, FUNCTION und LINE bestimmte Bereiche zur Messung vorsehen oder aus der Meßliste streichen. Hinter dem Befehl folgt der jeweilige Name (bei Zeilen eine Aufzählung mit Kommata oder Bereichsangabe mit Bindestrich; bei Funktionen und Prozeduren – für beide wird FUNCTION verwendet – muß auf die Groß-/Kleinschreibung geachtet werden), und dann kann ein „ADD“ oder „DELETE“ folgen. Alle Angaben sind immer relativ zum letztgenannten übergeordneten Objekt. Die folgenden Beispiele bringen Licht in das Dunkel (die kursiv gesetzten Bemerkungen dürfen nicht eingeschlossen werden):

```
PROGRAM TORUS.EXE ADD          Messung über das gesamte TORUS.EXE

PROGRAM TORUS.EXE DELETE       (dieser Befehl bewirkt nichts)
OBJECT TORDRAW ADD             Messung über alle Routinen in TORDRAW.OBJ
                               (=TORDRAW.BAS) aus TORDRAW.EXE

PROGRAM TORUS.EXE DELETE       (dieser Befehl bewirkt nichts)
OBJECT TORDRAW DELETE          (dieser Befehl bewirkt nichts)
FUNCTION Inside ADD            Messung über Funktion Inside in TORDRAW.OBJ
                               in TORDRAW.EXE

PROGRAM TORUS.EXE DELETE       (dieser Befehl bewirkt nichts)
OBJECT TORDRAW ADD             (Messung über TORDRAW.OBJ – wie oben)
FUNCTION Inside DELETE         Diesmal aber Funktion Inside weglassen
LINE 56,58,60-70               Zeilen 56, 58, 60-70 einschließen (sind ohnehin
                               durch OBJECT eingeschlossen, außer, wenn sie
                               innerhalb der ausgeschlossenen Prozedur stehen)
```

So können auch mehrere OBJ-Dateien hintereinander ein- oder ausgeschlossen werden. In den meisten Fällen werden Sie wahrscheinlich Daten über das gesamte Programm sammeln lassen oder – bei zeitintensiven Aktionen wie der Zeitmessung – nur eine ganz bestimmte Funktion im Auge haben. Beachten Sie, daß Zeilennummern immer von der ersten Zeile in der betr. Quelldatei ab gezählt werden, nicht vom Anfang der betr. Funktion oder Prozedur.

Nach der Aufzählung, welche Bereiche gemessen werden sollen, folgen noch die Angaben MAKEPBT gefolgt vom Dateinamen der zu erzeugenden PBT-Datei und MAKEPBI mit dem Namen der zu erzeugenden PBI-Datei. Die Befehle MERGEPBO und MERGEPBT (jeweils mit Dateinamen einer PBO- bzw. PBT-

Datei) können verwendet werden, um die Inhalte bestehender PBO- oder PBT-Dateien in die neu erzeugte PBT-Datei aufzunehmen.

Außerdem können Kommentarzeilen (beginnend mit #) in die Datei geschrieben werden, und INCLUDE-Befehle gefolgt vom Dateinamen einer anderen PCF-Datei fügen den Inhalt der genannten Datei an der Stelle ein, an der sie sich befinden.

16.3 Der Profiler für Bequeme

Da es, wie der Abschnitt 2 gezeigt hat, nicht immer leicht ist, den Profiler dahin zu bringen, wo man ihn haben will, liefert Microsoft das Programm VBDPROF.EXE mit, eine einfache Benutzeroberfläche für die drei Profiler-Programme.

Microsofts VBDPROF...

Diese ist nicht übersetzt worden und präsentiert sich dem neugierigen Programmierer so:

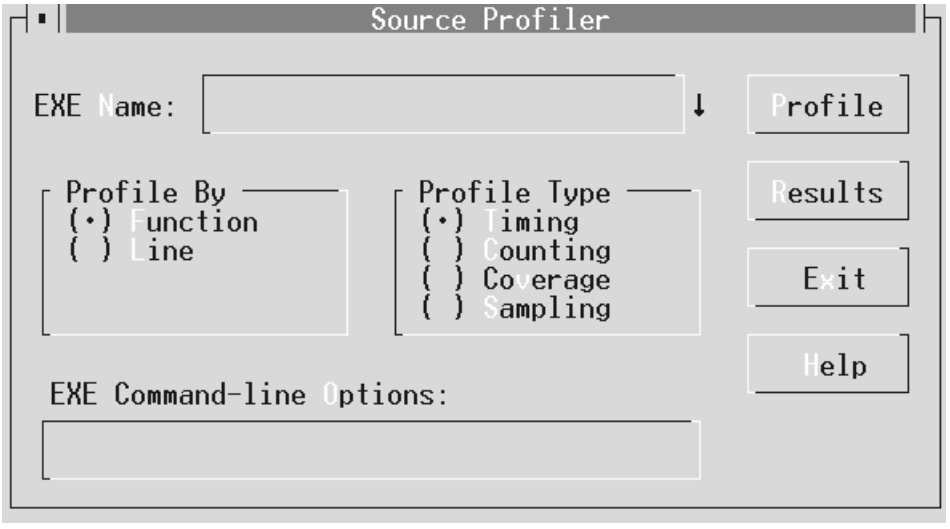


Abbildung 16–2: Die Oberfläche VBDPROF.EXE

Diese Applikation ist – unschwer zu erkennen – in VBDOS geschrieben. Sie können den Namen des EXE-Programms und evtl. Befehlszeilenoptionen angeben, auswählen, ob nach Funktionen/Prozeduren oder Zeilen gemessen wird und angeben, welche der vier Messungsarten sie wünschen (vgl. Beschreibung im Abschnitt über die PCF-Datei). Dann wählen Sie „Profile“, und VBDPROF erzeugt eine Batch-Datei namens PROF.BAT, die Sie (von Hand) starten müssen und die die benötigten Programme PREP, PROFILER, wieder PREP

und PLIST hintereinander aufruft. Wenn alles erledigt ist, startet die Prozedur wieder VBDPROF.EXE, und dieses zeigt Ihnen die Ergebnisse an.

...versus Ramms XPROF

Ich war mit VBDPROF nicht so ganz zufrieden, weil es mir nicht ermöglichte, die Sortierung der Ausgabedatei und das Ein- bzw. Ausschließen von einzelnen Objekten zu definieren. Daher habe ich kurzerhand eine eigene Version von VBDPROF kreiert und „XPROF“ getauft. XPROF ist als ausführbare Datei und auch im Quellcode auf der Diskette enthalten; einiges habe ich nicht ganz so sorgfältig dokumentiert, aber interessante Einblicke bietet der Quellcode vielleicht trotzdem – blättern Sie ihn einmal durch!

Die XPROF-Oberfläche ermöglicht zusätzlich zu dem, was auch VBDPROF kann, die Angabe der Stichprobenfrequenz (ein Switch bei PROFILE.EXE) und die Bestimmung der Sortierung für die Ausgabe (Switches bei PLIST.EXE). Experimentieren Sie mit diesen Einstellungen ruhig ein wenig. „Nur ausgeführte“ bedeutet, daß nur die Zeilen und Prozeduren ausgegeben werden sollen, die auch ausgeführt wurden.

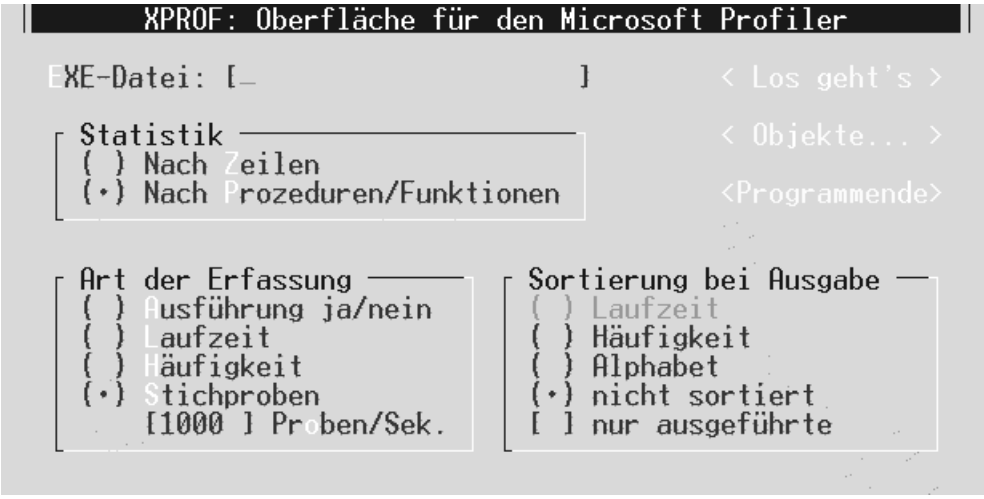


Abbildung 16–3: Die Hauptform von XPROF.EXE

Ein Klick auf den Knopf „Objekte“ öffnet ein Textfenster (in Wahrheit wird alles verborgen, was auf der Form ist, und ein vorher an gleicher Stelle positioniertes Textfeld wird angezeigt – eine brauchbare Methode, um die Formenanzahl gering zu halten). In diesem können Sie die Ein- bzw. Ausschlußangaben für die PCF-Datei verändern. Standardmäßig trägt XPROF dort drei Zeilen ein, die den Einschluß des gesamten angegebenen Programms verursachen.

XPROF speichert eine von Ihnen vorgenommene Einstellung inklusive der Ein- und Ausschlußangaben in einer Datei mit der Erweiterung .XPF. Immer, wenn Sie den Namen einer EXE-Datei angeben, prüft XPROF, ob ein entsprechendes XPF-File vorliegt und stellt alle Optionsfelder ggf. entsprechend ein.

Wenn Sie „Los geht's“ wählen, startet ein Profiler-Lauf. Obwohl XPROF auch eine Batchprozedur anlegt und aufruft (SHELL steht nicht zur Debatte, weil das zu messende Programm dann zu wenig Speicher zur Verfügung hat), müssen Sie nichts von Hand eingeben: XPROF manipuliert den Tastaturpuffer (vgl. Kapitel 22).

Nach Ablauf der Batchprozedur sollte XPROF wieder aufgerufen werden und erkennt an der Existenz von „$XPROF.BAT“, der von ihm erstellten temporären Batchprozedur, daß es jetzt an der Zeit ist, die Ergebnisse anzuzeigen. Diese werden von der Batchprozedur mittels PLIST in eine Datei mit der Erweiterung .XPO geschrieben.

XPROF zeigt dann die Ergebnisse des Profiler-Laufes an und ermöglicht auch ihren Ausdruck. Leider ist die Anzeige von mehr als 32 KB nicht möglich, so daß Sie bei umfangreichen Ergebnisdateien u. U. auf einen anderen Texteditor zurückgreifen müssen.

16.4 Profiler-Ausgabebeispiele

Das folgende Test-Programm erzeugt 200 Zufallszahlen, sortiert sie und zählt danach, wieviele Zahlen kleiner als 50 und wieviele größer als 25.000 sind:

```
CONST True = -1, False = 0
DIM Zahl(1 TO 200) AS INTEGER, SehrGross AS INTEGER, SehrKlein AS INTEGER

FOR i% = 1 TO 200:
   Zahl(i%) = RND * 30000
NEXT

Sortiere Zahl()

FOR i% = 1 TO 200
   IF Zahl(i%) < 50 THEN
      SehrKlein = SehrKlein + 1
   ELSEIF Zahl(i%) > 25000 THEN
      SehrGross = SehrGross + 1
   END IF
NEXT

PRINT "Sehr große Zahlen: "; SehrGross
PRINT "Sehr kleine Zahlen: "; SehrKlein
```

⇨

```
SUB Sortiere (Array() AS INTEGER)  ' einfaches Bubblesort - dauert ewig

   DIM Getauscht AS INTEGER
   DO
      Getauscht = False
      FOR i% = 2 TO UBOUND(Array)
         IF Array(i%) < Array(i% - 1) THEN
            SWAP Array(i%), Array(i% - 1): Getauscht = True
         END IF
      NEXT
   LOOP WHILE Getauscht

END SUB
```

Listing 16–1: PROFTEST.BAS

Listing nach Funktionen

Bei einer Zeitmessung nach Funktionen erzeugt der Profiler die folgende Ausgabedatei (Ich habe die Zeilen mit ' kommentiert):

```
Microsoft PLIST Version 1.20

Profile: Function timing, sorted by line.            ' In XPROF getroffene Auswahl
Date:    Thu Mar 25 22:08:42 1993                    ' war schon spät...

Program Statistics
------------------
    Total time: 121.214 milliseconds                 ' Gesamte Programmlaufzeit
                                                     ' (siehe Bemerkung unten)
    Time outside of functions: 43.452 milliseconds   ' Im Modulcode verbrachte Zeit
    Call depth: 1                                    ' max. Anzahl der verschach-
                                                     ' telten Prozedur-/Funktions-
                                                     ' aufrufe
    Total functions: 2                               ' Eigentlich haben wird nur
                                                     ' eine Prozedur; diese Anzeige
                                                     ' beruht auf einem Fehler
    Total hits: 1                                    ' von den angeblich zwei Pro-
                                                     ' zeduren wurde 1 aufgerufen
    Function coverage:  50.0%                        ' Das sind genau 50%

Module Statistics for g:\vbdpro\proftest.exe         ' Das eben war die Gesamt-
-------------------------------------------          ' statistik; es folgt eine
                                                     ' Modulstatistik
    Time in module: 77.762 milliseconds              ' Zeitdifferenz zu oben er-
                                                     ' klärt sich aus den hier
                                                     ' nicht berücksichtigten Ver-
                                                     ' waltungsroutinen von VBDOS
```

```
    Percent of time in module: 100.0%
    Functions in module: 2                          ' wie oben
    Hits in module: 1
    Module function coverage:  50.0% pfeil2

      Func       Func+Child        Hit                ' Func Time ist die Zeit, die
      Time   %      Time      %   count Function      ' in der Prozedur selbst ge-
-------------------------------------------------     ' braucht wurde; bei Func+
                                                      ' Child Time ist die Zeit
                                                      ' aller von hier aufgerufenen
                                                      ' Prozeduren addiert
                                                      ' Hit Count ist die Anzahl der
                                                      ' Aufrufe
    77.762 100.0     77.762 100.0     1 Sortiere (proftest.bas:23)
     0.000   0.0      0.000   0.0     0 b$sd (strdsp1.c:38)
```

Aus allen angegebenen Laufzeiten ist die Zeit, die der Profiler zum Aufzeichnen der Ergebnisse benötigte, so gut wie irgend möglich herausgerechnet worden. Dieses Programm lief tatsächlich etwa drei Sekunden; die Tatsache, daß hier eine Gesamtzeit von nur 121 ms angezeigt wird, bedeutet, daß die restlichen 2,88 Sekunden vom Profiler benötigt wurden. Allerdings irrt der Profiler insbesondere bei kurzen Zeiten oft – wie Sie im Vergleich zum nächsten Listing, das mit dem gleichen EXE-Programm erzeugt wurde, sehen.

Daß hier eine Routine „b$sd“ angezeigt wird, liegt an einem kleinen Fehler in der VBDOS-Library; die Entwickler haben versehentlich eine mit /zi kompilierte C-Routine eingebunden.

Listing nach Zeilen

Hier noch einmal derselbe Profiler-Lauf, diesmal allerdings mit Zeitmessung pro Zeile. Bei der Zeitmessung wird auch die Aufrufhäufigkeit mit ausgegeben. Wenn Sie „Häufigkeit“ wählen, wird nur diese Zahl angegeben, und wenn Sie den Modus „Ausführung ja/nein“ verwenden, sehen Sie nur ein Sternchen („wurde ausgeführt“) oder einen Punkt („nicht ausgeführt“) neben der Zeile.

```
Microsoft PLIST Version 1.20

Profile: Line timing, sorted by line.
Date:    Thu Mar 25 22:04:56 1993

Program Statistics
------------------
    Total time: 2288.393 milliseconds
    Time before any line: 14.661 milliseconds
```

```
    Total lines: 25                                    ' Insgesamt 25 Zeilen wurden
    Total hits: 84955                                  ' insgesamt 84955x ausgeführt
    Line coverage:  88.0%                              ' 88% der Zeilen ausgeführt

Module Statistics for g:\vbdpro\proftest.exe
-------------------------------------------
    Time in module: 2273.732 milliseconds
    Percent of time in module: 100.0%
    Lines in module: 25
    Hits in module: 84955
    Module line coverage:  88.0%

Source file: g:\vbdpro\proftest.bas

             Line        Hit
  Line       Time   %   count Source
-------------------------------------
    1:                        DECLARE SUB Sortiere (Array() AS INTEGER)
    2:                        CONST True = -1, False = 0
    3:                        DIM Zahl(1 TO 200) AS INTEGER
    4:                        DIM SehrGross AS INTEGER, SehrKlein AS INTEGER
    5:
    6:       0.040   0.0     1 FOR i% = 1 TO 200
    7:      11.486   0.5   200    Zahl(i%) = RND * 30000    ' Zeile braucht 11 ms
    8:       5.133   0.2   200 NEXT                         ' und wird 200x ausgef.
    9:
   10:       0.024   0.0     1 Sortiere Zahl()
   11:
   12:       0.023   0.0     1 FOR i% = 1 TO 200
   13:       4.790   0.2   200    IF Zahl(i%) < 50 THEN          ' Diese IF-Abfrage
   14:       0.023   0.0     1       SehrKlein = SehrKlein + 1   ' hat nur 1 Treffer
   15:       0.022   0.0     1    ELSEIF Zahl(i%) > 25000 THEN   ' diese aber 28
   16:       0.613   0.0    28       SehrGross = SehrGross + 1
   17:                            END IF
   18:       5.435   0.2   200 NEXT
   19:
   20:       4.278   0.2     1 PRINT "Sehr große Zahlen: "; SehrGross
   21:       0.152   0.0     1 PRINT "Sehr kleine Zahlen: "; SehrKlein
   22:
   23:       0.035   0.0     1 SUB Sortiere (Array() AS INTEGER)
   24:
   25:                           DIM Getauscht AS INTEGER ' Hier sehen Sie, was
   26:       0.019   0.0     1    DO                       ' was BubbleSort so lang-
   27:       4.030   0.2   184       Getauscht = False     ' langsam macht: 36616
                                                           ' Prüfungen für nur
                                                           ' 200 Zahlen!
   28:       6.025   0.3   184       FOR i% = 2 TO UBOUND(Array)
   29:     911.670  40.1 36616          IF Array(i%) < Array(i% - 1) THEN
   30:     334.190  14.7 10332            SWAP Array(i%),Array(i%-1):Getauscht=True
   31:                                  END IF
   32:     963.477  42.4 36616       NEXT
```

```
   33:       5.778   0.3   184    LOOP WHILE Getauscht
   34:
   35:       0.032   0.0     1 END SUB
   36:      16.457   0.7     1

Unable to open source file: ..\rt\strdsp1.c          ' der übliche Fehler
```

An diesem Beispiel wird auch deutlich, was „Optimieren aufgrund von Ausführungshäufigkeiten“ ist: In den Zeilen 13–16 wird die Abfrage „`IF Zahl(i%) < 50`“ 200mal ausgeführt, die Abfrage „`ELSEIF Zahl(i%) > 25000`“ weitere 199mal (der Profiler ist hier etwas mißverständlich in seiner Anzeige; es ist jedoch klar, daß die ELSEIF-Abfrage so oft ausgeführt wird wie die IF-Abfrage abzüglich aller vor ELSEIF stehenden, zutreffenden Bedingungen).

Wenn Sie nun beide Abfragen vertauschen und zuerst prüfen, ob die Zahl größer als 25.000 ist, würde die zweite Abfrage nur noch 172mal ausgeführt. Diese Verbesserung um insgesamt sieben Prozent macht hier natürlich nur Millisekunden aus und wirkt etwas kleinlich; an anderen Stellen kann eine solche Umstrukturierung aber durchaus viel Zeit sparen, insbesondere dann, wenn in der IF-Abfrage Funktionen aufgerufen werden.

Benutzerdefinierte Steuerelemente 17

Diesem Kapitel möchte ich eins vorwegschicken: Die Materie „benutzerdefinierte Steuerelemente“ ist ziemlich kompliziert. Ich verzichte deshalb an vielen Stellen auf lange Worte und bringe lieber ein paar Beispiele hintereinander, aus denen Sie hoffentlich recht schnell erkennen, worauf es ankommt.

17.1 Das Konzept

Beim Umgang mit den Standard-Steuerelementen hat man in VBDOS direkten Zugriff auf die Objekte: Man kann ihre Eigenschaften direkt abfragen und setzen, man kann Methoden auf sie anwenden und Ereignisprozeduren schreiben, die „vom Steuerelement selbst“ ausgelöst werden. Die folgende Tabelle verdeutlicht diese „direkte Interaktion“:

im Programm		*im Steuerelement*
`Knopf.Tag = "Hallo"`	→	*Tag*-Eigenschaft wird geändert
`Text$ = Knopf.Tag`	→	*Tag*-Eigenschaft wird ausgelesen
`Knopf.DRAG`	→	DRAG-Methode wird auf Steuerelement angewandt
Knopf_Click() wird ausgelöst	←	Benutzer klickt auf Steuerelement

Wenn Sie ein Steuerelement selbst programmieren, können Sie zwischen diese beiden Ebenen den von Ihnen programmierten Steuerelement-Code schieben. Das ist nicht für alle Ereignisse, Methoden und Eigenschaften zwingend; manche werden, wenn Sie keinen eigenen Code schreiben, von VBDOS einfach „wie üblich“ – zumeist wie bei einem Bildfeld – behandelt. Nehmen wir aber einmal an, Sie würden von allen Möglichkeiten, eigenen Code einzuschieben, Gebrauch machen. Dann müßte die Tabelle wie folgt erweitert werden (ich verwende ab jetzt den Terminus „Programm“ für das BASIC-Programm, in dem das benutzerdefinierte Steuerelement eingesetzt wird, und „Code des Steuerelements“ für das BASIC-Programm, mit dessen Hilfe das Steuerelement programmiert wird):

im Programm		*im Steuerelement-Code*		*im Steuerelement*
`Knopf.Tag = "Hallo"`	→	*StringSet*-Ereignis wird ausgelöst (↓) Aufruf der Routine SetStringProperty	→	*Tag*-Eigenschaft wird geändert

im Programm		*im Steuerelement-Code*		*im Steuerelement*
`Text$ = Knopf.Tag`	→	*StringGet*-Ereignis wird ausgelöst		
		(↓)		
		Aufruf der Routine GetStringProperty	→	*Tag*-Eigenschaft wird ausgelesen
`Knopf.DRAG`	→	*MthDrag*-Ereignis wird ausgelöst		
		(↓)		
		Aufruf der Routine InvokeMethod	→	DRAG-Methode wird auf Steuerelement angewandt
		CClick-Ereignis wird ausgelöst (zwei C!)	←	Benutzer klickt auf Steuerelement
		(↓)		
Knopf_Click() wird ausgelöst	←	Aufruf der Routine InvokeEvent		

Fast alles, was mit einem selbstdefinierten Steuerelement passieren kann – egal, ob von seiten des Programms aus, in dem es eingesetzt wird, oder von seiten des Benutzers – löst im Code des Steuerelements ein Ereignis aus. Auf dieses Ereignis kann der Code dann reagieren; so wird der Code in der *CClick*-Ereignisprozedur eines selbstdefinierten Steuerelements zum Beispiel aller Wahrscheinlichkeit nach durch einen Aufruf der Routine InvokeEvent ein *Click*-Ereignis im Programm auslösen und das Klicken des Benutzers so an das Programm „weiterreichen“. Das ist zwar wahrscheinlich, aber nicht notwendig: Genauso könnte ein selbstdefiniertes Steuerelement bei Mausklick ein Geräusch ertönen lassen und keine Ereignisprozedur im Programm aufrufen, oder es könnte im Programm anstatt eines *Click*-Ereignisses ein *Resize*-Ereignis (oder etwas beliebig anderes) auslösen. Deshalb sind die Abwärtspfeile (↓) in der Tabelle eingeklammert. Sie stellen den üblichen Ablauf dar, an den Sie sich aber nicht halten müssen, wenn Sie ein Steuerelement selbst definieren.

17.2 Das Dienstprogramm CUSTGEN

CUSTGEN.EXE wird nicht unbedingt benötigt, um Steuerelemente selbst zu programmieren; ich zeige hier jedoch nur die Methode mit CUSTGEN, weil sie einfacher ist.

CUSTGEN leistet für seine Größe von 260 KB nicht gerade viel: Es erzeugt lediglich nach Ihren Vorgaben eine Registrierungsdatei (in Assembler) und eine

Code-Schablone (in Assembler, C oder BASIC). Die Registrierungsdatei können Sie zwar modifizieren, aber in den meisten Fällen werden Sie sie unverändert lassen. Die Code-Schablone hingegen enthält lauter leere Prozeduren für die verschiedenen Ereignisse, und Sie müssen hier Ihr Steuerelement „hineinprogrammieren".

Im folgenden werde ich davon ausgehen, daß Sie BASIC-Codeschablonen erzeugen und Ihr Steuerelement mit VBDOS programmieren, obwohl das, wie erwähnt, auch mit Assembler oder C möglich wäre.

Was Sie mit CUSTGEN wählen

Am Anfang der Programmierung eines eigenen Steuerelements steht die Verwendung von CUSTGEN. Wenn Sie dieses Programm einsetzen, müssen Sie sich vorher überlegen,

- wie Ihr Steuerelement heißen soll,
- ob Ihr Steuerelement (wie ein Bildfeld oder ein Rahmen) in der Lage sein soll, andere Steuerelemente zu enthalten,
- für welche Ereignisse Sie im Code Ihres Steuerelements Code einfügen wollen,
- welche Ereignisse Ihr Steuerelement in einem Programm, in dem es eingesetzt wird, auslösen kann und
- welche Eigenschaften im Form-Designer (also zur Entwurfszeit) manipulierbar sein sollen, wenn eine Form mit Ihrem Steuerelement entworfen wird.

Welche Methoden Ihr Steuerelement unterstützt und wie es auf das Setzen und Abfragen von Eigenschaften reagiert, müssen Sie jetzt noch nicht festlegen. Dafür programmieren Sie später Ereignisprozeduren im Steuerelementcode.

CUSTGEN benutzen

Sie starten CUSTGEN einfach durch Eingabe dieses Namens, ohne weitere Parameter. Das Programm meldet sich mit seiner Haupt-Form (Abb. auf der nächsten Seite).

Sie geben zunächst den Namen des Steuerelements an und wählen dann unter Verwendung der Knöpfe ➔ und ➔ die Ereignisse aus, für die Sie im Code Ihres Steuerelements Ereignisprozeduren programmieren möchten (diese Ereignisse müssen unter „Schablonenereignisse" stehen). Dabei stehen auch zahlreiche Ereignisse zur Wahl, die Sie bisher von VBDOS nicht kennen. Ich werde sie später einzeln behandeln.

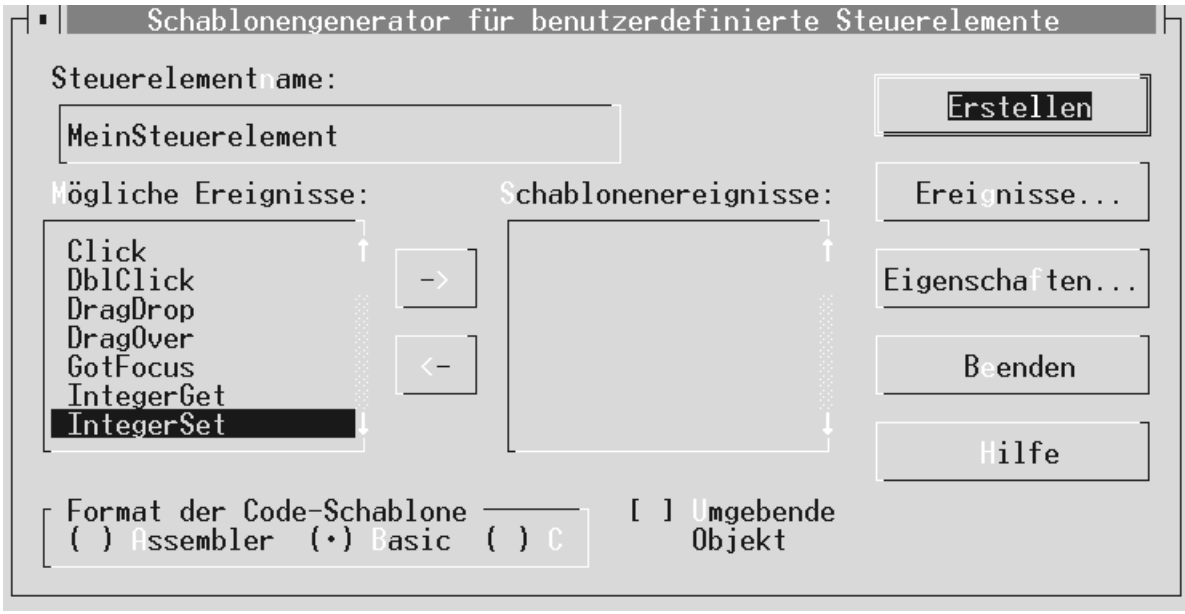


Abbildung 17–1: Die Hauptform von CUSTGEN

Danach können Sie durch Anwahl der „Ereignisse...“-Schaltfläche in die Form verzweigen, mit der Sie die Ereignisse wählen, die Ihr Steuerelement im Programm, in dem es verwendet wird, aufrufen kann:

Abbildung 17–2: Die Ereignisverfügbarkeitsform von CUSTGEN

Hier können Sie nur Ereignisse wählen, die aus VBDOS schon bekannt sind. Die Ereignisse, die Sie hier als „verfügbar“ markieren, erscheinen später in VBDOS im Dialogfeld „Ereignisprozeduren“, wenn Ihr Steuerelement ausgewählt wird.

Die Schaltfläche „Eigenschaften...“ auf der Hauptform verzweigt in eine ähnlich aussehende Eigenschaften-Auswahlform, mit der Sie festlegen können, welche Eigenschaften beim Entwurf im Form-Designer für Ihr Steuerelement einstellbar sein sollen.

Unabhängig von der Einstellung in CUSTGEN kann Ihr Steuerelement alle Eigenschaften, die es gibt, intern für sich verwenden. Auch kann später vom Programm aus auf alle Eigenschaften Ihres Steuerelements zugegriffen werden, wenn Sie das nicht explizit im Code Ihres Steuerelements mit einer Fehlermeldung beantworten. Die Eigenschaften-Einstellung in CUSTGEN hat also nur auf die Struktur Ihres Steuerelements innerhalb des Form-Designers Einfluß.

Nachdem Sie diese drei Einstellungen mit CUSTGEN vorgenommen haben, können Sie noch wählen, ob Ihr Steuerelement ein „Container"-Element sein soll, das wie ein Rahmen oder ein Bildfeld andere Steuerelemente enthalten kann.

Klicken Sie dann auf „Erstellen". CUSTGEN fragt Sie nach einem Namen für die Registrierungsdatei und die Codeschablone. Beide Dateien sollten nicht den gleichen „Vornamen" haben. CUSTGEN erzeugt dann beide Dateien. Die Codeschablone können Sie nun (wenn Sie eine BASIC-Schablone erstellt haben) mit VBDOS weiter bearbeiten, während Sie die Registrierungsdatei assemblieren müssen, um eine OBJ-Datei zu erhalten. (Wenn Sie selbst keinen Assembler besitzen, lohnt es sich wohl kaum, hierfür einen anzuschaffen, da Sie für jedes Steuerelement den Assembler nur ein einziges Mal benötigen. Sie finden bestimmt jemanden, der Ihnen – schlimmstenfalls gegen eine kleine Gebühr – die Registrierungsdatei assembliert.)

17.3 Besondere Ereignisse und Prozeduren

Alle Standardereignisse von VBDOS mit Ausnahme von *Load* stehen Ihrem Steuerelement zur Verfügung. Wenn Sie diese Ereignisse in CUSTGEN unter „Schablonenereignisse" eintragen, können Sie im Code Ihres Steuerelements eine Ereignisprozedur für das jeweilige Ereignis schreiben, die VBDOS dann aufruft, wenn es eintritt.

Die Ereignisprozeduren im Code des Steuerelements unterscheiden sich jedoch von denen der „gewöhnlichen" Ereignisprozeduren im Programm in wesentlichen Punkten:

- Die Ereignisprozeduren im Code des Steuerelements sind keine Prozeduren, sondern Funktionen mit einem INTEGER-Rückgabewert. Dieser Rückgabewert wird allerdings bei den Standard-Ereignissen nicht verwendet.
- Die Ereignisprozeduren im Code des Steuerelements beginnen mit einem „C" (z.B. `Schalter_CClick` statt `Schalter_Click`).
- Die Ereignisprozeduren im Code des Steuerelements haben zwei zusätzliche Parameter: `Ctrl AS CONTROL` und `ControlID AS INTEGER`. Das ist erforderlich, um verschiedene Instanzen desselben Steuerelements abzugrenzen. Sie pro-

grammieren im Code des Steuerelements nur eine einzige Ereignisprozedur für jedes Ereignis. Im Programm können später aber zum Beispiel fünf verschiedene Varianten Ihres Steuerelements auf einer Form stehen, genauso, wie man auch mehrere Schaltflächen oder mehrere Optionsfelder auf eine Form setzen kann. Daher muß jede Ereignisprozedur im Code des Steuerelements wissen, von welchem Steuerelement sie aufgerufen wurde.

Bis auf *Change*, *DropDown*, *PathChange*, *PatternChange* und *Resize* gilt dabei für alle Ereignisse: Wenn Sie keine Ereignisprozedur für das Ereignis im Code Ihres Steuerelements schreiben (und das Ereignis bei CUSTGEN nicht unter „Schablonenereignisse“ eingetragen haben), wird bei Eintreten des Ereignisses direkt die Ereignisprozedur im Programm aufgerufen. Auf diese Weise haben Sie, wie ich schon anfangs erläuterte, die Wahl, ob Sie ein bestimmtes Ereignis „abfangen“ möchten oder nicht.

In jedem Fall haben Sie die Möglichkeit, im Code des Steuerelements durch den Aufruf einer InvokeEvent-Prozedur eine Ereignisprozedur im Programm aufzurufen (mehr dazu später).

Custom-Ereignis

Neben den Standardereignissen gibt es noch ein Ereignis namens *Custom*, das einfach eine INTEGER-Zahl als Argument hat und das Sie im Programm aufrufen können. Dadurch können Sie ohne unübersichtliche Tricks spezielle Ereignistypen für eigene Steuerelemente definieren. Ich verwende dieses Ereignis im Beispiel „PrintBuffer“ am Ende dieses Kapitels.

Die Prozedur InvokeEvent

Um eines der bisher besprochenen Ereignisse in dem Programm, in dem Ihr Steuerelement eingesetzt wird, auszulösen, verwenden Sie die InvokeEvent-Prozedur. Es gibt für jedes Ereignis eine eigene Prozedur; neben den Argumenten, die diesen Ereignisprozeduren gewöhnlich übergeben werden, folgen noch die INTEGER-Werte CID (Identifikationsnummer des Steuerelements) und EID (Nummer des Ereignisses, Konstanten in CUSTINCL.BI). Um zum Beispiel das Ereignis *Change* für Ihr Steuerelement „Schalter“ auszulösen, wenn der Benutzer darauf klickt, würden Sie programmieren:

```
FUNCTION Schalter_CClick (Ctrl AS CONTROL, CID AS INTEGER) AS INTEGER
   InvokeChangeEvent CID, EVENT_Change  ' Konstante aus CUSTINCL.BI
END FUNCTION
```

Es erscheint unsinnig, daß hier gleich zweimal angegeben werden muß, daß es sich um das Change-Ereignis handelt – den Grund dafür lesen Sie unter „Doppelt gemoppelt“ weiter unten in diesem Kapitel.

Die InvokeEvent-Prozedur löst keinen Fehler aus, wenn eine Ereignisprozedur für das hierdurch ausgelöste Ereignis nicht im Programm enthalten ist – genauso, wie VBDOS ja auch keinen Fehler erzeugt, wenn Sie auf eine Schaltfläche klicken und keine *Click*-Ereignisprozedur vorhanden ist.

Schließlich gibt es noch eine Reihe von Spezialereignissen, die zwar im Code Ihres Steuerelements auftreten können, die aber im späteren Programm nicht verfügbar sind:

Die Get- und Set-Ereignisse

Von den Ereignissen *IntegerGet* und *IntegerSet*, *LongGet* und *LongSet*, *StringGet* und *StringSet* wird eines immer dann aufgerufen werden, wenn das Programm den Wert einer der Eigenschaften Ihres Steuerelements liest (Get) oder verändert (Set). Je nach Art der Eigenschaft wird das Integer-, Long- oder String-Ereignis ausgelöst, und an einem übergebenen Parameter können Sie erkennen, um welche Eigenschaft es sich handelt und welcher Wert ihr zugewiesen wird. Übergeben werden diesen Ereignisprozeduren folgende Parameter:

Parameter	*Datentyp*	*Bedeutung*
Ctrl	CONTROL	Das Steuerelement, von dessen Eigenschaften eine geändert oder gelesen werden soll
CID	INTEGER	Die „Control ID“ (Identifikationsnummer des Steuerelements); sie wird für eventuelle GetProperty- oder SetProperty-Aufrufe gebraucht
PID	INTEGER	Die „Property ID“ der betroffenen Eigenschaft (Identifikationsnummer; Konstanten sind in CUSTINCL.BI definiert)
Value	INTEGER, LONG oder STRING (je nach Ereignis)	Die Variable, die in die Eigenschaft geschrieben oder aus ihr gelesen werden sollte

Wenn Sie für diese Ereignisse keine Ereignisprozeduren schreiben (und sie nicht unter „Schablonenereignisse“ in CUSTGEN eintragen), kann das Programm später beliebig alle Eigenschaften Ihres Steuerelements manipulieren, ohne daß das Steuerelement darauf reagieren könnte.

Wenn Sie zum Beispiel erreichen wollen, daß Ihr selbstdefiniertes Steuerelement namens „Schalter“ sofort neu gezeichnet wird, wenn der Benutzer die

Caption-Eigenschaft ändert (das ist ja bei fast allen VBDOS-Steuerelementen der Fall), müßten Sie etwa so programmieren:

```
FUNCTION Schalter_CStringSet (Ctrl AS CONTROL, BYVAL CID AS INTEGER,
                              BYVAL PID AS INTEGER, Value AS STRING) AS INTEGER
   SetStringProperty Value, CID, PID
   IF PID = PROP_Caption THEN  ' Konstante PROP_Caption ist in CUSTINCL.BI definiert
      Schalter_CStringSet = Schalter_CPaint (Ctrl, CID) ' Aufruf der PAINT-Ereignis-
   END IF                                               ' Prozedur, wenn Caption ge-
                                                        ' ändert
END FUNCTION
```

Diese Funktion ruft, bevor Sie irgendetwas anderes tut, SetStringProperty auf. SetStringProperty ist eine Routine, mit der man eine String-Eigenschaft eines Steuerelements setzen kann (ebenso gibt es auch SetIntProperty und SetLongProperty). Dieser Aufruf ist nötig, weil VBDOS in dem Augenblick, in dem Sie selbst eine Ereignisprozedur für das StringSet-Ereignis schreiben, nicht mehr selbst die String-Eigenschaften des Steuerelements verändert, sondern das völlig Ihrer Routine überläßt.

Danach prüft die Routine, ob die geänderte Eigenschaft die *Caption*-Eigenschaft war. (Dieselbe Routine wird ja auch beim Ändern der *Text*- oder *Tag*-Eigenschaft aufgerufen.) Falls ja, wird nun die Paint-Ereignisprozedur im Code des Steuerelements aufgerufen, die Sie natürlich unabhängig davon selbst schreiben müssen, damit Ihr Steuerelement auch so angezeigt wird, wie Sie sich das vorstellen.

Die Tatsache, daß VBDOS auf die Eigenschaften nicht mehr selbst zugreift, sobald Sie dafür Routinen schreiben, können Sie nutzen, um Eigenschaften zu sperren. Nehmen wir an, Ihr selbstdefiniertes Steuerelement erlaubt keine STRING-Eigenschaften außer *Tag* (*Tag* sollten Sie nie sperren; in Kapitel 7 haben Sie gesehen, daß man *Tag* für vielerlei nützliche Zwecke „mißbrauchen“ kann). Sie möchten also erreichen, daß der Fehler 422 (*Eigenschaft nicht gefunden*) erzeugt wird, wenn das Programm z. B. auf die Text- oder Caption-Eigenschaft des Steuerelements zugreift. Dann programmieren Sie:

```
FUNCTION Schalter_CStringSet (Ctrl AS CONTROL, BYVAL CID AS INTEGER,
                              BYVAL PID AS INTEGER, Value AS STRING) AS INTEGER
   IF PID <> PROP_Tag THEN     ' PROP_Tag ist in CUSTINCL.BI definiert
      Schalter_CStringSet = 422
   ELSE
      SetStringProperty Value, CID, PID
   END IF
END FUNCTION
```

Hier wird die Möglichkeit genutzt, durch Rückgabe von Werten als Funktionswert der speziellen Ereignisprozeduren (die ja eigentlich Funktionen sind), einen Fehler auszulösen. Wenn später in einem Programm, das Ihr Steuerelement „Schalter“ benutzt, etwa der Befehl `Schalter1.Text = "Hallo"` auftaucht, ergibt das einen Fehler 422, genau so, als würde man z. B. versuchen, die *Text*-Eigenschaft einer Schaltfläche zu verwenden.

Was hier für die Set-Routinen erklärt wurde, gilt analog für die drei Get-Funktionen. Auch hier tut VBDOS von dem Augenblick an, in dem Sie eine IntegerGet-, LongGet- oder StringGet-Routine schreiben, nichts mehr, um den Wert einer Eigenschaft zu ermitteln, und Sie müssen selbst eine GetInteger-, GetLong- oder GetStringProperty-Routine einsetzen, um einen Wert zurückzugeben.

Die Methodenereignisse

Die Ereignisse *MthAddItem*, *MthCls*, *MthDrag*, *MthHide*, *MthMove*, *MthPrint*, *MthRefresh*, *MthRemoveItem*, *MthSetFocus* und *MthShow* treten ein, wenn aus dem Programm heraus eine der betreffenden Methoden auf Ihr Steuerelement angewandt wird. Auch hier können Sie eine Fehlernummer zurückgeben, die dann einen entsprechenden Fehler im Programm verursacht. Geben Sie 421 zurück, um den Fehler *Methode ist auf dieses Objekt nicht anwendbar* zu erzeugen. Ansonsten programmieren Sie das, was die Methode bewirken soll, in dieser Ereignisprozedur. Die Argumente der Methode werden der Ereignisprozedur als Parameter übergeben (vgl. DECLARE-Zeilen in der von CUSTGEN erzeugten Codeschablone).

Einzige Ausnahme ist die PRINT-Methode. Da hier beliebig viele Argumente übergeben werden können, ist es Ihnen nicht möglich, eine eigene Ereignisprozedur für die PRINT-Methode zu programmieren, die die übergebenen Argumente verwendet. Stattdessen stehen Sie, wenn Sie eine Ereignisprozedur *MthPrint* programmieren, vor der Wahl, entweder den Funktionswert 0 zurückzugeben (dann führt VBDOS die PRINT-Methode auf Ihr Objekt so aus wie auf ein Bildfeld) oder aber einen Fehlercode als Funktionswert zuzuweisen – dann führt VBDOS die PRINT-Methode nicht aus.

Wenn Sie für die Methodenereignisse keine Ereignisprozeduren schreiben, dann können die Methoden mit Ausnahme von ADDITEM, HIDE, REMOVEITEM und SHOW auf das selbstdefinierte Steuerelement wie auf ein Bildfeld angewendet werden.

Die Ereignisse Load und Unload

Diese Ereignisse stehen bei normalen Steuerelementen nicht zur Verfügung. Das *Load*-Ereignis tritt ein, wenn das Steuerelement zum ersten Mal in den Speicher geladen wird (mit dem LOAD-Befehl als Element eines Arrays oder einfach, weil die Form, auf der es steht, geladen wird). Verwenden Sie das *Load*-Ereignis, um z.B. Daten zu initialisieren; Code, der das Steuerelement anzeigt, gehört nicht hierhin, sondern ins *Paint*-Ereignis.

Das *Unload*-Ereignis tritt ein, wenn das Steuerelement aus dem Speicher gelöscht wird.

Wenn Sie beim *Load*-Ereignis einen Funktionswert zurückgeben, wird im Programm ein Fehler erzeugt, und der Load-Prozeß wird abgebrochen. Das Zurückgeben eines Fehlers beim *Unload*-Ereignis kann jedoch das Entfernen des Steuerelements aus dem Speicher nicht verhindern.

Last but not least: InvokeMethod

Sie haben jetzt alle speziellen Ereignisse, die Routinen Set- und GetProperty sowie InvokeEvent kennengelernt, und damit haben wir schon fast alles zusammen, was wir für ein erstes, eigenes Steuerelement benötigen – bis auf eines: Die InvokeMethod-Prozedur.

Sie wird verwendet, um eine Methode auf ein Steuerelement anzuwenden. Dabei müssen der Prozedur zunächst die Argumente übergeben werden, die die Methode naturgemäß erwartet, dann die Anzahl dieser Argumente, danach ein INTEGER-Code für das betroffene Steuerelement (CID) und ein INTEGER-Code für die Methode (MID, Konstanten stehen in CUSTINCL.BI).

Um zum Beispiel zu erreichen, daß ein selbstdefiniertes Steuerelement immer dann seinen Inhalt löscht, wenn der Benutzer darauf klickt, würden Sie die Click-Ereignisprozedur im Code des Steuerelements etwa so formulieren:

```
FUNCTION Schalter_CClick (Ctrl AS CONTROL, CID AS INTEGER) AS INTEGER
   InvokeCLSMethod 0, CID, METHOD_CLS ' Konstante aus CUSTINCL.BI
END FUNCTION
```

Der Wert 0 wird benötigt, um anzuzeigen, daß keine zusätzlichen Argumente übergeben werden (bei CLS ja gar nicht möglich).

Eine Ausnahme macht die PRINT-Methode, die wie folgt aufgerufen wird:

```
InvokePrintMethod x, y, fcol, bcol, text$, 5, CID, METHOD_PRINT
```

x und *y* sind dabei Spalte und Zeile, *fcol* und *bcol* Vorder- und Hintergrundfarbe, *text$* der auszugebende Text, CID die ControlID-Zahl des betroffenen Steuerelements und METHOD_PRINT eine Konstante aus CUSTINCL.BI.

Weitere Informationen über die etwas ungewöhnliche Aufrufkonvention von InvokeMethod finden Sie unter „Doppelt gemoppelt“ weiter unten in diesem Kapitel.

17.4 Eine erste Anwendung: „Schalter“

Als erstes Beispiel-Steuerelement habe ich einen Wahlschalter ausgesucht, der die Auswahl zwischen einer Anzahl von verschiedenen Einstellungen ermöglichen soll und anstelle von mehreren Optionsfeldern eingesetzt werden kann:

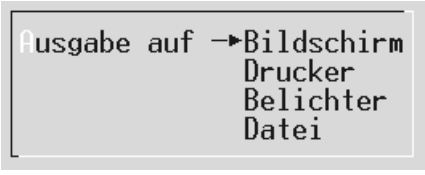


Abbildung 17–3: Das geplante Steuerelement „Schalter“

An dieses Steuerelement stelle ich folgende Anforderungen:

- Ein Klick auf den Schalter oder das Betätigen einer Taste soll die Auswahl um eins weiterstellen. Dabei soll ein *Change*-Ereignis ausgelöst werden.
- Über die Eigenschaft *Value* soll die aktuelle Einstellung abgefragt und auch gesetzt werden können. Beim Verändern der Eigenschaft soll ebenfalls ein *Change*-Ereignis ausgelöst werden.
- Die Eigenschaft *Caption* bestimmt, was links vom Pfeil steht, im Bild also „&Ausgabe auf“. Dieser Text wird jeweils mitbewegt, wenn sich die Einstellung ändert. Ein hervorgehobenes Zeichen als Zugriffstaste soll möglich sein.
- Die Eigenschaft *Text* bestimmt die Wahlmöglichkeiten, und zwar mit Schrägstrichen getrennt. Für ein Ergebnis wie in der Abbildung müßte also *Text* auf „Bildschirm/Drucker/Belichter/Datei“ gesetzt werden.

Die Basis mit CUSTGEN erzeugen

Den Grundstein für diesen „Schalter“ legen wir mit CUSTGEN. Rufen Sie das Programm auf, und wählen Sie folgende Ereignisse als Schablonenereignisse aus: *IntegerGet*, *IntegerSet*, *LongGet*, *LongSet*, *StringGet*, *StringSet*; *Click*, *KeyPress* und *Paint*. Welche Ereignisse und Eigenschaften Sie in den Dialogboxen als „zur Entwurfszeit verfügbar“ wählen, spielt zunächst einmal keine Rolle. Nennen Sie das Steuerelement „Schalter“.

CUSTGEN erzeugt für Sie die Registrierungsdatei (nennen Sie sie SCHALTRG.ASM) und eine Codeschablone (SCHALTER.BAS), die (bis auf Kommentarzeilen) leere Funktionen für die ausgewählten Ereignisse enthält. Im folgenden zeige ich, was Sie nun in die Funktionen eintragen müssen. Das komplette Listing SCHALTER.BAS finden Sie auf der Diskette.

LongGet und LongSet

Diese beiden Ereignisfunktionen sind die einfachsten. Da unser Schalter keine Eigenschaft unterstützt, die den Datentyp LONG hat (eine Übersicht über die Eigenschaften gibt es weiter unten im Kapitel), können wir bei jedem Zugriff auf eine LONG-Eigenschaft ausnahmslos den Fehler 422 (*Eigenschaft nicht gefunden*) zurückliefern.

```
FUNCTION Schalter_CLongGet (Ctrl AS CONTROL, BYVAL ControlID AS INTEGER, BYVAL
                            PropertyID AS INTEGER, Value AS LONG) AS INTEGER
   Schalter_CLongGet = 422
END FUNCTION
```

```
FUNCTION Schalter_CLongSet (Ctrl AS CONTROL, BYVAL ControlID AS INTEGER, BYVAL
                            PropertyID AS INTEGER, BYVAL Value AS LONG) AS INTEGER
   Schalter_CLongSet = 422
END FUNCTION
```

Natürlich wäre auch der völlige Verzicht auf diese Prozeduren denkbar. Dann können die LONG-Eigenschaften des Schalter-Steuerelements beliebig verändert werden, was auch nicht weiter tragisch ist, da wir sie nicht verwenden. Der Vollständigkeit halber sollte man aber Funktionen wie diese einbauen, damit in Programmen, die später den Schalter verwenden, nicht versehentlich eine der ununterstützten Eigenschaften verwendet wird.

IntegerGet und IntegerSet

Nicht ganz so einfach ist es mit IntegerGet und IntegerSet. Der Lesezugriff auf eine Anzahl von Eigenschaften des Schalters soll ungehindert möglich sein; im einzelnen sind dies *Value, ForeColor, BackColor, Left, Height, Top* und *Width*. Es gibt keinen Grund, einem Programm den Zugriff hierauf zu versagen.

Beim Schreibzugriff verhält es sich ebenso, mit einer Ausnahme: Wenn die Eigenschaft *Value* geändert wird, dann muß sich natürlich sofort das Aussehen unseres Schalters ändern (er wird weitergestellt). Außerdem soll, das hatten wir zuvor festgelegt, in diesem Fall auch in dem Programm, das den Schalter verwendet, ein *Change*-Ereignis ausgelöst werden.

```
FUNCTION Schalter_CIntegerGet (Ctrl AS CONTROL, BYVAL ControlID AS INTEGER,
                BYVAL PropertyID AS INTEGER, Value AS INTEGER) AS INTEGER
   SELECT CASE PropertyID
   CASE PROP_Value, PROP_ForeColor, PROP_BackColor, PROP_Left,
        PROP_Height, PROP_Top, PROP_Width
      GetIntProperty Value, ControlID, PropertyID
   CASE ELSE
       Schalter_CIntegerGet = 422
   END SELECT
END FUNCTION
```

```
FUNCTION Schalter_CIntegerSet (Ctrl AS CONTROL, BYVAL ControlID AS INTEGER,
          BYVAL  PropertyID AS INTEGER, BYVAL Value AS INTEGER) AS INTEGER
   SELECT CASE PropertyID
   CASE PROP_Value
      ' alten Wert auslesen und prüfen, ob sich überhaupt etwas ändert
      IF Value <> Ctrl.Value THEN
         ' neuen Wert setzen. neuen Status anzeigen und Change-Ereignis auslösen
         ' ein Wert größer als die Anzahl der Auswahlen wird durch MOD verhindert
         SetIntProperty Value MOD Ctrl.Height, ControlID, PropertyID
         SchalterStatusAnzeigen Ctrl, ControlID
         InvokeChangeEvent ControlID, EVENT_Change
      END IF
   CASE PROP_ForeColor, PROP_BackColor, PROP_Left, PROP_Height, PROP_Top, PROP_Width
      ' Eigenschaft einfach setzen
      SetIntProperty Value, ControlID, PropertyID
   CASE ELSE
       Schalter_CIntegerSet = 422
   END SELECT

END FUNCTION
```

In den SELECT CASE-Anweisungen werden die in CUSTINCL.BI definierten Konstanten (PROP_x) verwendet, um zwischen verschiedenen Eigenschaften zu unterscheiden.

In der Prozedur Schalter_CIntegerSet sehen Sie, daß ich auf die Eigenschaften *Height* und *Value* des Schalters einfach, wie das in VBDOS üblich ist, mit `Ctrl.Height` und `Ctrl.Value` zugreife, anstatt die Routine GetIntProperty aufzurufen. Der Unterschied zwischen

```
x = Ctrl.Height
```

und dem aufwendigeren

```
GetIntProperty x, ControlID, PROP_Height
```

liegt darin, daß die erste Variante den Umweg über das IntegerGet-Ereignis nimmt, die zweite dagegen nicht. Im Falle Height sind hier beide gleichwertig. Wollte ich aber zum Beispiel auf die INTEGER-Eigenschaft *Style* zugreifen (alle existierenden Eigenschaften dürfen ja für jedes Steuerelement verwendet

werden), so würde ein Aufruf `Ctrl.Style` von der oben programmierten IntegerGet-Prozedur mit einem Fehler quittiert, weil sie den Zugriff auf diese Eigenschaft nicht erlaubt. Die Verwendung von GetIntProperty unterliegt hingegen keinen Beschränkungen.

Aus dem gleichen Grunde ist das Lesen von *Value* mit `Ctrl.Value` in der IntegerSet-Prozedur durchaus möglich (in der Zeile `IF Value <> Ctrl.Value`), während ich zum Verändern der *Value*-Eigenschaft die SetIntProperty-Routine aufrufen muß, da bei einem Befehl wie `Ctrl.Value = x` sofort wieder das IntegerSet-Ereignis aufgerufen würde und eine endlose Rekursion einträte.

Auf die Prozedur „SchalterStatusAnzeigen“, die hier aufgerufen wird, um das Erscheinungsbild des Schalters der veränderten *Value*-Eigenschaft anzupassen, komme ich später zurück.

StringGet und StringSet

Diese beiden Routinen werden nach dem gleichen Muster wie schon die Integer-Versionen gestrickt:

```
FUNCTION Schalter_CStringGet (Ctrl AS CONTROL, BYVAL ControlID AS INTEGER, BYVAL ↪
         PropertyID AS INTEGER, Value AS STRING, BYVAL Index AS INTEGER) AS INTEGER

   SELECT CASE PropertyID
   CASE PROP_Tag, PROP_Caption, PROP_Text
      GetStringProperty Value, ControlID, PropertyID
   CASE ELSE: Schalter_CStringGet = 422
   END SELECT

END FUNCTION
```

```
FUNCTION Schalter_CStringSet (Ctrl AS CONTROL, BYVAL ControlID AS INTEGER, BYVAL ↪
         PropertyID AS INTEGER, Value AS STRING, BYVAL Index AS INTEGER) AS INTEGER

   SELECT CASE PropertyID
   CASE PROP_Tag
      SetStringProperty Value, ControlID, PropertyID
   CASE PROP_Caption, PROP_Text
      SetStringProperty Value, ControlID, PropertyID
      ' Diese Zuweisung hat nicht viel Sinn, das mache ich nur, weil Schalter_CPaint
      ' eine Funktion ist und ich keine Dummy-Variable verwenden will!
      Schalter_CStringSet = Schalter_CPaint(Ctrl, ControlID)
      SetIntProperty 0, ControlID, PROP_Value
   CASE ELSE
      Schalter_CStringSet = 422
   END SELECT

END FUNCTION
```

Beachten Sie, daß diese Funktionen ein Argument mehr als die Long- und Integer-Routinen haben: `Index AS INTEGER`. Dieses Argument wird benötigt, wenn das Programm, in dem das Steuerelement eingesetzt wird, auf die *List*-Eigenschaft zugreift, die ja wie ein Array gehandhabt wird.

Unsere StringGet-Routine erlaubt den Zugriff auf die Eigenschaften *Tag*, *Caption* und *Text* und erzeugt bei allen anderen einen Fehler.

In StringSet kann eine Änderung an der *Tag*-Eigenschaft ungehindert passieren (das sollten Sie immer so programmieren, damit das Programm, in dem Ihr Steuerelement verwendet wird, frei über die *Tag*-Eigenschaft verfügen kann). Änderungen an der *Text*- oder der *Caption*-Eigenschaft werden zwar auch durchgeführt, allerdings wird hier zusätzlich die *Value*-Eigenschaft auf 0 gesetzt und das Steuerelement durch einen Aufruf des *Paint*-Ereignisses neu gezeichnet. Das Rücksetzen von *Value* ist erforderlich, weil sich durch Änderung der *Text*-Eigenschaft ja die Anzahl der Wahlmöglichkeiten im Schalter verringern könnte, und *Value* dann womöglich einen Wert hätte, der größer als diese Anzahl ist.

Verwechseln Sie diesen Aufruf an die Ereignisfunktion Schalter_CPaint nicht mit dem Aufruf von Ereignissen durch InvokeEvent. Hier ging es darum, die im Code des Steuerelements enthaltene Paint-Ereignisprozedur aufzurufen, die wir später noch definieren. Ein Befehl wie `InvokePaintEvent ControlID, EVENT_PAINT` hätte stattdessen die Paint-Ereignisprozedur im Programm, das das Steuerelement später verwendet, aufgerufen – das soll hier nicht geschehen.

Click und KeyPress

Diese beiden Ereignisse können wir nun sehr leicht abhandeln. Der Schalter sollte, wenn der Benutzer darauf klickt oder eine Taste drückt, während er den Fokus hat, einfach weitergestellt werden. Dazu bedarf es nur weniger Zeilen:

```
FUNCTION Schalter_CClick (Ctrl AS CONTROL, BYVAL ControlID AS INTEGER) AS INTEGER
   Ctrl.Value = Ctrl.Value + 1
END FUNCTION
```

```
FUNCTION Schalter_CKeyPress (Ctrl AS CONTROL, BYVAL ControlID AS INTEGER,
         KeyAscii AS INTEGER) AS INTEGER
   Ctrl.Value = Ctrl.Value + 1
END FUNCTION
```

Hier wird durch einen Zugriff auf `Ctrl.Value` der Wert um eins erhöht. Dadurch ruft VBDOS intern zunächst IntegerGet auf, das wiederum GetIntProperty verwendet, um den gewünschten Wert zu liefern. Der so erhaltene Wert wird hier um eins erhöht und wieder in `Ctrl.Value` zurückgeschrieben. Dabei erfolgt

intern ein Aufruf von IntegerSet, und IntegerSet erledigt – wie oben programmiert – alles, was bei einer Wertänderung des Schalters notwendig ist. Auch das Auslösen des Change-Ereignisses im Programm, das den Schalter enthält, ist Sache der IntegerSet-Prozedur. Wie müssen noch nicht einmal prüfen, ob durch die Addition von 1 der Wert zu hoch wird, da IntegerSet hier den MOD-Operator anwendet.

Das Finale: Paint und SchalterStatusAnzeigen

Die beiden verzwicktesten Prozeduren habe ich bis zum Schluß aufgehoben: Das Paint-Ereignis, das von VBDOS automatisch aufgerufen wird, wenn es einen Schalter auf den Bildschirm zeichnen muß, und SchalterStatusAnzeigen, die Prozedur, die ich geplant habe, um eine Wertänderung des Schalters auf den Bildschirm zu bringen.

Die Paint-Routine soll den Schalter neu zeichnen. Dazu wird die *Text*-Eigenschaft nach Schrägstrichen durchsucht. Um aber zu wissen, an welche Stelle innerhalb des Schalters die so gefundenen Wahlmöglichkeiten geschrieben werden, muß erst die Länge des Textes links vom Pfeil ermittelt werden, der in der *Caption*-Eigenschaft steht.

Das ist so einfach mit LEN nicht zu machen, da es möglich sein soll, das &-Zeichen zur Bestimmung der Zugriffstaste in Caption zu verwenden. Bei einem Caption-Text von „L&aut && Leise“ würde ja z.B. nur „Laut & Leise“ angezeigt. Um auch der SchalterStatusAnzeigen-Routine lästige Prüfungen zu ersparen, ist es sinnvoll, eine der nicht benutzten String-Eigenschaften des Schalters – hier habe ich *SelText* gewählt – zu verwenden und darin den „echten“ *Caption*-Text zu speichern. Außerdem wird in die ebenfalls sonst ungenutzte Eigenschaft *Style* eingetragen, an welche Stelle im Text ein Zeichen hervorgehoben werden muß. Da wir den Zugriff auf *SelText* und *Style* in den entsprechenden Get- und Set-Ereignissen (weiter oben) ja verhindert haben, kann das Schalter-Steuerelement intern mit den Eigenschaften tun, was es will, ohne Sorge haben zu müssen, daß das Programm später dazwischenfunkt.

Nach der Einstellung dieser Eigenschaften sorgt die Paint-Prozedur durch einen Aufruf an SetAttribute (wird später noch behandelt) dafür, daß das Betätigen der mit & markierten Zugriffstaste auch wirklich bewirkt, daß der Schalter den Fokus bekommt. Da sich die Zugriffstaste durch eine Änderung der *Caption*-Eigenschaft ebenfalls ändern kann, ist es durchaus sinnvoll, diesen Aufruf hier unterzubringen.

Nachdem die Wahlmöglichkeiten unter Verwendung der PRINT-Methode angezeigt wurden, wird noch durch SetAttribute die Cursorform eingestellt, und SchalterStatusAnzeigen kümmert sich dann um den Rest.

```
FUNCTION Schalter_CPaint (Ctrl AS CONTROL, BYVAL ControlID AS INTEGER) AS INTEGER

   DIM StartSpalte AS INTEGER, AktuellPos AS INTEGER, SchraegStrich AS INTEGER
   DIM Text AS STRING, Zaehler AS INTEGER, i AS INTEGER, TastenPos AS INTEGER

   ' Erstmal alles löschen:
   InvokeCLSMethod 0, ControlID, METHOD_Cls
   IF LEN(Ctrl.Text) = 0 THEN EXIT FUNCTION

   ' SelText-Eigenschaft einstellen, die von SchalterStatusAnzeigen verwendet wird
   Text = Ctrl.Caption + " → "
   FOR i = 1 TO LEN(Text) - 3
      IF MID$(Text, i, 1) = "&" THEN
         Text = LEFT$(Text, i - 1) + MID$(Text, i + 1)
         IF MID$(Text, i, 1) <> "&" THEN TastenPos = i: EXIT FOR
      END IF
   NEXT
   SetStringProperty Text, ControlID, PROP_SelText: SetIntProperty TastenPos,
                     ControlID, PROP_Style
   ' Mit SetAttribute festlegen, daß die durch & markierte Taste die Zugriffstaste
   ' sein soll
   IF TastenPos THEN SetAttribute ControlID, ATTR_AccessKey, ASC(MID$(Text,
                                 TastenPos))

   ' Text-Eigenschaft nach Querstrichen trennen und Text mit PRINT-Methode ausgeben
   StartSpalte = LEN(Text) + 1: AktuellPos = 0
   DO
      SchraegStrich = INSTR(AktuellPos + 1, Ctrl.Text, "/")
      IF SchraegStrich = 0 THEN SchraegStrich = LEN(Ctrl.Text) + 1
      Text = MID$(Ctrl.Text, AktuellPos + 1, SchraegStrich - 1 - AktuellPos)
      InvokePrintMethod StartSpalte, Zaehler, Ctrl.ForeColor, Ctrl.BackColor,
                        Text, 5, ControlID, METHOD_Print
      Zaehler = Zaehler + 1: AktuellPos = SchraegStrich
   LOOP UNTIL AktuellPos > LEN(Ctrl.Text)

   ' Höhe des Steuerelements je nach Anzahl der in Text-Eigenschaft enthaltenen
   ' Elemente anpassen
   IF Ctrl.Height <> Zaehler THEN Ctrl.Height = Zaehler

   ' Cursor einstellen
   SetAttribute ControlID, ATTR_TextCursor, TC_Underscore

   SchalterStatusAnzeigen Ctrl, ControlID

END FUNCTION
```

SchalterStatusAnzeigen ermittelt durch die ungenutzte *Mode*-Eigenschaft, welche Einstellung des Schalters zuletzt aktiv war, und schreibt in die entsprechende Zeile einige Leerzeichen. Dann wird der in *SelText* enthaltene „echte" *Caption*-Text an der nun aktuellen Zeile ausgegeben, und zuletzt wird noch das hervorgehobene Zeichen mit der dafür in *SCREEN.ControlPanel* festgelegten Farbe geschrieben. Außerdem ändert SchalterStatusAnzeigen die Werte von *CurrentX* und *CurrentY*, da diese bestimmen, wo VBDOS den blinkenden Cursor anzeigt.

```
' Diese Prozedur wird immer aufgerufen, wenn sich die Schaltereinstellung ändert.
SUB SchalterStatusAnzeigen (Ctrl AS CONTROL, CID AS INTEGER)

   DIM LetzterStand AS INTEGER, Text AS STRING, TastenPos AS INTEGER
   DIM AlteFarbe AS INTEGER

   ' ungenutzte Mode-Eigenschaft wird mißbraucht, um letzten Schalterstand zu
   ' speichern:
   GetIntProperty LetzterStand, CID, PROP_Mode

   ' ungenutzte SelText-Eigenschaft wird benutzt, um die "wahre" Beschriftung
   ' (ohne eventuelles &-Zeichen, dafür mit Pfeil) zu speichern
   GetStringProperty Text, CID, PROP_SelText

   ' ungenutzte Style-Eigenschaft speichert, an welcher Stelle in Text ein Buchstabe
   ' hervorgehoben werden soll
   GetIntProperty TastenPos, CID, PROP_Style

   ' Letzter Stand wird gelöscht und neuer wird geschrieben
   InvokePrintMethod 0, LetzterStand, Ctrl.ForeColor, Ctrl.BackColor,
                     SPACE$(LEN(Text)), 5, CID, METHOD_Print
   InvokePrintMethod 0, Ctrl.Value, Ctrl.ForeColor, Ctrl.BackColor, Text, 5,
                     CID, METHOD_Print
   IF TastenPos THEN
      AlteFarbe = Ctrl.ForeColor ' Diese Mimik wird benötigt, weil der Aufruf...
      InvokePrintMethod TastenPos - 1, Ctrl.Value, SCREEN.ControlPanel(ACCESSKEY_
        FORECOLOR), Ctrl.BackColor, MID$(Text, TastenPos, 1), 5, CID, METHOD_Print
      Ctrl.ForeColor = AlteFarbe ' ... fehlerhafterweise ForeColor verändert!
   END IF

   ' Neuer Stand wird in Mode gespeichert; Cursor wird gesetzt
   SetIntProperty Ctrl.Value, CID, PROP_Mode
   SetIntProperty Ctrl.Value, CID, PROP_CurrentY
   SetIntProperty 0, CID, PROP_CurrentX

END SUB
```

Kompilieren und starten

Das war's! Sie können das Schalter-Steuerelement jetzt benutzen. Auf der Diskette finden Sie die bereits assemblierte Registrierungsdatei (SCHALT-

RG.OBJ). Kompilieren Sie SCHALTER.BAS mit dem Befehl `BC SCHALTER;` (wobei Sie auf Wunsch den Switch /G2 oder /G3 angeben können, was hier aber wenig bringt). Mit `LIB SCHALTER +SCHALTER+SCHALTRG` erhalten sie eine Objectcode-Library, und mit `LINK /Q SCHALTER+SCHALTRG,,,VBDOSQLB` können Sie eine Quick Library für VBDOS erstellen.

Wenn Sie nun VBDOS mit `VBDOS /L SCHALTER` starten, steht Ihnen im Form-Designer das zusätzliche Steuerelement „Schalter“ zur Verfügung, das Sie wie gewohnt auf Ihrer Form plazieren können. Selbstdefinierte Steuerelemente werden beim Entwurf nur als Bildfelder angezeigt; das wahre Aussehen ergibt sich erst später durch die PAINT-Ereignisprozedur, die wir programmiert haben.

Überraschungen und Korrekturen

Zu Anfang Ihrer Karriere als Programmierer von Steuerelementen werden Sie immer wieder erleben, daß VBDOS nicht so reagiert, wie Sie es erwartet haben. Fast immer werden Sie Ihr Steuerelement in Folge einer Testphase nachbessern müssen. Das bisher hier entworfene Schalter-Steuerelement weist zum Beispiel folgende Eigenschaften auf, die teils erfreulich sind, teils nicht und sicherlich nicht alle vorhersehbar waren:

- Die *BorderStyle*-Eigenschaft funktioniert hartnäckig, ohne daß wir dafür Code programmiert haben. Allerdings zeigt sie immer einen hervorgehobenen Rahmen an, auch dann, wenn wir in der Paint-Ereignisprozedur den Befehl `SetIntProperty 0, ControlID, PROP_BorderStyle` verwenden. Außerdem enthält *Height* die äußere Höhe des Schalters (inkl. Rahmen), so daß wir im *Paint*-Ereignis *Height* um zwei vergrößern müssen, wenn *BorderStyle* nicht 0 ist, und im *IntegerSet*-Ereignis muß es `Height 2` statt `Height` heißen, wenn ein Rahmen gesetzt ist.
- Im Form-Designer kann zwar ein Wert für die *Text*-Eigenschaft eingestellt werden, dieser wird jedoch offenbar nicht gespeichert. Benutzer unseres Steuerelements sind also darauf angewiesen, die Wahlmöglichkeiten erst zur Laufzeit durch eine Zuweisung an die *Text*-Eigenschaft festzulegen.
- Wenn wir in *IntegerSet* und *IntegerGet* den Zugriff auf die Eigenschaften *Visible* und *Enabled* zulassen, dann sind folgende Effekte zu beobachten: Wenn die *Enabled*-Eigenschaft auf FALSE gesetzt wird, funktioniert der Schalter nicht mehr, ohne jedoch seine Farbe automatisch zu wechseln (das ist dann wohl wieder unsere Sache); wird die *Visible*-Eigenschaft im Programm auf FALSE gesetzt, verschwindet unser Schalter sofort, wird aber nicht automatisch wieder angezeigt, wenn *Visible* auf TRUE gesetzt wird.
- Das Anwenden beliebiger Methoden auf unseren Schalter führt zu seltsamen Resultaten. Um das zu verhindern, müßten wir die Methodenereignisse

MthCls, *MthDrag* usw. ebenfalls als Schablonenereignisse beim Schalter-Steuerelement definieren und bei ihrem Aufruf einen Fehler „Methode ist auf dieses Objekt nicht anwendbar“ zurückgeben.
- Das *Paint*-Ereignis wird offenbar jedesmal aufgerufen, wenn der Schalter den Fokus erhält. Dadurch flackert er unangenehm, weil wir ein CLS im *Paint*-Ereignis eingebaut haben.

Oftmals erlebt man Überraschungen, weil unklar ist, die Behandlung welcher Eigenschaften und Ereignisse VBDOS selbst übernimmt (BorderStyle im Beispiel) und worum man sich selbst kümmern muß. Etwas mehr – aber noch lange nicht alle, denn vieles ist in dieser Hinsicht unsicher – Klarheit vermittelt der nächste Abschnitt.

17.5 Eigenschaften und weitere Details

Liste der Eigenschaften

Wie bereits erwähnt, stehen alle Eigenschaften, die VBDOS für Steuerelemente kennt, einem von Ihnen definierten Steuerelement zur Verfügung (mehr jedoch nicht – Sie können keine neuen, eigenen Eigenschaften erstellen). In CUSTGEN entscheiden Sie, welche davon im Form-Designer angezeigt werden, und in Ihren Integer-, String- und LongGet und -Set-Routinen können Sie Fehler zurückliefern, wenn zur Laufzeit auf Eigenschaften zugegriffen wird, die Ihr Steuerelement nicht unterstützt (oder nicht preisgeben will, weil es sie intern für andere Zwecke verwendet).

Folgende Eigenschaften sind insgesamt verfügbar:

INTEGER-Eigenschaften

–32768 bis 32767:
Action, BackColor, Column, ControlID, *CurrentX*, *CurrentY*, *Height*, *Index*, LargeChange, *Left*, ListCount, ListIndex, Max, Min, Mode, Row, *ScaleHeight*, *ScaleWidth*, SelLength, SelStart, SmallChange, *TabIndex*, *Top*, Value, *Width*

0 bis 255:
Alignment, *BorderStyle*, *DragMode*, *ForeColor*, *MousePointer*, ScrollBars, Style

–1 und 0:
Archive, *AutoRedraw*, Checked, *Enabled*, Hidden, MultiLine, Normal, ReadOnly, Sorted, System, *TabStop*, *Visible*

STRING-Eigenschaften
CtlName, Drive, FileName, List, Path, Pattern, SelText, Tag, Text, TypeID (schreibgeschützt)

LONG-Eigenschaften
Interval

Die in der Tabelle kursiv gesetzten Eigenschaften werden, wenn Sie keine Sonderbehandlung vorsehen, wie bei einem Bildfeld verarbeitet. Die Eigenschaft *BackColor* wirkt auch wie bei einem Bildfeld, löscht jedoch bei Änderung nicht das gesamte Feld; die Eigenschaft *Interval* funktioniert wie bei einem Timer-Steuerelement (jedes selbstdefinierte Steuerelement hat also prinzipiell die Fähigkeit, in regelmäßigen Zeitabständen Timer-Ereignisse auszulösen, und das werden wir uns im nächsten Abschnitt zunutze machen).

Die Eigenschaft *List( )* wird von selbstdefinierten Steuerelementen zwar unterstützt (eigens dafür besitzen die Ereignisse *StringSet* und *StringGet* das Argument *Index*); während Sie bei allen anderen Eigenschaften jedoch einfach mit Set- und GetProperty die Werte setzen und lesen können, ohne sich darum zu kümmern, wo sie gespeichert werden, obliegt Ihnen die Verwaltung der *List*-Eigenschaft selbst. Mehr dazu finden Sie unter „Variablen in Steuerelementen“ weiter unten.

Die Prozedur SetAttribute

Mittels der SetAttribute-Prozedur können Sie für Ihr Steuerelement einige spezielle Verhaltensweisen festlegen, die nicht über Eigenschaften steuerbar sind.

Der Aufruf von SetAttribute lautet

```
SetAttribute ControlID, AttributID, Wert
```

wobei Sie für *ControlID* die ControlID-Nummer des betroffenen Steuerelements verwenden. Für *AttributID* stehen folgende Konstanten aus CUSTINCL.BI zur Verfügung:

Konstante	*Verwendung*
ATTR_AccessKey	Geben Sie für *Wert* den ASCII-Code der Taste an, die Sie als Zugriffstaste für das Steuerelement bestimmen wollen (bei Betätigen von Alt und dieser Taste erhält Ihr Steuerelement den Fokus)
ATTR_AcceptFocus	Geben Sie für *Wert* TRUE an, wenn das Steuerelement den Fokus erhalten kann, FALSE, wenn nicht

Konstante	*Verwendung*
ATTR_TrapArrowKeys	Geben Sie für *Wert* TRUE an, wenn das Steuerelement die Pfeiltasten bei KeyDown und KeyUp erkennen soll; geben Sie FALSE an, wenn ein Betätigen der Pfeiltasten (wie bei Schaltflächen, Options- und Kontrollfeldern) den Fokus in der TabIndex-Reihenfolge vorwärts (Pfeil ab/Pfeil rechts) oder rückwärts (Pfeil auf/Pfeil links) bewegen soll.
ATTR_TextCursor	Geben Sie für Wert eine der folgenden Konstanten an: TC_NoCursor, wenn Ihr Steuerelement keinen Cursor zeigen soll, TC_Underscore für den Standard-Cursor, TC_Block, wenn Ihr Steuerelement einen Blockcursor zeigen soll.

Variablen in Steuerelementen

Solange Sie nur automatische Variablen in den Ereignisprozeduren Ihres Steuerelements verwenden (wie ich im Schalter-Beispiel), ergeben sich keinerlei Schwierigkeiten. Sobald Sie allerdings statische lokale Variablen oder globale Variablen im Code Ihres Steuerelements definieren, betreten Sie eine zusätzliche Komplexitätsstufe. Sie programmieren ja nur einen Steuerelement-Code, später kann Ihr Steuerelement aber beliebig oft auf einer Form existieren. Angenommen, wir hätten im Schalter-Beispiel die *Click*-Ereignisprozedur so formuliert...

```
FUNCTION Schalter_CClick (Ctrl AS CONTROL, BYVAL ControlID AS INTEGER) AS INTEGER
   STATIC BereitsAufgerufen
   IF NOT BereitsAufgerufen THEN BereitsAufgerufen = True: BEEP
   Ctrl.Value = Ctrl.Value + 1
END FUNCTION
```

... und damit im Sinn gehabt, daß ein Schalter immer dann einen Pieps von sich gibt, wenn er das erste Mal angeklickt wird. Das geht so lange gut, wie nur ein einziger Schalter in einer Anwendung benutzt wird. In dem Moment aber, wo mehrere Schalter existieren, gilt die obige *Click*-Prozedur für alle „Instanzen" des Schalters, und es würde nur gepiept, wenn zum ersten Mal auf *irgendeinen* Schalter geklickt wird. Die einfachste Lösung für solche Probleme ist es, eine nicht genutzte Eigenschaft des Steuerelements zur Speicherung der internen Daten zu verwenden. Im Schalter-Beispiel wurde so mit *SelText*, *Style* und *Mode* verfahren.

Besonders brisant wird es allerdings, wenn Ihr Steuerelement eine größere Datenmenge selbst verwalten muß, wenn Sie zum Beispiel die *List*-Eigenschaft unterstützen möchten. Nehmen wir an, Sie programmieren zum Beispiel eine

zweispaltige Listbox. Dann kommen Sie nicht umhin, ein Array für die Listeneinträge zu definieren; meistens werden Sie ein globales, auf Modulebene definiertes, dynamisches Array wählen, das bei Bedarf mit REDIM ausgeweitet werden kann:

```
DIM SHARED ListenEintrag(0 TO 0) AS STRING
```

Das ist aber genau der falsche Ansatz: Sobald ein Programm zwei oder mehr Ihrer zweispaltigen Listboxen verwendet, teilen sich alle das gleiche Array, und das kann nur schiefgehen. Stattdessen müssen Sie wie folgt vorgehen:

```
DIM SHARED ListenEintrag(0 TO MaxEintraege, 1 TO 1) AS STRING
DIM SHARED ControlIDListe(1 TO 1) AS INTEGER
DIM SHARED AnzahlElemente AS INTEGER
```

`MaxEintraege` ist eine Konstante, die festlegt, wieviele Einträge eine Liste maximal haben darf, und in der zweiten Dimension speichert das Array einen Code für das Steuerelement, zu dem die jeweilige Liste in der ersten Dimension gehört. Über das Array *ControlIDListe* kann zu einer gegebenen ControlID (die ja ein Steuerelement eindeutig identifiziert, aber nicht notwendigerweise von 0 bis x durchnumeriert ist) das entsprechende Array-Element gefunden werden. Dazu wird das *Load*-Ereignis des Steuerelements etwa so programmiert:

```
FUNCTION Element_CLoad (Ctrl AS CONTROL, BYVAL ControlID AS INTEGER) AS INTEGER
   AnzahlElemente = AnzahlElemente + 1
   IF AnzahlElemente > UBOUND(ControlIDListe) THEN
      REDIM PRESERVE ControlIDListe(1 TO AnzahlElemente) AS INTEGER
      REDIM PRESERVE ListenEintrag(0 TO MaxEintraege, 1 TO AnzahlElemente) AS STRING
   END IF
   ' Die ControlID des neuen Elements in der Liste speichern
   ControlIDListe(AnzahlElemente) = ControlID
   ' ... weiterer Code
```

Um dann z. B. auf ein Element der List-Eigenschaft zuzugreifen, kann *StringGet* so formuliert werden (*StringSet* analog):

```
FUNCTION Element_CStringGet (Ctrl AS CONTROL, BYVAL ControlID AS INTEGER, BYVAL ↵
         PropertyID AS INTEGER, Value AS STRING, BYVAL Index AS INTEGER) AS INTEGER

   DIM Gefunden AS INTEGER
   SELECT CASE PropertyID
   CASE PROP_List
      ' Die übergebene ControlID in der Liste suchen
      FOR i% = 1 TO AnzahlElemente
         IF ControlIDListe(i%) = ControlID then Gefunden = i%: EXIT FOR
      NEXT
      IF Gefunden = 0 THEN Element_CStringGet = 999: EXIT FUNCITON ' Fehler!
```

⇨

```
      Value = ListenEintrag(Index, Gefunden)
   CASE ELSE
      ' andere String-Eigenschaften
   END SELECT

END FUNCTION
```

Auch hier teilen sich alle Instanzen Ihres Steuerelements ein und dasselbe Array, aber diesmal ist das Array zweidimensional, und jede Instanz hat sozusagen eine Dimension für sich alleine.

Leider haben Sie bei dieser Vorgehensweise nicht die Möglichkeit, den Speicher wirklich dynamisch zu verwalten. REDIM PRESERVE erlaubt nur die Änderung der höchsten Dimension des Arrays. Es ist also möglich, neuen Speicher zu reservieren, wenn neue Steuerelemente hinzukommen; dies muß aber immer in einer festgelegten Blockgröße (hier `MaxEintraege`) geschehen und kann sich nicht der tatsächlichen Anzahl der Einträge in einem Steuerelement anpassen.

Doppelt gemoppelt

Die Prozeduren InvokeMethod, InvokeEvent, SetProperty und GetProperty sind Universalprozeduren, die je nachdem, welche Eigenschaft gelesen oder gesetzt, welche Methode oder welches Ereignis aufgerufen wird, unterschiedliche Anzahlen und Typen von Argumenten verarbeiten können. Sie können diese Prozeduren direkt im Code Ihres Steuerelements verwenden. Die Typüberprüfung von VBDOS ist hier jedoch abgeschaltet, und wenn Sie versehentlich ein Argument zu wenig übergeben, BYVAL vergessen oder ein Argument mit falschem Datentyp verwenden, wird VBDOS mit an Sicherheit grenzender Wahrscheinlichkeit abstürzen. Deshalb erzeugt CUSTGEN in der Code-Schablone eine Anzahl von DECLARE-Zeilen wie die folgende:

```
DECLARE SUB InvokeCLSMethod ALIAS "INVOKEMETHOD" (BYVAL NumArgs AS INTEGER, BYVAL CID AS INTEGER, BYVAL MthID AS INTEGER)
```

Hier wird sozusagen ein „Doppel“ der Prozedur InvokeMethod geschaffen, das für den Aufruf der CLS-Methode genau die richtige Anzahl von Parametern erwartet. So können Sie im Code des Steuerelements statt

```
InvokeMethod BYVAL 0, BYVAL CID, BYVAL Method_CLS
```

jetzt schreiben:

```
InvokeCLSMethod 0, CID, Method_CLS
```

Zwar ist es lästig, daß man die dadurch im Prinzip obsolet gewordene Anzahl der Argumente (3) und die Konstante für die Methode (`Method_CLS`) noch immer angeben muß; Sie haben jedoch den Vorteil der automatischen Typüberprü-

fung und des Verzichts auf BYVAL. Vergleichbare ALIAS-Prozeduren sind für alle Methoden, alle Ereignisse und den Zugriff auf STRING-, INTEGER- und LONG-Eigenschaften vorgesehen.

Einzig bei der PRINT-Methode ist es u.U. sinnvoll, doch auf das „normale“ InvokeMethod zurückzugreifen, um sich die Übergabe von Parametern zu ersparen. Der vollständige PRINT-Aufruf lautet:

```
InvokePrintMethod x, y, fcol, bcol, text$, 5, CID, METHOD_PRINT
```

Wenn Sie nun aber an der durch *CurrentX* und *CurrentY* festgelegten Position in den Standardfarben einen Text ausgeben wollen, können Sie auch schreiben:

```
InvokeMethod BYVAL text$, BYVAL 1, BYVAL CID, BYVAL METHOD_PRINT
```

17.6 Einfach, aber nützlich: „PrintBuffer“

Zum Abschluß dieses Kapitels, bevor ich Sie eigenen Experimenten überlasse und Ihnen schon jetzt viel Spaß mit diesem „Randgebiet“ der VBDOS-Programmierung wünsche, stelle ich ein etwas untypisches Steuerelement vor: Einen Druckpuffer, der selbsttätig ihm übergebene Daten häppchenweise an den Drucker schickt, etwa so, wie das in WINDOWS der „Druck-Manager“ tut. Sie werden staunen, mit wie wenig Aufwand eine solche doch schon recht anspruchsvolle Aufgabe lösbar ist.

Definition

Der „PrintBuffer“ wird ein Steuerelement sein, das mit Tastatur und Maus nichts zu schaffen hat. Er reagiert einzig und allein auf eine Änderung seiner Text-Eigenschaft. Sobald man etwas in die Text-Eigenschaft einträgt, beginnt PrintBuffer, in bestimmten Zeitintervallen diesen Text kilobyteweise an den Drucker zu senden. Die Zeitintervalle werden durch die Value-Eigenschaft in Viertelsekunden festgelegt.

Wenn der Text „verbraucht“ ist oder ein Fehler am Drucker auftritt, erzeugt PrintBuffer ein *Custom*-Ereignis mit dem Wert 0 (alles o.k.) oder 1 (Fehler am Drucker).

In der mit CUSTGEN erzeugten Schablone müssen hier nur die Get- und Set-Ereignisse (String, Integer, Long), das Paint- und das Timer-Ereignis aktiviert werden.

Die Ereignisse LongGet, LongSet, IntegerGet und IntegerSet

Die Long-Ereignisse dienen auch hier nur zum Abschmettern von Zugriffen, daher spare ich mir den Abdruck. Die Integer-Ereignisse erlauben die Verwendung der Eigenschaften Value, ForeColor, BackColor und Visible.

```
FUNCTION PrintBuffer_CIntegerGet (Ctrl AS CONTROL, BYVAL ControlID AS INTEGER,
                    BYVAL PropertyID AS INTEGER, Value AS INTEGER) AS INTEGER

   SELECT CASE PropertyID
   CASE PROP_Value, Prop_Visible, Prop_ForeColor, Prop_BackColor
      GetIntProperty Value, ControlID, PropertyID
   CASE ELSE
      PrintBuffer_CIntegerGet = 422
   END SELECT

END FUNCTION
```

```
FUNCTION PrintBuffer_CIntegerSet (Ctrl AS CONTROL, BYVAL ControlID AS INTEGER,
                BYVAL PropertyID AS INTEGER, BYVAL Value AS INTEGER) AS INTEGER

   SELECT CASE PropertyID
   CASE PROP_Value
      IF Value < 1 OR Value > 260 THEN
         PrintBuffer_CIntegerSet = 380
      ELSE
         SetIntProperty Value, ControlID, PropertyID
      END IF
   CASE Prop_Visible, Prop_ForeColor, Prop_BackColor
      SetIntProperty Value, ControlID, PropertyID
   CASE ELSE
      PrintBuffer_CIntegerSet = 422
   END SELECT

END FUNCTION
```

Bei der Zuweisung eines Wertes an die *Value*-Eigenschaft wird geprüft, ob der Wert 260 nicht überschreitet (da 260/4 = 65 die maximale Timer-Zeit ist). Falls doch, wird ein Fehler 380 (*ungültiger Eigenschaftswert*) zurückgegeben.

Die Ereignisse StringGet und StringSet

StringGet erlaubt das Abfragen der Eigenschaften *Tag* und *Text*. Mit StringSet können beide gesetzt werden; wird dabei ein nichtleerer String in *Text* eingetragen, setzt StringSet die *Interval*-Eigenschaft auf den 250fachen Wert der *Value*-Eigenschaft, so daß von nun an alle *Value*/4 Sekunden ein Timer-Ereignis ausgelöst wird.

```
FUNCTION PrintBuffer_CStringGet (Ctrl AS CONTROL, BYVAL ControlID AS INTEGER, ↩
   BYVAL PropertyID AS INTEGER, Value AS STRING, BYVAL Index AS INTEGER) AS INTEGER
   IF PropertyID = PROP_Text OR PropertyID = PROP_Tag THEN
      GetStringProperty Value, ControlID, PropertyID
   ELSE
      PrintBuffer_CStringGet = 422
   END IF

END FUNCTION
```

```
FUNCTION PrintBuffer_CStringSet (Ctrl AS CONTROL, BYVAL ControlID AS INTEGER, ↩
   BYVAL PropertyID AS INTEGER, Value AS STRING, BYVAL Index AS INTEGER) AS INTEGER
   IF PropertyID = PROP_Text THEN
      SetStringProperty Value, ControlID, PropertyID
      IF LEN(Value) THEN SetLongProperty Ctrl.Value * 250, ControlID, PROP_Interval
   ELSEIF PropertyID = PROP_Tag THEN
      SetStringProperty Value, ControlID, PropertyID
   ELSE
      PrintBuffer_CStringSet = 422
   END IF

END FUNCTION
```

Das Timer-Ereignis

Langsam arbeiten wir uns zum „harten Kern“ des PrintBuffers vor. Das *Timer*-Ereignis prüft beim Aufruf zunächst, ob überhaupt noch Zeichen in der *Text*-Eigenschaft übrig sind. Wenn nicht, wird im Programm, das den PrintBuffer verwendet, ein *Custom*-Ereignis ausgelöst, um ihm mitzuteilen, daß der Druck beendet ist, und die *Interval*-Eigenschaft wird auf 0 gesetzt, damit künftig keine *Timer*-Ereignisse mehr ausgelöst werden.

Ansonsten werden die ersten 1000 Bytes der *Text*-Eigenschaft an den Drucker geschickt (das aufrufende Programm kann über *Printer.PrintTarget* festlegen, wohin die Ausgabe geht). Tritt dabei ein Fehler auf, wird ein *Custom*-Ereignis ausgelöst. Ansonsten wird die Text-Eigenschaft um 1000 Bytes verkürzt.

Außerdem ruft das Timer-Ereignis die PrintBufferAnzeige-Routine auf, die für das Erscheinungbild des Steuerelements verantwortlich ist (siehe weiter unten).

```
FUNCTION PrintBuffer_CTimer (Ctrl AS CONTROL, BYVAL ControlID AS INTEGER) AS INTEGER

    DIM Drucken AS STRING
    PrintBufferAnzeige Ctrl, ControlID, True ' später erläutert

    GetStringProperty Drucken, ControlID, PROP_Text
```

```
    IF LEN(Drucken) = 0 THEN
       SetIntProperty 0, ControlID, PROP_Interval
       ' Anzeige verstecken
       PrintBufferAnzeige Ctrl, ControlID, 0:
       ' Custom-Event 0 bedeutet: Alles klar, Text gedruckt
       InvokeCustomEvent 0, ControlID, EVENT_Custom
    ELSE
       ON LOCAL ERROR RESUME NEXT
       Printer.PRINT LEFT$(Drucken, 1000);
       IF ERR THEN
          ' Custom-Event 1 bedeutet: Fehler beim Drucken
          ' hierdurch wird Custom natürlich, wenn das aufrufende Programm den Fehler
          ' nicht behebt oder die Text-Eigenschaft auf einen Leerstring setzt,
          ' in regelmäßigen Zeitabständen aufgerufen!
          InvokeCustomEvent 1, ControlID, EVENT_Custom
       ELSE
          IF LEN(Drucken) > 1000 THEN
             SetStringProperty MID$(Drucken, 1001), ControlID, PROP_Text
          ELSE
             SetStringProperty "", ControlID, PROP_Text
          END IF
       END IF
    END IF

END FUNCTION
```

Die Optik des PrintBuffers

Schließlich fehlen nur noch zwei Prozeduren: Die für das Paint-Ereignis und die PrintBufferAnzeige-Routine. Diese prüft zunächst, ob die Visible-Eigenschaft auf TRUE gesetzt ist. Wenn ja, zeigt sie in der oberen linken Ecke des Steuerelements einen sich im Uhrzeigersinn drehenden Strich an (es spielt also keine Rolle, wie groß das Steuerelement entworfen wird – genutzt wird immer nur die obere linke Ecke).

```
SUB PrintBufferAnzeige (Ctrl AS CONTROL, CID AS INTEGER, Aktiv AS INTEGER)

   DIM Wert AS INTEGER, Zeichen AS STRING * 1
   IF Ctrl.Visible THEN
      IF Aktiv THEN
         GetIntProperty Wert, CID, PROP_Style: Wert = (Wert + 1) MOD 4
         SetIntProperty Wert, CID, PROP_Style
         SELECT CASE Wert
         CASE 0: Zeichen = "¦"
         CASE 1: Zeichen = "/"
         CASE 2: Zeichen = "-"
         CASE 3: Zeichen = "\"
         END SELECT
```

```
      ELSE
         Zeichen = " "
      END IF
      InvokePrintMethod 0, 0, Ctrl.ForeColor, Ctrl.BackColor, Zeichen, 5,
                        CID, METHOD_PRINT
   END IF
END SUB
```

```
FUNCTION PrintBuffer_CPaint (Ctrl AS CONTROL, BYVAL ControlID AS INTEGER) AS INTEGER

   PrintBufferAnzeige Ctrl, ControlID, 0

END FUNCTION
```

Die Anzeigeroutine verwendet auch hier eine ungenutzte Eigenschaft – *Style* – um den zuletzt angezeigten Strich zu speichern und bei einem erneuten Aufruf das entsprechende Folgezeichen anzuzeigen.

PrintBuffer in der Praxis

Erstellen Sie die Libraries analog zur Vorgehensweise beim Schalter-Beispiel. Sie finden die Registrierungsdatei für den PrintBuffer unter PRBUFRG.OBJ auf der Diskette; das komplette Listing steht in PRBUFFER.BAS. Diesmal müssen Sie beim Kompilieren den Switch /X angeben, da wir ON LOCAL ERROR verwendet haben.

Abschließend sehen Sie hier ein kleines Beispiel für die Nutzung des PrintBuffers. Es geht davon aus, daß Sie eine Form erzeugt haben, die eine Schaltfläche namens „Drucken“, ein Textfeld namens „Datei“ und einen PrintBuffer namens „Druckpuffer“ enthält.

```
SUB Drucken_Click ()

   DIM Temp AS STRING
   ON LOCAL ERROR RESUME NEXT
   OPEN DateiName.Text FOR INPUT AS #1
   IF ERR THEN
      MSGBOX "Datei nicht gefunden.", 0, "Fehler"
   ELSE
      IF LOF(1) > 32000 THEN
         MSGBOX "Datei zu groß.", 0, "Fehler"
         ' man könnte allerdings auch erstmal 32 KB einlesen und im Custom-Ereignis
         ' dann immer, wenn der Puffer meldet, er sei fertig, noch etwas nachladen!
      ELSE
         Temp = SPACE$(LOF(1)): GET #1, 1, Temp
         DruckPuffer.Text = Temp ' fängt jetzt automatisch zu drucken an!
      END IF
   END IF
```

```
   CLOSE 1
END SUB
```

```
SUB DruckPuffer_Custom (Typ AS INTEGER)

   SELECT CASE Typ
   CASE 0
      MSGBOX "Alles ok - Druck fertig.", 0, "OK"
   CASE 1
      SELECT CASE MSGBOX("Fehler beim Drucken!", 5, "Fehler") ' Wiederh./Abbrechen
      CASE 2 ' Abbrechen
         DruckPuffer.Text = ""
      END SELECT
      ' Bei Wiederholen keine Aktion nötig - Druckpuffer macht das alleine
   END SELECT

END SUB
```

Listing 17–1: PRBDEMO.FRM

CodeView & fremde Sprachen 18

18.1 Assembler – was ist das?

Ich beginne mit einem Abschnitt für all diejenigen, die bisher nicht mit Assembler gearbeitet haben. Ich kann hier natürlich keine umfassende Einführung geben, aber ich will versuchen, Ihnen die Sache so schmackhaft zu machen, daß Sie sich am Ende des Abschnitts vielleicht doch irgendwo billig einen Assembler kaufen (nehmen Sie zum Experimentieren ruhig eine ältere Version, viel hat sich da nicht getan – nur OBJ-Files sollte sie erzeugen...) und ein wenig damit programmieren.

Assembler im ursprünglichen Sinn ist die reine Maschinensprache, also alles, was der Prozessor direkt verstehen kann. Im „reinen“ Assembler gibt es keine Variablennamen, sondern nur „die Zahl, die an Adresse 813Ah steht“ und „der Text, der an der Adresse steht, die das DX-Register enthält“. Halt – ein paar Variablen gibt es schon, und zwar die Speicherplätze, die im Prozessor selbst bereitstehen. Man nennt sie auch Register, und sie haben so klangvolle Namen wie AX, BX, CX, DX, BP oder SP – insgesamt sind es rund 16.

Nun programmiert heute niemand mehr in „reinem“ Assembler; vielmehr erlauben alle fortgeschrittenen Assembler (Microsofts MASM spätestens seit der Version 5) die Verwendung von Variablennamen, Sprungadressen und anderen Kleinigkeiten, die dem Programmierer viel Routinearbeit abnehmen.

Assembler-Befehle

Einer der Gründe, warum Assembler für viele ein Buch mit sieben Siegeln ist, sind seine zuweilen kryptischen Befehle. Wenn ich hier MOV, INT, RET, REP, STOSW, CMP und JNZ nenne, habe ich noch nicht einmal die seltsamsten herausgegriffen. Zum Glück gibt es unter den Hunderten von Assembler-Befehlen (mit jedem neuen Prozessor gibt es zusätzliche, und alte werden niemals gestrichen, damit die Programme von 1969 auch im Jahr 2069 noch laufen) eine relativ kleine „Grundausstattung“, mit der man schon sehr viele Probleme lösen kann.

Einer der wichtigsten Assemblerbefehle ist zweifelsohne MOV. MOV ist in Assembler so etwas wie das Gleichheitszeichen in BASIC und dient der Zuweisung von Werten sowohl an Register als auch an Adressen im Speicher. Einige Beispiele hierzu sehen Sie auf der folgenden Seite:

`MOV AX, BX`	Kopiert den Inhalt des Registers BX in das Register AX
`MOV BX, [AX]`	Kopiert den Inhalt der INTEGER-Variablen, die an der Adresse steht, die in AX gespeichert ist, in Register BX
`MOV [DX], 5`	Schreibt die Zahl 5 in die INTEGER-Variable, deren Adresse in DX gespeichert ist

Gut gestapelt ist halb gewonnen: Der Stack

Sicher haben Sie schon einmal vom „Stack-Speicher“ gehört. Während Sie sich in BASIC aber kaum darum kümmern müssen (der Speicher läuft allenfalls einmal über, wenn Sie zu viele Rekursionen programmieren), ist der Stack in Assembler der wichtigste Speicherbereich überhaupt. Fast alle Datenübergaben zwischen Prozeduren laufen über den Stack – auch in BASIC, da merken Sie es bloß nicht. Der Stack ist ein Speicherbereich, in dem jede Prozedur und jedes Programm wahllos Daten ablegen kann, vorausgesetzt, die Daten werden in der gleichen Reihenfolge wieder abgeholt. Bei einem Prozeduraufruf werden die Parameter, die die aufgerufene Prozedur erwartet, einfach auf den Stack gelegt, und die aufgerufene Prozedur holt sich die Daten dort ab. (Mit „Daten“ sind hier fast immer INTEGER-Werte gemeint, die also 2 Byte umfassen; längere Einheiten teilt man oft auf oder übergibt nicht die Daten selbst, sondern nur die Adresse des Speicherplatzes, an dem sie zu finden sind.)

Das Register SP ist eigens dafür reserviert, immer auf das letzte Element des Stacks zu zeigen („Stack Pointer“), und spezielle Assemblerbefehle existieren, um den Stack zu manipulieren: PUSH legt den Wert eines Registers auf den Stack (und verändert SP entsprechend), POP holt einen Registerwert vom Stack (und ändert ebenfalls SP). Beispiele:

`PUSH SI`	Schreibt den Wert des Registers SI auf den Stack
`POP AX`	Holt einen Wert vom Stack und schreibt ihn in AX
`PUSH AX` `POP BX`	Diese beiden Befehle zusammen wirken genau wie `MOV BX, AX`: Der Wert von AX wird auf den Stack gelegt und dann von dort in BX geholt.

Ein komplizierter Assemblerbefehl: MOVSB

Um schnell zu einem eindrucksvollen Beispiel vorzudringen, greife ich jetzt noch einen nicht ganz so trivialen Assembler-Befehl auf: MOVSB, ein Kürzel für „Move String Data: Byte“. Der MOVSB-Befehl kopiert ein Byte von der Adresse in DS:SI an die Adresse in ES:DI und erhöht SI und SD danach je um

1. (Man kann dafür sorgen, daß die Werte vermindert werden, aber auch das lassen wir hier außer acht.)

Wenn in Assembler eine Adresse – wie hier – mit einem Doppelpunkt geschrieben wird, dann handelt es sich um eine „Far-Adresse“, die aus einer Segment- und einer Offsetadresse besteht. Vor dem Doppelpunkt steht die Segment-, hinter dem Doppelpunkt die Offsetadresse. In BASIC können Sie die Segmentadresse einer Variablen mit VARSEG und die Offsetadresse mit VARPTR ermitteln.

Dem Befehl MOVSB kann noch das Wort REP vorangestellt werden, dann vermindert MOVSB jedesmal, wenn ein Byte kopiert wird, den Wert im CX-Register um 1 und kopiert so lange, bis CX den Wert 0 hat.

Eine kleine Anwendung

Ich wollte Ihnen mit wenig Aufwand vorführen, wozu man mit Assembler in der Lage ist, und in der Tat kennen Sie jetzt schon alle Befehle, um eine sehr nützliche Assembler-Routine zu programmieren. Ich muß allerdings ein wenig ausholen, um ihre Anwendung plausibel zu machen:

Nehmen wir an, Sie haben eine „Multimedia“-Datenbank programmiert, die eine Adreßkartei mit digitalisierten Fotos der Personen speichern kann. Sie verwenden ISAM für den Datenzugriff, und Sie haben einen Zugriffstyp wie folgt definiert:

```
TYPE AdressenTyp
   Name AS STRING * 40
   ...
   Konterfei(1 TO 1000) AS DOUBLE
END TYPE
```

Das Array *Konterfei()* soll jeweils das Bild der Person speichern, und es soll mit PUT (im Grafikmodus) auf dem Bildschirm angezeigt werden. Nun stellen Sie fest, daß der PUT-Befehl jedoch nur mit einfachen Arrays funktioniert und nicht mit Arrays, die in einem Typ enthalten sind:

```
DIM Adresse AS AdressenTyp
DIM HilfsArray(1 TO 1000) AS DOUBLE
PUT (100, 100), Adresse.Konterfei, PSET  ' geht nicht: Fehlermeldung!
PUT (100, 100), HilfsArray, PSET         ' würde klappen
```

Schweren Herzens beißen Sie in den sauren Apfel und programmieren eine Routine „CopyArray“, die mit einer FOR-Schleife die 1000 Elemente des Arrays kopiert:

```
SUB CopyArray(Typ AS AdressenTyp, Array() AS DOUBLE)
   FOR i%=1 TO 1000: Array(i%)=Typ.Konterfei(i%): NEXT
END SUB
```

Damit haben Sie das Problem gelöst, allerdings mit einem nicht gerade geringen Zeitaufwand zur Laufzeit. Lassen Sie sich nun die Alternative in Assembler vorführen: Wir schreiben eine Assemblerprozedur, der die Segment- und Offsetadresse für Quelle und Ziel des Kopiervorgangs sowie die Anzahl der zu kopierenden Bytes übergeben werden und die den Kopiervorgang mittels des MOVSB-Befehls ausführt:

```
.Model Medium, Basic    ; Diese Zeile sagt dem MASM, daß wir eine Routine für
                        ; BASIC programmieren
.Code                   ; wir fangen gleich mit der Routine an
CopyData Proc, VonSeg: Word, VonAdr: Word, NachSeg: Word, NachAdr: Word, Laenge: Word
; der Name der Routine ist "MoveData", und sie empfängt 5 Parameter vom Typ "WORD"
; (2-Byte-Ganzzahl, wie INTEGER in BASIC, jedoch ohne Vorzeichen)
Push SI                 ; Die Register SI, DI und DS werden auf den Stack gelegt,
Push DI                 ; um sie später wiederherstellen zu können, da VBDOS ab-
Push DS                 ; stürzt, wenn man ihm in den Registern herumpfuscht
Mov  DS, VonSeg         ; Wie es der MOVSB-Befehl erwartet, werden
Mov  SI, VonAdr         ; hier DS, SI, ES und DI mit der Quell- bzw.
Mov  ES, NachSeg        ; Zieladresse beschrieben
Mov  DI, NachAdr
Mov  CX, Laenge         ; in CX kommt die Anzahl Bytes
Rep  MovSb              ; jetzt CX mal ein Byte kopieren
Pop DS                  ; und nun die drei Register wieder vom Stack
Pop DI                  ; holen, damit VBDOS die Werte vorfindet, die
Pop SI                  ; es erwartet, und nicht unsere
Ret                     ; Ende der Prozedur - Rückkehr ins Programm
CopyData Endp           ; Zeichen für den Assembler, daß Prozedur zu Ende
End                     ; Zeichen für den Assembler, daß keine Prozedur mehr folgt
```

Das war schon alles. Sie sehen, daß wir einigen Aufwand treiben müssen, um die Register für VBDOS unverfälscht zu lassen; mehr dazu im zweiten Abschnitt dieses Kapitels.

Diese Assemblerdatei kann man jetzt mit dem Befehl `ML -c COPYDATA.ASM` zu einer OBJ-Datei assemblieren lassen. Das BASIC-Programm wird um die Zeile

```
DECLARE SUB CopyData (BYVAL StartSeg%, BYVAL StartAdr%, BYVAL
                      ZielSeg%, BYVAL ZielAdr%,BYVAL Laenge%)
```

ergänzt, und die Prozedur CopyArray muß nur noch umformuliert werden, so daß sie die Segment- und Offsetadressen des Start- und des Zielarrays ermittelt und übergibt:

```
SUB CopyArray(Typ AS AdressenTyp, Array() AS DOUBLE)
   CopyData VARSEG(Typ.Konterfei(1)), VARPTR(Typ.Konterfei(1)),
            VARSEG(Array(1)), VARPTR(Array(1)), 8000

END SUB
```

Sie müssen nun noch dafür sorgen, daß die vom Assembler erzeugte OBJ-Datei mit LINK zum Programm hinzugefügt oder in eine Quick Library eingebaut wird, und schon können Sie das Programm wie vorher verwenden.

Mit einem kleinen Unterschied: Während die alte CopyArray-Routine rund eine Sekunde brauchte, um (auf einem 486er) die 1000 Elemente zu kopieren, schafft die neue es in 0,08 Sekunden. Und dabei haben wir noch gar nicht alle Möglichkeiten ausgeschöpft; es gibt in Assembler nämlich auch Befehle, die mehrere Bytes auf einmal kopieren und dadurch noch etwas schneller sind.

Weitere Einsatzgebiete für CopyData

Die Routine CopyData finden Sie fertig assembliert als COPYDATA.OBJ auf der Diskette; eine Quick Library erstellen Sie mit dem Befehl `LINK /Q COPYDATA,,,VBDOSQLB;`. Sie können CopyData für beliebige Kopieroperationen einsetzen, nicht nur, um Arrays ineinander zu kopieren.

Bei CGA-, EGA- und VGA-Karten zum Beispiel beginnt der Bildschirmspeicher für den Textmodus an der Adresse 0 im Segment B800h und umfaßt 4000 Bytes. Die folgenden Funktionen können den Bilschirmspeicher in einen String kopieren und auch den Bildschirm anhand des Strings wiederherstellen:

```
FUNCTION GetBildschirm () AS STRING

   DIM Temp AS STRING
   Temp = SPACE$(4000) ' Unbedingt vorher den benötigten Platz reservieren.
                       ' Assembler kopiert einfach die Daten, ohne zu fragen,
                       ' ob er vielleicht etwas überschreibt!
   ' Achtung: String-Adressen nicht mit VARPTR/VARSEG, sondern mit SSEG/SADD holen!
   CopyData &HB800, 0, SSEG(Temp), SADD(Temp), 4000
   GetBildschirm = Temp

END FUNCTION
```

```
SUB PutBildschirm (Puffer AS STRING)

   ' Prüfen, ob auch kein Fehler gemacht wurde:
   IF LEN(Puffer) <> 4000 THEN EXIT SUB
   CopyData SSEG(Puffer), SADD(Puffer), &HB800, 0, 4000

END SUB
```

Listing 18–1: SCHIRM.BAS

Ein letztes Schmankerl: EMS nutzen

EMS-Speicher ist ein Trick, mit dem erweiterter Speicher jenseits der 1 MB-Grenze in gewöhnlichen Programmen, die nur auf Speicher innerhalb des ersten Megabyte zugreifen können, verwendet werden kann.

Voraussetzung für die Verwendung von EMS (vgl. Kapitel 8) ist ein geladener EMS-Treiber, zum Beispiel EMM386.EXE.

Bei der Verwendung von EMS wird der gesamte EMS-Speicher (der beliebig groß sein kann) in einzelne „Seiten“ zu je 16 KB eingeteilt. Innerhalb des Standardspeichers von 1 MB reserviert sich der EMS-Treiber ein Fenster von 64 KB, in das er bis zu vier Seiten aus dem gesamten EMS hardwaremäßig einblenden lassen kann.

Ein Programm, das EMS benutzen will, geht wie folgt vor:

- Es prüft, ob ein EMS-Treiber installiert ist.
- Es teilt dem EMS-Treiber mit, wieviele Speicherseiten es reservieren möchte, und der EMS-Treiber gibt daraufhin ein „Handle“, eine eindeutige Identifikationsnummer zurück, die das Programm künftig angeben muß, um auf den reservierten Speicher zuzugreifen.
- Um auf eine der reservierten Seiten zuzugreifen, gibt das Programm dem EMS-Treiber den Befehl, diese Seite an eine der vier möglichen Positionen in das 64 KB große Fenster einzublenden. Dann kann das Programm mit gewöhnlichen Speichermanipulationen auf diese Seite zugreifen, als wäre dort im Fenster wirklich Speicher vorhanden – in Wahrheit liegt der Speicher, auf den das Programm durch das Fenster zugreift, irgendwo im EMS-Speicher.
- Das Programm kann hintereinander beliebige Seiten einblenden und manipulieren.
- Wenn das Programm den EMS nicht mehr benötigt, kann es dem EMS-Treiber den Befehl zur Freigabe des reservierten Speichers geben.

Die Befehle an den EMS-Treiber werden über Interrupts erteilt. Ohne die CopyData-Routine ist es trotzdem nicht möglich, EMS von BASIC aus zu benutzen, da der EMS-Treiber zwar Speicherplatz zur Verfügung stellt, BASIC aber keine Methode kennt, dort irgendetwas zu speichern.

Auf der Diskette finden Sie ein Programm namens EMS.BAS, das Routinen zur Verfügung stellt, mit denen Sie beliebige Speicherbereiche in den EMS kopieren und aus dem EMS lesen können. Da jedoch immer nur komplette EMS-Seiten reserviert werden können, lohnt es sich selten, kleinere Bereiche in den EMS auszulagern. Vielmehr ist es sinnvoll, große Arrays oder Strings, die bestenfalls eine durch 16 KB teilbare Länge haben, in den EMS auszulagern.

Die Routinen selbst drucke ich hier nicht ab, wohl aber ein Beispiel zum Umgang mit ihnen:

```
' Wichtig: Immer zuerst prüfen, ob überhaupt ein EMS-Treiber geladen ist, sonst
' stürzt das Ganze ab:
IF NOT EMSGeladen THEN PRINT "Kein EMS.": END

' Funktion EMSFrei ermittelt, wieviel KB EMS frei sind
PRINT "Sie haben"; EMSFrei; "KB EMS frei."

' Testarray dimensionieren und mit Zahlen füllen
DIM Array(1 TO 100) AS INTEGER: FOR i% = 1 TO 100: Array(i%) = i%: NEXT

PRINT "Das Array enthält die Elemente"; Array(1); "-"; Array(100)
PRINT "Es wird im EMS gespeichert."

' Größe des Arrays ist 200 (100 Elemente à 2 Byte). Im EMS werden trotzdem 16 KB
' belegt, da nur ganze Seiten reserviert werden können.
' Funktion EMSSpeichern speichert einen Bereich im EMS ab; übergeben werden die
' Adresse und Länge. Zurückgegeben wird der Handle, unter dem gespeichert wurde.
Handle% = EMSSpeichern(VARSEG(Array(1)), VARPTR(Array(1)), 200)

PRINT "Der Handle ist"; Handle%: PRINT "Jetzt sind noch "; EMSFrei; "KB EMS frei."
PRINT "Das Array wird gelöscht."

ERASE Array: PRINT "Es enthält jetzt die Elemente"; Array(1); "-"; Array(100)

PRINT "Das Array wird wieder aus dem EMS geladen."
' EMSLaden lädt unter Angabe von Handle, Adresse und Länge einen Speicherbereich
' aus dem EMS.
EMSLaden Handle%, VARSEG(Array(1)), VARPTR(Array(1)), 200
PRINT "Es enthält jetzt die Elemente"; Array(1); "-"; Array(100)

PRINT "Der EMS wird freigegeben.": EMSFreigeben Handle%
PRINT "Jetzt sind wieder "; EMSFrei; "KB EMS frei."
```

Listing 18–2: (Ausschnitt aus) EMS.BAS

Resumée

Jetzt haben Sie die „Schokoladenseite“ von Assembler kennengelernt. Mit Subroutinen in Assembler erreichen Sie kompaktere und schnellere Programme und können Operationen durchführen, die BASIC nicht bietet. Wir haben mit ganz wenigen Assembler-Zeilen Dinge möglich gemacht, von denen man als BASIC-Programmierer sonst nur träumen kann.

Natürlich ist Assembler nicht immer die erste Wahl; die Routinen sind nicht sehr übersichtlich, und wenn ich Ihnen verriete, wie oft mein Rechner zwischendurch abgestürzt ist, als ich an den Beispielen für dieses Kapitel bastelte, verlören Sie womöglich völlig das Vertrauen. Assembler glänzt dadurch, daß

Sie damit *alles* programmieren können, was denkbar ist – darunter allerdings auch alles, was sinnlos oder gar falsch ist.

Im Abschnitt über CodeView weiter hinten in diesem Kapitel sehen Sie ein Beispiel für den Assembler-Code, den BC.EXE aus einem gewöhnlichen BASIC-Programm erzeugt.

Ohne Frage ist Assembler ein nützliches Werkzeug, das es dem BASIC-Programmierer ermöglicht, professionelle und effiziente Applikationen zu erstellen. Schon wenige Assemblerkenntnisse genügen, um die Grenzen von BASIC zu überwinden, wie dieser Abschnitt gezeigt hat. Probieren Sie es!

18.2 Details zum Umgang mit Assembler

Dieser Abschnitt richtet sich an all jene, die schon über Assemblerkenntnisse verfügen. Stichwortartig gebe ich einige Hinweise zur Schnittstelle zwischen BASIC und Assembler.

Registerwerte erhalten

Sorgen Sie dafür, daß Ihre Assembler-Routinen die Werte in den Registern SI, SS, DI, DS, BP und dem Flag-Register wiederherstellen, bevor sie beendet werden. Das aufrufende BASIC-Programm wird sonst abstürzen.

Strings

Strings werden in BASIC vergleichsweise aufwendig gespeichert (vgl. Kapitel 8). Wenn Sie eine Assembler-Routine schreiben, die einen BASIC-String lesen oder beschreiben soll, ist es am einfachsten, Sie übergeben der Routine die in BASIC mittels SSEGADD (oder SSEG/SADD) ermittelte Adresse des Strings und die mit LEN ermittelte Länge. Dadurch kann Ihre Assemblerroutine direkt auf die Stringdaten zugreifen. Ohne Verwendung von SSEG und SADD würde der Assemblerroutine die Adresse des Stringdeskriptors übergeben; Sie müssen den String dann mit den weiter unten beschriebenen Routinen manipulieren.

Stringmanipulationen von Assembler aus

VBDOS stellt in seinen Libraries einige Routinen zur gemischtsprachlichen Programmierung bereit, die die Handhabung von Strings vereinfachen sollen. Wenn Sie diese Routinen verwenden, sparen Sie sich das mühsame Dekodieren

der Stringdeskriptoren. Außerdem behält sich Microsoft die Änderung der Stringdeskriptoren vor, so daß Ihre über diese Routinen hinausgehenden Bemühungen womöglich mit der nächsten VBDOS-Version überholt sind.

Routine	*Argumente/Rückgabe*	*Nutzen*
StringAddress	Eingabe: Near-Adresse eines Stringdeskriptors (WORD) auf dem Stack Ausgabe: Far-Adresse der String-Daten in DX:AX	Verwenden Sie StringAddress, um zur von BASIC beim Aufruf der Assemblerroutine übergebenen Near-Adresse des Stringdeskriptors die Far-Adresse der Stringdaten selbst zu erhalten.
StringLength	Eingabe: Near-Aresse eines Stringdeskriptors (WORD) auf dem Stack Ausgabe: Länge des Strings in AX	Diese Routine ermittelt zur gegebenen Near-Adresse eines Stringdeskriptors die Länge des zugehörigen Strings.
StringAssign (1)	Eingabe: Deskriptoradresse (DWORD), 0 (WORD), Zieladresse (DWORD), Länge (WORD) auf dem Stack	Kopiert *Länge* Zeichen des BASIC-Strings, dessen Deskriptoradresse angegeben wird, an die Zieladresse.
StringAssign (2)	Eingabe: Stringadresse (DWORD), Länge (WORD), Deskriptoradresse (DWORD), 0 (WORD) auf dem Stack	Kopiert *Länge* Zeichen ab der gegebenen Stringadresse in einen String, dessen Deskriptoradresse übergeben wird, und setzt dessen Länge auf *Länge*. Wenn die übergebene Deskriptoradresse auf einen mit Nullen gefüllten 4-Byte-Bereich zeigt, wird der BASIC-String neu erzeugt, und der Deskriptor für den neuen String wird an die übergebene Deskriptoradresse geschrieben.
StringRelease	Eingabe: Near-Adresse des Stringdeskriptors (WORD) auf dem Stack	Gibt den Speicher, den der BASIC-String, auf den der übergebene Deskriptor zeigt, belegt, frei.

Diese Routinen verändern den Inhalt der Register AX, BC, CX, DX, ES und Flags.

BYVAL, SEG und ALIAS

Mittels der Schlüsselworte BYVAL und SEG kann man die Art der Datenübergabe an Prozeduren beeinflussen. Sie können entweder im DECLARE-Befehl angegeben werden – dann wirken sie auf jeden Prozeduraufruf – oder beim CALL-Befehl, wenn die Prozedur aufgerufen wird.

Ein gewöhnlicher CALL-Aufruf an eine Prozedur in Maschinensprache legt Near-Adressen der Parameter (oder Stringdeskriptoren) auf den Stack. Bei der Übergabe von Arrayelementen aus dynamischen Arrays, die nicht im DGROUP-Segment gespeichert werden und daher nicht über eine Near-Adresse erreichbar sind, wird zuerst eine Kopie des Array-Elements in DGROUP angelegt, und die Near-Adresse dieser Kopie wird übergeben.

Um stattdessen die Far-Adresse auf dem Stack zu übergeben, verwenden Sie entweder den Befehl CALLS (übergibt alle Werte als Far-Adressen), oder Sie verwenden das Schlüsselwort SEG vor dem betreffenden Parameter. Hierdurch können Sie viel Schreibarbeit sparen. Unsere Beispielzeilen aus dem ersten Abschnitt dieses Kapitels...

```
DECLARE SUB CopyData(BYVAL ss%,BYVAL so%,BYVAL ds%,BYVAL do%,BYVAL l%)

SUB CopyArray(Typ AS AdressenTyp, Array() AS DOUBLE)
   CopyData VARSEG(Typ.Konterfei(1)), VARPTR(Typ.Konterfei(1)), VARSEG(Array(1)), ↩
            VARPTR(Array(1)), 8000
END SUB
```

...kann man leichter auch so formulieren:

```
DECLARE SUB CopyData(SEG Quelle#, SEG Ziel#, BYVAL Laenge%)

SUB CopyArray(Typ AS AdressenTyp, Array() AS DOUBLE)
   CopyData Typ.Konterfei(1), Array(1), 8000
END SUB
```

Einziger Nachteil an dieser Form ist, daß wir durch den DECLARE-Befehl nun die Vielseitigkeit der Routine eingeschränkt haben, da sie nur noch DOUBLE-Zahlen als Start- und Zieladresse akzeptiert. Eine Möglichkeit, dies zu umgehen, ist zwar ein typenloses DECLARE...

```
DECLARE SUB CopyData

SUB CopyArray(Typ AS AdressenTyp, Array() AS DOUBLE)
   CopyData SEG Typ.Konterfei(1), SEG Array(1), BYVAL 8000
END SUB
```

... aber das ist sehr fehlerträchtig, weil man bei jedem Aufruf die korrekten SEG- und BYVAL-Schlüsselwörter angeben muß und VBDOS überhaupt keine Prüfung (nicht einmal der Argumentanzahl) vornimmt.

Eine Möglichkeit, die strenge Prüfung mit der Flexibilität zu kombinieren, bietet ALIAS. Hiermit können verschiedene Aufrufversionen ein- und der selben Routine definiert werden, etwa so wie auf der folgenden Seite:

```
DECLARE SUB CopyDataDouble ALIAS "CopyData" (SEG Quelle#, SEG Ziel#, BYVAL Laenge%)
DECLARE SUB CopyDataAdresse ALIAS "CopyData"(BYVAL ss%, BYVAL so%,
                                             BYVAL ds%, BYVAL do%, BYVAL l%)

SUB CopyArray(Typ AS AdressenTyp, Array() AS DOUBLE)
   CopyDataDouble Typ.Konterfei(1), Array(1), 8000
END SUB
```

Diese DECLARE-Zeilen vereinbaren eine Version namens CopyDataDouble, die zwei DOUBLE-Zahlen als Start- und Zieladresse nimmt, und eine andere, der man ganz universell die mit VARPTR/VARSEG ermittelten Adressen übergeben kann. Ähnlich geht auch das Hilfsprogramm CUSTGEN zur Erzeugung selbstdefinierter Steuerelemente vor (vgl. Abschnitt „Doppelt gemoppelt“ am Ende des Kapitels 17).

Interne Routinen und Variablen

Wenn es Ihnen gelingt, Informationen über die internen BASIC-Routinen zu bekommen, können Sie quasi jeden BASIC-Befehl auch von einem Assemblerprogramm auslösen und auf interne Variablen zugreifen. Sie brauchen dazu nur die betreffenden Prozeduren oder Variablen in Ihrem Assemblerprogramm als EXTRN zu definieren. In einigen Fällen werden Sie allerdings feststellen, daß die so erzeugten Assemblerprogramme nur in kompilierten Programmen, nicht aber als Quick Library unter VBDOS funktionieren. Das liegt daran, daß unter VBDOS nicht der volle Umfang der internen Routinen zur Verfügung steht; VBDOS erledigt einige Aufgaben mit eigenen Methoden.

Welche Prozeduren und Variablen VBDOS intern verwaltet, können Sie (wenn Sie Assembler-Erfahrung haben) relativ leicht ermitteln, indem Sie CodeView verwenden, um ein BASIC-Programm und den dazu vom Compiler erzeugten Assembler-Code synoptisch zu betrachten.

Einige Beispiele für eingebaute BASIC-Routinen sind:

Routine	*Eingabe/Ausgabe*	*Nutzen*
B$CSRL	Eingabe: keine Ausgabe: WORD im AX-Register	Ermittelt die aktuelle Zeile des Cursors (CSRLIN).
B$FPOS	Eingabe: beliebiges WORD auf dem Stack Ausgabe: WORD im AX-Register	Ermittelt die aktuelle Cursorspalte.
B$FRI2	Eingabe: –1, –2 oder –4 als WORD auf dem Stack Ausgabe: DWORD in DX:AX	Ermittelt den freien Speicher (Argument wie bei der BASIC-Funktion FRE).

Routine	*Eingabe/Ausgabe*	*Nutzen*
B$STDL	Eingabe: WORD auf dem Stack Ausgabe: keine	Löscht den String, auf den der Deskriptor zeigt, dessen Near-Adresse übergeben wird.
B$SETM	Eingabe: DWORD (mit Vorzeichen) auf dem Stack Ausgabe: DWORD in DX:AX	Funktioniert genau wie die BASIC-Funktion SETMEM.

Diese Routinen kann man meistens auch direkt in BASIC einsetzen. Anstatt des Befehls `a$ = ""`, einer intern vergleichsweise aufwendigen Stringzuweisung, könnte man schreiben:

```
DECLARE SUB StringLoeschen ALIAS "B$STDL" (Text  AS STRING)
StringLoeschen a$
```

In diesem Fall spart es einige Bytes im EXE-Programm und vielleicht auch ein wenig Zeit; oft lohnen solche Tricks allerdings die Mühe nicht.

18.3 BASIC und C

Wenn Sie in C Unterprogramme zur Nutzung mit BASIC schreiben möchten, haben Sie vor allem zwei Sonderregeln zu beachten:

- Sie müssen im DECLARE-Befehl in BASIC das Schlüsselwort CDECL angeben oder aber in C Ihre Prozeduren als „pascal“ oder „fortran“ definieren;
- Wenn Sie CDECL verwenden, wird ein Punkt im Prozedurnamen (BASIC) für C durch einen Unterstrich ersetzt, dem Prozedurnamen wird ein führender Unterstrich vorangestellt, und der Name wird in Kleinbuchstaben umgewandelt.

Eine Anzahl von C-Funktionen dürfen nicht verwendet werden, wenn Sie Prozeduren für BASIC schreiben. Dazu zählen getenv, putenv, system, spawn und exec. Wenn Sie malloc im Medium-Speichermodell oder nmalloc verwenden, müssen Sie einen globalen COMMON SHARED-Datenblock mit dem Namen NMALLOC definieren, der beliebige BASIC-Daten (am besten ein Array) enthält, das Sie so groß dimensionieren, daß die beiden alloc-Funktionen darin den angeforderten Speicher reservieren können.

BASIC speichert Arrays spaltenweise; Sie können das durch einen Switch (/R) beim Compiler und bei VBDOS jedoch ändern.

Weiterhin gelten für C-Subroutinen die Richtlinien aus 18.2 entsprechend.

18.4 CodeView

Das in der professionellen Ausgabe von VBDOS mitgelieferte CodeView-Programm ist ein „symbolischer Debugger“. Sie können ein kompiliertes EXE-Programm, auch und vor allem dann, wenn es Bestandteile enthält, die in anderen Sprachen geschrieben wurden, in CodeView schrittweise ablaufen lassen, Haltepunkte setzen und Variablenwerte abfragen, fast genau so, wie es in VBDOS mit reinen BASIC-Programmen möglich ist.

Voraussetzungen für den Einsatz von CodeView

Um Ihr Programm mit CodeView sinnvoll zu bearbeiten, müssen Sie beim Kompilieren den Switch /zi angeben und mit /CO linken. Wenn Ihr Programm Assemblerroutinen oder Routinen aus Libraries verwendet, müssen diese ebenfalls mit /zi assembliert bzw. kompiliert sein; andernfalls können Sie keine Details über die aufgerufenen Routinen in CodeView sehen.

CodeView starten

Starten Sie CodeView durch Eingabe von CV, gefolgt vom Namen des EXE-Programms, das Sie verfolgen möchten. Wählen Sie dann „Open Module“ aus dem „File“-Menü. CodeView zeigt ein Fenster mit allen Quelldateien an, die in Ihrem Programm Verwendung finden; wählen Sie die Quelldatei, die angezeigt werden soll (die hier häufig angezeigte Datei STRDSP.C ignorieren Sie einfach – die Microsoft-Entwickler haben versehentlich ein Modul mit Codeview-Informationen in die Library gesteckt).

Sie können maximal zwei verschiedene Quelltexte gleichzeitig laden; wenn Sie einen Quelltext in das zweite Code-Fenster laden wollen, wählen Sie dieses vorher über das „Window“-Menü aus.

Ansichtssache

Über den Menüpunkt „Source 1 Window...“ bzw. „Source 2 Window...“ des Menüs „Options“ können Sie bestimmen, welche Ansicht des geladenen Quelltextes – ich nehme an, es ist ein BASIC-Programm – CodeView Ihnen zeigt.

Die Auswahl „Source“ zeigt hierbei nur den Quelltext an; „Assembly“ begnügt sich mit dem vom Compiler erzeugten Assembler-Code, und „Mixed Source and Assembly“ – der aufschlußreichste Modus – zeigt jeweils eine BASIC-Zeile und darunter den dafür erzeugten Assembler-Code an.

Programme ausführen

Sie können das geladene Programm in CodeView mit den gleichen Tasten ausführen, die Sie auch in VBDOS verwenden. F5 startet das Programm mit voller Geschwindigkeit; F8 führt nur die nächste Anweisung aus, F10 ebenso, überspringt dabei jedoch Prozeduraufrufe als „black box“. Mit F9 können Sie einen Haltepunkt setzen oder entfernen, und F7 führt Ihr Programm bis zu der Stelle aus, an der der Cursor gerade steht.

Eine Entsprechung zu „Ablauf verfolgen“ in VBDOS findet sich hier im „Run“-Menü: Wählen Sie „Animate“, um Ihr Programm langsam vor Ihren Augen ablaufen zu lassen. Im Menü „Options“ können Sie die Geschwindigkeit einstellen, mit der das geschieht („Trace Speed“).

Wenn Sie im „Mixed Source and Assembly“-Modus arbeiten, dann legt die Option „Mixed Mode: Source“ bzw. „Mixed Mode: Assembly“ im „Source Window...“-Dialogfenster des Menüs „Options“ fest, ob die Tasten F8 und F10 die nächste BASIC-Anweisung oder die nächste Assembler-Anweisung ausführen (eine BASIC-Anweisung kann ja in mehreren Assembler-Anweisungen resultieren).

Über „Edit Breakpoints“ aus dem „Data“-Menü können Sie eine Liste aller gesetzten Haltepunkte sehen und beliebig verändern.

Die Ausgabe des Programms

CodeView ist üblicherweise bemüht, Ihnen sowohl den Quellcode als auch das, was Ihr Programm auf den Bildschirm ausgibt, anzuzeigen. Dazu ist standardmäßig die „Screen Swap“-Option aus dem „Options“-Menü aktiviert. Das bedeutet, daß CodeView alles, was Ihr Programm auf den Bildschirm schreibt, in einem Puffer zwischenspeichert und vor der Ausführung eines Befehls diesen Puffer auf den Bildschirm bringt, um nach Ende der Befehlsausführung wieder den CodeView-Schirm anzuzeigen. Für diesen Bildschirmpuffer wird insbesondere im Grafikmodus einiges an Speicherplatz benötigt.

Wenn Sie auf die Bildschirmausgabe verzichten können, schalten Sie „Screen Swap“ aus. CodeView meldet dann bei Versuchen, auf den Bildschirm zu schreiben, daß die Ausgabe verlorengegangen ist. (Sollte einmal der CodeView-Bildschirm durch eine Programmausgabe gestört werden, geben Sie den Befehl @ im Befehlsfenster ein.)

Falls Ihr Programm nur im Textmodus arbeitet und Sie eine CGA-, EGA-oder VGA-Karte verwenden, können Sie durch Aufruf von CodeView mit dem Switch /F erreichen, daß anstelle des Zwischenspeichers einfach eine andere

Bildschirmseite der Grafikkarte verwendet wird. Das ist schneller und spart Speicherplatz.

Mit der Taste F4 können Sie, wie in VBDOS auch, die Ausgabe Ihres Programms sehen.

Ein kleines Beispiel

Zur Erläuterung der CodeView-Anzeige habe ich ein kleines Programm geschrieben, das die Zahlen 100 bis 200 mittels einer Prozedur auf dem Bildschirm ausgibt. In CodeView wird das Programm wie folgt dargestellt (auf die rechte Spalte komme ich später noch zu sprechen):

„Mixed"-Anzeige ohne „Show Symbolic Name"	*mit „Show Symbolic Name"*
`1:  DECLARE SUB DruckeZahl (Wert AS INTEGER)`	
`2:  OPTION EXPLICIT`	
`3:  DIM i AS INTEGER`	
`4:`	
`5:  FOR i = 100 TO 200`	
`34A8:0030  MOV  AX,0064`	`MOV  AX,0064`
`34A8:0033  JMP  0043`	`JMP  DRUCKEZAHL-0010 (0043)`
`6:  DruckeZahl i`	
`34A8:0036  MOV  AX,0056`	`MOV  AX,0056`
`34A8:0039  PUSH  AX`	`PUSH  AX`
`34A8:003A  CALL  34A8:0053`	`CALL  DruckeZahl (34A8:0053)`
`7:  NEXT`	
`34A8:003F  MOV  AX,WORD PTR [0056]`	`MOV  AX,WORD PTR [0056]`
`34A8:0042  INC  AX`	`INC  AX`
`34A8:0043  MOV  WORD PTR [0056],AX`	`MOV  WORD PTR [0056],AX`
`34A8:0046  CMP  AX,00C8`	`CMP  AX,00C8`
`34A8:0049  JLE  0036`	`JLE  DRUCKEZAHL-001D (0036)`
`8:`	
`9:  END`	
`34A8:004B  CALL  34B1:0495`	`CALL  B$CEND (34B1:0495)`
`34A8:0050  JMP  0070`	`JMP  DRUCKEZAHL+001D (0070)`
`10:`	
`11: SUB DruckeZahl (Wert AS INTEGER)`	
`34A8:0053  MOV  CX,0000`	`MOV  CX,0000`
`34A8:0056  MOV  BX,0000`	`MOV  BX,0000`
`34A8:0059  CALL  34B1:0247`	`CALL  B$ENRA (34B1:0247)`
`12:  PRINT Wert;`	
`34A8:005E  MOV  SI,WORD PTR [BP+06]`	`MOV  SI,WORD PTR [Wert]`
`34A8:0061  PUSH  WORD PTR [SI]`	`PUSH  WORD PTR [SI]`
`34A8:0063  CALL  34B1:0036`	`CALL  B$PSI2 (34B1:0036)`
`13: END SUB`	
`34A8:0068  CALL  34B1:0298`	`CALL  B$EXSA (34B1:0298)`

Sie sehen, daß für einige BASIC-Befehle überhaupt kein Assemblercode generiert wird. Die Zeilen 1, 2 und 3 des BASIC-Programms enthalten nur „nicht ausführbare Befehle“.

Dann aber beginnt der ausführbare Teil: Der FOR-Befehl wird in zwei Assemblerinstruktionen umgewandelt. `MOV AX,0064` bedeutet nichts weiter, als die Zahl 100 in das Register AX zu schreiben (100 ist hexadezimal genau 64h). Dann folgt ein JMP (genau das, was in BASIC GOTO ist) an die Adresse 43h. Da die Assemblerzeilen mit Adressen versehen sind, ist leicht zu erkennen, daß dieser Sprung in die Zeile `MOV WORD PTR [0056], AX` verzweigt. Hier wird der Wert des Registers AX in eine 2-Byte-Variable (WORD) an die Speicherstelle 56 geschrieben. Die Speicherstelle 56h ist aber nichts anderes als unsere Variable i – was leicht zu erkennen ist, weil wir keine andere Variable verwenden. (Da Code und Daten in zwei verschiedenen Segmenten gespeichert werden, hat die Speicherstelle 56h nichts mit dem `MOV BX,0000`-Befehl zu tun, der im Programm an Adresse 56h steht.)

Bis jetzt ist also der Variablen i der Wert 100 zugewiesen worden. Nun folgt ein Vergleichsbefehl: `CMP AX, 00C8` vergleicht den Wert im Register AX (der ja ebenfalls noch 100 ist) mit dem Wert C8h (200). Die nächste Zeile – JLE heißt „Jump if less or equal“ – springt an die Adresse 36h, wenn der Vergleich ergab, daß AX kleiner oder gleich 200 ist.

Die drei Assemblerbefehle ab Adresse 36 schreiben den Wert 56h – das ist genau die Adresse der Variablen, die übergeben wird – in das Register AX, speichern es auf dem Stack und rufen eine Routine an Adresse 53h auf.

Nach der Rückkehr aus dieser Routine wird der Wert von Speicherstelle 56h in das Register AX kopiert, AX wird um eins erhöht, und weiter geht es bei der Adresse 43h, die wir oben schon einmal hatten. Dieser Zyklus wird also so lange ausgeführt, bis im Register AX die Zahl 201 steht, dann wird noch eine Routine an Adresse 34B1:0495h aufgerufen, und es erfolgt ein Sprung an die Adresse 70h. Diese ist im Listing nicht mehr enthalten; ab dieser Adresse beginnen noch einige BASIC-Routinen, die das Programm ordnungsgemäß beenden.

Unsere Subroutine „DruckeZahl“ ab Adresse 53 beschränkt sich offensichtlich darauf, einige interne BASIC-Routinen der Reihe nach aufzurufen (PRINT ist eine recht komplizierte Anweisung, die einiges „Drumherum“ erfordert).

Im „Source Window“-Dialogfeld aus dem Menü „Options“ können Sie ein Kontrollfeld namens „Show Symbolic Name“ aktivieren. Dann versucht CodeView, bei jedem Zugriff auf eine bestimmte Adresse den Namen einer Prozedur oder Variablen, auf die sich der Zugriff beziehen könnte, herauszufinden und zeigt diesen Namen an. Das Resultat sehen Sie in der rechte Spalte des Listings; wäh-

rend CodeView bei den CALL-Befehlen und Variablenzugriffen mit seinen Namen durchaus hilfreich ist, wirken Fehlschläge wie `JMP DRUCKEZAHL-0010` in der zweiten Zeile eher störend. Hier hat CodeView festgestellt, daß an eine Adresse gesprungen wird, die 16 Byte vor dem Beginn von DRUCKEZAHL liegt und meinte, uns das mitteilen zu müssen – inhaltlich hat dieser Sprung aber mit DRUCKEZAHL gar nichts zu tun, und daher ist es gut, daß wenigstens in Klammern die richtige Adresse angezeigt wird.

Außerdem hat CodeView trotz dieser Einstellung nicht den Namen der Variablen „i“ angezeigt – aber wenn Sie länger mit CodeView arbeiten, werden Sie sich schnell an solche Kleinigkeiten gewöhnen. Manche finden CodeView gerade wegen seiner Unvollkommenheit sympathisch...

Variablen anzeigen

Wenn Sie den aktuellen Wert einer Variablen sehen wollen, setzen Sie den Cursor auf den Variablennamen und drücken Sie Shift+F9. Wenn Sie eine Variable längere Zeit überwachen möchten, können Sie „Add Watch“ wählen und über das „Windows“-Menü das „Watch“-Fenster öffnen, in dem der Variablenwert dann dauerhaft angezeigt und verändert wird.

Mit Registern funktioniert das ebenso; sie können allerdings auch über das „Windows“-Menü ein Fenster eigens für Register öffnen.

Schließlich bietet das „Locals“-Fenster die Möglichkeit, die jeweiligen lokalen Variablen der Prozedur oder Code-Einheit, die gerade ausgeführt wird, anzuzeigen. Auf diese Weise ersparen Sie sich zahlreiche „Expression not available“-Fehler, die im Watch-Fenster angezeigt würden, wenn eine lokale Variable einer anderen Codeeinheit anzuzeigen ist.

Für den Zugriff auf Stringvariablen gelten besondere Regeln, die in der mit VBDOS gelieferten Datei LIESMICH.TXT erläutert sind.

Weitere CodeView-Features

Im „Calls“-Menü werden, wenn das Programm unterbrochen wurde, alle Prozeduren angezeigt, die zu diesem Zeitpunkt aufgerufen sind, und durch Auswahl einer Prozedur können Sie an die betreffende Stelle im Code springen.

Sie können bis zu zwei „Memory“-Fenster öffnen, in denen Sie den Speicherinhalt beliebiger Adressen auf verschiedene Weise anzeigen lassen können (wird gesteuert über „Memory Windows“ aus dem „Options“-Menü).

Mit „Set Runtime Arguments“ aus dem Menü „Run“ können Sie den Inhalt von COMMAND$ verändern.

Eine Liste der Switches, mit denen CodeView aufgerufen werden kann, finden Sie im Anhang B.

Alle Aktionen, die Sie mit CodeView ausführen können, sind nicht nur über Tastenkombinationen und Menüs, sondern auch über Befehle zugänglich, die in das „Command“-Fenster eingegeben werden. Der Befehl „G“ zum Beispiel führt das Programm aus; gefolgt von einem Prozedurnamen („G DruckeZahl“) stoppt er das Programm, sobald die benannte Prozedur erreicht wird. Einige Aktionen lassen sich über Befehle einfacher ausführen als über Menüs und Tastenkombinationen.

CodeView speichert alle Einstellungen (Haltepunkte, Status der einzelnen Fenster, „Options“-Einstellungen) in einer Datei namens CURRENT.STS und stellt sie beim nächsten Aufruf wieder her.

Oberflächliches 19

Die Benutzerschnittstelle ist das A und O der meisten Programme. Häufig richtet sich heute nicht mehr die Oberfläche nach dem Programm, sondern das Programm nach der gewünschten Oberfläche. Dieser Tatsache trägt VBDOS mit seinem Form-Designer und der ereignisgesteuerten Programmierung Rechnung. Das heißt aber nicht, daß man hiermit nichts mehr falsch machen kann.

Vielleicht ist es Ihnen auch schon einmal so gegangen: Sie tüfteln in Tage-, ja wochenlanger Kleinarbeit eine Benutzeroberfläche für Ihr Programm aus, präsentieren sie stolz dem Kunden, einem Versuchskaninchen oder dem zukünftigen Benutzer, und der – kommt damit nicht zurecht. Sie können das gar nicht verstehen; der Benutzer nimmt all die tollen Fähigkeiten, die Sie Ihrer Software spendiert haben, nicht in Anspruch und verlangt dafür an anderer Stelle mehr Komfort.

Ich habe es schon oft erlebt, daß Programmierer – mich selbst eingeschlossen – bei der Entwicklung einer Anwendung zu leicht den Anwender aus den Augen verlieren, daß die technische Perfektion des Programms wichtiger wird als die Frage, ob der Benutzer damit überhaupt etwas anfangen kann.

Im Mittelpunkt dieses Kapitels sollen ausnahmsweise also nicht Visual BASIC und der Programmierer, sondern der Anwender und seine Ansprüche stehen.

19.1 Sprache

Versuchen Sie in jedem Fall, das Programm streng einsprachig zu halten. Da VBDOS jetzt auch deutsche Fehlermeldungen kennt, fällt es nicht schwer, Software komplett auf Deutsch zu entwickeln.

Falls Sie jedoch auch Software in einer anderen Sprache programmieren, sollten Sie jedes deutsche Wort vermeiden. Ich arbeitete vor kurzem mit einem englischen Programm, das sich plötzlich mit einer schwedischen Fehlermeldung verabschiedete – so ähnlich fühlen sich wahrscheinlich Menschen, die kein Deutsch können, wenn sie mit einer teilweise deutschen Oberfläche konfrontiert werden. Unterschätzen Sie nicht den psychologischen Effekt: Zu einem Programm, das den Benutzer in kritischen Situationen „verläßt“, nicht mehr seine Sprache spricht, schöpft er kein Vertrauen mehr.

VBDOS enthält einige deutsche Texte, die Sie nicht verändern können, und zwar im sogenannten „Systemmenü“ (vgl. *ControlBox*-Eigenschaft im Objekt-Referenzteil) und auch in der MSGBOX-Routine. Einzig die Standard-MSGBOX mit der „OK“-Schaltfläche ließe sich noch in einem englischen Programm

verwenden. Wenn Sie auf eine englische Version des Systemmenüfeldes angewiesen sind, können Sie in der Library bzw. dem Runtime-Modul mit einem Diskeditor die deutschen Zeichenketten durch (gleichlange) englische ersetzen – obwohl das nur eine Notlösung sein sollte.

Multilinguale Systeme

Wenn Sie mehrsprachige Software programmieren, also Programme, von denen es Versionen in verschiedenen Sprachen geben soll, haben Sie prinzipiell zwei Möglichkeiten, dies technisch zu realisieren.

Sie können einerseits mit einer Text-Datenbank arbeiten, die – durch Zahlen codiert – alle Texte und Meldungen enthält. Zur Laufzeit werden die Texte dann aus der Datenbank gelesen, mit einer Routine, wie ich sie am Ende von Kapitel 10 vorgestellt habe.

Diese Methode ist recht mühsam, insbesondere deshalb, weil Sie beim Bearbeiten Ihres Programms immer nur Code-Nummern vor sich haben und dadurch die Übersicht über das Programm und die Bearbeitbarkeit leiden.

Ein anderer Ansatz, mit dem ich seit einigen Jahren arbeite, um englische und deutsche Programme zu erzeugen, ist recht trivial: Jeder Programmzeile, die in der englischen Version anders lauten muß, geht eine Zeile voran, die mit REM $E beginnt und danach den veränderten Inhalt aufweist, etwa so:

```
CLS
COLOR 7, 0
BEEP
REM $E LOCATE 5, 28: PRINT "This is a centered line."
LOCATE 5, 24: PRINT "Dies ist eine zentrierte Zeile."
```

Zum Kompilieren verwende ich eine Batchprozedur, die, wenn die englische Version erzeugt werden soll, mittels eines kleinen selbstgeschriebenen Programms den Quelltext nach REM $E-Zeilen absucht, das REM $E entfernt und die folgende Zeile löscht. Das Ganze spielt sich in einer temporären Datei ab, die danach wieder gelöscht wird.

Diese Methode hat den Vorteil, daß Sie die deutsche Programmversion ohne Schwierigkeiten und ohne Veränderungen am Quelltext erzeugen können (die REM $E-Zeilen werden ja ignoriert). Außerdem ist es sehr einfach, nicht nur Strings, sondern beliebige Befehle „sprachabhängig“ zu formulieren, wie Sie am LOCATE im Beispiel sehen.

Der Nachteil ist, daß Sie zwei unterschiedliche EXE-Dateien für die beiden Sprachen erhalten, und nicht ein Programm, das beides kann.

Schwierig wird es nur bei den Formen: Hier werden Sie nicht umhin können, für beide Sprachen eine eigene Form zu entwerfen.

19.2 Installation

Jede Software, die aus mehr als einem einzelnen EXE-Programm besteht und nicht von Ihnen persönlich installiert wird, sollte ein Installationsprogramm besitzen, und sei es nur eine Batch-Prozedur. Sonst können Sie die seltsamsten Dinge erleben: Manche Benutzer verzichten zum Beispiel entschieden darauf, Verzeichnisse anzulegen, so daß sich im Hauptverzeichnis der Festplatte mindestens ein Dutzend verschiedenster Kleinprogramme tummeln. Andere installieren jedes Programm versehentlich gleich mehrmals, tragen jedes Verzeichnis in die PATH-Umgebungsvariable ein und wundern sich dann, warum sie eine Fehlermeldung erhalten – und so weiter.

Sie sollten, wenn irgend möglich, ein Installationsprogramm namens INSTALL.EXE oder SETUP.EXE auf der (ersten) Diskette Ihres Programms vorsehen, das alle Aufgaben, die in Zusammenhang mit der Installation anfallen, erledigt. Im folgenden finden Sie hierzu einige Anregungen.

Konfiguration

Falls die Anzahl der installierten Dateien von bestimmten Angaben des Benutzers abhängig ist, sollten Sie diese Angaben zunächst abfragen und vom Benutzer bestätigen lassen. (Einige Informationen über die Systemkonfiguration könnte das Install-Programm auch selbsttätig einholen; vgl. dazu Kapitel 22.) Danach sollte der Installationsvorgang starten, in dessen Verlauf keine weiteren Fragen gestellt werden, bis auf die Aufforderung, die nächste Diskette einzulegen.

Untersuchung der Festplatte

Bevor die Installation beginnt, sollte(n) die Festplatte(n) des Systems, auf dem die Installation läuft, geprüft werden. Erstens ist zu ermitteln, ob genügend Platz zur Verfügung steht (eine Routine dafür findet sich in Kapitel 22).

Hinweis: Online-Kompressionsprogramme wie z. B. STACKER oder auch ein Zusatzprogramm zu DR-DOS geben als „freien Platz“ eine geschätzte Größe zurück. Es kann also sein, daß Sie vor der Installation genügend Platz auf der Festplatte gefunden haben, während der Installation aber dennoch ein „Platte voll“-Fehler auftritt; andererseits kann aber auch eine reibungslose Installation möglich sein, obwohl der freie Platz dagegen spricht. Verhindern Sie also nicht, daß Ihr Programm bei zu wenig freiem Platz installiert wird, sondern warnen Sie den Benutzer nur, und bieten Sie ihm die Möglichkeit, trotzdem fortzufahren.

Zweitens wäre es sinnvoll, zu prüfen, ob schon eine Version des zu installierenden Programms (möglicherweise eine ältere) installiert ist. Falls ja, könnte man den Benutzer fragen, ob sie durch die neue ersetzt werden soll. Je nach Programm muß man dazu evtl. alle Verzeichnisse durchsuchen (mit der INHALT.BAS-Sammlung aus Kapitel 22; alle existierenden Laufwerke ermitteln Sie am besten, indem Sie ein nicht sichtbares Laufwerkslistenfeld verwenden).

Drittens könnte Ihr Programm, falls Sie mit dem Standard-Runtime-Modul VBDRT10.EXE (VBDRT10E bzw. -A.EXE in der Profi-Version) arbeiten, prüfen, ob das Runtime-Modul – evtl. als Bestandteil eines anderen installierten Programms – bereits auf der Festplatte des Benutzers steht. Hierzu reicht es meist, die PATH-Umgebungsvariable zu untersuchen (eine Routine hierfür ist in Kapitel 10 beschrieben). Das Runtime-Modul müßte dann nicht mehr installiert werden.

Verteilung der Dateien auf die Disketten

Falls das Installationsprogramm einfach alle Disketten auf die Festplatte kopiert, spielt die Verteilung der Dateien auf die Disketten keine Rolle. Sie können sich für die Erzeugung des „Master“-Diskettensatzes ein kleines Programm schreiben, daß das Verzeichnis einliest, auf dem die Dateien stehen, und die optimale Verteilung der Dateien auf die Disketten (maximale Auslastung jeder Diskette) berechnet. Dazu verwendet man einen Backtracking-Algorithmus, der so lange Dateien hinzunimmt, bis die volle Diskettengröße (fast) erreicht ist. Beachten Sie allerdings, daß eine Datei auf der Diskette immer etwas mehr Speicher verbraucht, als sie wirklich lang ist (Aufrundung auf volle Cluster). Die Clustergröße erfahren Sie mit CHKDSK oder dem Listing in Kapitel 22, „Freier Platz auf einem Datenträger“.

Wenn Ihr Programm allerdings nicht immer alle Dateien kopiert, lohnt es sich, ein Installationsprogramm zu schreiben, das aus einer Textdatei liest, welche Datei auf welcher Diskette steht. Dann können Sie das Programm mühelos –

durch einfaches Auswechseln der Textdatei – auch für andere Diskettengrößen oder, wenn Sie so flexibel programmieren wie Microsoft, für ganz andere Programme verwenden. (Microsoft verwendet für fast alle DOS-Programme das gleiche SETUP; die Texte wie „Willkommen zur BASIC-Installation“ stehen in einer Datei.)

Kopieren und Stückeln von Dateien

Natürlich kann man Dateien mit dem Befehl „SHELL COPY“ kopieren. Eleganter geht es aber, wenn man eine eigene Routine verwendet:

```
FUNCTION DateiKopieren (Quelle AS STRING, Ziel AS STRING) AS INTEGER

   DIM nr1 AS INTEGER, nr2 AS INTEGER, Erledigt AS LONG, Laenge AS LONG
   DIM Puffer AS STRING, MaxPuffer AS INTEGER, ZeitCode AS DOUBLE

   ON LOCAL ERROR RESUME NEXT
   MaxPuffer = 32767: Puffer = SPACE$(MaxPuffer)
   DO WHILE ERR = 7 OR ERR = 14  ' Testen, wieviel Speicher wir belegen können
      MaxPuffer = MaxPuffer - 2000: ERR = 0: Puffer = SPACE$(MaxPuffer)
   LOOP
   ZeitCode = DateiZeitCode(Quelle)        ' ist im Kapitel 22 abgedruckt

   ON LOCAL ERROR GOTO CopyFehler
   nr1 = FREEFILE: OPEN Quelle FOR BINARY AS #nr1: nr2 = FREEFILE
   OPEN Ziel FOR OUTPUT AS #nr2: CLOSE nr2: OPEN Ziel FOR BINARY AS #2

   ' SEEK #2, LOF(2) + 1 (diesen Befehl zum Anhängen verwenden)
   Laenge = LOF(nr1)
   DO WHILE Laenge - Erledigt > MaxPuffer ' MaxPuffer-Blocks lesen, bis Rest kleiner
      GET #nr1, , Puffer: PUT #nr2, , Puffer: Erledigt = Erledigt + MaxPuffer
   LOOP
   IF Laenge > Erledigt THEN              ' Rest einlesen, wenn noch vorhanden
      Puffer = SPACE$(Laenge - Erledigt): GET #nr1, , Puffer: PUT #nr2, , Puffer
   END IF
   CLOSE nr1, nr2
   SetzeDateiZeitCode Ziel, ZeitCode      ' ist im Kapitel 22 abgedruckt

   DateiKopieren = 0   ' Funktionswert 0 = alles o.k.
   EXIT FUNCTION

CopyFehler:
   DateiKopieren = ERR ' Fehler als Funktionswert zurückgeben

END FUNCTION
```

Listing 19–1: DATEIKOP.BAS

Diese Kopierroutine kann auch leicht erweitert werden, so daß man eine Datei, die auf mehrere Disketten verteilt wurde, wieder zusammensetzen kann. Dazu braucht man nur die Befehle `OPEN Ziel FOR OUTPUT AS #nr2` und `CLOSE nr2` zu entfernen und dafür den auskommentierten SEEK-Befehl zu aktivieren. Dann hängt die Routine die Quelldatei an die Zieldatei an, falls diese schon existert.

Die Funktion bewahrt das Datum der Datei mit Hilfe von zwei Routinen, die im Kapitel 22 vorgestellt werden (FILEDATE.BAS auf der Diskette).

Komprimieren

Echte Komprimier-Routinen sind in BASIC schwer zu programmieren, weil man sich auf die Ebene der Bitmanipulationen herabbegeben muß. Ich verweise Sie hier an die einschlägigen Toolbox-Hersteller, die diverse in Assembler programmierte Komprimierungsalgorithmen anbieten.

Trotzdem können Sie auch so schon einiges tun, um den Platzbedarf Ihres Programms auf den Installationsdisketten gering zu halten. So können Sie z. B., falls Ihr Programm eine große Anzahl kleiner Dateien umfaßt, mehrere Dateien zusammenfügen und erst bei der Installation die große Datei wieder zerlegen – das spart den „Aufrundungsplatz“ auf der Diskette. Falls Ihr Programm sequentielle Dateien mit Zahlen darin verwendet, können Sie diese leicht komprimieren, indem Sie sie in Binärdateien umwandeln. Verwenden Sie z.B. die MK*x*$- und CV*x*-Funktionen, um Zahlen umzuwandeln.

Änderung von Konfigurationsdateien

Wenn Ihr Program bei der Installation Veränderungen an irgendwelchen Dateien auf der Festplatte des Benutzers vornimmt – zum Beispiel AUTOEXEC.BAT oder CONFIG.SYS – gehört es zum guten Ton, ihn vorher die Änderungen begutachten und notfalls verwerfen zu lassen. Im Zweifel wird der Benutzer sein System besser kennen als Sie, und Laien werden sich hüten, Ihrem Installationsprogramm einen Wunsch abzuschlagen.

Zwar ist es ein leichtes, das Verzeichnis, in dem Ihr Programm installiert wurde, an die PATH-Einstellung in AUTOEXEC.BAT anzufügen; angesichts der Tatsache, daß das sehr viele Programme tun, ist das inzwischen aber eher „out“. Die Zeile in AUTOEXEC.BAT darf nur 255 Zeichen umfassen, und jede unnötige Eintragung in der PATH-Variablen verlangsamt das System. Überlegen Sie, ob es nicht viel besser ist, anhand der PATH-Variablen ein Verzeichnis auszumachen, in das Sie eine Batchprozedur schreiben, mittels derer Ihr Programm aufgerufen wird. So wird in den meisten Fällen der gleiche Effekt er-

reicht – der Benutzer kann Ihr Programm von überall her aufrufen –, ohne daß der PATH unnötig verlängert wurde.

Installation von Zusatzkomponenten

Ein vernünftiges Installationsprogramm sollte, sofern Ihr Programm einige Teile beinhaltet, die nur auf Wunsch des Benutzers installiert werden, jederzeit die Möglichkeit der Nachinstallation bieten. Hierzu sollte das Installationsprogramm bei späterem Aufruf erkennen, daß das Programm bereits auf der Festplatte steht und welche Komponenten installiert sind; sodann kann der Benutzer wählen, welche Komponenten er zusätzlich installieren möchte.

Deinstallation

Auch wenn es uns Programmierern in der Seele weh tut: Manche Benutzer möchten eventuell das Programm einmal wieder von der Platte entfernen, und nicht alle sind mit DOS oder einem beliebigen Utility so vertraut, daß sie sich selbst ans Löschen wagen.

Deshalb sollte ein gutes Installationsprogramm auch das installierte Programm von der Festplatte entfernen können und dabei alle Änderungen, die auf der Festplatte des Benutzers vorgenommen wurden, rückgängig machen. Beachten Sie dabei, daß es nicht reicht, z. B. einfach den Zustand der AUTOEXEC.BAT wiederherzustellen, der vor der Installation vorlag – denn vielleicht sind inzwischen andere Programme installiert worden, die ebenfalls Änderungen durchgeführt haben.

Vielmehr muß das Installationsprogramm genau prüfen, welche Zeilen, welche Abschnitte im PATH-Befehl usw. von ihm selbst eingefügt wurden und diese entfernen.

19.3 Hilfe und Dokumentation

Die Hilfe am Bildschirm ist ein ganz eigenes Thema. Jedes Unternehmen hat hier seine eigene Philosophie. Viele bieten überhaupt keine Hilfstexte an, und zwar nicht nur aus Bequemlichkeit, sondern auch aus einer ganz handfesten Überlegung heraus: Programme, die so gut mit Hilfefunktionen ausgestattet sind, daß man auf ein Handbuch verzichten kann, lassen sich wesentlich leichter raubkopieren. Programme, für deren kryptische Benutzeroberfläche man sich ständig eine Pinwand mit 4 Seiten Tastaturkürzeln neben den Bildschirm stellen muß – solche Programme sind heute erstaunlicherweise meist im Preisbereich

jenseits der Fünftausendmarkgrenze angesiedelt – mag niemand sich einfach so kopieren.

Auch die Dokumentation wird völlig unterschiedlich gehandhabt. Manche verzichten völlig darauf und verweisen auf die umfangreiche Online-Hilfe. Microsoft hat erkannt, daß der Benutzer nicht an den Wert eines Produktes glaubt, wenn er nicht einige Kilo Papier bekommt. Die gedruckten Handbücher enthalten denn auch in vielen Bereichen wortwörtlich die gleichen Texte wie die Online-Hilfe – sogar die Fehler wurden übernommen.

Inhalt der schriftlichen Dokumentation

Es war von mir nicht nur so dahingesagt, sondern durchaus ernstgemeint: Viele Kunden akzeptieren eine Software erst als Wert, wenn sie mit Material daherkommt – sprich: dickes Handbuch, viele Disketten, eleganter Schuber. Wenn Ihr Programm wirklich so gut und selbsterklärend ist, daß sich eine echte Dokumentation weitgehend erübrigt, können Sie einfach ein dickes Buch mit lauter Bildschirmausdrucken und Anwendungsbeispielen Ihres Programms zusammenstellen, die vielsagend beschriftet sind.

Zumeist wird sich aber doch Dokumentationsbedarf ergeben; ein allgemein recht sinnvoller Ansatz ist eine Dokumentation, die folgende Bereiche abdeckt:

- Das im Programm zum Einsatz kommende Fachwissen. Dieser Punkt ist je nach Anwendung unterschiedlich umfangreich; in einem Desktop-Publishing-Programm zum Beispiel, das eine Domäne ausgebildeter Spezialisten plötzlich einer breiten Masse zugänglich macht, sollte auch einiges über die Typografie vermittelt werden. In einem Programm hingegen, das ohnehin nur von ausgebildeten Spezialisten verwendet wird, kann dieser Bereich sehr kurz ausfallen (evtl. in Form eines Glossars).
- Ein Teil, der sich weniger mit dem Programminhalt als vielmehr mit der Oberfläche beschäftigt: Wie starte ich das Programm? Wie wähle ich einen Punkt aus einem Menü? Wie schalte ich zur Grafikanzeige um? Wie gebe ich Daten ein?
- Ein Teil, der die Leistungsfähigkeit des Programms allgemein beschreibt („Geladene Zeitreihen können als Grafik oder Tabelle ausgedruckt werden“) und anhand von umfangreichen Beispielen einzelne Aspekte aus der Praxis aufgreift („Von der Tabelle zur Grafik in 10 Schritten“).
- Ein Nachschlageteil, in dem jede Funktion des Programms stichwortartig mit Einsatzbereich und Grenzen genannt ist. Auch wenn es schwerfällt, sollte hier klar zu lesen sein, was das Programm *nicht* kann. Oftmals neigen die Handbuchschreiber dazu, solche Informationen wegzulassen („...ist in diesem Modus nur eingeschränkt möglich...“, „... wird nicht automatisch unterstützt...“),

so daß dem Benutzer ein Hoffnungsschimmer bleibt, der sich nach dem Durchlesen des kompletten Handbuchs als sehr blaß erweist.
- Ein umfangreiches, vollständiges, sorgfältig erstelltes Stichwortverzeichnis (freundlicher Gruß an Microsoft).

Art der Online-Hilfe

Wenn Sie überhaupt eine Online-Hilfe anbieten wollen, haben Sie grundsätzlich zwei Möglichkeiten:

Die erste, einfachere ist die, bei Betätigung einer Hilfetaste einfach in einen Programmteil zu verzweigen, der es ermöglicht, die gesamte Hilfedatei durchzublättern. Versehen Sie die Hilfedatei am Anfang mit einem Verzeichnis und bieten Sie die Möglichkeit, im Text nach einem Stichwort zu suchen (mit der INSTR-Funktion geht das vergleichsweise flott, wenn es Ihnen gelingt, die Hilfedatei komplett in den Speicher zu laden; ansonsten müssen Sie evtl. eine Unterteilung in Häppchen vornehmen, die in den Speicher passen).

Eine solche Lösung braucht nicht viel mehr als eine Form mit einem Textfeld.

Die zweite Möglichkeit ist etwas aufwendiger: Kontextsensitive Hilfe. „Kontextsensitiv“ bedeutet, daß das Programm in der Lage sein sollte, dem Benutzer immer die Ausschnitte aus der Hilfedatei anzuzeigen, die für Ihn wahrscheinlich gerade interessant sind. Das ist nicht immer leicht, und wenn es danebengeht, ist das Ergebnis meist schlechter als bei der oben beschriebenen Primitivhilfe. Es ist für den Benutzer nämlich äußerst nervenaufreibend, wenn er Hilfe anfordert, weil er nicht weiß, was er im Feld „Polarität“ eintragen soll, stattdessen aber eine Hilfe-Seite angezeigt wird, die ihm mitteilt, er möge auf dieser Bildschirmseite doch die Kenndaten für das gewünschte Bauteil eingeben. Andererseits wird er auch kein Interesse haben, zu erfahren, daß er mit der Pfeil-nach-rechts-Taste den Cursor bewegen und mit ESC das Eingabefeld löschen kann.

Sie sehen, daß die kontextsensitive Hilfe oft ein schmaler Grat ist, auf dem Ihr Programm mit seiner fehlenden Intelligenz zu erraten sucht, was der Benutzer wissen will. Ein funktionsfähiges Querverweissystem wirkt hier wahre Wunder: Sie könnten auf einer Hilfeseite z. B. „Mehr Details“ oder eine „Übersicht“ anbieten – ganz so, wie Sie es von VBDOS her kennen, oder vielleicht noch besser.

Realisation der kontextsensitiven Hilfe

Wenn Sie die Profi-Version von VBDOS besitzen, ist eine Datei namens HELP.FRM im Lieferumfang enthalten (im Kapitel 21 ist sie etwas eingehender beschrieben). Ich rate Ihnen allerdings, dieses Tool allenfalls für kleine Programme oder zu Anschauungszwecken einzusetzen; es ist nicht besonders ausgereift.

Die kontextsensitive Hilfe ist ein kniffliges Projekt. Zwei Kernaspekte möchte ich kurz herausgreifen: Die Speicherung der Daten und die Ermittlung des anzuzeigenden Themas.

Datenspeicherung bei der kontextsensitiven Hilfe

Eine Möglichkeit der Datenspeicherung ist in diesem Falle eine Textdatenbank, wie ich sie am Ende von Kapitel 10 vorgestellt habe.

Wenn Sie keinen Komprimierungsalgorithmus für die Speicherung der Hilfedaten einsetzen möchten, können Sie allerdings auch eine gewöhnliche Textdatei für den eigentlichen Hilftstext verwenden, die mit speziellen Steuerzeichen strukturiert ist. In einer zweiten Datei, einer binären Indexdatei, könnten alle Hilfethemen und Stichworte alphabetisch abgelegt sein. So kann – per binärem Suchen – ein bestimmtes Thema sehr schnell gefunden werden; in der Indexdatei müßte zusätzlich die Information enthalten sein, an welcher Position das zugehörige Hilfethema in der Textdatei steht und wie lang es ist; dann kann das Thema aus der Textdatei (die als Binärdatei geöffnet wird) sofort gelesen werden.

Beim Programmstart müßte stets überprüft werden, ob die Indexdatei existiert und ob sie auf dem aktuellen Stand ist (Datum der Indexdatei mit Datum der Textdatei vergleichen).

Ein Datensatz der binären Indexdatei könnte so aussehen:

```
TYPE IndexType
   Thema AS STRING * 40
   Anfang AS LONG
   Laenge AS INTEGER
END TYPE
```

Angenommen, Sie haben aus der Indexdatei den auf das gesuchte Thema passenden Datensatz gefunden, dann würden Sie mit

```
OPEN Textdatei FOR BINARY AS #DateiNummer
Puffer = SPACE$(Index.Laenge)
```

```
GET #DateiNummer, Index.Anfang, Puffer
CLOSE #DateiNummer
```

auf den zugehörigen Text zugreifen können.

Welches Thema anzeigen?

Wenn Sie eine kontextsensitive Hilfe realisieren, ist es wichtig, stets zu wissen, welches Hilfethema gerade „aktuell“ ist. Eine einfache Möglichkeit, dies festzulegen, wäre, eine COMMON SHARED-Variable zu verwenden und dieser immer dann, wenn sich die Situation verändert, einen neuen Wert zuzuweisen, an der die Hilfsfunktion erkennt, wozu Hilfe angezeigt werden soll.

Wenn Sie allerdings Formen und Steuerelemente einsetzen, dann haben Sie noch eine andere Möglichkeit: Definieren Sie einfach ein Menü mit einem Menüpunkt „Hilfe“, dem Sie die Taste F1 als Abkürzungstaste zuordnen. Dann wird *Hilfe_Click* immer aufgerufen, wenn der Benutzer F1 drückt, egal, welches Element gerade den Fokus hat. Tragen Sie ferner in die *Tag*-Eigenschaft jedes Objektes einen eindeutigen Hilfe-Code ein. In der Hilfe-Prozedur können Sie dann durch Abfrage der Eigenschaft *SCREEN.ActiveControl.Tag* feststellen, von wo aus Hilfe angefordert wurde.

Inhalt der Online-Hilfe

Es liegt auf der Hand, daß es keinen Sinn hat, zu viele Details in die Online-Hilfe zu packen. Beispiele und Schritt-für-Schritt-Anleitungen sind in gedruckter Form wesentlich praktischer. Stattdessen sollte die Online-Hilfe so kurz und präzise wie möglich schildern, welche Möglichkeiten in der aktuellen Situation zur Verfügung stehen und was sie bewirken.

Sie haben es leichter, Hilfstexte für bestimmte Situationen zu verfassen, wenn Sie sich dabei den Benutzer vorstellen, der meistens ungefähr weiß, was er will, aber nicht, wie er es mit ihrem Programm erreicht.

Erfahrungsgemäß ist das Verfassen von Hilfstexten besonders leicht, wenn Sie die Gelegenheit haben, auf zwei Rechnern gleichzeitig zu arbeiten: auf dem einen läuft das zu dokumentierende Programm, auf dem anderen schreiben Sie die Anleitung.

19.4 Intuitive Oberflächen

Der Begriff „intuitive Bedienung“ steht für das höchste Ziel jeder Progammoberfläche: Der Benutzer soll sein Ziel durch lauter Schritte erreichen, die so selbstverständlich sind, daß man über sie nicht mehr nachdenkt.

Wenn auf der Straße ein Gullydeckel fehlt, dann wissen Sie „intuitiv“, ohne weiter über die Physik zu sinnieren, daß Sie besser außen herum gehen sollten, da Sie sonst Bekanntschaft mit dem städtischen Kanalsystem machen werden. Jeder Mensch wird diese „Intuition“ haben, nicht nur Sie, weil Sie schon länger in der Stadt wohnen.

Eine „intuitive“ Programmoberfläche müßte also von jedem Menschen – ausgenommen vielleicht jenen, die öfters in Kanäle fallen – bedienbar sein. Es gehört nicht viel Erfahrung dazu, dies als Utopie zu erkennen. Dennoch kann man die Qualität einer Benutzeroberfläche zumindest ungefähr daran messen, ob das, was dem Benutzer in den Sinn kommt, auch zu dem führt, was er im Sinn hat.

Ich will Sie hier zu einigen Gedanken über die „Psyche“ des Benutzers und über Ergonomie anregen, die Ihnen vielleicht in eigenen Programmen nützlich sind:

Fangen Sie nicht die Maus

Meine ersten mausbedienbaren Programme sahen aus wie immer, nur daß man den Menübalken jetzt durch Auf- und Abbewegen der Maus verschieben konnte. Ein Verbrechen an der Maus, wie ich heute weiß: Der Benutzer fühlt sich verunsichert, wenn der Mauszeiger sich nicht so bewegt, wie er das erwartet. Der Mauszeiger sollte immer über den ganzen Bildschirm frei bewegbar sein. Es macht nichts, wenn beim Klicken außerhalb eines bestimmten Bereichs nichts passiert, aber es stört, wenn man die Maus gar nicht aus diesem Bereich herausbewegen kann.

Maus contra Tastatur

Mausfreundliche Programme zeichnen sich dadurch aus, daß man sie weitgehend ohne Tastatur bedienen *kann*, die Tastatur aber dennoch nicht völlig verbannt ist. Bieten Sie, wo immer möglich, Auswahllisten an, die der Tastaturliebhaber aber auch durch Direkteingabe umgehen kann (in VBDOS: Kombinationsfelder). Verwenden Sie Bildlaufleisten oder Größer- und Kleiner-Zeichen, auf die man mit der Maus klicken kann, wenn es um eine Zahleneingabe geht.

Die professionelle Ausgabe von VBDOS enthält als Beispiel für benutzerdefinierte Steuerelemente das (allerdings verbesserungswürdige) Steuerelement

„Spin“, mit dem man Zahlenbereiche unter Verwendung der Maus einstellen kann.

Manche Benutzer sind so mausfixiert, daß sie, würde man ihnen eine Tastatur einblenden, auch Texte mit der Maus eingeben würden. Andere sind froh, wenn Sie ihnen ein paar Tastenkürzel zuwerfen und sie nicht dauernd auf den Bildschirm sehen müssen. Für Tastaturbenutzer ist das ein wichtiger Aspekt: Der Programmablauf muß so gleichförmig sein, daß man das Programm bedienen kann, ohne auf den Bildschirm zu sehen.

Für Tastaturbenutzer ist es in ereignisgesteuerten Programmen extrem wichtig, daß Sie die *TabIndex*-Eigenschaften der Objekte auf einer Form sinnvoll einstellen, damit der Fokus nicht scheinbar planlos zwischen den Objekten herumspringt, wenn man die Tab-Taste betätigt.

Nutzen Sie den Gewohnheitseffekt

Der Mensch ist ein Gewohnheitstier. Verunsichern Sie den Benutzer nicht, indem Sie ihm immer ein gleichaussehendes Menü zeigen, in dem mal diese, mal jene Auswahlmöglichkeiten zu sehen sind. Gestalten Sie vielmehr die Menüs so, daß sie aufgrund von Farbe, Position auf dem Bildschirm, Rahmenart o. ä. sofort erkennbar sind. Ein Untermenü wird sofort als Untermenü erkannt, wenn im Hintergrund noch das (überlappte) Hauptmenü zu sehen ist; wird das Untermenü genau über dem Hauptmenü angezeigt und unterscheidet sich lediglich durch die Überschrift, dauert es länger, bis der Benutzer sieht, woran er ist.

Wenn dasselbe Menü bestimmte Punkte manchmal enthält und manchmal nicht, verzichten Sie darauf, die Menüpunkte völlig zu streichen (das Menü sieht dann zu „anders“ aus), sondern färben Sie sie grau oder markieren Sie sie.

Definieren Sie niemals Tasten unterschiedlich. Gehen Sie insbesondere bei Verwendung der Funktionstasten F1 bis F10 sicher, daß ein- und dieselbe Taste nicht in jedem Kontext eine andere Bedeutung hat!

Schreiben Sie „smarte“ Programme

„Smart“ – das ist ein Wort, das die deutsche Sprache so nicht kennt. Im Englischen hat es zwar ein wenig den Nebensinn „intelligent“, ist aber nicht ganz so dick aufgetragen. Wer kann schon ein intelligentes Programm schreiben – aber „smart“, das sind die vielen kleinen Annehmlichkeiten, die dem Benutzer das Leben leichter machen: Daß er das Datum nicht mit einer Null vor dem Monat schreiben und nicht das Jahrhundert mit angeben muß, daß es aber auch nichts ausmacht, wenn er es dennoch tut. Daß ihm eine Liste von Dateinamen gezeigt

wird, wenn er keinen eingibt. Daß sinnvolle Standards in Felder eingetragen werden, die er mit ENTER übergeht. Daß der Benutzer bei einem Grafikprogramm sofort eine Rohgrafik sieht und diese später durch tausend Einstellungen verändern kann, anstatt daß er erst blindlings tausend Einstellungen vornimmt und dann eine verhunzte Grafik sieht, und so weiter. „Smart“ ist es, wenn Sie versuchen, zu ahnen, was der Benutzer tun wird, und Ihr Programm ihm dann den roten Teppich ausrollt.

Die leidigen Farben

Ich werde nie den Augenblick vergessen, als mir ein Programmierer, der jahrelang mit einem monochromen Bildschirm gearbeitet hatte, sein erstes farbiges Programm vorführte. Das war nicht farbig, sondern bunt – und das Schlimmste: man konnte die Farben nicht verstellen.

Für wie gemäßigt Sie Ihren Geschmack auch halten mögen, geben Sie dem Benutzer zumindest ein paar Möglichkeiten, die Farben selbst zu wählen. Mit dem PALETTE-Befehl können Sie sogar statt der 16 Standardfarben eine beliebige andere 16er Kombination aus mindestens 64 verschiedenen Farben zusammenstellen. Nachdem ich anfangs dem Benutzer alle Wahlmöglichkeiten gelassen und so manchen vor die Qual der Wahl von einem guten Dutzend Farben für verschiedene Anzeigezwecke gestellt hatte (eine Farbe für Standardtext, eine für die Rähmchen, eine für die Fehler, eine für die Warte-Anzeige...), biete ich nun in meinen Programmen meist drei, vier voreingestellte und aufeinander abgestimmte Kombinationen an.

Wie gesagt, können Sie mit PALETTE recht ansprechende Ergebnisse erzielen; pro Farbe gibt es hier dann nicht nur „hell“ und „normal“, sondern auch himmelblau, apfelgrün, bordeauxrot und zitronengelb – in dieser Zusammenstellung allerdings nicht zu empfehlen.

Algorithmen für den Alltag 20

Immer wieder kommt es vor, daß ein Programm wegen eines unnötigen Flaschenhalses zu langsam läuft. Man bemüht sich, so schnellen Code wie möglich zu programmieren, und übersieht eine einzelne kleine Stelle, wegen der dann das ganze Programm ins Stocken kommt. Um solche Probleme vermeiden zu helfen, sollen hier einige Standard-Algorithmen in ihrer BASIC-Implementation vorgestellt werden.

20.1 Sortieren

Quicksort

Der Quicksort-Algorithmus, seit seiner Erfindung durch C.A.R. Hoare gewiß zum am intensivsten benutzten Standard-Sortieralgorithmus avanciert, sortiert ein Array durch Teilung. Er wählt ein beliebiges Referenz-Element, teilt den Gesamtbereich an dieser Stelle in zwei Teilbereiche, wobei links vom Referenz-Element alle kleineren und rechts alle größeren Elemente einsortiert werden. Dann sortiert Quicksort rekursiv die zwei Teilbereiche, so lange, bis die Teilbereiche nur noch ein Element umfassen – das Array ist sortiert. Quicksort ist der Array-Sortieralgorithmus mit der besten Allround-Leistung, er kann prinzipiell überall eingesetzt werden.

Die vorliegende Quicksort-Implementation ist selten anzutreffen, da Quicksort normalerweise rekursiv formuliert wird, der abgedruckte Algorithmus jedoch nicht rekursiv arbeitet. Diese Version benötigt etwas weniger Speicher als die rekursive, weil sie, wenn sie anfängt, einen Teilbereich zu sortieren, nur seine Grenzen speichern muß, während die rekursive Version alle lokalen Variablen zwischenspeichern müßte. Deshalb ist die nichtrekursive Ausführung ein kleines bißchen schneller.

```
SUB QuickSort (Zeile() AS STRING, Anzahl AS INTEGER)

   CONST MaxMerk = 15   ' reicht zum Sortieren von 2^15 Elementen

   DIM Links AS INTEGER, Rechts AS INTEGER
   DIM MerkAnzahl AS INTEGER, Vergleich AS STRING
   DIM MerkL(1 TO MaxMerk) AS INTEGER, MerkR(1 TO MaxMerk) AS INTEGER

   MerkAnzahl = 1
   MerkL(1) = LBOUND(Zeile): MerkR(1) = Anzahl
```

⇨

```
   DO
      Links = MerkL(MerkAnzahl): Rechts = MerkR(MerkAnzahl)
      MerkAnzahl = MerkAnzahl   1
      DO
         i% = Links: j% = Rechts
         Vergleich = Zeile((Links + Rechts) \ 2)
         DO
            DO WHILE Zeile(i%) < Vergleich: i% = i% + 1: LOOP
            DO WHILE Vergleich < Zeile(j%): j% = j%   1: LOOP
            IF i% <= j% THEN
               SWAP Zeile(i%), Zeile(j%)
               i% = i% + 1: j% = j%   1
            END IF
         LOOP UNTIL i% > j%
         ' Auf den "Stack" muß unbedingt der größere Teil gelegt werden; der
         ' kleinere wird sofort bearbeitet. Sonst wäre in Extremfällen der
         ' "Stack" zu schnell voll.
         IF j%   Links < Rechts   i% THEN
            IF i% < Rechts THEN
               MerkAnzahl = MerkAnzahl + 1
               MerkL(MerkAnzahl) = i%: MerkR(MerkAnzahl) = Rechts
            END IF
            Rechts = j%
         ELSE
            IF Links < j% THEN
               MerkAnzahl = MerkAnzahl + 1
               MerkL(MerkAnzahl) = Links: MerkR(MerkAnzahl) = j%
            END IF
            Links = i%
         END IF
      LOOP WHILE Links < Rechts
   LOOP UNTIL MerkAnzahl = 0

END SUB
```

Listing 20–1: QSORT.BAS

Beim Aufruf müssen diesem Algorithmus nur die zu sortierenden Elemente – im Beispiel ein String-Array – und deren Anzahl übergeben werden. Da es nicht möglich ist, mehr als 32.767 Elemente in einem Array zu haben, reichen INTEGER-Typen für die interne Verwaltung völlig aus.

Der Algorithmus stellt mit LBOUND die untere Grenze des Arrays fest und sortiert von ihr bis zur angegebenen Element-Anzahl.

Die Konstante *MaxMerk*, die den internen Puffer festlegt, kann verkleinert werden; die Anzahl der zu sortierenden Elemente darf $2^{MaxMerk}$ nicht übersteigen.

Sollen statt Strings andere Datentypen (zum Beispiel selbstdefinierte) sortiert werden, müssen nur die Kopfzeile und sämtliche Zeilen geändert werden, in denen die Variable „Vergleich“ auftaucht. Dieser Quicksort-Algorithmus kann so-

gar zum Sortieren größerer Datenmengen (in einem RANDOM-File, in Datensätzen mit fester Länge) auf der Festplatte dienen. Man muß dann alle Zeilen, die auf das zu sortierende Array zugreifen, so umschreiben, daß sie stattdessen die Festplatte benutzen, eventuell für die internen Variablen (*Rechts*, *Links*, *MerkR*, *MerkL* usw.) den Datentyp LONG wählen und darüberhinaus die Konstante *MaxMerk* anpassen.

Ich werde jedoch noch einen Algorithmus behandeln, der für solche Aufgaben meist besser geeignet ist.

Insertsort

Insertsort, Sortieren durch Einfügen, ist ein relativ trivialer Algorithmus. Er wird bei Größenordnungen ab 100 Elementen gewiß von Quicksort übertroffen; wer es darauf anlegt, sollte aber zumindest bei kleinen Datenmengen einmal beide Verfahren gegeneinander arbeiten lassen. Je nach Vorsortierung der Daten und Größe der Elemente kann Insertsort unter Umständen seinem berühmten Kollegen den Rang ablaufen.

In einem Fall wäre es sicher Verschwendung, Quicksort zu benutzen: Wenn bereits ein sortiertes Array vorliegt und nur ein neues Element an der richtigen Stelle eingefügt werden muß – das ist in vielen Anwendungen der Fall – hat Insertsort die besten Karten. Der folgende Algorithmus nimmt das letzte Array-Element und sortiert es an die Stelle, an die es gehört. Es wird davon ausgegangen, daß alle anderen Elemente bereits sortiert vorliegen.

Um eine komplette Insertsort-Routine zu erhalten, müßten Sie hier nur noch eine Schleife mehr einbauen, so daß erst das zweite Element einsortiert wird, dann das dritte usw.

```
SUB InsertSort (Zeile() AS STRING, Anzahl AS INTEGER)

   DIM Vergleich AS STRING

   Vergleich = Zeile(Anzahl)
   FOR i% = Anzahl   1 TO LBOUND(Zeile) STEP  1
      IF Zeile(i%) < Vergleich THEN
         Zeile(i% + 1) = Vergleich
         EXIT SUB
      END IF
      Zeile(i% + 1) = Zeile (i%)
   NEXT
   Zeile(LBOUND(Zeile)) = Vergleich

END SUB
```

Listing 20–2: ISORT.BAS

Die in VBDOS eingebauten Listen- und Kombinationsfelder sortieren ihre internen Listen ebenfalls mit „Insertsort“, wenn die *Sorted*-Eigenschaft auf TRUE gesetzt ist; das bietet sich an, weil neue Elemente ja immer mit der ADDITEM-Methode hinzugefügt werden. Bei größeren Datenmengen (und wenn es auf die Zeit ankommt) kann man einige Sekunden sparen, wenn man die Daten vorher sortiert und sie dann in der richtigen Reihenfolge mit ADDITEM an die Liste anfügen läßt. Wichtig: Bei ADDITEM als zweiten Parameter die Position angeben, damit VBDOS gar nicht erst versucht, das Element einzusortieren.

Bucketsort

Bucketsort ist ein überaus primitiver Algorithmus, den man vielleicht gerade deswegen manchmal übersieht. Er kann nur sehr selten eingesetzt werden, aber dort, wo man ihn benutzen kann, ist er in puncto Geschwindigkeit allen anderen überlegen.

Bucketsort kann nur für Sortieraufgaben eingesetzt werden, bei denen die zu sortierenden Elemente als Array-Indizes benutzt werden können. Bucketsort kann also nur Zahlen (und mit einigen Abwandlungen auch vielleicht einzelne ASCII-Zeichen) sortieren, und zwar nur ganze Zahlen (also INTEGER), und auch diese nur, wenn die Zahlen alle in einem Bereich liegen, der vorher bekannt sein muß und nicht mehr als 32.767 Elemente enthalten darf. Bucketsort funktioniert so:

Wie der Name schon andeutet, verteilt Bucketsort alle Zahlen, die sortiert werden sollen, auf verschiedene „Eimer“, wobei für jede Zahl, die überhaupt vorkommen kann, ein Eimer (hier ein Arrayelement) benötigt wird. Wenn dieser Vorgang abgeschlossen ist, werden die Eimer der Reihe nach wieder geleert und in das Zahlen-Array geschrieben.

Dieser Bucketsort-Routine müssen das Array der zu sortierenden Zahlen (mit *Anzahl* Elementen, beginnend bei 1), ihre Anzahl und der kleinste und größte mögliche Wert übergeben werden. (Es macht nichts, wenn *Min* und *Max* zu groß gewählt werden; ihr Abstand darf aber nicht größer als 32.766 sein.)

```
SUB Bucketsort(Zahl() AS INTEGER, Anzahl AS INTEGER, Min AS INTEGER, Max AS INTEGER)

   DIM Eimer(Min TO Max) AS INTEGER, Zaehler AS INTEGER

   ' Zahlen auf Eimer verteilen
   FOR i% = 1 TO Anzahl
      Eimer(Zahl(i%)) = Eimer(Zahl(i%)) + 1
   NEXT

   Zaehler = Min
```

⇨

```
    ' Eimer der Reihe nach leeren
    FOR i% = 1 TO Anzahl
       DO UNTIL Eimer(Zaehler) > 0: Zaehler = Zaehler + 1: LOOP
       Zahl(i%) = Zaehler: Eimer(Zaehler) = Eimer(Zaehler)   1
    NEXT

    ' Eimer-Array-Speicherplatz freigeben
    ERASE Eimer

END SUB
```

Listing 20–3: BUCKET.BAS

Sortieren durch Mischen („Mergesort“)

Bisher habe ich drei Sortieralgorithmen bearbeitet: Den universellen QuickSort und zwei einfachere Algorithmen für Sonderfälle, Insertsort und Bucketsort.

Wie sortiert man Datenmengen, die so umfangreich sind, daß sie nicht in ein Array geladen werden können?

Eine einfache Möglichkeit besteht darin, eine der Array-Sortiermethoden – vorzugsweise Quicksort – derart umzuschreiben, daß sie nicht auf Elemente eines Arrays, sondern stattdessen auf bestimmte Felder einer Datei mit konstanter Satzlänge (OPEN FOR RANDOM) zugreift. Das ist eine Methode, die funktioniert – allerdings, je nach Datenträger, sehr langsam. Vor allem sind viele Daten, die sortiert werden müssen, nicht mit konstanter Satzlänge gespeichert – man müßte sie zuvor umformatieren.

Sortieren durch Mischen arbeitet mit sequentiellen Dateien (OPEN FOR INPUT). Die ursprüngliche Datei wird in einem ersten Schritt in *n* temporäre Dateien aufgeteilt, wobei nicht immer ein Element, sondern ein sogenannter Lauf von Elementen kopiert wird, das heißt, eine Gruppe von aufeinanderfolgenden Elementen, die in sich schon sortiert sind (die Zahlenfolge 3 56 2 76 78 89 22 45 2 4 9 besteht zum Beispiel aus den vier Läufen [3 56], [2 76 78 89], [22 45] und [2 4 9]). Danach wird der Inhalt der *n* temporären Dateien wieder in eine Datei zusammengemischt. Die hier entstehende Datei enthält wieder alle Elemente, allerdings nur noch etwa ein *n*-tel der Läufe, die die ursprüngliche Datei hatte. Es wird so lange fortgefahren, bis nur noch ein einziger Lauf vorhanden ist. Dann ist die Datei sortiert.

Das Verfahren hat den großen Vorteil, daß es „Teilsortierungen“ erkennt und zeitsparend berücksichtigt.

```
SUB MergeSort (FileName AS STRING)
   CONST False = 0, True =  1
   CONST ExtBuffer = 10, IntBuffer = 1024, TempFileCode = "MSORT.$"

   DIM AktuellesFile AS INTEGER, InputFile AS INTEGER, OutputFile AS INTEGER
   DIM TempFile(1 TO ExtBuffer) AS INTEGER, LaufZaehler AS INTEGER
   DIM LaufEnde(1 TO ExtBuffer) AS INTEGER, DateiEnde(1 TO ExtBuffer) AS INTEGER
   DIM Zeile AS STRING, LetzteZeile AS STRING
   DIM NaechsteZeile(1 TO ExtBuffer) AS STRING

   DO
      ' 1. Schritt: Der Inhalt des zu sortierenden Files wird auf alle temporären
      ' Files verteilt. Dabei wird immer ein ganzer Lauf kopiert.
      InputFile = FREEFILE
      OPEN FileName FOR INPUT AS #InputFile LEN = IntBuffer
      FOR i% = 1 TO ExtBuffer
         TempFile(i%) = FREEFILE
         OPEN TempFileCode + FORMAT$(i%) FOR OUTPUT AS #TempFile(i%) LEN = IntBuffer
      NEXT

      AktuellesFile = 1: LetzteZeile = ""
      DO UNTIL EOF(InputFile)
         LINE INPUT #InputFile, Zeile
         IF Zeile < LetzteZeile THEN
            ' Ein neuer Lauf beginnt. Wähle eine andere temporäre Datei:
            AktuellesFile = AktuellesFile + 1
            IF AktuellesFile > ExtBuffer THEN AktuellesFile = 1
         END IF
         LetzteZeile = Zeile: PRINT #TempFile(AktuellesFile), Zeile
      LOOP

      ' 2. Schritt: Das Kopieren ist beendet; jetzt werden die temporären Dateien
      ' wieder zusammengemischt. Dabei wird stets das kleinste von allen "wartenden"
      ' Elementen ausgewählt und in die Ergebnisdatei geschrieben. Es wird laufweise
      ' gemischt, das heißt, der erste Lauf aus der ersten temporären Datei wird
      ' nur mit den ersten Läufen der anderen temporären Dateien (und mit keinem
      ' zweiten Lauf) gemischt. Deshalb werden Dateien, bei denen der aktuelle Lauf
      ' schon zu Ende gelesen ist, bei der Elementauswahl nicht berücksichtigt.
      CLOSE InputFile: OPEN FileName FOR OUTPUT AS #InputFile LEN = IntBuffer
      FOR i% = 1 TO ExtBuffer
         CLOSE TempFile(i%)
         OPEN TempFileCode + FORMAT$(i%) FOR INPUT AS #TempFile(i%) LEN = IntBuffer
      NEXT

      ' Das Feld NaechsteZeile() wird mit der ersten Zeile jeder Datei gefüllt
      FOR i% = 1 TO ExtBuffer
         IF NOT EOF(TempFile(i%)) THEN
            LINE INPUT #TempFile(i%), NaechsteZeile(i%) ' ?pfeil1?
            LaufEnde(i%) = False: DateiEnde(i%) = False
         ELSE
            LaufEnde(i%) = True: DateiEnde(i%) = True
         END IF
      NEXT
```

```
' Auf geht's!
LaufZaehler = 0: NeuLauf = False
DO
   Kleinstes = 0
   DO
      FOR i% = 1 TO ExtBuffer
         IF NOT LaufEnde(i%) THEN
            IF Kleinstes = 0 THEN
               Kleinstes = i%
            ELSEIF NaechsteZeile(Kleinstes) > NaechsteZeile(i%) THEN
               Kleinstes = i%
            END IF
         END IF
      NEXT
      IF Kleinstes = 0 THEN
         IF NOT NeuLauf THEN
            ' Wenn kein kleinstes Element gefunden wird, weil alle Läufe zu
            ' Ende sind, wird in jeder Datei der nächste Lauf angebrochen.
            ' (Eine Datei, die schon völlig zu Ende ist, weil sie vielleicht
            ' einen Lauf weniger enthält als die anderen, soll weiter
            ' ignoriert werden.)
            FOR i% = 1 TO ExtBuffer: LaufEnde(i%) = DateiEnde(i%): NEXT
            NeuLauf = True
            LaufZaehler = LaufZaehler + 1
         ELSE
            ' Wenn unmittelbar nach einem Neuanfang (siehe obige Bemerkung)
            ' noch immer kein kleinstes Element gefunden wird, muß das daran
            ' liegen, daß alle Dateien zu Ende sind. Dann wird die Schleife
            ' verlassen, und das Mischen ist beendet.
            Kleinstes =  1
         END IF
      ELSE
         NeuLauf = False
      END IF
   LOOP WHILE Kleinstes = 0

   IF Kleinstes > 0 THEN
      LetzteZeile = NaechsteZeile(Kleinstes)
      PRINT #InputFile, LetzteZeile
      IF EOF(TempFile(Kleinstes)) THEN
         LaufEnde(Kleinstes) = True: DateiEnde(Kleinstes) = True
      ELSE
         LINE INPUT #TempFile(Kleinstes), NaechsteZeile(Kleinstes)
         IF NaechsteZeile(Kleinstes) < LetzteZeile THEN
            LaufEnde(Kleinstes) = True
         END IF
      END IF
   END IF
LOOP UNTIL Kleinstes =  1

CLOSE InputFile
```

```
      FOR i% = 1 TO ExtBuffer: CLOSE TempFile(i%): NEXT

      ' Ein Sortier- und ein Mischvorgang sind abgeschlossen. Beim Mischen wird ge-
      ' zählt, wieviele Läufe gemischt wurden. In dem Augenblick, in dem jede tempo-
      ' räre Datei nur noch einen (oder gar keinen) Lauf enthielt, fand das letzte
      ' Mischen statt; wurde hingegen mehr als ein Lauf gemischt, muß nun von neuem
      ' verteilt werden.
   LOOP UNTIL LaufZaehler = 1

   ' Löschen der temporären Dateien
   FOR i% = 1 TO ExtBuffer
      KILL TempFileCode + LTRIM$(STR$(i%))
   NEXT

END SUB
```

Listing 20–4: MSORT.BAS

Der abgedruckten Routine für das Sortieren durch Mischen muß nur der Dateiname der zu sortierenden Datei übergeben werden. Die Konstante *IntBuffer* legt fest, wie groß der Dateibuffer für sämtliche OPEN-Anweisungen sein soll. *ExtBuffer* gibt an, wieviele temporäre Dateien das Programm benutzen darf. Es müssen mindestens zwei sein. Zur maximalen Anzahl der gleichzeitig geöffneten Dateien siehe den Eintrag zu FREEFILE im Disketten-Referenzteil bzw. die Routine in Kapitel 22.

IntBuffer sollte zwischen 512 und 4096 liegen; andere Werte sind möglich (siehe Eintrag zu OPEN im Disketten-Referenzteil), aber zumeist weniger effizient. Der optimale Wert für *ExtBuffer* hängt von Anzahl und Art der sortierten Daten ab. Werte zwischen 5 und 10 ergeben im allgemeinen die besten Resultate. Zu hohe Werte können das Zeitverhalten verschlechtern. Die besten Werte für beide Buffer-Konstanten sollten am besten in der Praxis durch Ausprobieren ermittelt werden.

Die Konstante *TempFileCode* ist der Name für die temporären Dateien; der Algorithmus hängt eine Zahl von 1 bis *ExtBuffer* hinten an, achten Sie also darauf, daß sich danach noch gültige Dateinamen ergeben. Sie können aus *TempFileCode* auch eine Variable bilden und diese dann abhängig von der Betriebssystemvariablen TMP setzen, in die auf vielen Systemen ein Verzeichnis für temporäre Dateien eingetragen wird (zu Betriebssystemvariablen siehe den Eintrag ENVIRON im Disketten-Referenzteil).

Für den Einsatz in der Praxis müßte für die Routine noch ein Error-Handler geschrieben werden, da es während des Sortierens passieren könnte, daß die Festplatte (bzw. das Medium, auf dem die temporären Dateien erzeugt werden) voll wird.

20.2 Suchen

Binäres Suchen

Das binäre Suchen ist die effizienteste Suchmethode für Arrays. Es setzt voraus, daß das zu durchsuchende Array sortiert ist. Es benötigt nur maximal $\log_2 n$ Vergleichsschritte bei *n* Elementen, könnte also aus einer Liste aller Einwohner der Bundesrepublik mit höchstens 27 Vergleichen einen bestimmten Namen heraussuchen.

Das binäre Suchen funktioniert ähnlich wie der Quicksort-Algorithmus. Auch hier wird der zu durchsuchende Bereich so lange eingeschränkt, bis er nur noch ein Element enthält.

Diese Version benötigt als Argumente das zu durchsuchende Array, die Nummer des höchsten Elements und den String, nach dem gesucht werden soll. Als Funktionswert wird die Nummer des gefundenen Elements zurückgegeben oder *Anzahl + 1*, wenn keines gefunden wurde.

```
FUNCTION SuchBin (Such AS STRING, Zeile() AS STRING, Anzahl AS INTEGER) AS INTEGER

   DIM Links AS INTEGER, Rechts AS INTEGER, Mitte AS INTEGER

   Links = LBOUND(Zeile): Rechts = Anzahl
   DO
      Mitte = (Links + Rechts) \ 2
      IF Zeile(Mitte) < Such THEN
         Links = Mitte + 1
      ELSE
         Rechts = Mitte
      END IF
   LOOP UNTIL Links = Rechts
   ' Hier eventuell Zeilen einfügen   siehe Text
   IF Zeile(Links) = Such THEN
      SuchBin% = Links
   ELSE
      SuchBin% = Anzahl + 1
   END IF

END FUNCTION
```

Listing 20–5: SUCHBIN.BAS

Zu beachten ist, daß das binäre Suchen in der vorliegenden Form bei mehreren passenden Array-Einträgen *irgendeinen* davon finden würde, das heißt, es ist ungewiß, ob der erste, der zweite oder der dritte von drei Meiers in der Adreßdatei gefunden würde. Für solche Anwendungen, bei denen der Suchschlüssel nicht eindeutig ist, müßten an der markierten Stelle noch die Zeilen

```
   DO UNTIL Links = LBOUND(Zeile)
      IF Zeile(Links   1) = Zeile(Links) THEN
         Links = Links   1
      ELSE
         EXIT DO
      END IF
   LOOP
```

(Zusatz zu SUCHBIN.BAS) SUCHBINZ.BAS

eingefügt werden, damit stets die Nummer des *ersten* passenden Schlüssels zurückgegeben wird.

Das binäre Suchen kann natürlich auch für das Auffinden eines Datensatzes in einer sortierten RANDOM-Datei benutzt werden.

Binäres Suchen in sequentiellen Dateien

Da das binäre Suchen ein sehr effizienter Algorithmus ist, eignet er sich gut, um große Datenmengen zu durchsuchen. Häufig ist es aber schwierig, große Datenmengen als Random-Access-Datei zu speichern, weil durch die dafür erforderliche konstante Satzlänge viel Platz verschwendet wird. Ein Sprachwörterbuch zum Beispiel, das Wörter vom Englischen ins Deutsche übersetzen soll, kann es sich nicht leisten, für jedes englische und deutsche Wort die maximal denkbare Länge an Bytes zu reservieren.

Für solche und ähnliche Fälle bietet es sich an, eine gewöhnliche ASCII-Datei zu nehmen und in jede Zeile ein englisches Wort gefolgt von seiner deutschen Übersetzung einzutragen, mit einem beliebigen Trennzeichen dazwischen.

Diese Datei wird nun (mit Mergesort) sortiert, und danach kann mit einer Spezialversion des binären Suchens auf sie zugegriffen werden.

```
STATIC FUNCTION Such (File AS INTEGER, Gesucht AS STRING) AS STRING

   CONST MaxZeilenLaenge = 80, BufferGroesse = MaxZeilenLaenge * 2 + 2

   DIM Buffer AS STRING * BufferGroesse, Zeile AS STRING, Pruefen AS STRING
   DIM Mitte AS LONG, Links AS LONG, Rechts AS LONG,
   DIM Anfang AS INTEGER, Ende AS INTEGER

   Rechts = LOF(File): Links = 1
   Such = ""

   DO
      Mitte = (Links + Rechts) \ 2
      GET #File, Mitte, Buffer
      IF Mitte = 1 THEN
         Anfang = 1
      ELSE
```

```
         ' Vor dem Anfang einer neuen Zeile steht immer die Kombination CHR$(13) +
         ' CHR$(10), also fängt die nächste Zeile hinter dem nächsten CHR$(10) an:
         Anfang = INSTR(Buffer, CHR$(10)) + 1
      END IF
      Pruefen = MID$(Buffer, Anfang, LEN(Gesucht))

      IF Pruefen > Gesucht THEN
         Rechts = Mitte   1
      ELSEIF Pruefen < Gesucht THEN
         Links = Mitte + 1
      ELSE
         Ende = INSTR(Anfang, Buffer, CHR$(13))
         IF Ende = 0 THEN Ende = LEN(Buffer) + 1
         Such$ = MID$(Buffer, Anfang, Ende   Anfang)
         EXIT DO
      END IF
   LOOP UNTIL Links = Rechts

END SUB
```

Listing 20–6: SUCHBIND.BAS

Dieser Algorithmus benötigt eine bereits geöffnete Datei mit den sortierten Zeilen; die Dateinummer wird ihm mitgeteilt. Er gibt auch nicht, wie das binäre Suchen für Arrays, eine Nummer, sondern gleich die gefundene Zeile zurück. (Eine Nummer würde nichts nützen, da man ja in einer sequentiellen Datei nie auf einen bestimmten Datensatz direkt zugreifen kann.) Dabei wird eine Zeile gesucht, die mit dem angegebenen Suchtext (*Gesucht*) beginnt. Im Beispiel unseres Wörterbuchs müßte man also das englische Wort gefolgt von dem Trennzeichen angeben, und zurück käme die vollständige Zeile mit englischem Wort, Trennzeichen und der Übersetzung.

Die Funktionsweise ist ähnlich der des binären Suchens für Arrays, nur wird hier nicht eine Zeile gezielt herausgenommen, sondern einfach eine bestimmte Anzahl von Bytes ab einer bestimmten Stelle, und dann wird aus dieser Anzahl eine Zeile „herausgeschält“. Das ist möglich, weil in einer ASCII-Datei alle Zeilen durch die Kombination `CHR$(13) + CHR$(10)`, also Carriage Return und Linefeed, getrennt sind. Um sicherzugehen, daß in jedem Falle im ausgewählten Bereich eine komplette Zeile enthalten ist, muß er um zwei Bytes länger sein als das Doppelte der maximalen Zeilenlänge. Damit das Programm damit arbeiten kann, müssen Sie die maximale Zeilenlänge in die Konstante *MaxZeilenLaenge* eintragen. Je größer diese ist, desto langsamer arbeitet der Algorithmus – allerdings ist er von Natur aus so fix, daß nicht viel schiefgehen kann.

Phonetische Suche

Völlig aus dem Rahmen der gewöhnlichen Suchalgorithmen fällt ein Problem, das in der Praxis nicht selten ist: Die Suche nach Strings, die „so ähnlich“ sind wie ein eingegebener Referenz-String. Damit lassen sich zuweilen verblüffende Effekte erzielen. Ein Online-Rechercheprogramm, mit dem ich vor gut 5 Jahren gearbeitet habe (in einer Zeit also, in der „Künstliche Intelligenz“ und WINDOWS noch nicht Mode waren), reagierte auf Fehleingaben stets höflich mit der Meldung „Befehl unbekannt. Meinen Sie vielleicht ....?“.

Aber auch für die Suche in Listen ist es sicherlich sinnvoll, dem unvollkommenen Menschen, der das Programm benutzt, Hilfsmittel an die Hand zu geben, die seinem Erinnerungsvermögen besser angepaßt sind als ein Suchalgorithmus, dem man den gesuchten Namen *ganz genau* mitteilen muß.

Hierzu bedient man sich sogenannter „phonetischer“ Suchalgorithmen, obwohl diese Bezeichnung nicht ganz treffend ist, da hier ja nicht nach dem Klang, sondern nur nach der Schreibweise eines Wortes gesucht wird.

Man läßt den Benutzer den Suchbegriff eingeben und ermittelt dann für jedes Wort in der Datenbank einen Ähnlichkeitswert, an dem man erkennt, ob die Wörter „fast gleich“ oder „sehr verschieden“ sind (der Zeitgeist würde das heute vielleicht „Fuzzy Logic“ nennen – die Algorithmen sind aber wesentlich älter als der Begriff). Der oder die Suchschlüssel, die dem eingegebenen Wort am ähnlichsten sind, werden ausgesucht. Da man jeden Begriff in der Liste mit dem eingegebenen Suchbegriff vergleichen muß, ist die phonetische Suche natürlich sehr viel langsamer als ein Suchverfahren, das den exakten Suchschlüssel findet.

Es gibt verschiedene Methoden zur Ermittlung des Ähnlichkeitswertes zweier Worte. Weit verbreitet ist die sogenannte „Levenshtein-Distanz“. Sie gibt an, wieviele Veränderungen man an dem Vergleichswort mindestens vornehmen muß, um den Suchschlüssel zu erhalten. Als „Veränderungen“ sind dabei das Einfügen und Löschen eines Buchstabens und das Ersetzen eines Buchstabens durch einen anderen zugelassen.

Die Levenshtein-Distanz von „VISUAL BASIC FÜR DOS“ und „VBDOS“ ist zum Beispiel 15; „Levenshtein“ und „Lefstein“ haben nur eine Distanz von 4.

Einen Schritt weiter geht noch die „gewichtete Levenshtein-Distanz“, bei der es möglich ist, den drei Operationen verschiedene Gewichte zu verleihen, so daß z. B. eine Ersetzung so schwer wiegt wie zwei oder drei Einfügungen. Das kommt der Realität etwas näher, da man öfter in der Erinnerung Buchstaben hinzuphantasiert oder vergißt, seltener aber sich an falsche Buchstaben erinnert.

Die Realisation der gewichteten Levenshtein-Distanz in BASIC ist theoretisch (mit einer rekursiven Prozedur) kein Problem. In der Praxis erweist sich die allerdings als ziemlich langsam, und da man bei vielen Anwendungen gleiche mehrere hundert Distanzen nacheinander berechnen wird, ist das ein schweres Handicap. Wie so oft hilft hier die Formulierung als iterativer Algorithmus weiter – eine gebrauchsfertige Version der „Levenshtein-Distanz“ mit einigen Anmerkungen finden Sie unter LEVENSTN.BAS auf der Diskette.

20.3 Veränderungen an den Sortier- und Suchalgorithmen

Datentypen

Bei allen abgedruckten Algorithmen (bis auf Bucketsort) wird der Datentyp STRING gesucht bzw. sortiert. Natürlich kann auch jeder andere Datentyp benutzt werden; man muß nur einige Deklarationen ändern. Bei selbstdefinierten Typen muß man außerdem die Vergleiche so abändern, daß nicht die ganze Variable, sondern nur der relevante Schlüssel verglichen wird (statt `IF Zeile(a) < Zeile(b)` müßte es dann `IF Element(a).Name < Element(b).Name` heißen o. ä).

Sortierfolge und Indexlisten

Der Kern all dieser Algorithmen (bis auf Bucketsort...) sind die Vergleichsoperatoren <, = und >. Durch Änderungen an den Vergleichsstellen können Sie zum Beispiel erreichen, daß ein Array ab- und nicht aufsteigend sortiert wird. Für das binäre Suchen wären bei einer solchen Änderung allerdings umfangreichere Anpassungen nötig.

Wenn Sie beispielsweise Namen sortieren, würden alle unsere Sortieralgorithmen nicht gerade Telefonbuch-konform arbeiten: Umlaute stehen am Ende des Alphabets, ein kleines A kommt erst hinter dem großen Z usw. – die typische ASCII-Sortierung eben. Um das zu umgehen, haben Sie zwei Möglichkeiten.

Die erste: Sie fügen jedem zu sortierenden Element noch einen Sortierschlüssel bei, in dem nur Großbuchstaben vorkommen und Umlaute als AE, OE, UE und SS umschrieben sind, und lassen dann basierend auf diesem Schlüssel sortieren und suchen. Das geht fast genauso schnell wie ein normales Sortieren, verbraucht aber mehr Speicher. Wenn Sie ohnehin Datensätze mit einer Länge von 500 Bytes sortieren, werden Ihnen die zusätzlichen paar Bytes nichts ausmachen; handelt es sich aber um kurze Sätze mit vielleicht nur 30 Bytes, braucht diese Methode fast doppelt so viel Speicher.

Noch besser ist es sogar, die Sortierschlüssel in einem separaten Array bzw. in einer separaten Datei anzulegen, zusammen mit einer Zahl, die auf ein bestimmtes Element des eigentlichen Datenbestandes verweist. Dann muß der Datenbestand selbst überhaupt nicht sortiert werden, sondern es reicht, die Indexdatei bzw. das Indexfeld zu sortieren. Das geht noch schneller, weil dann beim Sortieren nur der Sortierschlüssel selbst hin- und herkopiert wird und nicht der ganze Datensatz. Im Lieferumfang der professionellen Ausgabe ist ein Programm namens QSORT.BAS enthalten, das mit Quicksort und Indexlisten arbeitet.

Die zweite Möglichkeit: Sie ersetzen die Vergleiche in den Sortier- und Suchalgorithmen durch Funktionsaufrufe. Statt `IF Zeile(a) < Zeile(b)` schreiben Sie nun `IF Kleiner(Zeile(a), Zeile(b))` und definieren eine Funktion `Kleiner AS INTEGER`, die dann und nur dann den Wert TRUE annimmt, wenn das erste Argument nach Telefonbuch-Sortierung kleiner als das zweite ist. Diese Routine müßte dazu dann bei jedem Aufruf die übergebenen Strings umwandeln, was eine Unmenge an Zeit verbraucht. Trotzdem kann man die Routine geschickt programmieren, so daß sie zum Beispiel nicht erst beide Strings umwandelt, sondern nur buchstabenweise arbeitet und den Vergleich abbricht, sobald das Ergebnis feststeht – im Durchschnitt muß dann nur ein Bruchteil der Zeichen wirklich umgewandelt werden. Oder, das Einfachste und wohl auch Schnellste: Sie nehmen einfach die ISAM-Routine `TEXTCOMP` (nur in der professionellen Version, siehe ISAM-Referenzteil). Aber seien Sie gewarnt: Wenn Sie die ISAM-Routinen nicht im Runtime-Modul, sondern im EXE-Programm selbst haben wollen (BC PROGRAMM /O), und selbst wenn Sie außer der `TEXTCOMP`-Funktion keinen einzigen ISAM-Befehl benutzen, zahlen Sie für diese Bequemlichkeit mit etwa 90 KB zusätzlichen EXE-Speicherplatzes. Denn genau so lang sind alle ISAM-Routinen zusammen, und ISAM wird vom Compiler nur komplett angeboten: Entweder Sie nehmen alles, oder Sie verzichten.

20.4 Rekursive Programmierung

Rekursive Programmierung erfordert in besonderem Maße analytisches Denken. Bei wirklich rekursiven Problemen liegt der Löwenanteil der Arbeit in der Analyse des Problems und nicht im Programmieren.

Als Beispiel für rekursive Programmierung wird immer wieder gern die Fakultätsfunktion genommen. Die Fakultät n! läßt sich wie folgt definieren:

```
n! = n·(n 1)·(n 2)·(n 3) ... ·3·2·1
```

Es ist kein Problem, eine BASIC-Funktion zu programmieren, die mittels einer FOR...NEXT-Schleife die Fakultät einer beliebigen Zahl ausrechnet. Das soll aber jetzt nicht unser Ziel sein. Die Fakultät besitzt nämlich noch eine andere Definition:

$n! = 1$ für $n = 1$ und $n! = n \cdot (n-1)!$ für $n > 1$

Diese Definition führt zu einer *rekursiven* Funktion:

```
FUNCTION Fakultaet (Zahl AS INTEGER) AS DOUBLE

   IF Zahl = 1 THEN
      Fakultaet# = 1
   ELSE
      Fakultaet# = Fakultaet#(Zahl   1) * Zahl
   END IF

END FUNCTION
```

Listing 20–7: FAKULT.BAS

Diese Funktion ruft sich selbst auf, um die Fakultät der Vorgängerzahl zu ermitteln und multipliziert das Ganze dann mit dem Argument *Zahl*.

In diesem Beispiel fällt es gar nicht auf, aber bei umfangreicheren Problemen kann die rekursive Programmierung sehr viel Programmierarbeit ersparen. Die Prozedur DirectoryRek aus Kapitel 22 ruft sich zum Beispiel selbst auf, um Dateilisten von Unterverzeichnissen anzulegen.

Rekursive Algorithmen bieten sich häufig dann an, wenn das Gesamtproblem sich in gleichartige Teilprobleme aufteilen läßt. Es reicht dann, eines dieser Teilprobleme sozusagen exemplarisch zu lösen, sich eine geeignete Verkettung auszudenken, und schon ist das Gesamtproblem gelöst. Die Mächtigkeit der Rekursion tritt noch deutlicher hervor, wenn ich nun etwas diffizilere Probleme per Backtracking löse.

Backtracking

Es gibt Probleme, die man nicht einfach mit einer simplen Formel oder Schleife lösen kann. Als Beispiel will ich vereinfacht eine Aufgabe nennen, die im Rahmen des Bundeswettbewerbs Informatik einmal gestellt wurde: Eine Schatztruhe enthält eine Anzahl von Münzen mit verschiedenen, ganzzahligen Werten. Deren Gesamtwert sei eine gerade Zahl. Diese Münzen sollen in zwei Truhen so verteilt werden, daß beide Truhen den gleichen Gesamtwert (nämlich jeweils die Hälfte des ursprünglichen Gesamtwerts) enthalten.

Es ist prinzipiell unklar, ob es *überhaupt* eine Lösung für das Problem gibt. Es wäre ja denkbar, daß die Münzen sich gar nicht „gerecht“ verteilen lassen. Solche Aufgaben werden häufig mit einem sogenannten Backtracking-Algorithmus gelöst, einem „trial and error“-Verfahren. Das folgende Schaubild erläutert die Funktion eines typischen Backtracking-Algorithmus:

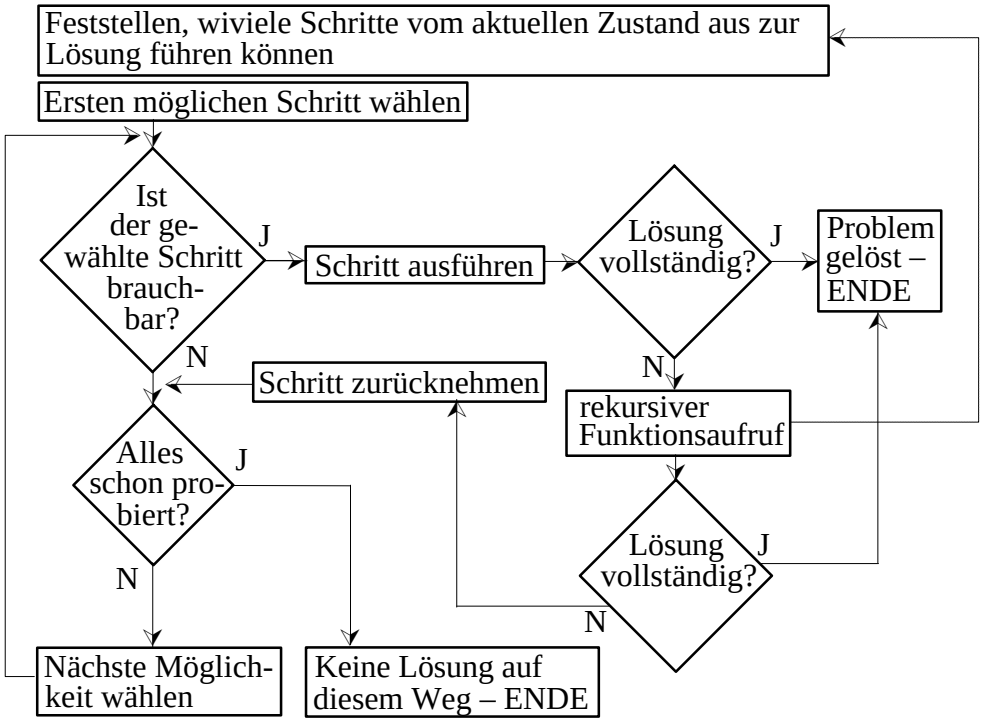


Abbildung 20–1: Ein typischer Backtracking-Grundriß

Die Funktion stellt zunächst fest, wieviele Schritte in Richtung der Lösung gemacht werden können (im Beispiel: Wie viele Münzen genommen werden könnten, sich also noch in der Ursprungskiste befinden). Dabei ist es wichtig, so viele Schritte wie möglich jetzt schon auszuschließen; ich komme später darauf zurück.

Dann wird der erste der möglichen Schritte ausgeführt. Wenn danach die angestrebte Gesamtlösung erreicht ist, kann das Programm beendet werden (oder die Prozedur wird verlassen). Wenn nicht, ruft die Funktion sich nun selbst auf, um an der durch den eben vollzogenen Schritt veränderten Position weiterzuarbeiten. Ist nach der Rückkehr aus dem rekursiven Aufruf die Gesamtlösung erreicht, kann das Programm beendet oder die Prozedur verlassen werden. Ist das nicht der Fall, dann weiß man jetzt, daß der vollzogene Schritt zumindest an dieser Stelle zu keiner Lösung führt. Er wird rückgängig gemacht, und der nächste wird ausprobiert, so lange, bis alle Schritte durchprobiert sind. Danach wird die Prozedur gegebenenfalls erfolglos verlassen.

Niklaus Wirth definiert in seinem Standardwerk „Algorithmen und Datenstrukturen“*: „Das charakteristische Merkmal [eines Backtracking-Algorithmus] ist folgendes: Man versucht Schritte in Richtung Ziel und zeichnet sie auf. Stellt sich später heraus, daß sie in eine Sackgasse führen, so macht man sie wieder rückgängig und löscht die Aufzeichnungen.“

In unserem Beispiel mit der Schatzkiste sähe die Lösung etwa so aus:

```
SUB NimmMuenze(Erfolg AS INTEGER)

   DIM TochterErfolg AS INTEGER
   SHARED SollWert AS INTEGER, SummeGenommenerMuenzen AS INTEGER
   Erfolg = TRUE

   ' Ist das Problem gelöst?
   IF SummeGenommenerMuenzen = SollWert THEN EXIT FUNCTION

   ' gibt es überhaupt eine Möglichkeit?
   IF SummeGenommenerMuenzen < SollWert THEN
      DO UNTIL AlleMuenzenAusprobiert
         ' eine noch nicht probierte und noch nicht genommene Münze als genommen
         ' markieren und SummeGenommenerMuenzen erhöhen (hier nicht weiter
         ' ausgeführt)
         ' Lösung des Problems nun an die Tochterprozedur delegieren
         NimmMuenze TochterErfolg

         IF TochterErfolg = TRUE THEN
            ' Prozedur mit Erfolg verlassen (zum Finden aller Lösungen Prozedur hier
            ' nicht verlassen, sondern Münze wieder zurücklegen, als ob es keine
            ' Lösung gegeben hätte, und vorher die Lösung vermerken)
            EXIT FUNCTION
         ELSE
            ' die oben markierte Münze wieder "zurücklegen" (also als nicht genommen
            ' markieren und SummeGenommenerMuenzen wieder um deren Wert verringern)
            ' (hier nicht weiter ausgeführt)
         END IF
      LOOP
   END IF
   ' wenn die Prozedur bis jetzt noch nicht verlassen
   ' wurde, gibt es wohl keine Lösung mehr:
   NimmMuenze = FALSE

END FUNCTION
```

Listing 20–8

* Niklaus Wirth: „Algorithmen und Datenstrukturen“, Pascal-Version, Stuttgart 1975; ein unbedingt empfehlenswertes Buch, wenn Sie sich tiefergehend mit Such- und Sortieralgorithmen beschäftigen möchten. Leider benutzt ein großer Teil der in diesem auf PASCAL zugeschnittenen Buch abgedruckten Algorithmen Zeiger, einen Datentyp, der in BASIC nicht zur Verfügung steht.

Das Programm ist in „Pseudo-Code“ geschrieben, also nicht lauffähig (und daher auch nicht auf der Diskette enthalten). Es nimmt so lange Münzen aus der Kiste, bis der Gesamtwert der genommenen Münzen genau die Hälfte des ursprünglichen Gesamtwerts erreicht hat. Die Routinen zum „Nehmen“ und „Zurücklegen“ sind hier, ebenso wie die globalen Datenstrukturen, nicht weiter ausgeführt.

Zuweilen sind Backtracking-Verfahren die einzige Möglichkeit, ein Problem zu lösen. Aber auch diese Medaille hat eine Kehrseite. Angenommen, ich lasse den Computer das Münzen-Beispiel rechnen, und angenommen, ich bin so tückisch, ihm 40 verschiedene Münzen so vorzugeben, daß es keine Lösung gibt. Angenommen ferner, daß der Computer in einer Sekunde 1.000 Münzen nehmen und wieder zurücklegen kann (womit ich seine Leistung wohl schon etwas überbewerte).

Wissen Sie, wie lange er dann brauchen würde, um festzustellen, daß es keine Lösung gibt? Über den Daumen gepeilt (ich nehme an, daß im Durchschnitt 20 Münzen kombiniert werden), wird der Rechner nach etwa 278 Milliarden Jahren seinem greisen Operator das Ergebnis mitteilen. Sie verstehen, was ich sagen will: Man sollte Vorsicht walten lassen. Sie können sich den Aufruf einer solchen rekursiven Funktion oder Prozedur vorstellen wie die Wurzel eines Baumes. Vom Ausgangspunkt wird für jeden möglichen Zug eine neue Funktion aufgerufen, die ihrerseits wieder weitere Funktionen aufruft usw.

Der Trick, um 278-Milliarden-Jahre-Kaffeepausen zu vermeiden, besteht darin, daß man diese Verästelungen der Wurzel schon möglichst weit oben abschneidet, daß man schon vorher weiß, ob es Sinn hat, diesen Pfad weiter zu beschreiten.

Konkret werden solche Tests an der Stelle eingebaut, an der geprüft wird, ob es möglich ist, einen weiteren (anderen) Zug zu machen. Im Münzen-Programm wäre das zum Beispiel: Wenn es zwei Münzen mit dem Wert 5 gibt und das Nehmen der einen nicht zum Erfolg geführt hat, wird das Nehmen der anderen es auch nicht tun. Sie schmunzeln? Diese IF-Abfrage spart einige Millionen, wenn nicht Milliarden Jahre!

Diese Abfrage ist in der Tat der Knackpunkt, der über die Güte eines Backtracking-Algorithmus entscheidet. Man kann sie in unserem Programm zum Beispiel auf mehrere Stufen ausdehnen. Wenn ich eine 2 und danach eine 3 genommen habe und damit keinen Erfolg hatte, kann ich es mir sparen, eine 3 und danach eine 2 zu probieren usw.

Es gilt, genau abzuwägen, wie kompliziert die Abfrage sein kann – denn zu komplizierte Berechnungen an dieser Stelle könnten sich wieder negativ auf das Zeitverhalten auswirken. Häufig kann man es sich auch leisten, ein Limit zu set-

zen, die Rekursion also zum Beispiel nach einer bestimmten Zeit oder von einer bestimmten Rekursionstiefe an nicht mehr weiterzuverfolgen.

Wenn Sie rekursiv arbeiten, versuchen Sie immer abzuschätzen, wie lange die Prozedur im schlimmsten Falle arbeiten wird, und bauen Sie nötigenfalls Möglichkeiten zum Abbruch der Prozedur nach einer gewissen Zeit ein.

Der oben dargestellte Algorithmus bricht sofort ab, wenn er eine Lösung gefunden hat. Manchmal ist es aber wünschenswert, *alle* möglichen Lösungen einer Aufgabe zu erhalten. Man muß dann einfach, anstatt die Funktion sofort zu verlassen, wenn ein rekursiv gelaufener Prozeß „Erfolg“ meldet, die gefundene Lösung festhalten (in einem Array; in einer Datei) und so fortfahren, als hätte es keine Lösung gegeben.

Rekursive und nicht-rekursive Algorithmen

Es gibt keinen rekursiven Algorithmus, der sich nicht auch nicht-rekursiv programmieren ließe.

Der am Anfang dieses Kapitels abgedruckte Quicksort-Algorithmus oder das binäre Suchen sind dem Problem nach eher rekursive Algorithmen. In beiden Fällen wurde hier jedoch die nicht-rekursive Variante gewählt, die (zumindest bei Quicksort) sehr viel komplizierter aussieht und auch meist schwerer zu verstehen ist als die rekursive.

Funktionen, die sich sehr oft selbst aufrufen (eine lange Rekursionskette bilden), belasten den Stack-Speicher im DGROUP-Segment sehr, denn für jeden Aufruf müssen alle lokalen Variablen der Prozedur sowie einige weitere Informationen dort abgespeichert werden.

Formuliert man einen solchen Algorithmus nicht-rekursiv, so gelingt es meist, den Speicherverbrauch zu reduzieren, wobei man ein Array benutzt, das von BASIC ja sogar im Far-Speicher abgelegt werden kann. Dadurch sind nicht-rekursive Algorithmen meist ein wenig sparsamer als ihre rekursiven Pendants.

Sie sollten jedoch nie versuchen, ein der Art nach rekursives Problem nicht-rekursiv anzugehen. Vielmehr sollten Sie zunächst einen funktionierenden rekursiven Algorithmus programmieren und diesen dann, wenn Sie hoffen, ihn durch nicht-rekursive Programmierung verbessern zu können, umformulieren. Dadurch haben Sie auch bessere Möglichkeiten der Fehlersuche; zu leicht wird ein nicht-rekursiver Algorithmus zum gefürchteten „Spaghetti-Code“.

Gebrauchsfertige Formen 21

21.1 Die „Common Dialogues“

Beiden Versionen von VBDOS liegen die CMNDLG-Dateien (CMNDLG.BAS, CMNDLGF.FRM, CMNDLG.BI, CMNDLG.QLB, CMNDLG.LIB und in der Profi-Ausgabe CMNDLGA.LIB) bei. Sie enthalten einfache Routinen für Standard-Dialoge, die in vielen Programmen benötigt werden.

Um die CMNDLG-Form zu verwenden, können Sie entweder die Dateien CMNDLG.BAS und CMNDLGF.FRM zu Ihrem Projekt hinzufügen, oder Sie starten VBDOS mit der Quick Library CMNDLG (VBDOS /L CMNDLG). In beiden Fällen müssen Sie in Ihrem Programm die Include-Datei CMNDLG.BI per $INCLUDE-Befehl laden, um Zugriff auf die Prozedurdeklarationen zu haben.

Bei den „Common Dialogues“ handelt es sich um Formen (genauer: um eine einzige Form, aber das ist hier nicht so wichtig), die durch einen gewöhnlichen Prozeduraufruf aktiviert und gebunden angezeigt werden. Die Programmausführung hinter dem Prozeduraufruf wird erst nach Schließen der Form fortgesetzt.

Dies ist ein klassisches Beispiel dafür, wie Sie fertige Programme nachträglich durch einfache Prozeduraufrufe mit Formen „bestücken“ können. Eine andere Möglichkeit wurde im Kapitel 7 gezeigt; durch die Verwendung ungebundener Formen lassen Sie dem Benutzer die Möglichkeit, während eine Form angezeigt wird, mit einer anderen Form des Programms weiterzuarbeiten.

Das Verfahren, das hier angewandt wird, eignet sich am besten für Dialogboxen, bei denen ein sinnvolles Weiterarbeiten ohne Beenden der Dialogbox nicht sinnvoll ist. Bei „Datei Laden“ oder „Datei speichern“ ist das meistens der Fall; Formen wie „Farbeinstellung“ sind da schon fragwürdiger und würden sich eher als ungebundene Form eignen. Das erfordert allerdings eine stärkere Anpassung des Programms als die hier verwendete Methode.

Es empfiehlt sich, vor einer ernsthaften Verwendung der CMNDLG-Routinen den Modulcode aus CMNDLG.BAS zu entfernen, der für eine nette Demonstration beim Programmaufruf sorgt. Dieser würde spätere EXE-Programme nur unnötig belasten, da er dort nicht verwendet wird. Außerdem ist es möglich, die CMNDLG.QLB-Library mit den Switches /G2 oder /G3 neu zu erstellen, wenn man mit einem 286- oder 386-Rechner arbeitet. Die Quick Library wird nur von Ihnen verwendet und muß nur für Ihr System optimiert sein; wenn Sie jedoch auch die Objectcode-Library CMNDLG.LIB mit den auf 286er oder 386er ab-

gestimmten Dateien neu erzeugen, werden die von Ihnen erzeugten EXE-Programme nur auf solchen (oder höheren) Rechnern laufen.

Zur Neuerstellung der Library ist wie folgt vorzugehen:

```
BC CMNDLG /X;
BC CMNDLGF /X;
LIB CMNDLG.LIB -+CMNDLG-+CMNDLGF;
LINK /Q CMNDLG+CMNDLGF,,,VBDOSQLB;
```

(Evtl. an die BC-Aufrufe den Switch /G2 oder /G3 anhängen; für die Alternate Math-Library der professionellen Version zusätzlich die ersten beiden Zeilen mit /FPa und die dritte mit CMNDLGA.LIB statt CMNDLG.LIB ausführen.)

Registrieren

Die Routinen der CMNDLG-Form laufen schneller, wenn Sie vor dem ersten Aufruf den Befehl `CmnDlgRegister Ergebnis%` ausführen, der die Form schon einmal lädt, ohne sie jedoch anzuzeigen. In *Ergebnis%* wird zurückgegeben, ob das Laden geklappt hat (TRUE = ok, FALSE = Fehler). Verwenden Sie diesen Befehl, der übrigens jederzeit mit `CmnDlgClose` rückgängig gemacht werden kann, nicht, dann wird die Form bei jedem Prozeduraufruf geladen und danach aus dem Speicher entfernt. Das benötigt mehr Zeit und weniger Speicher.

Datei drucken

Die Dialogbox der Routine FilePrint ist dafür gedacht, den Benutzer das Ziel eines Druckvorgangs bestimmen zu lassen.

Sie hat folgende Parameter (in dieser Reihenfolge):

Parameter	*Datentyp*	*Inhalt*
Kopien	INTEGER	Die vom Benutzer eingegebene Kopienzahl. Sie können einen Wert vorgeben.
ForeColor, BackColor	INTEGER	Die Vorder- und Hintergrundfarbe für die Dialogbox.
Abbruch	INTEGER	wird als TRUE zurückgegeben, wenn der Benutzer das Dialogfeld abgebrochen hat.

Das Ergebnis, also die Auswahl des Benutzers, wird sofort in die *PrintTarget*-Eigenschaft des Spezialobjekts PRINTER eingetragen.

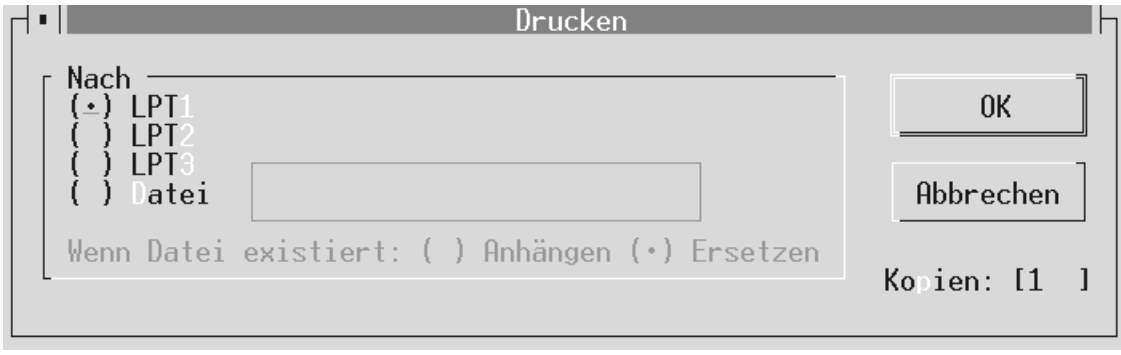


Abbildung 21–1: Die „Drucken“-Dialogbox aus CMNDLG

Datei laden und speichern

Die Routinen FileOpen und FileSave ermöglichen es dem Benutzer, einen beliebigen Dateinamen auszuwählen. Bei FileOpen können nur Namen von existierenden Dateien gewählt werden, bei FileSave ist auch die Eingabe eines neuen Namens möglich.

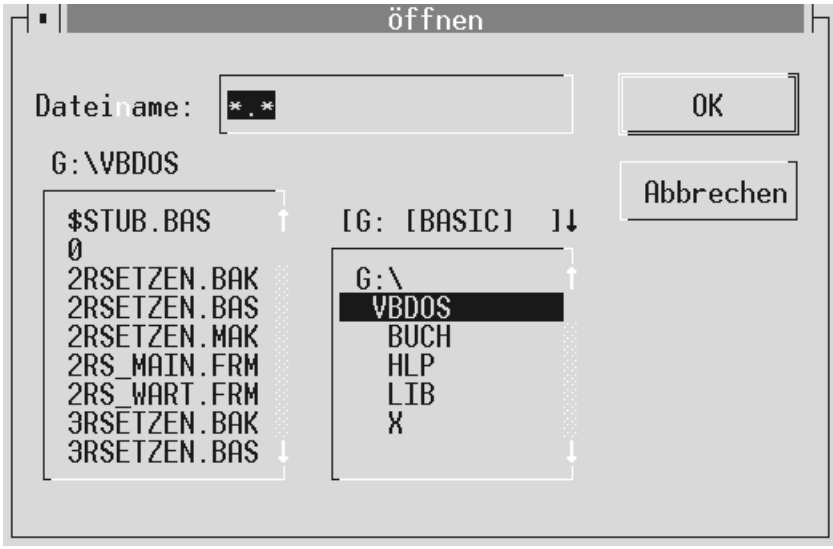


Abbildung 21–2: Die „Datei Öffnen“-Dialogbox aus CMNDLG

Beide Routinen verwenden folgende Parameter (in dieser Reihenfolge):

Parameter	*Datentyp*	*Inhalt*
Dateiname	STRING	Der Name der Datei; übergeben Sie einen leeren String oder eine Vorgabe. Die Prozedur gibt den gewählten Namen zurück.
PfadName	STRING	Laufwerk und Pfad der Datei. Übergeben Sie einen Leerstring oder eine Vorgabe; die Prozedur gibt den Pfad der gewählten Datei zurück.

Parameter	*Datentyp*	*Inhalt*
Maske	STRING	Die Dateimaske, zu der anfangs die Dateien angezeigt werden (z. B. *.TXT). Der Benutzer kann sie jedoch ändern.
Ueberschrift	STRING	Die Überschrift der Dialogbox. Geben Sie keine an, wird „Öffnen“ bzw. „Speichern“ gewählt.
ForeColor, BackColor	INTEGER	Die Vorder- und Hintergrundfarbe für die Dialogbox.
Flags	INTEGER	Nicht genutzt (für eigene Erweiterungen vorgesehen).
Abbruch	INTEGER	wird als TRUE zurückgegben, wenn der Benutzer das Dialogfeld abgebrochen hat

Farbeinstellung

Die Dialogbox zur Farbeinstellung wird mit der Prozedur ColorPalette aufgerufen und bietet dem Benutzer die 16 Bildschirmfarben zur Auswahl an. Sie hat folgende Parameter:

Parameter	*Datentyp*	*Inhalt*
Farbe	INTEGER	Die ausgewählte Farbe.
ForeColor, BackColor	INTEGER	Die Vorder- und Hintergrundfarbe für die Dialogbox.
Abbruch	INTEGER	wird als TRUE zurückgegben, wenn der Benutzer das Dialogfeld abgebrochen hat.

Text suchen und ersetzen

Die Dialogboxen zum Suchen und Ersetzen von Text können mit ChangeText bzw. FindText aufgerufen werden.

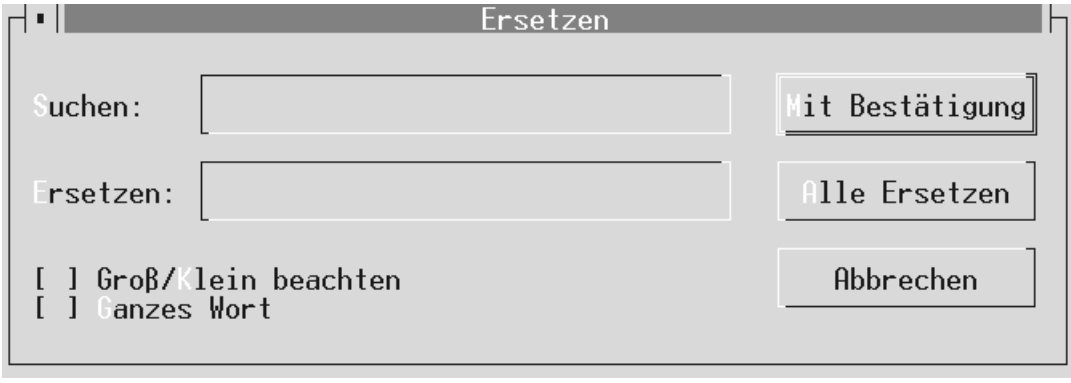


Abbildung 21–3: Die „Ersetzen“-Dialogbox

Die Parameter für „Ersetzen“ sind:

Parameter	*Datentyp*	*Inhalt*
Suchen	STRING	Der zu suchende Text.
Ersetzen	STRING	Der Ersatztext.
ForeColor, BackColor	INTEGER	Die Vorder- und Hintergrundfarbe für die Dialogbox.
Optionen	INTEGER	Wert 1: Groß-/Kleinschreibung beachten ist angewählt Wert 2: Ganzes Wort ist angewählt Wert 4: Alle Ersetzen ist angewählt (Summierung möglich, z.B. 3 = Ganzes Wort und Groß/Klein)
Flags	INTEGER	Wie oben, steuert hier aber nicht, welche Felder angewählt, sondern welche Felder anwählbar sind (also überhaupt gezeigt werden).
Abbruch	INTEGER	wird als TRUE zurückgegben, wenn der Benutzer das Dialogfeld abgebrochen hat.

Für „Suchen“ gelten dieselben Werte; „ErsatzText“ fällt jedoch weg, und der Wert 4 in den Feldern „Optionen“ und „Flags“ bedeutet „Suchrichtung aufwärts ist gewählt“ bzw. „Suchrichtung ist anwählbar“.

Über...

Die Routine About schließlich fungiert als bessere MSGBOX, der man einen Text übergibt, der über das Programm informiert.

Parameter	*Datentyp*	*Inhalt*
Text	STRING	Der anzuzeigende Text (wird automatisch umgebrochen)
ForeColor, BackColor	INTEGER	Die Vorder- und Hintergrundfarbe für die Dialogbox
Flags	INTEGER	Auf 1 setzen, wenn ein (etwas mißglücktes) Visual BASIC-Symbol angezeigt werden soll

21.2 Hilfe!

In der professionellen Ausgabe von VBDOS liefert Microsoft eine Hilfe-Toolbox mit, mit der Sie recht einfach eine Hilfefunktion in Ihren Programmen implementieren können, die mit gewöhnlichen Textdateien arbeitet.

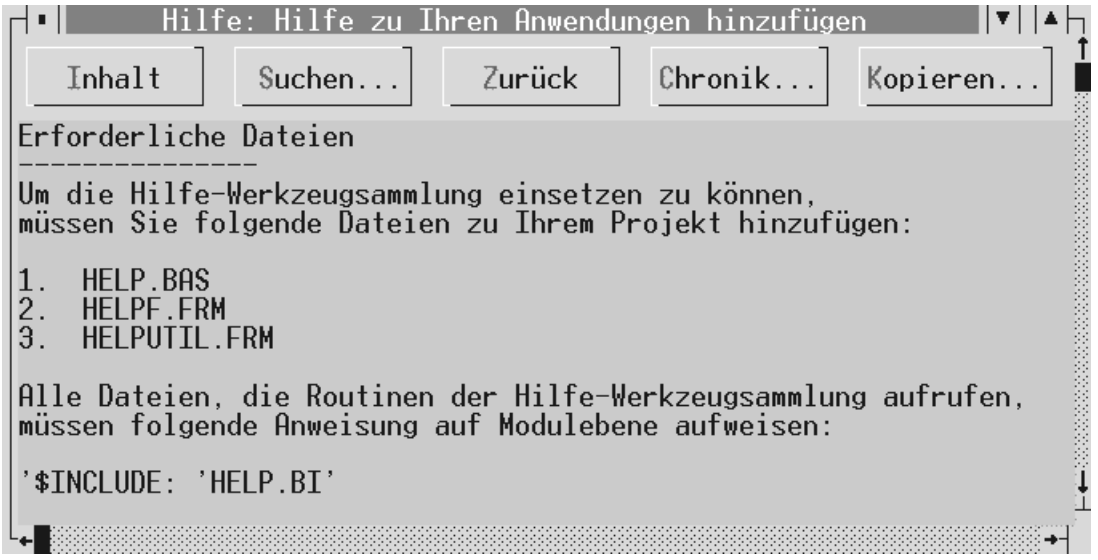


Abbildung 21–4: Das Hilfefenster der Help-Toolbox

Struktur der Hilfedateien

Grundlage dieser Toolbox sind Hilfedateien, die wie folgt aufgebaut sind:

```
.TOPIC
Themenname
beliebig viele textzeilen
.TOPIC
Themenname
... (usw.)
```

Dabei kann als „Themenname“ ein beliebiger, eindeutiger Name vergeben werden. Außerdem können Sie mitten im Text Querverweise auf andere Themen unterbringen, indem Sie den Themennamen in die Zeichen ◄ und ► (`CHR$(17)` und `CHR$(16)`) einschließen. Wenn der Benutzer diesen Querverweis dann mit der Tab-Taste ansteuert und ENTER drückt oder aber mit der Maus anklickt, wird das entsprechende Hilfethema angezeigt. Zum Beispiel:

```
.TOPIC
Taschenrechner verwenden
Den Taschenrechner benutzen Sie, indem Sie...
blabla...
.TOPIC
Kopfrechnen
Kopfrechnen ist eine schwierige Sache. Sie sollten stattdessen
lieber einen ◄Taschenrechner verwenden►.
Ansonsten können Sie auch...
blabla...
```

Funktionsweise der Toolbox

Wenn Sie die Hilfe-Toolbox in einem eigenen Programm einsetzen wollen, müssen Sie die Dateien HELP.BAS, HELPF.FRM und HELPUTIL.FRM laden oder aber aus diesen Dateien eine Library erstellen. Entfernen Sie in jedem Falle den Modulcode in HELP.BAS (ab Zeile 116), denn der verlängert Ihre EXE-Dateien und Quick Libraries unnötig.

Laden Sie in jedem Programm, das die Hilfe-Toolbox verwenden soll, HELP.BI mit einem $INCLUDE-Befehl.

Am Anfang des Programms rufen Sie dann die **HelpRegister**-Prozedur auf:

```
HelpRegister DateiName$, Erfolg%
```

In *DateiName$* schreiben Sie den Namen der zu ladenden Hilfedatei, in *Erfolg%* wird zurückgegeben, ob es geklappt hat (TRUE) oder nicht (FALSE). Gründe für ein Fehlschlagen sind oft ungültiger Dateiname oder mangelnder Speicher.

Die HelpRegister-Prozedur liest die angegebene Datei einmal durch und sucht dabei nach .TOPIC-Zeilen. Sie speichert jeden Themennamen zusammen mit der Dateiposition, an der er gefunden wurde, in einem Array. Bei einer durchschnittlichen Länge eines Themennamens von 30 Zeichen müßte so die Speicherung von gut 500 Hilfethemen möglich sein (da die Hilfethemen zeitweise im Array und in einem Listenfeld gespeichert werden, belegen sie doppelt Platz).

Mit der Prozedur **HelpClose** können Sie das Hilfesystem jederzeit wieder aus dem Speicher entfernen.

Die Prozedur **HelpShowTopic** öffnet das Hilfe-Fenster und zeigt ein bestimmtes Hilfethema an:

```
HelpShowTopic Thema$
```

Thema$ muß dabei genau dem in der Datei verwendeten Themennamen entsprechen (lediglich die Groß- und Kleinschreibung darf abweichen).

Die Hilfe-Form wird dabei nicht gebunden angezeigt, so daß der Benutzer durchaus andere Aktionen in Ihrem Programm ausführen kann, während die Hilfe am Bildschirm sichtbar ist.

HelpShowTopic durchsucht die im Array gespeicherte Liste von oben nach unten nach dem angegebenen Thema. Wenn man in HelpRegister das Themen-Array sortieren würde, könnte man hier das binäre Suchen verwenden und – zumindest bei einer großen Anzahl von Hilfethemen – viel Zeit sparen. Wenn das Thema gefunden ist, wird es aus der Hilfedatei gelesen und angezeigt.

Dabei ist zu beachten, daß sich die dauerhafte gespeicherte Themenliste und der Text des gerade angezeigten Themas den Speicher im Formsegment teilen müssen – zusammen dürfen beide nur etwa 60 KB belegen.

Mit den Schaltflächen am oberen Rand des Hilfe-Fensters kann der Benutzer eine Liste aller Hilfethemen abrufen, zum zuletzt angezeigten Hilfethema wechseln, eine Liste der letzten 20 Hilfethemen abrufen oder Text aus der Hilfe in das CLIPBOARD-Objekt kopieren.

Welche dieser Schaltflächen aktiv sind und in welchen Farben sich das Ganze abspielt, können Sie durch einen Aufruf an die Prozedur **HelpSetOptions** festlegen (lesen Sie die Details bitte im Quelltext nach). Durch den Aufruf der Prozedur **HelpSearch** können Sie auch aus dem Programm heraus direkt eine Liste aller Themen anzeigen lassen.

Die Hilfe-Toolbox als Studienobjekt...

Wenn Sie sich die Zeit nehmen, den Quellcode der Hilfe-Toolbox durchzulesen, können Sie eventuell einige interessante Anregungen herausholen. Zum Beispiel wird der Hilfstext nicht, wie es vielleicht auf den ersten Blick aussieht, in einem Textfeld angezeigt, sondern mit PRINT direkt auf die Form geschrieben. Das hat den Nachteil, daß man stets den gesamten Text im Speicher haben muß und auch Code schreiben muß, der sich darum kümmert, das Bild neu aufzubauen, wenn die Bildlaufleisten betätigt werden.

Nötig war das deshalb, weil die in Dreiecke eingeschlossenen Querverweise im Text farblich hervorgehoben werden sollen. Textfelder bieten diese Möglichkeit nicht. Um den Aufwand in Grenzen zu halten, wird der Text nicht automatisch umgebrochen (wie das bei einem Textfeld der Fall ist).

Außerdem ist das typische Toolbox-Konzept (eine Register-Prozedur, die anfangs gestartet wird, und einige globale Variablen, an denen die anderen Prozeduren erkennen, ob „registriert“ wurde) durchaus nachahmenswert, weil die Toolbox dadurch ohne großen Aufwand in eigene Programme eingebunden werden kann.

Auch das Verfahren, „normale“ Textdateien als Hilfedatei zu verwenden, halte ich für keine schlechte Idee.

...aber nicht als Fertiglösung

Zu einer ausgereiften Hilfe-Lösung fehlen der Toolbox aber noch einige Features: So ist es zum Beispiel nicht möglich, „versteckte“ Querverweise zu ver-

wenden (was tun Sie, wenn Sie in fünf verschiedenen Hilfethemen den Querverweis „mehr Details“ haben, der aber jedesmal auf einen anderen Detail-Text zeigen soll?), und ebenso fehlt eine Möglichkeit, mehrere Themennamen für ein und dasselbe Thema zu vergeben.

In der Praxis hat es sich außerdem bewährt, neben der Liste aller Hilfethemen auch noch eine Liste von Stichworten zu erstellen. Dabei kann ein Thema beliebig viele Stichworte enthalten, und in der Stichwortliste wird einfach gespeichert „Stichwort x kommt in Hilfethema y in Zeile z vor“. So kann der Benutzer dann neben einer Liste von Themen (dem Inhaltsverzeichnis) auch noch eine Liste von Stichworten (den Index) abrufen, in dem durchaus mehrere Einträge auf das gleiche Thema verweisen, aber eventuell auf verschiedene Zeilen bzw. Abschnitte darin.

Um einer möglichen Speicherknappheit durch die hohe Zahl von Themen und Stichworten zu entgehen, ist es außerdem besser, diese Datenfelder in eine Indexdatei auf der Festplatte zu verlegen.

Falls Sie jetzt den Satz „Ein Programm, das all dies leistet, finden Sie auf der beiliegenden Diskette“ erwarten, muß ich Sie ausnahmsweise enttäuschen: So etwas habe ich für VBDOS selbst noch nicht. Allerdings: Ein Programm, das Ansätze meiner Ideen realisiert, finden Sie auf der Diskette unter RHELP.BAS.

21.3 Das Setup-Programm

Ebenfalls nur der professionellen Ausgabe hat Microsoft das Programm SETUP (SETUP.BAS, SETUPSTS.FRM, SETUPPTH.FRM, SETUPOPT.FRM, SETUPMSG.FRM) beigelegt, mit dem Sie ein einfaches Installationsprogramm für Ihre Software herstellen können.

Sie können alle programmspezifischen Einstellungen, die dafür benötigt werden, in der Prozedur **InitSetup** vornehmen. Durch entsprechende Zuweisungen legt diese Prozedur fest,

- welche Überschriften angezeigt werden,
- welche Farben verwendet werden,
- welche Meldungen am Beginn und Ende der Installation angezeigt werden,
- welches Verzeichnis als Zielverzeichnis vorgegeben wird,
- wieviel Speicherplatz zur Installation des Programms erforderlich ist,
- welche Auswahlmöglichkeiten der Benutzer hat und
- welche Dateien bei welcher Auswahlmöglichkeit von welcher Diskette kopiert werden sollen.

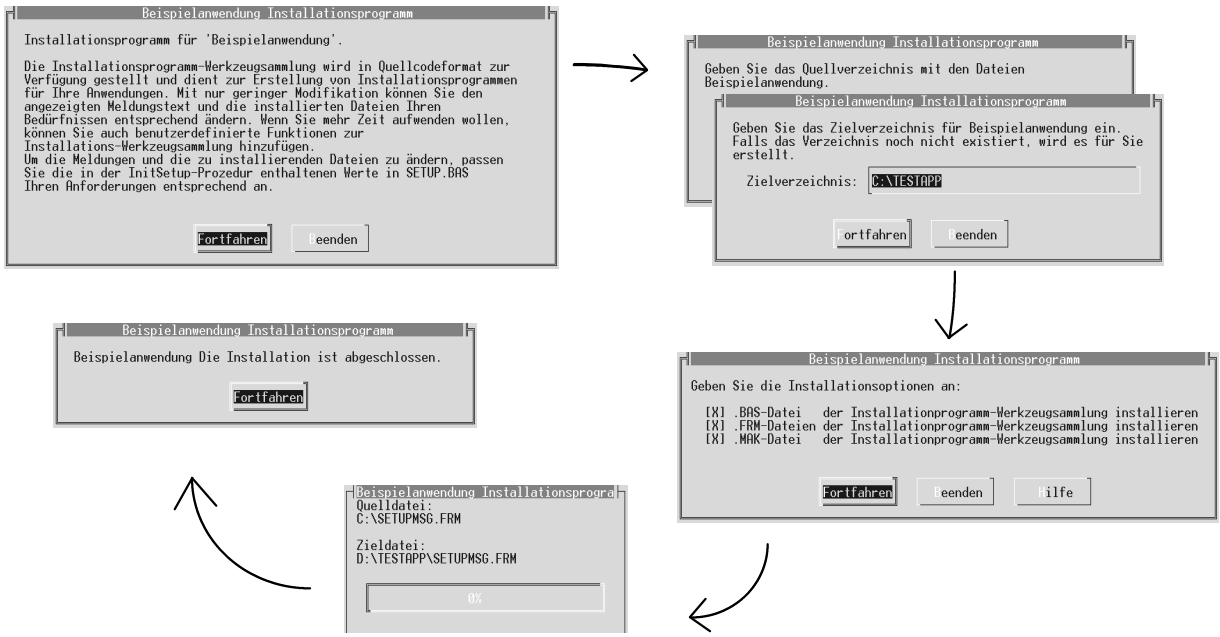


Abbildung 21–5: Ein Beispieldurchlauf des SETUP-Programms

Um die Verwendung dieses Programms etwas einfacher zu gestalten, sollten Sie auf jeden Fall die **InitSetup**-Prozedur ab Zeile 69 so ändern, daß die Anzahl der Disketten und Dateien sowie die Information, auf welcher Diskette sich eine Datei befindet und bei welcher Installationsoption sie kopiert werden soll, aus einer Datei eingelesen wird.

Die Prüfung, ob das Ziellaufwerk genügend freie Speicherkapazität hat, erfolgt hier anhand einer vorgegebenen Zahl und nicht abhängig davon, welche Installationsoptionen der Benutzer ausgewählt hat.

Das Setup-Programm ist eine nette Dreingabe, bedarf aber aus meiner Sicht und in Anbetracht dessen, was ich in Kapitel 19 über die Installation von Programmen geschrieben habe, noch einiger Verbesserungen. Immerhin können Sie damit für kleine Programme schnell eine kleine Installation erstellen, die besser als eine Batchprozedur ist.

21.4 Eine universelle Wartemeldung

Es kommt nicht gerade selten vor, daß man dem Benutzer Informationen über einen laufenden Vorgang anzeigen will; als besonders elegant gilt es in diesen Tagen, das mit einem Prozentbalken zu tun:

Abbildung 21–6: Die Form WARTEN.FRM in Aktion

Eine solche Form zu programmieren, ist nicht sonderlich schwer; in meinem Beispiel – auf der Diskette unter WARTEN.FRM – habe ich jedoch einige Methoden verwandt, die für Sie interessant sein können.

Flaschenhals

Um zu vermeiden, daß das Programm, in das die Warte-Form eingebaut wird, zuerst noch eine INCLUDE-Datei laden und mehrere verschiedene Prozeduren, SHOW- und HIDE-Methoden und so weiter aufrufen muß, läuft die gesamte Kommunikation zwischen Ihrem Programm und der Warteform über ein Bezeichnungsfeld namens *Steuerung*, das in der Abbildung nicht sichtbar ist, weil seine *Visible*-Eigenschaft auf FALSE gesetzt wurde.

Dieses Feld kann über seine *Caption*-Eigenschaft Befehle aus Ihrem Programm empfangen. Folgende Befehle sind verfügbar:

Befehl	*Wirkung*
INIT *zahl*	Zukünftige Prozentberechnungen werden am Wert *zahl* orientiert.
TEXT *text*	Als Warte-Meldung wird der Text *text* angezeigt.
SHOW	Die Warteform wird angezeigt.
HIDE	Die Warteform wird ausgeblendet.
zahl	Die *zahl* wird anhand des bei INIT übergebenen Wertes in Prozent umgerechnet und in der Warteform angezeigt.

Wie Sie diese „Befehle“ in der Praxis einsetzen, werden Sie gleich im Beispiel sehen.

Darüber hinaus kann Ihr Programm über die Eigenschaft *Tag* der Warten-Form den Status der Form abfragen:

Warten.Tag	*bedeutet*
OK	Form wird angezeigt
"" (leer)	Form wird nicht angezeigt
ABBRUCH	Benutzer hat auf „Abbrechen“ geklickt

Dadurch, daß die gesamte Steuerung der Form über das unsichtbare Element *Steuerung* läuft, ist die Einbindung in ein bestehendes Programm äußerst einfach. Man benötigt keinen INCLUDE-Befehl; lediglich `$FORM Warten` muß in die Programme eingetragen werden, die die Form verwenden sollen.

Einfacher geht es nicht

Betrachten Sie das folgende Beispielprogramm:

```
'$FORM Warten
FUNCTION DateiDrucken (DateiName AS STRING) AS INTEGER

    DIM DateiNummer AS INTEGER, Zeile AS STRING
    DateiNummer = FREEFILE
    OPEN DateiName FOR INPUT AS #DateiNummer

    Warten.Steuerung.Caption = "INIT" + STR$(LOF(DateiNummer))
    Warten.Steuerung.Caption = "TEXT Datei wird gedruckt..."
    Warten.Steuerung.Caption = "SHOW"

    DO UNTIL EOF(DateiNummer) OR Warten.Tag = "ABBRUCH"
       LINE INPUT #DateiNummer, Zeile
       PRINTER.PRINT Zeile
       Warten.Steuerung.Caption = STR$(SEEK(DateiNummer))
    LOOP

    ' FALSE zurückgeben, wenn abgebrochen; sonst TRUE:
    DateiDrucken = NOT (Warten.Tag = "ABBRUCH")
    Warten.Steuerung.Caption = "HIDE"

END FUNCTION
```

Listing 21–1: WARTTEST.BAS

Mit drei klar verständlichen Befehlen wird hier die Warte-Form aktiviert; als Bezugsgröße wird die Länge der Datei, die gedruckt wird, übergeben. Während des Datei-Ausdrucks wird *Warten.Steuerung* stets die gerade erreichte Dateiposition mitgeteilt, aus der dort eine Prozentzahl relativ zur anfangs übergebenen Größe ermittelt wird. Natürlich könnte man auch andere Größen zur Berechnung der Prozentzahl verwenden, z. B. die Anzahl der insgesamt vorhandenen und bisher gelesenen Zeilen (falls bekannt).

Hinter den Kulissen

Die Realisation einer solchen Schnittstelle mit Hilfe eines Bezeichnungsfeldes baut darauf auf, daß immer dann, wenn die *Caption*-Eigenschaft des Bezeich-

nungsfeldes geändert wird, ein *Change*-Ereignis für das Steuerelement eintritt. In WARTEN.FRM findet sich denn auch die Prozedur:

```
SUB Steuerung_Change ()

   STATIC Gesamt AS LONG: DIM Dummy AS INTEGER

   SELECT CASE LEFT$(Steuerung.Caption, 4)
   CASE "INIT"
      Gesamt = VAL(MID$(Steuerung.Caption, 5)): Prozent.Caption = "0"
   CASE "TEXT"
      Nachricht.Caption = LTRIM$(MID$(Steuerung.Caption, 5))
   CASE "SHOW"
      Visible = True: Tag = "OK"
   CASE "HIDE"
      Visible = False: Tag = ""
   CASE ELSE
      Prozent.Caption = FORMAT$(VAL(Steuerung.Caption) * 100 \ Gesamt)
   END SELECT
   Dummy = DOEVENTS()

END SUB
```

Listing 21–2: (Ausschnitt aus) WARTEN.FRM

Ferner existiert eine *Prozent_Change*-Prozedur, die immer dann, wenn die Prozentzahl geändert wird, auch den Balken neu berechnet. Der Balken befindet sich ebenfalls in einem Bezeichnungsfeld und wird aus den Zeichen „█“ (`CHR$(219)`) und „▌“ (`CHR$(221)`) zusammengesetzt.

Nur Vorteile?

Die Vorgehensweise ist – aus meiner Sicht – an Eleganz kaum zu überbieten. Es gibt aber handfeste Gründe dafür, warum sie nicht überall verwendet wird: Sie benötigt mehr Speicher (durch das zusätzliche Steuerelement) und vor allem mehr Zeit (durch die mehrfache Stringzuweisung und das dauernde Aufrufen von *Steuerung_Change*) als eine Lösung, die zum Beispiel mit verschiedenen Prozeduren für die verschiedenen „Befehle“ arbeitet.

Nicht immer ist also ein solches Schnittstellenkonzept, wie ich es hier vorgeführt habe, sinnvoll; häufig ist es jedoch eine angenehme Alternative zu vielen Prozeduren und Include-Dateien oder dem unübersichtlichen Zugriff auf verschiedene Eigenschaften der Objekte.

Tricks mit Interrupts und Systemadressen 22

Gleich zu Anfang eine Warnung: Mit Interrupts lassen sich eine Anzahl von Dingen bewerkstelligen, die im Standard-BASIC nicht – oder nur mit Umständen – möglich sind. Andererseits sollten Sie aber auch bedenken, daß die Möglichkeit, mit Interrupts zu arbeiten, in VB für WINDOWS eingeschränkt ist; Sie werden deshalb unter Umständen Probleme bekommen, wenn Sie ein Programm, das viele Interrupts verwendet, nach WINDOWS portieren möchten.

22.1 Warum und wie man Interrupts benutzt

Es gibt eine Anzahl von Aufgaben, die mit den Standard-Befehlen und Funktionen von BASIC nicht gelöst werden können. Dazu gehören zum Beispiel die Feststellung der Systemkonfiguration oder kompliziertere Directory-Abfragen (die Funktion DIR$ und die Datei-Listenfelder bieten auch nicht alle Möglichkeiten, zum Beispiel kann man Datum und Uhrzeit einer Datei nicht ermitteln).

Interrupts sind Funktionen des DOS oder des BIOS („Basic Input/Output System“). Daneben kann man über Interrupts auch Funktionen geladener Treiberprogramme aufrufen, zum Beispiel des Maus- und des EMS-Treibers (vgl. Routinen zum EMS-Zugriff in Kapitel 18). Manche Interrupts haben viele verschiedene Funktionen. Welche davon man aufrufen möchte, muß man dem Interrupt dann in einem der Register mitteilen.

Wenn man weiß, wie es zu bewerkstelligen ist, sind Interrupts fast so einfach aufzurufen wie normale BASIC-Funktionen. Natürlich muß man auch wissen, welche Interrupts überhaupt verfügbar sind. Dieses Kapitel stellt eine Auswahl von Interrupts vor, die von besonderem Interesse für BASIC-Programmierer sein können; komplette Interruptlisten finden sich in diversen technischen PC-Kompendien.

Datenübergabe bei Interrupt-Aufrufen

Wie bei normalen Funktionsaufrufen muß bei Interrupt-Aufrufen zumeist auch eine Variablenübergabe stattfinden. Einer Interrupt-Funktion kann man jedoch nicht einfach Variablen übergeben, sondern sie erwartet, daß alle Variablen (oder zumindest deren Adressen) in den Registern des Prozessors stehen. Die „Register“ sind feste Speicherplätze, auf die man mit BASIC nicht direkt zugreifen kann. Ein Register umfaßt 16 Bits, also 2 Bytes; man kann demzufolge genau eine Integer-Zahl hineinschreiben. Ich will mich hier nicht weiter mit den

Registern und ihrer Funktion auseinandersetzen. Wichtig ist nur, daß es 10 für uns interessante Register gibt: AX, BX, CX, DX, DI, SI, BP, DS, ES und das Flag-Register. Wie eben erwähnt, paßt in jedes Register eine INTEGER-Zahl. Weil es aber manchmal gar nicht nötig ist, einen so großen Bereich zu benutzen (–32.768 bis 32.767), werden die ersten vier Register AX, BX, CX und DX zuweilen zweigeteilt: aus AX werden AH (H für High) und AL (L für Low), aus BX werden BH und BL und so weiter. Wenn man vom AH-Register spricht, meint man die ersten 8 Bits des AX-Registers (entspricht in BASIC `AX \ 256`), und mit dem AL-Register sind die letzten 8 Bits von AX gemeint (in BASIC `AX MOD 256`); das gilt auch für BX, CX und DX. Wenn einem Interrupt INTEGER-Zahlen übergeben werden, können diese einfach in die Register geschrieben werden, und auch, wenn ein Interrupt INTEGER-Daten zurückliefert, kann das direkt über die Register geschehen. Wenn es allerdings darum geht, Strings oder größere Datenmengen zu übergeben, muß man hier mit Zeigern arbeiten.

Wenn zum Beispiel einem Interrupt ein Dateiname übergeben werden muß, so erwartet er gewöhnlich in einem der Register die Segment- und in einem anderen die Offset-Adresse des Strings. Die Segmentadresse eines BASIC-Strings ermitteln Sie mit SSEG, die Offsetadresse mit SADD. Die so erhaltenen Werte schreiben Sie in die entsprechenden Register. Dann kann der Interrupt auf Ihren String zugreifen. Bei Dateinamen ist es meistens so, daß sie mit einem CHR$(0) aufhören müssen, damit der Interrupt das Ende erkennen kann, weil ihm die Stringlänge nicht übergeben wird.

Gibt ein Interrupt selbst STRING-Informationen zurück, trägt also in einen String etwas ein, anstatt ihn nur zu lesen, ist die Prozedur dieselbe. Sie müssen dann aber auf jeden Fall dafür sorgen, daß der String schon *vor* dem Interrupt-Aufruf so lang ist (füllen Sie ihn mit Leerzeichen o.ä.), daß er garantiert alle Zeichen aufnehmen kann, die der Interrupt hineinschreiben wird. Andernfalls ist es ziemlich sicher, daß Ihr Programm sich früher oder später mit einen Fehler verabschiedet, weil der Speicherbereich verfälscht wurde.

Die Routinen INTERRUPT und INTERRUPTX

Eine kleine Assembler-Routine ist für den Interrupt-Aufruf zuständig. Diese Routine heißt INTERRUPT oder (wenn man auch die Register DS und ES benutzt) INTERRUPTX und wird in der Library VBDOS.LIB bzw. der Quick Library VBDOS.QLB zur Verfügung gestellt. Wenn man mit ihr in der Programmierumgebung VBDOS arbeiten will, muß man VBDOS mit dem Parameter /L aufrufen; beim Kompilieren ohne VBDOS muß für den Linker die VBDOS.LIB-Library angegeben werden.

Diese Assembler-Routine hat drei Parameter: Die Nummer des aufzurufenden Interrupts, die Registerwerte, die ihm übergeben werden sollen, und die Registerwerte, die er zurückgibt. In BASIC sieht das so aus:

```
INTERRUPT Nummer%, RegEin, RegAus
```

Meistens kann man für *RegEin* und *RegAus* dieselbe Variable benutzen, da man nicht mehr wissen muß, was man dem Interrupt als Eingabe geliefert hat, wenn er abgelaufen ist und die Ausgabe vorliegt. *RegEin* und *RegAus* sind Variablen vom Typ `RegType`. Dieser Typ ist wie folgt definiert:

```
TYPE RegType
   AX AS INTEGER
   BX AS INTEGER
   CX AS INTEGER
   DX AS INTEGER
   BP AS INTEGER
   SI AS INTEGER
   DI AS INTEGER
   FLAGS AS INTEGER
END TYPE
```

Hinweis: Es gibt auch noch einen Typ `RegTypeX`, der zusätzlich die Register DS und ES enthält, und die dazugehörige Routine `INTERRUPTX`. Beide Typdefinitionen sind in der Include-Datei VBDOS.BI enthalten.

Um einen Interrupt aufzurufen, definiert man also zwei Registervariablen vom Typ `RegType`, schreibt in die entsprechenden Elemente der ersten Variable hinein, was der Interrupt an Daten erwartet, ruft dann die INTERRUPT-Routine mit Interruptnummer und den beiden Registervariablen auf und kann danach die Ergebniswerte aus der zweiten Registervariablen auslesen.

Ein einführendes Beispiel

Angenommen, Sie möchten mit dem Interrupt 21h feststellen, welche DOS-Version gerade läuft. (Interruptnummern werden hier immer hexadezimal angegeben. Ab einem bestimmten Wissensstand *muß* man einfach mit hexadezimalen Zahlen um sich werfen, und den haben Sie, wenn Sie sich mit Interrupts beschäftigen...) Dafür müssen Sie als Eingabe-Register AH auf 30h setzen und erhalten dann als Ausgabe in AL die Hauptnummer (vor dem Punkt) und in AH die Unternummer (hinter dem Punkt). Sie programmieren also:

```
' aus Bequemlichkeit die TYPE-Definitionen aus dem INCLUDE-File einlesen:
REM $INCLUDE:'VBDOS.BI'

' Registervariablen definieren:
DIM RegEin AS RegType, RegAus AS RegType

' Eingabe-Register AH, also die ersten 8 Bits von RegEin.AX, auf 30h setzen:
RegEin.AX = &H30 * 256
' (Dabei wird zwar AL auf Null gesetzt, aber das ist ja hier egal.)
' Kürzer hätte man auch schreiben können: RegEin.AX = &H3000

' Interrupt aufrufen:
INTERRUPT &H21, RegEin, RegAus

' Ergebnis auswerten:
PRINT "Die DOS-Version ist";
PRINT RegAus.AX \ 256; "."; RegAus.AX MOD 256

' Ende.
```

Listing 22–1: DOSVER.BAS

Schwierigkeiten mit der Integer-Rechnung

Solange die Werte, die übergeben werden sollen, kleiner als 32.768 sind, gibt es keine Probleme. Einige Interrupts verarbeiten aber auch höhere Zahlen. Ein Register faßt schließlich alle Zahlen von 0 bis 65.535 (außerhalb BASIC sind solche Zahlen als „unsigned Integer“ bekannt). Ein BASIC-INTEGER reicht aber von –32.768 bis 32.767; darüber gibt es nichts mehr. Das höchste Bit hat bei BASIC nicht den Wert 32.768 (wie üblich), sondern –32.768. Strenggenommen zählt BASIC so: 0, 1, 2, ..., 32.766, 32.767, –32.768, –32.767, –32.766, ..., –3, –2, –1. Für die Interrupt-Aufrufe muß jedoch einfach von 0 bis 65.535 durchgezählt werden. Das bedeutet: Wenn Sie einem Interrupt eine Zahl *x* übergeben wollen, die größer als 32.767 ist, müssen Sie in die Registervariable *x* – 65.536 eintragen; wenn eine Registervariable nach einem Interrupt-Aufruf eine Zahl enthält, die kleiner als 0 ist, müssen Sie zunächst 65.536 addieren, bevor Sie die Zahl weiterverarbeiten können. Die Routine *UnsignedInt* im Listing FREI.BAS (später in diesem Kapitel) kann zur korrekten Interpretation der Ergebnisse eines Interrupt-Aufrufes herangezogen werden.

Die Teilung der Register AX, BX, CX und DX in je ein H- und ein L-Register bewältigen Sie am besten so:

```
AX = <Wert für AH> * 256 + <Wert für AL>
```

Wenn Sie die alten AX-Wert nicht ändern und nur AL oder nur AH neu setzen möchten, schreiben Sie

```
AX = <Wert für AH> * 256 + AX AND 255
```

oder

```
AX = (AX AND -256) + <Wert für AL>
```

Ebenso werden aus AX wieder einzelne Register gelesen:

```
PRINT "AH ist"; AX \ 256
PRINT "AL ist"; AX MOD 256
```

Auch hierbei müssen Sie natürlich beachten, daß AH nicht größer werden darf als 127; ansonsten müssen Sie für AX zunächst eine temporäre LONG-Variable benutzen und später 65.536 abziehen, da der Befehl `AX = AH * 256` sonst zu einem Überlauf-Fehler führen würde.

22.2 Mehr als 15 Dateien gleichzeitig öffnen

Unabhängig von der Eintragung „FILES=xxx“ in der Datei CONFIG.SYS können in BASIC nur 15 Dateien gleichzeitig geöffnet werden. Für die meisten Zwecke ist das ausreichend, aber zuweilen ist dieses Limit auch lästig.

Die folgenden Zeilen sorgen für mehr Luft:

```
REM $INCLUDE: 'VBDOS.BI'

SUB MehrDateien (Anzahl AS INTEGER)

   DIM Register AS RegType
   Register.AX = &H6700
   Register.BX = Anzahl + 5
   Interrupt &H21, Register, Register

END SUB
```

Listing 22–2: MEHRDAT.BAS

Beachten Sie dabei, daß die Zahl, die Sie in das BX-Register schreiben, nicht größer sein darf als die FILES-Eintragung in der DATEI CONFIG.SYS. Außerdem stehen Ihnen in BASIC immer fünf Dateinummern weniger zur Verfügung, als Sie mit dem Interruptaufruf anfordern, da DOS fünf für sich selbst abzweigt – deshalb addiert die Routine 5.

Die Prozedur funktioniert nicht, wenn Ihr Programm unter WINDOWS aufgerufen wurde. Ob das der Fall ist, läßt sich mit ENVIRON$ (vgl. Eintrag im Disketten-Referenzteil) feststellen.

22.3 Freier Platz auf einem Datenträger

Die folgende Funktion *FreierPlatz* ermittelt den freien Platz auf einem Datenträger. Mit einem Leerstring als Argument wird der freie Platz auf dem aktuellen Laufwerk zurückgegeben; stattdessen kann aber auch ein Laufwerksbuchstabe als Argument angegeben werden. Wenn der angegebene Datenträger ungültig ist, gibt die Funktion –1 zurück.

Ebenfalls zu diesem Listing gehört die Funktion *UnsignedInt*, die eingesetzt werden kann, wenn Interrupts Zahlen zurückgeben, die größer als 32.767 sind, weil BASIC diese dann als negative darstellt, obwohl sie eigentlich zwischen 32.768 und 65.535 liegen.

Durch eine kleine Modifikation (siehe Bemerkungszeile) kann man die Routine auch verwenden, um die Größe einer Zuordnungsheinheit, also die Größe, auf deren Vielfaches jede Dateigröße aufgerundet wird, zu ermitteln.

```
REM $INCLUDE: 'VBDOS.BI'

FUNCTION FreierPlatz (Laufwerk AS STRING) AS LONG

   DIM Reg AS RegType
   Reg.AX = &h3600
   IF Laufwerk = "" THEN Reg.DX = 0 ELSE Reg.DX = ASC(UCASE$(Laufwerk)) - 64
   Interrupt &h21, Reg, Reg
   IF Reg.AX = -1 THEN
      FreierPlatz = -1
   ELSE
      ' UnsignedINT(Reg.AX) * UnsignedInt(Reg.CX) ist die Größe einer Zuordnungs-
      ' einheit auf dem Datenträger!
      FreierPlatz = UnsignedInt(Reg.AX) * UnsignedInt(Reg.BX) * UnsignedInt(Reg.CX)
   END IF
END FUNCTION
```

```
FUNCTION UnsignedInt (Zahl AS INTEGER) AS LONG

   ' konvertiert eine BASIC-Integer-Zahl (von -32768 bis 32767) in einen "unsigned
   ' Integer", wie er von DOS-Interrupts meist zurückgegeben wird (0 bis 65535).
   ' Das Ergebnis muß in BASIC als LONG-Zahl dargestellt werden.
   IF Zahl < 0 THEN
      UnsignedInt = CLNG(Zahl) + 65536
      ' CLNG notwendig, da sonst Überlauf-Fehler auftritt, weil Ergebnis einer
      ' INTEGER-Berechnung größer 32767
   ELSE
      UnsignedInt = Zahl
   END IF
END FUNCTION
```

Listing 22–3: FREI.BAS

22.4 Lesen und Setzen von Dateiattributen

Neben dem Dateinamen, der Größe und dem Zeitpunkt der Erstellung speichert DOS für jede Datei noch ein Attribut-Byte, das wie folgt aufgeschlüsselt wird:

Bit	*Wert*	*Attribut*
1	1	Read-Only (Datei kann nicht gelöscht/geschrieben werden)
2	2	Hidden (Datei erscheint nicht im DIR-Verzeichnis)
3	4	System (Datei erscheint nicht im DIR-Verzeichnis)
4	8	Volume Label (Dateiname ist die Datenträgerbezeichnung)
5	16	Directory (Dateiname ist Unterverzeichniseintrag)
6	32	Archive (Archiv-Attribut ist gesetzt) Das Archiv-Attribut wird von DOS automatisch auf 1 gesetzt, wenn eine Datei erzeugt oder verändert wird. Einige Backup-Programme nutzen das und setzen alle Archiv-Attribute der gesicherten Dateien auf 0, dann müssen beim nächsten Backup nur die Dateien mit Archiv-Attribut = 1 kopiert werden.

Die folgenden Routinen ermöglichen es, das Attribut-Byte einer Datei zu setzen oder zu lesen:

```
REM $INCLUDE: 'VBDOS.BI'

FUNCTION AttributLesen (DateiName AS STRING) AS INTEGER

   DIM DOSDateiName AS STRING, Register AS RegTypeX

   DOSDateiName = DateiName + CHR$(0)
   Register.AX = &H4300
   Register.DS = SSEG(DOSDateiName): Register.DX = SADD(DOSDateiName)
   InterruptX &H21, Register, Register

   IF Register.Flags AND 1 THEN
      ' Es ist ein Fehler aufgetreten. Hier evtl. Fehlerbehandlung einbauen!
   END IF
   AttributLesen = Register.CX

END FUNCTION
```

```
' Achtung: Das Volume Label- und das Directory-Attribut dürfen mit dieser Routine
' nicht verändert werden!
SUB AttributSetzen (DateiName AS STRING, Attribut AS INTEGER)

   DIM DOSDateiName AS STRING, Register AS RegTypeX

   DOSDateiName = DateiName + CHR$(0)
   Register.AX = &H4301
```

```
   Register.DS = SSEG(DOSDateiName): Register.DX = SADD(DOSDateiName)
   Register.CX = Attribut
   InterruptX &H21, Register, Register

   IF Register.Flags AND 1 THEN
      ' Es ist ein Fehler aufgetreten. Hier evtl. Fehlerbehandlung einbauen!
   END IF

END SUB
```

Listing 22–4: ATTRIBUT.BAS

22.5 Lesen und Setzen von Datum und Uhrzeit einer Datei

Manchmal ist es wichtig, Uhrzeit und Datum einer Datei zu lesen oder zu setzen – zum Beispiel beim Kopieren einer Datei, bei dem diese Informationen erhalten werden sollen, oder dann, wenn man eine Datei verändern will, ohne sich durch das geänderte Datum zu „verraten“. Hier eine Funktion, die Datum und Zeit einer Datei als Zeitcode zurückgibt, und eine Prozedur, die Datum und Uhrzeit ebenso setzen kann:

```
FUNCTION DateiZeitCode (DateiName AS STRING) AS DOUBLE

   DIM Tag AS INTEGER, Monat AS INTEGER, Jahr AS INTEGER
   DIM Sek AS INTEGER, Min AS INTEGER, Stunde AS INTEGER
   DIM Register AS RegType, DateiNummer AS INTEGER

   DateiNummer = FREEFILE: OPEN DateiName FOR INPUT AS #DateiNummer

   Register.AX = &H5700: Register.BX = FILEATTR(DateiNummer, 2)
   INTERRUPT &H21, Register, Register: CLOSE DateiNummer

   Tag = TeilZahl(Register.DX, 1, 5): Monat = TeilZahl(Register.DX, 6, 9)
   Jahr = 1980 + TeilZahl(Register.DX, 10, 16)
   Sek = TeilZahl(Register.CX, 1, 5) * 2: Min = TeilZahl(Register.CX, 6, 11)
   Stunde = TeilZahl(Register.CX, 12, 16)
   DateiZeitCode = DATESERIAL(Jahr, Monat, Tag) + TIMESERIAL(Stunde, Min, Sek)

END FUNCTION
```

```
SUB SetzeDateiZeitCode (DateiName AS STRING, ZeitCode AS DOUBLE)

   DIM Tag AS INTEGER, Monat AS INTEGER, Jahr AS INTEGER
   DIM Sek AS INTEGER, Min AS INTEGER, Stunde AS INTEGER
   DIM Register AS RegType, DateiNummer AS INTEGER

   DateiNummer = FREEFILE: OPEN DateiName FOR INPUT AS #DateiNummer
   Register.CX = SignedInt(SECOND(ZeitCode) \ 2 + 32& * MINUTE(ZeitCode) +
                 2048& * HOUR(ZeitCode))
```

```
   Register.DX = SignedInt(DAY(ZeitCode) + 32& * MONTH(ZeitCode) + 512& *
                 (YEAR(ZeitCode) - 1980))
   Register.AX = &H5701: Register.BX = FILEATTR(DateiNummer, 2)
   INTERRUPT &H21, Register, Register
   CLOSE DateiNummer

END SUB
```

```
FUNCTION SignedInt (Zahl AS LONG) AS INTEGER

   IF Zahl < 32768 THEN SignedInt = Zahl ELSE SignedInt = Zahl - 65536

END FUNCTION
```

```
FUNCTION TeilZahl% (Zahl AS INTEGER, VonBit AS INTEGER, BisBit AS INTEGER)
   ' ist in Listing INHALT.BAS abgedruckt
END FUNCTION
```

Listing 22–5: FILEDATE.BAS

22.6 Eine Datei abschneiden

Kurze Knobelaufgabe für den versierten BASIC-Programmierer: Wie entfernen Sie das letzte Zeichen einer Datei?

Gar nicht – BASIC kann zwar Dateien verlängern, aber nicht kürzen. Sie müssen die alte Datei umbenennen, den Inhalt (bis auf das eine Byte) in eine neue Datei kopieren und dann die temporäre Datei löschen. Das ist nicht nur langwierig, sondern auch gefährlich: Bei Stromausfall ist nämlich gar nichts mehr von der zu kürzenden Datei übrig.

Das Abschneiden von Dateien ist aber immer dann notwendig, wenn Sie mit einer RANDOM- oder BINARY-Datei arbeiten und diese verkürzt werden soll; schließlich soll die Datenbank eines Programms auch schrumpfen, wenn Daten gelöscht werden (und nicht bloß wachsen, wenn welche hinzukommen).

Die folgende Routine schafft Abhilfe:

```
REM $INCLUDE: 'VBDOS.BI'

SUB DateiAbschneiden (DateiName AS STRING, NeueLaenge AS LONG, ASCII AS INTEGER)

   DIM DateiNummer AS INTEGER, EOFMarkierung AS STRING * 1
   DIM Register AS RegType

   DateiNummer = FREEFILE: EOFMarkierung = CHR$(26)
   OPEN DateiName FOR BINARY AS #DateiNummer

   SEEK #DateiNummer, NeueLaenge + 1
```

```
   IF ASCII THEN PUT #DateiNummer, NeueLaenge + 1, EOFMarkierung
   Register.AX = &H4000
   Register.BX = FILEATTR(DateiNummer, 2)   ' findet den DOS-Handle heraus
   Register.CX = 0
   INTERRUPT &H21, Register, Register
   CLOSE DateiNummer

END SUB
```

Listing 22–6: FILECUT.BAS

Sie nutzt den Trick, daß DOS eine Datei abschneidet, wenn man ihm aufträgt, „Nichts“ in die Datei zu schreiben. Sie übergeben der Routine einfach den Dateinamen, die gewünschte neue Länge und einen True/False-Parameter, der TRUE sein sollte, wenn es sich um eine sequentielle Datei handelt. Diese muß zwar nicht unbedingt, sollte aber mit einem Dateiende-Kennzeichen abgeschlossen werden, und das hängt die Routine dann noch an.

22.7 Dateinamen komplettieren

Mit den DOS-Dateinamen ist es so ein Kreuz: In bestimmten Fällen können die Namen „..\BAUER.EXE“, „C:\VBDOS\BAUER.EXE“ und „F:BAUER.EXE“ die gleiche Datei bezeichnen (dann nämlich, wenn man sich im Verzeichnis C:\VBDOS\HLP befindet und ein `SUBST F: C:\VBDOS`-Befehl aktiv ist). Das kann in einigen Situationen äußerst lästig sein.

Die folgende Funktion löst alle ASSIGN-, SUBST- und Verzeichnisreferenzen auf und gibt immer einen String der Form

```
Laufwerk:\[Verzeichnispfad\]Dateiname
```

zurück (Fragezeichen und Sternchen im Dateinamen werden dabei allerdings nicht weiter berücksichtigt). Im obigen Beispiel würde sie für alle drei Eingaben „C:\VBDOS\BAUER.EXE“ zurückliefern.

```
FUNCTION KanonischerName (DateiName AS STRING) AS STRING

   DIM DOSDateiName AS STRING, NeuerName AS STRING * 255
   DIM Register AS RegTypeX

   DOSDateiName = DateiName + CHR$(0)
   Register.AX = &H6000

   Register.SI = SADD(DOSDateiName): Register.DS = SSEG(DOSDateiName)
   Register.di = VARPTR(NeuerName): Register.es = VARSEG(NeuerName)
   InterruptX &H21, Register, Register
```

```
   IF Flags AND 1 THEN  ' Fehler!
      KanonischerName = ""
   ELSE
      KanonischerName = LEFT$(NeuerName, INSTR(NeuerName, CHR$(0)) - 1)
   END IF

END FUNCTION
```

Listing 22–7: KANONAME.BAS

22.8 Inhaltsverzeichnis

Hier stelle ich eine ganze Reihe von Funktionen vor, die dazu dienen, ein Inhaltsverzeichnis von der Festplatte zu lesen.

Die Routinen *FindFirstFile* und *FindNextFile* benutzen Interrupts, um DOS nach Dateien suchen zu lassen, und schreiben das Ergebnis in einen BASIC-String. Sie entsprechen etwa DIR$, sind aber weitaus mächtiger, da sie zum Beispiel auch versteckte Dateien oder Verzeichnisse finden und außerdem mehr Informationen über die gefundenen Dateien zur Verfügung stellen.

GetAttrFile, *GetDateFile*, *GetTimeFile*, *GetSizeFile* und *GetNameFile* haben die Aufgabe, die gewünschten Informationen aus dem String zu extrahieren, den DOS übergibt. *GetTimeCode* ist eine Funktion, die *GetDateFile* und *GetTimeFile* ersetzt, indem sie direkt den vollständigen Zeitcode der Datei ermittelt und zurückgibt.

Ebenfalls von großer Bedeutung ist die Routine *TeilZahl*, die für viele Interrupt-Aufrufe nützlich ist. Zuweilen geben Interrupts nämlich einfach eine INTEGER-Zahl zurück, in der verschiedene Informationen gespeichert sind. Mit TeilZahl können Sie bestimmte Bit-Gruppen aus einer INTEGER-Zahl auslesen. Wenn es zum Beispiel heißt „... zurückgegeben wird die Anzahl der Schnittstellen in den Bits 3-7 von AX“, ergibt ein Aufruf von `TeilZahl(AX, 3, 7)` diese Zahl. Die umfangreichsten Routinen schließlich, *Directory* und *DirectoryRek*, stellen Anwendungen der anderen Prozeduren und Funktionen dar. Beide Prozeduren erzeugen ein Verzeichnis bestimmter, angegebener Dateien (zum Beispiel `*.DOC`). *Directory* tut dies für ein festgelegtes Verzeichnis und gibt nur die reinen Dateinamen zurück, während *DirectoryRek* auch alle Subdirectories des angegebenen Verzeichnisses untersucht und deshalb Dateinamen mit vollständiger Directory-Angabe zurückliefert.

Bei beiden Prozeduren enthält die Variable *DirName* den Namen des Verzeichnisses, von dem ausgegangen wird. *DirName* sollte mit einem Backslash enden (zum Beispiel `C:\`). *Maske* ist die Bezeichnung, zu der passende Dateien gesucht werden sollten (beispielsweise `ARC*.*`). In das String-Feld *FileName()*

werden die Dateinamen eingetragen, und mit UBOUND(FileName) kann man abfragen, wie viele es sind, da das Array dynamisch erweitert wird.

```
REM $INCLUDE: 'VBDOS.BI'

' Bei Aufruf muß FileName() mit 0 TO irgendwas dimensioniert sein
SUB Directory (DirName AS STRING, Maske AS STRING, FileName() AS STRING)

   DIM Fehler AS INTEGER, Anzahl AS INTEGER

   Anzahl = 0: Fehler = 0
   ' Attribut = 39 sucht alle Dateien inkl. Hidden + System + Read-Only
   FindFirstFile DirName + Maske, 39, Fehler

   DO UNTIL Fehler
      Anzahl = Anzahl + 1 ' Jetzt prüfen, ob FileName ausreichend dimensioniert ist;
                          ' wenn nicht, Platz für 50 weitere Namen schaffen
      IF Anzahl>UBOUND(FileName) THEN REDIM PRESERVE FileName(0 TO Anzahl+50) AS STRING
      FileName(Anzahl) = GetNameFile: FindNextFile Fehler
   LOOP

   ' Zum Schluß die Obergrenze von FileName() korrigieren, so daß das aufrufende
   ' Programm die Anzahl erkennen kann:
   REDIM PRESERVE FileName (0 TO Anzahl) AS STRING

END SUB
```

```
' Achtung: Für Aufruf von DirectoryRek ist wichtig, daß
' 1. DirName+Maske zusammen eine gültige Dateibezeichnung (die auch bei DIR erlaubt
'    wäre) ergeben
' 2. FileName mit 0 TO 0 (also "leer") dimensioniert ist
SUB DirectoryRek (DirName AS STRING, Maske AS STRING, FileName() AS STRING)

   DIM Fehler AS INTEGER, Anzahl AS INTEGER, NeuDirName AS STRING
   DIM SubDirectory(0 TO 0) AS STRING  ' wird bei Bedarf mit REDIM erweitert

   Fehler = 0: Anzahl = UBOUND(FileName)
   ' Attribut = 39 sucht alle Dateien inkl. Hidden + System + Read-Only
   FindFirstFile DirName + Maske, 39, Fehler
   DO UNTIL Fehler
      Anzahl = Anzahl + 1
      IF Anzahl>UBOUND(FileName) THEN REDIM PRESERVE FileName(0 TO Anzahl+50) AS STRING
      FileName(Anzahl) = DirName + GetNameFile: FindNextFile Fehler
   LOOP

   ' Vor dem rekursiven Aufruf die Größe von FileName justieren (könnte ja um bis
   ' zu 49 zu groß sein)
   REDIM PRESERVE FileName(0 TO Anzahl) AS STRING
```

```
   ' jetzt Subdirectories suchen:
   Fehler = 0: SubDirectoryZaehler = 0
   ' Attribut = 16 sucht Directories
   FindFirstFile DirName + "*.*", 16, Fehler
   DO UNTIL Fehler
      ' zusätzliche Abfrage, da auch Dateien mit Attribut > 16 gefunden werden:
      IF GetAttrFile AND 16 THEN
         ' es ist ein Subdirectory gefunden worden.
         IF LEFT$(GetNameFile, 1) <> "." THEN
            REDIM PRESERVE SubDirectory(0 TO 1 + UBOUND(SubDirectory))
            SubDirectory(UBOUND(SubDirectory)) = GetNameFile
         END IF
      END IF
      FindNextFile Fehler
   LOOP

   ' Alle Dateien sind in Liste. Rekursiv werden die Subdirectories abgearbeitet:
   FOR i% = 1 TO UBOUND(SubDirectory)
      NeuDirName = DirName + SubDirectory(i%) + "\"
      DirectoryRek NeuDirName, Maske, FileName(), Anzahl
   NEXT

END SUB
```

```
SUB FindFirstFile (Maske AS STRING, Attribut AS INTEGER, ErrorCode AS INTEGER)

   SHARED DTA AS STRING
   DIM Register AS RegTypeX, FileName AS STRING

   ' Interrupt &H21, Funktion &H1A: DOS die Adresse der DTA (Disk Transfer Area)
   ' mitteilen, in der es seine Datei-Informationen ablegen kann:
   DTA = SPACE$(128)
   Register.ax = &H1A * 256: Register.ds = SSEG(DTA): Register.dx = SADD(DTA)
   InterruptX &H21, Register, Register

   ' Interrupt &H212, Funktion &H4E: Erste Datei suchen, die paßt
   FileName = Maske + CHR$(0)
   Register.ax = &H4E * 256: Register.cx = Attribut
   Register.ds = SSEG(FileName): Register.dx = SADD(FileName)
   InterruptX &H21, Register, Register

   IF Register.flags AND 1 THEN
      ' Fehler! (Keine Datei vorhanden oder Directory-Angabe falsch)
      ErrorCode = Register.ax: EXIT SUB
   END IF
   ErrorCode = 0

END SUB
```

```
SUB FindNextFile (ErrorCode AS INTEGER)
   SHARED DTA AS STRING
   DIM Register AS RegTypeX
   ' Interrupt &H21, Funktion &H1A: DOS die Adresse der DTA (Disk Transfer Area)
   ' mitteilen, in der es seine Datei-Informationen ablegen kann:
```

```
   ' (Muß dauernd aufgerufen werden, weil BASIC von Zeit zu Zeit den String DTA im
   ' Speicher herumschiebt und dann die DOS bekannte Adresse nicht mehr stimmt!)
   Register.ax = &H1A * 256: Register.ds = SSEG(DTA): Register.dx = SADD(DTA)
   InterruptX &H21, Register, Register

   ' Interrupt &H21, Funktion &H4F: Nächste passende Datei suchen
   Register.ax = &H4F * 256
   InterruptX &H21, Register, Register
   IF Register.flags AND 1 THEN
      ' Fehler! (Keine Datei mehr gefunden)
      ErrorCode = Register.ax
   END IF

END SUB
```

```
FUNCTION GetAttrFile() AS INTEGER

   SHARED DTA AS STRING
   GetAttrFile = ASC(MID$(DTA, 22))

END FUNCTION
```

```
SUB GetDateFile (Monat AS INTEGER, Tag AS INTEGER, Jahr AS INTEGER)

   SHARED DTA AS STRING
   DIM Datum AS INTEGER

   Datum = CVI(MID$(DTA, 25, 2))          ' Datum steht in Bytes 25-26 der DTA
   Tag = TeilZahl(Datum, 1, 5)            ' Tag in den Bits 1-5 des Datums
   Monat = TeilZahl(Datum, 6, 9)          ' Monat in Bits 6-9 des Datums
   Jahr = 1980 + TeilZahl(Datum, 10, 16)  ' Jahr relativ zu 1980 in Bits 10-16

END SUB
```

```
FUNCTION GetNameFile() AS STRING

   SHARED DTA AS STRING
   GetNameFile = MID$(DTA, 31, INSTR(31, DTA, CHR$(0)) - 31)

END FUNCTION
```

```
FUNCTION GetSizeFile() AS LONG

   SHARED DTA AS STRING
   GetSizeFile = ASC(MID$(DTA, 27)) + 256 * ASC(MID$(DTA, 28)) + 65536
                 * ASC(MID$(DTA, 29)) + 16777216 * ASC(MID$(DTA, 30))

END FUNCTION
```

```
FUNCTION GetTimeCode AS DOUBLE

   GetTimeFile h%, m%, s%
   GetDateFile mm%, d%, y%
   GetTimeCode = TIMESERIAL(h%, m%, s%) + DATESERIAL(mm%, d%, y%)

END FUNCTION
```

```
SUB GetTimeFile (Stunde AS INTEGER, Min AS INTEGER, Sekunde AS INTEGER)

   SHARED DTA AS STRING
   DIM Zeit AS INTEGER

   Zeit = CVI(MID$(DTA, 23, 2))       ' Zeitangabe steht auf Bytes 23-24 der DTA
   Sekunde = TeilZahl(Zeit, 1, 5) * 2 ' Sekunde ist in den ersten 5 Bits
                                      ' gespeichert, allerdings in 2-Sek.-Schritten
   Min = TeilZahl(Zeit, 6, 11)        ' Minute in den Bits 6-11
   Stunde = TeilZahl(Zeit, 12, 16)    ' Stunde in den Bits 12-16

END SUB
```

```
' Funktion TeilZahl ermittelt eine Zahl, die in einem bestimmten Bit-Bereich einer
' INTEGER-Zahl gespeichert ist. Gültige Bitnummern sind 1 bis 16. Die kleinste
' Zahl, die zurückgegeben wird, ist 0, die größte ist 2 ^ (BisBit - VonBit).
FUNCTION TeilZahl (Zahl AS INTEGER, VonBit AS INTEGER, BisBit AS INTEGER) AS INTEGER

   z% = 0
   FOR i = VonBit TO BisBit
      ' Erhebliche Zeitgewinne möglich, wenn statt 2 ^ ... geschrieben wird:
      ' ZerierPotenz(...) - das Array muß dann nur einmal initialisiert werden
      IF Zahl AND (2 ^ (i - 1)) THEN z% = z% + 2 ^ (i - VonBit)
   NEXT
   TeilZahl = z%

END FUNCTION
```

Listing 22–8: INHALT.BAS

22.9 Feststellen, ob SHARE geladen ist

Der OPEN-Befehl mit dem Zusatz „ACCESS“ und auch die LOCK- und UNLOCK-Befehle funktionieren nur, wenn das DOS-Dienstprogramm SHARE.EXE geladen ist.

Um gleich bei Programmstart festzustellen, ob SHARE geladen wurde, können Sie die folgende Funktion verwenden:

```
FUNCTION ShareGeladen () AS INTEGER

   DIM Register AS RegType
   Register.AX = &H1000
   INTERRUPT &H2F, Register, Register
   ShareGeladen = (Register.AX AND 255) = 255

END FUNCTION
```

Listing 22–9: SHARE.BAS

In der letzten Zeile benutze ich eine elegante Kurzform für folgenden Ausdruck:

```
IF (Register.AX AND 255) = 255 THEN
   ShareGeladen = TRUE
ELSE
   ShareGeladen = FALSE
END IF
```

Wenn in einer Zeile mehrmals das Gleichheitszeichen auftritt, nimmt BASIC das erste als Indikator für eine Zuweisung (in vielen anderen Sprachen schreibt man dann ja „:=“), und alle weiteren werden als Teil eines logischen Ausdrucks (einer „Behauptung“) genommen, deren Wahrheitsgehalt BASIC prüft und mit –1 (TRUE) oder 0 (FALSE) einsetzt.

22.10 Konfiguration des Systems

Dieses Beispielprogramm gibt einige Konfigurationsdaten aus. Wegen der unterschiedlichen Konfigurationsinformation bei PCs und ATs funktioniert die Coprozessor-Angabe nur bei ATs; alle anderen Daten sind für beide Rechnertypen korrekt. Das Programm benutzt die Routine *TeilZahl*, die schon im „INHALT“-Listing abgedruckt war und deshalb hier nicht mehr angegeben wird.

```
REM $INCLUDE: 'VBDOS.BI'

DIM Register AS RegType

' Interrupt &H11: Feststellen der Konfiguration
Interrupt &H11, Register, Register

PRINT "Anzahl der Diskettenlaufwerke:";
IF Register.AX AND 1 THEN
   ' System besitzt Diskettenlaufwerke, Anzahl steht in den Bits 5-6 von AX
   PRINT TeilZahl(Register.AX, 5, 6)
ELSE
   ' System besitzt keine Diskettenlaufwerke
   PRINT "0"
END IF

' Bei Coprozessor oder 486er ist Bit 2 von AX gesetzt
PRINT "Coprozessor bzw. i486 DX-Prozessor: ";
IF Register.AX AND 2 THEN
   PRINT "installiert"
ELSE
   PRINT "nicht installiert"
END IF
```

⇨

```
' Bits 10-12 von AX enthalten die Anzahl der COM-Schnittstellen
PRINT "Serielle Schnittstellen:"; TeilZahl(Register.AX, 10, 12)

' Bits 15-16 enthalten die Anzahl der LPT-Schnittstellen
PRINT "Drucker-Schnittstellen:"; TeilZahl(Register.AX, 15, 16)

' Interrupt &H12: Feststellen der Speichergröße
Interrupt &H12, Register, Register
PRINT "Speicher unter 1 MB:"; Register.AX; "KB"
```

Listing 22–10: CONFIG.BAS

22.11 PrtSc verhindern

Üblicherweise wird durch Betätigen der „Druck“- oder „PrtSc“-Taste (bei älteren Tastaturen mußte man zusätzlich noch „Shift“ drücken) eine Kopie des Bildschirminhalts an den Drucker gesendet. Das führt im Grafikmodus zu unerwünschten Resultaten, wenn kein geeigneter Treiber geladen ist, und auch sonst ist es zuweilen nicht sinnvoll, daß der Benutzer diese Taste drückt (z. B., wenn Ihr Programm selbst eine bessere Hardcopy-Routine enthält). Mit dem simplen Befehl

```
DEF SEG = 0: POKE 1280, 1
```

können Sie den Ausdruck einer Hardcopy verhindern. Mit

```
DEF SEG = 0: POKE 1280, 0
```

wird die Taste wieder aktiviert.

22.12 Den Tastaturpuffer beschreiben

Sie haben sicher schon bemerkt, daß Sie üblicherweise bis zu 15 Zeichen auf der Tastatur schreiben können, obwohl noch ein Programm läuft, und diese Zeichen dann erscheinen, wenn das nächste Mal eine Eingabe erfolgt.

Diese Annehmlichkeit bewerkstelligt ein Tastaturpuffer, der von DOS bereitgestellt wird. Sie können eigene Daten in den Tastaturpuffer schreiben und diesen auch auslesen:

```
FUNCTION LiesTastaturPuffer () AS STRING

   DIM Ergebnis AS STRING

   DEF SEG = 0
   Anfang = 1024 + PEEK(1050)  ' Die Startadresse des Puffers
   Ende = 1024 + PEEK(1052)    ' Die Endadresse (kann kleiner als Anfang sein)
```

```
   DO UNTIL Anfang = Ende
      Ergebnis = Ergebnis + CHR$(PEEK(Anfang))
      Anfang = Anfang + 2: IF Anfang > 1084 THEN Anfang = 1054 ' Ringpuffer!
   LOOP

   LiesTastaturPuffer = Ergebnis

END FUNCTION
```

```
SUB SchreibeTastaturPuffer (Text AS STRING)

   DIM Zaehler AS INTEGER
   IF LEN(Text) > 15 THEN EXIT SUB  ' Mehr als 15 Zeichen sind nicht erlaubt!

   DEF SEG = 0: POKE 1050, 30: POKE 1052, 30 + 2 * LEN(Text)

   FOR Zaehler = 1 TO LEN(Text)
      POKE 1052 + 2 * Zaehler, ASC(MID$(Text, Zaehler, 1))
   NEXT

END SUB
```

Listing 22–11: TPUFFER.BAS

Die Funktion „LiesTastaturPuffer“ gibt einfach den aktuellen Inhalt des Puffers zurück, und „SchreibeTastaturPuffer“ schreibt einen bis zu 15 Zeichen langen Text in den Tastaturpuffer. Löschen können Sie den Tastaturpuffer jederzeit auch ohne Tricks: Schreiben Sie einfach `DO: LOOP WHILE LEN(INKEY$)`.

Zwischen SHELL und RUN

Insbesondere ist dieses Vorgehen geeignet, um andere Programme zu starten und ihnen Parameter zu übergeben, was ja mit RUN sonst nicht möglich ist:

```
SchreibeTastaturPuffer "CHKDSK /F" + CHR$(13)
SYSTEM
```

Diese Kombination beendet das BASIC-Programm und ruft danach das DOS-Dienstprogramm CHKDSK mit dem Parameter /F auf (CHR$(13) wirkt wie ein Druck auf die Enter-Taste).

Sie können diese Technik weiterentwickeln und damit auch vorher erzeugte Batchprozeduren aufrufen, die am Ende wieder Ihr Programm starten. Auf diese Weise läßt sich ein Effekt bewirken, der ähnlich wie der des SHELL-Befehls ist: Fremde Programme aufrufen und danach zum eigenen Programm zurückkehren. Ich verwende eine solche Technik im Programm XPROF.FRM, das Sie auf der Diskette finden und das in Kapitel 16 beschrieben wird.

Der Vorteil gegenüber SHELL ist, daß bei dieser Technik Ihr Programm völlig aus dem Speicher verschwindet und das aufgerufene Programm allen Speicher zur Verfügung hat, während bei SHELL das aufrufende Programm ja im Speicher verbleibt. Die Speicher-Räumung bietet zwar auch der RUN-Befehl, aber bei diesem gibt es keine Möglichkeit, danach wieder in das aufrufende Programm zu verzweigen.

Beachten Sie aber eines: Wenn Ihr BASIC-Programm nicht direkt von DOS, sondern aus einer Batchprozedur oder aus einem anderen Programm (z. B. WINDOWS) heraus aufgerufen wird, kehrt die Befehlsausführung nach einem SYSTEM dorthin zurück, und die in den Tastaturpuffer geschriebenen Zeichen können ihre Wirkung nicht entfalten.

Erweiterte Tastencodes

Wenn Sie sich die Routinen genauer ansehen, merken Sie, daß nur jedes zweite Byte (nur gerade Adressen) aus dem Tastaturpufferbereich gelesen wird. Die ungeraden Adressen benutzt DOS für Tastenkombinationen. Bei Kombinationen mit der Alt-Taste steht beispielsweise auf der geraden Adresse eine 0, auf der ungeraden dahinter der Scancode der gedrückten Taste (z.B. 30 für die Taste „A“). Eine Liste der Scancodes finden Sie im Anhang; so können Sie durch eine leichte Erweiterung der Routinen auch erweiterte Tastencodes verarbeiten, wenn es nötig ist.

22.13 Text auf Formen schreiben

VBDOS stellt, auch während der Formenanzeige, diverse Möglichkeiten zur Verfügung, auf den Bildschirm zu schreiben. Es ist jedoch nicht erlaubt, „gewaltsam“ mit dem PRINT-Befehl zu operieren, während Formen angezeigt werden.

Falls es doch einmal notwendig werden sollte, können Sie diese Routine verwenden:

```
SUB InterPrint (Zeile AS INTEGER, Spalte AS INTEGER, Text AS STRING,
                Farbe AS INTEGER)

   DIM Register AS RegType

   Register.DX = (Zeile - 1) * 256 + Spalte - 1: Register.CX = 1

   IF Farbe >= 0 THEN
      FOR Zaehler = 1 TO LEN(Text)
         Register.AX = &H200: Register.BX = 0: Register.DX = Register.DX + 1
```

```
         Interrupt &H10, Register, Register
         Register.AX = &H900 + ASC(MID$(Text, Zaehler)): Register.BX = Farbe
         Interrupt &H10, Register, Register
      NEXT
   ELSE
      Register.BX = 0
      FOR Zaehler = 1 TO LEN(Text)
         Register.AX = &H200: Register.DX = Register.DX + 1
         Interrupt &H10, Register, Register
         Register.AX = &HA00 + ASC(MID$(Text, Zaehler))
         Interrupt &H10, Register, Register
      NEXT
   END IF

END SUB
```

Listing 22–12: INTERPRN.BAS

Sie übergeben der Prozedur die Bildschirmzeile und -spalte, an der die Ausgabe erfolgen soll, dann den Text (der nicht länger sein darf, als die angegebene Zeile fassen kann) und schließlich noch ein Farb-Byte. Übergeben Sie –1 als *Farbe*, verwendet die Routine die Unterfunktion 0Ah des Interrupts 10h und schreibt damit den Text auf dem Bildschirm, ohne die Farbeinstellung an den jeweiligen Positionen zu ändern. Übergeben Sie einen positiven Farbwert, wird die Unterfunktion 09h desselben Interrupts verwendet, die die Zeichen in der von Ihnen angegebenen Farbe schreibt. Dabei können Sie den benötigten Wert für *Farbe* so ausrechnen: `Farbe = Vordergrund + 16 * Hintergrund`, wobei für Vordergrund und Hintergrund je die Werte von 0 bis 15 erlaubt sind (eine Liste der Farben finden Sie im Objekt-Referenzteil unter „BackColor“).

Beachten Sie allerdings, daß jeglicher Text, den Sie mittels der InterPrint-Routine ausgeben, von VBDOS gelöscht wird, wenn es die Form neu zeichnet oder verschiebt, oder wenn die REFRESH-Methode für das Steuerelement, auf das der Text geschrieben wurde, aufgerufen wird.

22.14 Das System booten

Wenn Ihr Programm Änderungen an der AUTOEXEC.BAT- oder CONFIG.SYS-Prozedur vornimmt, möchten Sie vielleicht vom Programm aus direkt einen „Reboot“ auslösen. Das erreichen Sie – wenn Sie die Library VBDOS.LIB beim Linken angeben bzw. VBDOS mit /L starten und die VBDOS.BI-Datei mit $INCLUDE laden – durch folgenden Aufruf:

```
DEF SEG = &HFFFF: Absolute 0
```

Dadurch wird ein „Kaltstart“ des Systems verursacht, das heißt, es beginnt mit dem Selbsttest. Wenn Sie stattdessen einen „Warmstart“ wünschen, der nicht so lange dauert, schicken Sie der oben genannten Zeile noch folgende voran:

```
DEF SEG = 0: POKE &H472, 52: POKE &H473, 18
```

Allerdings reagieren auf diesen Unterschied nicht alle Systeme; manche führen dennoch den längeren „Kaltstart“ durch.

Hinweis: Manche Cache-Programme schreiben Daten zeitverzögert auf die Platte. Bauen Sie, um sicherzugehen, vor einen solchen Reboot-Aufruf eine Pause von einigen Sekunden ein.

Tricks zur Optimierung 23

In diesem Kapitel finden Sie einige Hinweise, wie sich auch ohne den Einsatz von Toolboxen oder Assembler-Unterprogrammen einiges an Zeit oder EXE-Speicherplatz einsparen läßt.

23.1 Die Geschwindigkeit optimieren

286-/386-spezifischer Code

Verwenden Sie, wenn es möglich ist, den /G2- oder den /G3-Switch. Die erzeugten Programme laufen dann nur auf Rechnern mit den Prozessoren 80286 und aufwärts, sind aber schneller und kürzer als gewöhnlich mit diesen Prozessoren, da sie neue, schnellere Befehle verwenden, die es beim 8086/8088 noch nicht gab.

Geben Sie sich jedoch nicht der Illusion hin, ein Programm, das mit /G2 oder /G3 kompiliert wurde, nutze die besseren Fähigkeiten des Prozessors voll aus. Die Switches haben keine allzugroße Wirkung, da sie nur auf die Maschinensprachbefehle wirken, die der Compiler direkt erzeugt. Die meisten BASIC-Befehle werden aber über Funktionsaufrufe an die BASIC-Libraries abgewickelt, und die Libraries sind leider nicht in einer Spezialversion für diese höheren Prozessoren verfügbar (obwohl das technisch durchaus machbar wäre).

Wenn Sie nur die Standard-Version von VBDOS haben, können Sie die Switches /G2 und /G3 trotzdem verwenden: Benutzen Sie das BCPATCH-Programm auf der Diskette.

Alternate Math-Library (nur Profi-Version)

Wenn das Zielsystem mit Sicherheit keinen Coprozessor besitzt, sollte die Alternate Math-Library (siehe Kapitel 15) benutzt werden; auf Rechnern ohne Coprozessor ist sie bei der Fließkomma-Rechnung ein bißchen schneller als die Emulator-Libray.

Arrays

Statische Felder (REM $STATIC) sind schneller als dynamische (REM $DYNAMIC), und dynamische Felder in Programmen, die ohne /Ah kompiliert wurden, sind schneller als solche in /Ah-Programmen. Wenn Sie ein Programm mit

/Ah kompilieren, werden alle Array-Zugriffe in diesem Programm langsamer (Ausnahme: Arrays von Strings mit variabler Länge); wenn Sie aber nur in wenigen Routinen auf ein großes Array zugreifen müssen, dann lagern Sie diese Routinen in ein eigenes Modul aus und kompilieren Sie nur dieses eine Modul mit /Ah (das geht nicht von VBDOS aus, weil VBDOS, wenn es mit /Ah aufgerufen wird, alle Module mit /Ah kompiliert).

GOTO und GOSUB – alt, aber rüstig

GOTO, der Schrecken eines jeden Struktur-Fans, ist nichtsdestotrotz der schnellste Sprungbefehl von allen. An besonders zeitkritischen Stellen hat er deshalb noch immer seine Rechtfertigung gegenüber Strukturen wie FOR...NEXT, DO...LOOP etc., und manchmal kann es sich sogar lohnen, ein SUB oder eine FUNCTION, die nur von einer Stelle aufgerufen wird, in eine mit GOTO adressierte Befehlsgruppe zu verwandeln.

GOSUB ist zwar auch verschrieen, da es längst durch SUB...END SUB ersetzt wurde; trotzdem ist es geringfügig schneller als sein modernes Pendant. An Stellen, an denen man also auf lokale Fehlerbehandlung, lokale Variablen etc. verzichten kann, lohnt sich noch der Einsatz von GOSUB.

Datenübergabe bei Prozeduraufrufen

Beim Aufrufen von SUBs und FUNCTIONs sollten so wenig Parameter wie möglich übergeben werden. Die Übergabe von Werten aus dynamischen Arrays und von Strings mit fester Länge sollte als Werteparameter geschehen (Klammern um den Parameter setzen oder BYVAL verwenden – siehe Eintrag zu CALL im Disketten-Referenzteil), um die Geschwindigkeit des Aufrufs zu erhöhen.

Statische Subroutinen

Der Zusatz STATIC bei SUBs und FUNCTIONs sorgt für schnellere Ausführung, da die Variablen nicht jedesmal neu erzeugt werden. Allerdings benötigt diese Variante aus dem gleichen Grund mehr Speicherplatz.

Zeilennummern und -labels

Je weniger Zeilennummern/Zeilenlabels im Code vorhanden sind, desto besser kann der Compiler optimieren, weil er dann weniger damit rechnen muß, daß ein Sprung an eine bestimmte Stelle erfolgt. Im Listing

```
i& = k% * 2 + 5
j& = k% * 2 + 5
```

würde BC zum Beispiel nur einmal Code für die Berechnung des Ausdrucks erzeugen und das Ergebnis dann i& und j& zuweisen. Befände sich zwischen beiden Zeilen jedoch ein Label oder eine Zeilennummer, wäre das nicht möglich, weil BC nicht weiß, ob vielleicht irgendwann per GOTO, RESUME o. ä. an diese Zeile gesprungen wird.

Arrays statt Funktionen

Wenn man genug Speicher zur Verfügung hat, lohnt es sich zuweilen, die Funktionswerte einer oft benötigten Funktion (zum Beispiel trigonometrische oder auch selbstgeschriebene Funktionen) in einem Array abzulegen, da dieses sehr viel schneller im Zugriff ist. Das ist natürlich nur dann möglich, wenn man nur eine feste und nicht allzu große Zahl von Funktionswerten benötigt.

Ein extremes Beispiel hierfür sind die Zweierpotenzen: Wenn Ihr Programm öfter mit dem ^-Operator Zweierpotenzen berechnet, können Sie die Geschwindigkeit des Programms enorm erhöhen, wenn Sie ein Array ZweierPotenz() anlegen, das am Anfang einmal initialisiert wird, anstatt jedesmal die Potenz neu zu berechnen.

Leerstrings

Abfragen wie `IF a$ = ""` oder `LOOP WHILE a$ = ""`, die nicht gerade selten vorkommen, laufen schneller, wenn sie sich nicht auf den Inhalt, sondern auf die Länge des Strings beziehen (besser ist also: `IF LEN(a$) = 0` bzw. `LOOP WHILE LEN(a$) = 0`). Das erzeugt auch weniger EXE-Code.

Auswahl der Datentypen

Achten Sie darauf, niemals unnötig zu große Datentypen zu benutzen. Die Reihenfolge in der Verarbeitungsgeschwindigkeit (von schnell nach langsam) ist: INTEGER, LONG, CURRENCY (bei Zuweisung, Addition, Subtraktion), SINGLE, DOUBLE, CURRENCY (bei Multiplikation und Division). Für Strings lassen sich keine genauen Angaben machen, sicher ist aber, daß selbst

die schnellsten unter ihnen – Strings mit fester Länge 1 – noch langsamer sind als INTEGER-Zahlen. Strings mit fester Länge sind stets schneller als solche mit variabler Länge.

Schleifen

Verwenden Sie insbesondere für Schleifenvariablen den kleinstmöglichen Datentyp. In den meisten Fällen reicht ein INTEGER für die Zählervariable aus.

Vermeiden Sie jeden unnötigen Befehl in einer Schleife. Häufig lohnt es sich zum Beispiel, den Wert einer in der Schleife benutzten Funktion schon zuvor zu ermitteln und in einer Variablen zu speichern. Statt dieses Konstruktes...

```
FOR i = 0 TO Length - 1
   PRINT CHR$(PEEK(i + Offset))
NEXT
```

...sollten Sie lieber schreiben:

```
FOR i = Offset TO Length - 1 + Offset
   PRINT CHR$(PEEK(i))
NEXT
```

Beim ersten Beispiel wird insgesamt *Length* mal zu dem sich verändernden *i* die Variable *Offset* addiert. Im zweiten Beispiel sind nur zwei Additionen fällig (der Ausdruck `Length - 1 + Offset` wird als Zielwert einer FOR...NEXT-Schleife nur einmal berechnet).

IF...THEN-Abfragen

Formulieren Sie IF...THEN- und CASE-Strukturen mit Bedacht. Prüfen Sie die Bedingung, die am häufigsten eintreten wird, zuerst. Welche Bedingungen wie häufig eintreten, können Sie mit dem Profiler (nur professionelle Ausgabe, vgl. Kapitel 16) ermitteln.

String-Verknüpfungen

Vermeiden Sie unnötige String-Verbindungen mit dem Plus-Operator. `PRINT a$ + b$` verschwendet viel Rechenzeit gegenüber `PRINT a$; b$`, weil hier zuerst ein temporärer String angelegt werden muß. Frevelhaft ist das folgende Konstrukt, bei dem gleich 256mal ein String an den anderen gehängt werden muß:

```
z$ = ""
FOR a% = 0 TO 255
   z$ = z$ + CHR$(a%)
NEXT
```

Viel einfacher und mehr als doppelt so schnell erreicht man den gleichen Effekt so:

```
z$ = SPACE$(256)
FOR a% = 0 TO 255
   MID$(z$, a%, 1) = CHR$(a%)
NEXT
```

23.2 Programmgröße und Speicherplatz optimieren

Konstanten

Überall, wo Variablen feste Werte haben, die sich im Verlauf des Programms nicht ändern, sollten Sie statt ihrer symbolische Konstanten benutzen. Das spart Speicherplatz im DGROUP-Segment und verkleinert auch das EXE-File, weil symbolische Konstanten bereits beim Kompilieren in Skalare verwandelt werden können. Konstanten sparen auch Zeit.

Statische Subroutinen

DGROUP kommt es zugute, wenn man SUBs und FUNCTIONs möglichst nicht als STATIC vereinbart. Dann müssen die lokalen Variablen dieser Routinen nur so lange in DGROUP verbleiben, wie sie aktiv sind, und nicht auch während des gesamten weiteren Programmablaufs. STATIC-Routinen sind allerdings, wie vorher erwähnt, etwas schneller in der Ausführung. Dies ist also einer der Punkte, an denen man sich zwischen Speicher-Einsparung und Geschwindigkeitsvorteilen entscheiden muß.

Dateipuffer bei sequentiellen Dateien

Kleine Dateipuffer sparen Speicherplatz im String-Bereich. Ein OPEN-Befehl für sequentielle Dateien (INPUT, OUTPUT, APPEND) ohne LEN-Zusatz belegt 512 Bytes als Puffer. Durch einen LEN-Zusatz mit kleinerer Zahl kann dieser Bedarf verringert werden. Je größer der Puffer, desto schneller jedoch der Zugriff auf die Datei.

Variablen nach dem SCREEN-Befehl

Vermeiden Sie es, dem SCREEN-Befehl Variablen folgen zu lassen. Der Compiler weiß dann nicht, welche Werte die Variable annehmen kann, und bindet die Routinen für sämtliche SCREEN-Modi ein. Benutzen Sie stattdessen Konstanten im SCREEN-Befehl, oder verwandeln Sie eine `SCREEN a%`-Anweisung in `SELECT CASE a% / CASE 1: SCREEN 1` etc., wenn es Modi gibt, die Sie nicht benötigen. Bestimmte SCREEN-Modi können auch mittels Verzicht-Files ausgeschlossen werden; siehe „Verzicht-Files“ im Abschnitt 3.

Fließkomma-Arithmetik

Wenn Sie ohne Runtime-Modul (mit /O) kompilieren, benötigt die Fließkommaarithmetik-Einheit – egal, ob Sie mit Emulator- oder mit Alternate Math-Library arbeiten – etwa 10 KB in der EXE-Datei. Sobald irgendwo im Programm Variablen bzw. Konstanten vom Typ SINGLE oder DOUBLE, einer der Befehle SIN, COS, TAN, ATN, LOG, EXP, VAL, WINDOW, DRAW, TIMER, RANDOM, INPUT, READ, PMAP, POINT, PRINT USING, SOUND oder CIRCLE oder der Divisions-Operator / auftauchen, wird unweigerlich die gesamte Fließkomma-Einheit eingebunden. In vielen Fällen läßt sich das bei kleineren Programmen vermeiden. Bei der Division von INTEGER- und LONG- Zahlen kann man statt / auch den Integer-Divisionsoperator \ benutzen, der ohnehin um ein Vielfaches schneller arbeitet.

Event Trapping

Event Trapping, also die Verwendung von ON *event* GOSUB-Befehlen, ist eine platzraubende Sache. Mit /W kompiliert, wird für jedes Label und jede Zeilennummer ein Event-Test erzeugt, bei /V für jeden einzelnen Befehl. Selbst ein OFF-Befehl (KEY OFF etc.) verhindert das nicht, da der Compiler nicht wissen kann, wann das Trapping später ein- und wann es ausgeschaltet sein wird. Wenn es Stellen im Programm gibt, an denen das Event-Trapping ohnehin nie eingeschaltet sein wird oder nicht benötigt wird, verwenden Sie auf jeden Fall EVENT OFF, um dem Compiler dies mitzuteilen; dann wird zumindest für Teilbereiche die Produktion von Event-Tests verhindert.

Datenübergabe bei BASIC-Befehlen

Die Verwendung vieler BASIC-Befehle wird intern wie ein Prozeduraufruf gehandhabt. Dabei wird fast immer eine feste Anzahl von Variablen übergeben,

auch wenn der Befehl nicht immer gleich viele Argumente braucht. Der LOCATE-Befehl kann z. B. bis zu fünf Argumente haben, meistens wird er aber nur mit zweien verwendet. Trotzdem wird für den Befehl `LOCATE 5, 7` intern ein Funktionsaufruf wie etwa `CALL LOCATE(5, 7, 1, 1, 1)` generiert, und jedes übergebene Argument benötigt einige Bytes im EXE-Programm.

Wenn Sie den LOCATE-Befehl sehr oft mit nur zwei Argumenten verwenden, ist es sinnvoll, eine Prozedur zu schreiben:

```
SUB LOCATE2(x AS INTEGER, y AS INTEGER)
   LOCATE x, y
END SUB
```

Rufen Sie nun die Prozedur LOCATE2 in Ihrem Programm anstelle des LOCATE-Befehls auf, wird jedesmal der Code für die Übergabe von 3 Argumenten gespart.

Ein ähnliches Verfahren ist bei vielen anderen BASIC-Befehlen möglich.

Error Trapping

ON ERROR ist ein Speicherfresser; wenn Sie mit /X kompilieren (also die Verwendung von RESUME und RESUME NEXT ermöglichen), wird jeder einzelne Befehl zur potentiellen Rücksprungadresse durch RESUME – 4 Bytes Adreßcode pro Befehl werden generiert. Verzichten Sie auf RESUME und RESUME NEXT, und benutzen Sie dann nur /E beim Kompilieren, wenn möglich (Sie müßten alle RESUME und RESUME NEXT-Befehle durch RESUME *zeilennummer/-label* ersetzen). Noch besser ist es, die Routinen, die ON ERROR benötigen, getrennt zu kompilieren und dann erst beim Linken zum Rest des Programms hinzuzufügen.

Verzicht auf Coprozessor-Emulation

Wenn Sie sicher sind, daß das Zielsystem einen Coprozessor besitzt, kann die Library 87.LIB beim Linken angegeben werden. Dadurch werden die Routinen, die für eine Emulation des Coprozessors im Falle seines Fehlens zuständig sind, nicht in die EXE-Datei eingebunden.

Statische und dynamische Felder

Gerade in kleinen Programmen kostet es bis zu 5 KB an zusätzlichen Verwaltungsroutinen im EXE-File, wenn man statt statischen dynamische Felder verwendet (was in kleinen Programmen noch dazu zumeist unnötig ist).

Compiler- und Linker-Schalter

Switches, die beim Kompilieren und Linken angegeben werden, haben großen Einfluß auf die Programmgröße. Die Compiler-Switches /Ah, /D, /E, /Es, /O, /V, /W, /X, /Zd und /Zi vergrößern unter Umständen das EXE-File. Die Option /S kann die Programmgröße je nach Programmgestaltung nach oben oder unten verändern – siehe nächster Abschnitt –, und /G2 und /G3 sorgen in jedem Fall für etwas kleinere Programme. Beim Linken drücken /NON, /E und /F/PACKC die Programmgröße nach unten, während sie durch /CO, /NOF/NOP, /PADD und /PADC erhöht werden kann. Welche Einsparungen allein durch Switches noch möglich sind, zeigt die folgende Tabelle anhand von zwei verschieden großen Programmen aus dem Lieferumfang von VBDOS:

BC-Switch	*LINK-Switch*	*CHRTDEMO.EXE Größe*	*QSORT.EXE Größe*
–	–	324432	51630
/G2	–	320406	51566
/G3	–	320278	51550
–	/F/PACKC	318710	51118
–	/EX	297518	40976
/G3	/NON/EX/F/PACKC	291008	40534
wie oben; zusätzlich aber CHART.LIB durch Kompilieren mit /G3 neu erstellt		289008	
wie oben; CHRTDEMO.BAS und CHRTDATA.BAS mit /S kompiliert		288914	
wie oben, aber die RND-Funktion in QSORT.BAS durch eine eigene Routine ersetzt, die ohne Fließkommaeinheit auskommt			30210
zusätzlich noch Verzicht-Files NOCOM, NOEDIT, TSCNIO und NOLPT gelinkt			23436

Der Compiler-Switch /S

Üblicherweise merkt sich der Compiler BC.EXE alle String-Konstanten in Ihrem Programm und ersetzt mehrfach vorkommende durch Verweise. Enthält Ihr Programm zweimal den Befehl `PRINT "Guten Tag"`, wird dieser String nur einmal gespeichert, und für weitere Vorkommen werden Verweise auf den ersten String eingebaut. Dadurch wird nicht nur Ihr Programm kürzer, sondern BC braucht auch selbst mehr Hauptspeicher, während es Ihr Programm kompiliert.

Der Switch /S stellt diese „String-Verweise“ ab. Dadurch wird das resultierende EXE-Programm zumeist länger, insbesondere dann, wenn viele gleiche Stringkonstanten darin vorkommen. BC braucht allerdings auch beim Kompilieren weniger Speicherplatz, so daß Sie zuweilen Programme, bei denen BC „Nicht genug Speicher“ meldet, mit /S noch kompilieren können.

Zusätzlich – und das macht ihn so undurchsichtig – fügt er der Switch /S zwei kurze Routinen zur Stringzuweisung und -verknüpfung an ihr Programm an, die mit weniger Aufwand aufgerufen werden können als die in der BASIC-Library enthaltenen. Das bringt einen Platzvorteil, der sich umso stärker auswirkt, je mehr Stringzuweisungen und -verknüpfungen Ihr Programm enthält.

Bei Programmen mit vielen Stringoperationen, aber wenig gleichen Stringkonstanten wird /S also unter dem Strich ein kürzeres EXE-File ermöglichen, während Programme mit wenig Stringoperationen oder sehr vielen gleichen Stringkonstanten durch /S länger werden. Probieren Sie es einfach aus; oft ist es so, daß einige Module eines Projekts besser mit, andere besser ohne /S kompiliert werden.

23.3 LINK über die Schulter geschaut

Im Kapitel 6 ist LINK bereits behandelt worden; wesentliche Aufgabe von LINK ist es, aus diversen OBJ-Dateien ein ausführbares EXE-Programm zu produzieren und dabei alle benötigten Routinen aus den angegebenen Libraries einzubinden.

Obwohl LINK sich hier ja schon Mühe gibt, enthalten Ihre Programme oft Code, der niemals ausgeführt wird, weil LINK mehr einbindet, als nötig wäre. Dieses Phänomen ist Thema dieses Abschnittes.

Granularität

LINK kann nur ganze OBJ-Dateien aus der Library auswählen, nicht Teile davon. Wenn Sie also in einer Library drei OBJ-Dateien untergebracht haben, von denen jede zwanzig SUBs oder FUNCTIONs besitzt, dann reicht es schon, daß eine der zwanzig Routinen benötigt wird, damit das ganze OBJ-File mit allen zwanzig Routinen aus der Library eingebunden wird.

Die Chart-Library besteht im wesentlichen aus den zwei Quelldateien FONT.BAS und CHART.BAS; wenn Sie nur eine einzige Routine (z. B. ChartScatter) aus der Chart-Library aufrufen, wird der gesamte Code beider Module mit eingebunden: ChartScatter ist in CHART.OBJ enthalten, daher wird

CHART.OBJ komplett eingebunden, und irgendwo in CHART.OBJ befinden sich Aufrufe an FONT-Routinen, daher wird auch FONT.OBJ benötigt.

Deshalb ist es, wenn Sie ohnehin mit Libraries arbeiten, zumeist sinnvoll, möglichst jede Subroutine einzeln zu kompilieren und in die Library aufzunehmen, anstatt alle Subroutinen in wenigen BAS-Files zu kompilieren. Das hieße, um beim ersten Beispiel zu bleiben, daß Sie nicht drei OBJ-Dateien mit je 20, sondern 60 OBJ-Dateien mit je einer Routine darin verwenden sollten. Die Library wird zwar dadurch länger, weil mehr Verwaltungsaufwand geleistet werden muß, aber dafür kann später beim Herstellen von EXE-Dateien besser selektiert werden, was eingebunden wird und was nicht. Das macht sich bei EXE-Files, die nicht alle Routinen aus der Library benötigen, in der Größe bemerkbar.

Microsoft spricht von *hoher Granularität*, wenn die Library aus vielen OBJ-Dateien besteht, so daß der Linker relativ genau selektieren kann. Die Granularität einer Library können Sie feststellen, indem Sie LIB benutzen, um eine Liste aller enthaltenen Routinen zu erstellen.

Inhalt einer Library

Ein Beispiel: Der Befehl `LIB VBDOS,VBDOS.LST;` legt eine Liste aller Routinen in VBDOS.LIB an und schreibt sie in die Datei VBDOS.LST. Diese sieht wie folgt aus:

```
ABSOLUTE..........absolute          INT86OLD..........int86old
INT86XOLD.........int86old          INTERRUPT.........intrpt
INTERRUPTX........intrpt

absolute          Offset: 00000010H  Größe von Code und Daten: cH
  ABSOLUTE

intrpt            Offset: 000000d0H  Größe von Code und Daten: 111H
  INTERRUPT         INTERRUPTX

int86old          Offset: 000002c0H  Größe von Code und Daten: 11eH
  INT86OLD          INT86XOLD
```

Eine solche Liste besteht aus zwei Teilen. Der erste enthält alphabetisch sortiert die Namen aller Routinen (Prozeduren und Funktionen) in der Library mit dem Namen des OBJ-Files, aus dem sie ursprünglich stammen.

Der zweite Teil besteht aus einer alphabetischen Liste aller OBJ-Dateien, die in die Library eingebunden wurden, mit den Routinen, die sich darin befinden.

Hier können Sie leicht die Granularität einer Library ablesen. Je mehr Routinen im Durchschnitt in einem OBJ-File enthalten sind, desto kleiner (und demzufolge ungünstiger) ist die Granularität der Library. In unserem Beispiel handelt es sich also um eine überdurchschnittlich granulare Library, denn fünf Routinen sind auf drei OBJ-Files verteilt.

Die wichtigste Library bei der Erzeugung von ausführbaren Programmen, VBDCL10E.LIB, enthält in der professionellen Version gut 3.000 Routinen, die auf etwa 360 OBJ-Dateien verteilt sind.

Die Crux beim LINKen

Betrachten Sie folgende drei Programme:

MAIN.BAS

```
DECLARE SUB Test1(z%)
FOR i% = 1 TO 100
  Test1 i%
NEXT
```

SUB1.BAS

```
DECLARE SUB Test2()
SUB Test1 (x%)
  PRINT x%;
  IF x% > 100 THEN Test2
END SUB
```

SUB2.BAS

```
SUB Test2 ()
  PRINT "Ich werde nie"
  PRINT "aufgerufen!"
END SUB
```

Wenn Sie nun mit den Befehlen

```
BC MAIN;
BC SUB1;
BC SUB2;
LIB PROZEDUR +SUB1+SUB2;
```

die Programme kompilieren und eine Library erzeugen und dann mit

```
LINK MAIN,,,PROZEDUR;
```

eine EXE-Programm erstellen, erhalten Sie ein lauffähiges Programm namens MAIN.EXE. Sie haben für jede Prozedur eine eigene OBJ-Datei angelegt und hoffen zu Recht, daß LINK nur einbindet, was wirklich benötigt wird. Leider trauen Sie LINK hier zuviel zu: Daß die Bedingung `IF x% > 100 THEN Test2` in der Prozedur Test1 niemals wahr wird und Test2 deshalb völlig überflüssig ist, kann LINK nicht erkennen und bindet deshalb SUB2.OBJ mit ein. Das würde selbst dann geschehen, wenn die Prozedur Test1 gar nicht selbst einen Aufruf an Test2 enthielte, sondern ein solcher Aufruf in einer nie aufgerufenen Prozedur namens „Dummy“ im Modul SUB1.BAS steckte.

Genau das, was wir hier im Kleinen „nachgespielt“ haben, ist auch das Problem bei vielen BASIC-internen Routinen. Wenn Sie zum Beispiel in Ihrem Programm den OPEN-Befehl mit einer Stringvariablen dahinter verwenden (`OPEN FileName$ FOR INPUT AS #1`), werden automatisch auch alle Routinen einge-

bunden, die zur Kommunikation über die serielle Schnittstelle erforderlich sind, denn *FileName$* könnte ja „COM1:9600,DS,CS,DT“ lauten – auch wenn das in Ihrem Programm definitv nie der Fall ist.

Wie funktionieren Verzicht-Dateien?

Nehmen wir an, Sie können oder wollen die Library PROZEDUR und erst recht den Quellcode von Test1 nicht ändern (weil Sie die Library auch noch mit anderen Programmen verwenden), möchten aber trotzdem verhindern, daß in unserem Beispielfall SUB2.OBJ eingebunden wird. Dann können Sie ein Verzicht-File schreiben. `SUB Test2: END SUB` ist alles, was darin stehen muß; nennen Sie es NOTEST2.BAS und kompilieren Sie es. Wenn Sie nun mit

```
LINK MAIN+NOTEST2,,,PROZEDUR;
```

linken, verhindern Sie, daß aus der Library PROZEDUR die Routine TEST2 eingebunden wird, weil LINK durch die „Leerprozedur“ gleichen Namens in NOTEST2 schon zufriedengestellt wird.

Solche Verzichtdateien sind natürlich auch für alle eingebauten BASIC-Routinen denkbar; sie machen aber nur dann Sinn, wenn Sie Programme erzeugen, die ohne Runtime-Modul arbeiten, oder wenn sie bei der Erstellung eines eigenen Runtime-Moduls angewandt werden. Ansonsten befinden sich alle BASIC-Routinen ja ohnehin im Runtime-Modul (VBDRT10.EXE in der Standardversion), und es würde das Programm nicht verkürzen, ein „Double“ einer solchen Prozedur als Leerprozedur einzuschließen.

Mitgelieferte Verzichtdateien

VBDOS enthält in der professionellen Ausgabe eine Anzahl von Verzichtdateien, die bestimmte Funktion unbenutzbar machen oder ihre Leistung verringern. Im Beispiel oben habe ich bereits die Unterstützung der seriellen Schnittstelle beim OPEN-Befehl erwähnt; sie kann mit NOCOM.OBJ abgeschaltet werden.

Die Standardausgabe enthält lediglich die Verzichtdatei SMALLER.OBJ. Unter Verwendung der Informationen in der Datei „VBDSTUB.TXT“ auf der Diskette und der Anleitung weiter hinten in diesem Kapitel können Sie sich jedoch beliebige Verzichtdateien selbst erzeugen. Hier folgt zunächst eine Liste der mitgelieferten Standard-Verzichtfiles:

File	*Wirkung*
NOEDIT	Die INPUT- und LINE INPUT-Anweisungen können, wenn sie zur Tastatureingabe benutzt werden, nur noch die Enter- und Backspace-Taste als Sondertasten verarbeiten, also sind keine Pfeiltasten und kein Einfügen mehr möglich. Einsparung: 1 KB
SMALLERR	Reduziert die Länge der Fehlermeldungen, die ausgegeben werden, wenn ein Fehler auftritt, der nicht von einer Fehlerbearbeitungsroutine abgefangen wird („Ungültiger Dateiname in Zeile...“ usw.); es werden nur noch Fehlernummern, keine Texte mehr ausgegeben. Die Funktion ERROR$ gibt ebenfalls nur noch Nummern zurück. Einsparung: 3 KB
NOCOM	Die seriellen Schnittstellen COM1: und COM2: können nicht mehr mit OPEN geöffnet werden. Wenn Sie weder OPEN "COM..." noch einen OPEN-Befehl mit einer Stringvariablen als Dateinamen benutzen, werden diese Routinen auch ohne Verzicht-File nicht eingebunden. Einsparung: 3 KB
NOLPT	Die parallelen Schnittstellen LPTx: können nicht mehr mit OPEN geöffnet werden; LPRINT, LPOS und die Möglichkeit, mit der PrtSc- oder Druck-Taste eine Hardcopy vom Bildschirm zu erzeugen, sind im Programm nicht mehr zugänglich. Wenn Sie weder LPRINT, LPOS, OPEN "LPT..." noch einen OPEN-Befehl mit einer Stringvariablen als Dateinamen im Programm haben, werden diese Routinen auch ohne Verzicht-File nicht eingebunden. Einsparung: 1 KB
NOFLTIN	INPUT, VAL und READ können nicht mehr auf Fließkommazahlen angewandt werden. Die Fließkommaeinheit wird dann nicht mehr dieser Befehle wegen benötigt. Der INPUT-Befehl akzeptiert dann keine Hexadezimal- und Oktalzahlen, keine Typenbezeichner und keine wissenschaftliche Zahlendarstellung mit E mehr. Einsparung: 2 KB unter normalen Umständen, 12 KB, wenn deshalb auf die Fließkommaarithmetik verzichtet werden kann
NOTRNEM	Entfernt die Unterstützung transzendenter Funktionen, im einzelnen: LOG, SQR, SIN, COS, TAN, ATN, EXP, ^ (Potenz), CIRCLE mit Anfangs- und/oder Endwinkel, DRAW mit A- oder T-Befehlen. Als einziges Verzicht-File ist dies keine OBJ-, sondern eine LIB-Datei; die Verwendung ist jedoch identisch. Einsparung: 1,5 KB selbst dann, wenn die genannten Funktionen nicht benutzt werden
TSCNIO	Sämtliche Grafikfunktionen werden nicht mehr unterstützt; außerdem werden Textzeichen, bevor sie auf den Bildschirm ausgegeben werden, nicht mehr daraufhin überprüft, ob es sich um Sonderzeichen (wie CHR$(12) = CLS) handelt. Einsparung: 3,5 KB, wenn das Programm keine Grafikbefehle enthält (sonst mehr)

File	*Wirkung*
NOGRAPH	Entfernt die gesamte Grafikunterstützung. NOGRAPH ist in TSCNIO bereits enthalten, und alle folgenden Grafik-Verzichtfiles sind in NOGRAPH bereits enthalten. Einsparung: Nur, wenn das Programm Grafikbefehle enthält
NOCGA	Entfernt Grafikunterstützung für SCREEN 1 und 2. Einsparung: 0,5 KB
NOHERC	Entfernt Grafikunterstützung für SCREEN 3. Einsparung: 0,5 KB
NOOGA	Entfernt Grafikunterstützung für SCREEN 4. Einsparung: 0,5 KB
NOEGA	Entfernt Grafikunterstützung für SCREEN 7–10. Einsparung: 2 KB
NOVGA	Entfernt Grafikunterstützung für SCREEN 11–13. Einsparung: 2 KB
NOEVENT	Kann nur für die Herstellung von Runtime-Modulen benutzt werden; entfernt die Unterstützung jeder Art von Event Trapping.
NOISAM	Kann nur für die Herstellung von Runtime-Modulen benutzt werden; entfernt die Unterstützung von ISAM.
NOFORMS	Kann nur für die Herstellung von Runtime-Modulen benutzt werden; entfernt die Unterstützung von Formen und Steuerelementen.

Verzichtdateien selbst erzeugen

„Achtung: Sie verlassen den Bereich der absturzfreien Programme!“ – Diese Warnung möchte ich vorausschicken, bevor ich fortfahre. Das Erstellen von Verzichtdateien in Eigenregie ist ein wenig mit Experimentieren verbunden und von Microsoft auch nicht geplant. Es ist nicht auszuschließen, daß Sie bei Ihren Versuchen das eine oder andere Mal den Rechner neu booten müssen, weil ein Programm „hängt“.

Ein dankbares Objekt für das Erstellen eigener Verzichtdateien sind die formbasierten Programme. Die gesamte „ereignisgesteuerte Programmierung“ wird nämlich en bloc in Ihr EXE-Programm eingebunden, auch dann, wenn Sie nur eine Form mit einer einzigen Schaltfläche verwenden.

Der erste Schritt zum Erzeugen von eigenen Verzichtdateien ist es, festzustellen, welche Teile aus der VBDOS-Library in das Programm unnötigerweise ein-

gebunden werden. Dazu brauchen Sie vor allem Intuition und ein paar Englischkenntnisse.

Verwenden Sie den Befehl `LIB VBDCL10.LIB,ROUTINEN.LST;`, um eine Liste aller Routinen in der wichtigsten VBDOS-Library zu erhalten. (Hängen Sie ein E an den Namen an, wenn Sie die Profi-Version benutzen.) Die Liste wird in die Datei ROUTINEN.LST geschrieben, die Sie sich gut aufheben sollten.

Nun beginnt die Suche nach Routinen, auf die man verzichten könnte. Die Liste enthält im ersten Teil alle Routinen alphabetisch sortiert, im zweiten Teil die Liste der Routinen pro OBJ-Datei in der Library. Nicht alle Namen sind so „sprechend“, daß man versteht, was sie bedeuten sollen. Ich fand in der Liste aber zum Beispiel folgenden Eintrag:

```
...
keysynth          Offset: 00015540H  Größe von Code und Daten: a5H
  B$ISALTKEYDOWN    B$SYNTHESIZESHIFTKEYS

listbox           Offset: 00015760H  Größe von Code und Daten: 2527H
  B$ADDLISTSTRING   B$ADDLISTSZ       B$DELETESZ        B$DISPLAYLISTBOX
  B$ENUMLISTBOXENTRIES                B$FILLLISTBOX     B$FLOCATEMATCH
  B$GETLISTBOXORIENTATION             B$GETLISTTEXT     B$GETONDEMAND
  B$HILITELISTSEL   B$INITLISTBOX     B$INITLISTBOXORIENTED
  B$INSERTSZ        B$LISTBOXWNDPROC  B$LPVDEREF        B$MOVEHORIZLEFT
  B$MOVESELECTION   B$MOVESELECTIONDOWN                 B$MOVESELECTIONLEFT
  B$MOVESELECTIONRIGHT                B$MOVESELECTIONUP
  B$REPLACESZ       B$RESETCONTENT    B$REVERTTOOOMLB   B$SCROLLHORIZLISTBOX
  B$SCROLLLISTBOX   B$SETLISTBOXORIENTATION             B$SETLISTDEFAULTS
  B$SETSCROLLWINDOW                   B$SETSZ

menucore          Offset: 00018ae0H  Größe von Code und Daten: 59H
  B$FENABLEMENUBAR
...
```

Ganz offensichtlich: Bei der OBJ-Datei LISTBOX handelt es sich um die Unterstützung für die Listenfelder, ob ihrer offensichtlichen Größe (2527h ≈9,5 KB) könnte sich hier eine Verzichtdatei lohnen. Nun folgt der zweite Schritt: Die Routinen werden probehalber der VBDOS-Library entfernt (vorher eine Sicherungskopie anfertigen). Der Befehl lautet in diesem Fall:

```
LIB VBDCL10 -LISTBOX;
```

(Wieder gilt, daß für die Profi-Version ein E angehängt werden muß.)

Nun kommt der Test. Nehmen Sie ein Programm, das kein Listenfeld verwendet (z.B. eine Form mit nur einer Befehlsschaltfläche), kompilieren und linken Sie sie wie gewohnt. Im Beispiel werden Sie folgende Meldungen erhalten (die

Meldungen erscheinen mehrmals, ich habe nur die verschiedenen übernommen):

```
Microsoft (R) Segmented Executable Linker  Version 5.31.009 Jul 27 1992
Copyright (C) Microsoft Corp 1984-1992.  Alle Rechte vorbehalten.

vbdcl10e.lib(..\forms\frmbrdr.asm) : Fehler L2029: 'B$LISTBOXWNDPROC' : Nicht
aufgelöste externe Symbole
vbdcl10e.lib(user\dlgutil.c) : Fehler L2029: 'B$GETLISTTEXT' : Nicht aufgelöste
externe Symbole
vbdcl10e.lib(..\forms\frmmove.asm) : Fehler L2029: 'B$INITLISTBOX' : Nicht
aufgelöste externe Symbole

3 Fehler wurden entdeckt
```

Das bedeutet, daß LINK tatsächlich versucht hat, das von uns gewaltsam entfernte File einzubinden, obwohl das Programm kein Listenfeld verwendet. Hätten wir eine Datei entfernt, die LINK ohnehin nicht einbindet, wären keine Fehler aufgetreten – ein Verzichtfile zu programmieren, hätte dann keinen Sinn.

Sie sehen, daß die drei gennanten „nicht aufgelösten externen Symbole“ Namen von Prozeduren aus der Datei LISTBOX.OBJ sind (vgl. Listing auf Seite 391). Diese drei werden also irgendwo erwähnt, obwohl wir kein Listenfeld verwenden. Wenn sie wirklich *aufgerufen* werden, dann wird das Programm wahrscheinlich abstürzen, wenn wir sie durch Leerprozeduren ersetzen. Es ist aber sehr wahrscheinlich, daß sie nicht aufgerufen, sondern bloß *erwähnt* werden – diesen Unterschied habe ich im Beispiel auf Seite 387 zu verdeutlichen versucht, in dem die Prozedur Test2 zwar erwähnt, aber nicht aufgerufen wurde.

Nun müssen wir nur noch eine OBJ-Datei erzeugen, die die drei Prozeduren B$LISTBOXWNDPROC, B$GETLISTTEXT und B$INITLISTBOX als Leerprozeduren zur Verfügung stellt. Das ist leider nicht so einfach, weil VBDOS keine Sonderzeichen (wie $) im Prozedurnamen erlaubt. Doch wo ein Wille ist, ist auch ein Weg...

Das STUBCOMP-Programm

Ich habe ein Programm geschrieben, daß es Ihnen auf komfortable Weise ermöglicht, beliebige Verzichtdateien selbst zu erzeugen: Quelltext und ausführbare Datei (STUBCOMP.FRM, STUBCOMP.EXE) finden Sie auf der Diskette. Ich werde seine Verwendung kurz am Beispiel des geplanten Verzichtfiles für die Listbox erklären:

Abbildung 23–1: STUBCOMP bei der Erzeugung von NOLISTBX.OBJ

Nachdem Sie im Feld „Name des Vezicht-Files“ einen beliebigen Namen eingetragen haben, können Sie im Textfeld „auszuschließende Prozeduren“ eine Liste von Prozedurnamen eintragen, die das Verzicht-File als Leerprozeduren enthalten soll. Dabei können Sie mit ' beginnend Kommentare einfügen. Überdies ist wählbar, ob das Verzicht-File wirklich Leerprozeduren („Absturz“) enthalten soll, oder ob es einen SOUND-, einen SYSTEM- oder einen ERROR-Befehl ausführen soll. Im letztgenannten Fall können Sie eine Fehlernummer angeben, um z. B. alle Ihre Verzicht-Files den Fehler Nr. 20.000 erzeugen zu lassen, wenn die ausgeschlossenen Prozeduren wider Erwarten doch aufgerufen werden.

STUBCOMP speichert die Einstellungen stets in Dateien mit der Extension „.STB“; wenn Sie die oben gezeigte Eintragung für NOLISTBX.OBJ machen, wird also eine Datei namens NOLISTBX.STB angelegt. Wenn Sie beim nächsten Mal in das Namensfeld „NOLISTBX“ eingeben, lädt STUBCOMP automatisch die STB-Datei, so daß Ihre letzten Einstellungen immer verfügbar sind.

Klicken Sie auf „Erzeugen“, dann ruft STUBCOMP den Compiler BC.EXE auf, um die gewünschte OBJ-Datei mit den Leerprozeduren zu erzeugen. Dabei werden, weil BC ja keine Sonderzeichen im Prozedurnamen erlaubt, alle Prozeduren zunächst in garantiert erlaubte Namen umbenannt, und wenn BC.EXE die OBJ-Datei erstellt hat, ersetzt STUBCOMP diese harmlosen Namen durch die, die ursprünglich gefordert waren.

Zurück zum Beispiel

Mit STUBCOMP.EXE ist es nun kein Problem, die Verzichtdatei NOLISTBX.OBJ (der Sie natürlich auch einen beliebigen anderen Namen geben können) zu erzeugen. Es folgt der letzte Test: Linken Sie dieselbe Datei, die Sie zuvor zum Ausprobieren verwendet haben, diesmal unter Verwendung der vollständigen Library VBDCL10.LIB (und nicht der von uns um LISTBOX.OBJ verkürzten), einmal mit und einmal ohne Verzichtdatei...

```
LINK TEST;
LINK TEST+NOLISTBX;
```

... und vergleichen Sie die Größen der entstehenden EXE-Dateien. Im Beispiel werden Sie feststellen, daß durch die Verwendung von NOLISTBX.OBJ das EXE-Programm tatsächlich um 10 KB kürzer wird, obwohl es von seiner Funktionsfähigkeit nichts einbüßt.

Fehlerquellen

Falls Sie stattdessen bei Verwendung Ihres selbstgebastelten Verzichtfiles Fehlermeldungen wie „Symbol mehrfach definiert“ erhalten, dann müssen Sie davon ausgehen, daß die ursprüngliche OBJ-Datei aus der Library doch noch eingebunden wurde und die von Ihnen als „Leerprozeduren“ zur Verfügung gestellten Prozeduren sich damit ins Gehege kommen. In unserem Beispiel würde das passieren, wenn aus irgendeinem Grunde etwa die Routine B$ADDLISTSZ aufgerufen würde. Diese ist in LISTBOX.OBJ enthalten, nicht aber in unserer Verzicht-Datei. Also muß LISTBOX.OBJ eingebunden werden, und die drei in der Verzicht-Datei enthaltenen Namen sind dann doppelt definiert. Abhilfe würde man hier schaffen, indem man auch B$ADDLISTSZ in die Verzicht-Datei schreibt. Schlimmstenfalls muß man alle in LISTBOX.OBJ enthaltenen Routinen als Leer-Doubles in NOLISTBX.OBJ einschließen.

Wenn Routinen, die auf diese Weise mit eigenen Verzicht-Files „gekappt“ wurden, trotzdem aufgerufen werden, stellt das eine große Gefahr dar. Wir können zwar die Prozedurnamen sehen und in das Verzicht-File übernehmen, wir wissen aber nicht, wieviele und welche Parameter der Prozedur – zum Beispiel B$INITLISTBOX – übergeben werden. Wird sie nun aufgerufen und, da leer, sofort wieder beendet, liegen immer noch die übergebenen Parameter auf dem Stack. Normalerweise würden sie dort nach dem Prozeduraufruf gelöscht; das passiert hier jedoch nicht. Dadurch wird der Stack schneller voll, und andere Prozeduren kommen u.U. durcheinander, weil der Stack nicht das enthält, was sie erwarten.

Daher bietet STUBCOMP die Möglichkeit an, einen SOUND- oder SYSTEM-Befehl ausführen zu lassen (Achtung: SOUND benötigt die Fließkommaarithmetik). So merken Sie sofort, wenn eine der von Ihnen ausgeschlossenen Prozeduren aufgerufen wird, und können auf ihren Ausschluß verzichten.

Nachbildung der Verzicht-Files für die Standardversion

Anhand der Informationen in der Datei VBDSTUB.TXT auf der Diskette können Sie, wenn Sie nur die Standardversion besitzen, einige der Profi-Verzichtdateien nachbilden. Dabei handelt es sich um NOCOM, NOLPT, NOGRAPH, NOVGA, NOEGA, NOCGA, NOHERC, NOOGA, NOFLTIN und NOTRNEM. Beachten Sie jedoch, daß diese selbsterzeugten Verzichtdateien nicht, wie die mitgelieferten der Profi-Version, „geordnete Fehler“ erzeugen, sondern u. U. zu einem Absturz führen, wenn das Funktionsmerkmal, das sie entfernen, dennoch benutzt wird.

23.4 Käufliche Performance

Bereits heute gibt es eine Vielzahl von Hilfsprogrammen und Toolboxen, die das Programmieren mit VBDOS vereinfachen, schneller oder effizienter machen sollen. Darunter sind zum Beispiel weitere Steuerelemente, Sammlungen von Routinen für die Grafikverarbeitung, das Ansteuern bestimmter Drucker, das Programmieren speicherresidenter Programme und vieles mehr.

Ich kann und will Ihnen hier keine Übersicht über den Markt vermitteln – das versucht Microsoft mit einer dem VBDOS beigelegten Broschüre –, sondern möchte Ihnen lediglich einige Ratschläge für die Auswahl geben.

Benutzerschnittstelle

Die meisten Schnittstellen-Toolboxen haben mit der hervorragenden ereignisgesteuerten Programmierung von VBDOS Ihre Berechtigung verloren. Es gibt bereits neue Toolboxen, die auf dem Formenkonzept aufbauen und zusätzliche Steuerelemente zur Verfügung stellen; falls Sie eventuell später einmal Ihre Programme nach VB für WINDOWS portieren möchten, sollten Sie darauf achten, daß der Hersteller entsprechende Routinen auch für VB für WINDOWS anbietet.

Wenn Sie unbedingt eine Benutzeroberfläche im Grafikmodus realisieren möchten, steht Ihnen das ereignisgesteuerte Konzept nicht zur Verfügung. Fast alle Toolboxen arbeiten hier mit „Polling“, das heißt, daß Ihr Programm in regel-

mäßigen Abständen bei der Toolbox „nachfragt“, ob etwas passiert ist. Dieses Konzept ist zwar leichter zu beherrschen, bietet aber nicht den leistungsfähigen, objektorientierten Ansatz von VBDOS. Sie werden schwerlich eine Toolbox finden, die es ermöglicht, so leicht zwischen Formgestaltung und Programmierung umzuschalten, wie VBDOS und der Form-Designer es tun.

Es gibt allerdings Toolboxen, die mit dem Form-Designer erstellte Formen einlesen und in ihr eigenes Format umsetzen können; so bleibt der Form-Designer als Entwurfswerkzeug erhalten.

Achtung, Far Strings!

Die meisten Toolboxen, die für QuickBASIC konzipiert wurden, sind mit VBDOS nicht lauffähig, weil VBDOS im Gegensatz zu QuickBASIC die sogenannten „Far Strings“ verwendet. Überzeugen Sie sich, daß die Toolbox, die Sie verwenden wollen, wirklich VBDOS-kompatibel ist.

Ausprobieren umsonst

Ein Toolbox-Hersteller kann Ihnen gefahrlos seine Toolbox als Quick Library (QLB) überlassen, da es keine Möglichkeit gibt, aus einer Quick Library eine normale Library zu erstellen. Sie werden die Quick Library zwar in VBDOS verwenden können, es wird Ihnen aber nicht möglich sein, ein EXE-Programm zu erstellen. Viele Toolbox-Anbieter zeigen sich denn auch bereit, die Quick Library quasi als „Demo-Version“ herauszurücken, so daß Sie den Leistungsumfang eines Produktes prüfen können, bevor Sie kaufen.

P.D.Q.

Ein einziges Produkt muß ich ausnahmsweise hier namentlich erwähnen, und will es mit einer kleinen Hommage verbinden: Vielleicht haben Sie den Namen Ethan Winer schon einmal in Zusammenhang mit BASIC gelesen. Ethan Winer ist *der* BASIC-Experte schlechthin, Verfasser von zahlreichen Büchern und Artikeln, Co-Autor für das „PC Magazine“ und das „BASIC Pro Magazine“ und, um auf P.D.Q. zurückzukommen, der Boß von Crescent Software, einer traditionsreichen Goldgrube für BASIC-Programmierer. Neben diversen Toolboxen bietet Crescent Software eine Library namens „P.D.Q.“ an (dreimal dürfen Sie raten, wer sie programmiert hat). P.D.Q. ist quasi ein Ersatz für die Standard-Routinen von VBDOS und wird anstelle der Library VBDCL10.LIB gelinkt. Programme, die mit P.D.Q. gelinkt wurden, sind wesentlich kürzer als die Ori-

ginale; mit P.D.Q. können Sie lauffähige, eigenständige BASIC-Programme schreiben, deren EXE-Version unter 1 KB groß ist.

P.D.Q. unterstützt zwar bei weitem nicht alle BASIC-Funktionen (in der letzten mir bekannten Version keine Formen und Steuerelemente, keine Grafik, keine Fließkommaberechnungen), läßt sich aber durch zusätzliche Libraries ausbauen und ist eben gerade für die Erzeugung kleiner Utility-Programme ein Muß.

Die Crescent Software-Produkte werden in Deutschland durch die Firma Zoschke Data vertrieben, sind aber auch im größeren Versandhandel erhältlich.

WINDOWS-Kompatibilität 24

24.1 Benutzeroberfläche

Der augenfälligste Unterschied zwischen den Plattformen „DOS“ und „WINDOWS“ ist die Benutzeroberfläche. WINDOWS läuft vollkommen im Grafikmodus, während VBDOS sich auf den Textmodus beschränkt. Allein diese Tatsache sorgt für eine Anzahl von Unterschieden:

Maßeinheit

In VBDOS werden alle Positionen, Höhen und Breiten in Zeichen (d. h. Zeilen und Spalten) gemessen. In WINDOWS wird stattdessen zumeist mit „Twips“ (1440 Twips = 1 Zoll) gearbeitet, obwohl auch andere Maßeinheiten möglich sind.

Bei der Umrechnung von Werten zwischen VBDOS und WINDOWS wird angenommen, daß jedes Zeichen 240 Twips (Höhe) mal 120 Twips (Breite) mißt; da in WINDOWS jedoch meist proportionale Schriftarten verwendet werden, kann das nur ein durchschnittlicher Wert sein.

Verarbeitung von Grafiken

In VB für WINDOWS können Bildfelder Grafiken enthalten, und auch die Spezialobjekte PRINTER und CLIPBOARD sind grafikfähig.

VB für WINDOWS unterstützt keinen der Grafikbefehle von VBDOS, weil es auch keinen speziellen „Grafikmodus“ mehr gibt. Stattdessen gibt es zusätzliche Methoden, die auf Bildfelder und Formen angewandt werden können, so zum Beispiel eine LINE- oder eine CIRCLE-Methode.

WINDOWS-Systemsteuerung ersetzt eigene Parameterdateien

Viele Angaben, die für alle WINDOWS-Programme gelten, werden unter WINDOWS mit der „Systemsteuerung“ eingestellt, so zum Beispiel die Bildschirmfarben, der verwendete Drucker und die Länderkennung. Solche Einstellungen (auf die in VBDOS z. T. über die *ControlPanel*-Eigenschaft des SCREEN-Objekts zugegriffen wird) müssen in Programmen mit VB für WINDOWS nicht mehr ermöglicht werden.

Ereignisgesteuerte Programmierung

Was in VBDOS noch eine Option war – nämlich die Verwendung von Formen und Objekten zur Ein- und Ausgabe von Daten – wird in VB für WINDOWS zum Zwang. Sie können die „normalen“ Ein-/Ausgabebefehle INPUT, LINE INPUT, INPUT$, INKEY$, WRITE, PRINT, CLS, LOCATE, POS, CSRLIN, COLOR, FILES u.ä. nicht mehr verwenden, sondern sind auf Formen und Methoden angewiesen.

Mehr Methoden, Steuerelemente und Eigenschaften

VB für WINDOWS bietet für die meisten Steuerelemente zusätzliche Eigenschaften und einige zusätzliche Steuerelemente und Methoden. Alle Steuerelemente, die Text anzeigen, können mit einer eigenen Schriftart und -größe versehen werden; Bildfelder (hier gibt es zwei Steuerelemente, „PictureBox“ und „Image“) können echte Bilder enthalten, die auch skalierbar sind; auf Symbolgröße verkleinerte Formen können wirklich ein Symbol anzeigen.

24.2 WINDOWS-bedingte Änderungen

Eine weitere Gruppe von Veränderungen folgt ebenfalls logisch aus dem Konzept, das WINDOWS zugrunde liegt:

Datenaustausch mit anderen Programmen

Über das Spezialobjekt CLIPBOARD – die „Zwischenablage“ – können in VB für WINDOWS Daten mit anderen Programmen ausgetauscht werden. Während Sie in VBDOS davon ausgehen können, daß CLIPBOARD immer das enthält, was Sie zuletzt hineingeschrieben haben, ist es unter WINDOWS immer möglich, zwischendrin ein anderes Programm zu starten und mit diesem den CLIPBOARD-Inhalt zu manipulieren.

WINDOWS unterstützt mehr Farben

WINDOWS liegt grundsätzlich ein Farbmodell mit 256^3 Farben zugrunde, die, wenn die Grafikkarte nicht so viele unterstützt, automatisch durch Mischen erzeugt werden. VB für WINDOWS bietet eine Auswahl hieraus an; in VB für WINDOWS gibt die Funktion QBCOLOR den Wert der VB-WINDOWS-Farbe zurück, der am nächsten an der übergebenen DOS-Farbe liegt. Wenn Sie also in Ihrem DOS-Programm statt der Farbe „7“ (für weiß) immer „QBCOLOR (7)“

schreiben, ist ein Teil WINDOWS-Kompatibilität schon eingebaut. Das Programm wird dadurch nur unbedeutend länger.

Systemnahe Programmierung

Wenn Sie ein Programm von VBDOS nach VB für WINDOWS übertragen wollen, ist alles schadhaft, was in irgendeiner Weise DOS-Spezialitäten ausnutzt oder direkt auf den Speicher zugreift; dazu zählen BLOAD, BSAVE, DEF SEG, PEEK, POKE, VARPTR, VARSEG, SSEG, SADD, INP, OUT, WAIT und Interruptaufrufe – um nur einige zu nennen. Ähnliche Funktionen lassen sich in WINDOWS oft durch Aufrufe der sogenannten „WINDOWS API-Funktionen“ (API = Application Program Interface) bewerkstelligen.

Hierzu muß ich Sie auf die WINDOWS-Fachliteratur oder die Systemprogrammiererhandbücher von Microsoft (berühmt-berüchtigt sind die vier Bände des „Software Development Kit“) verweisen. Zugleich erlaube ich mir eine Warnung: Wenn Sie den sicheren Bereich von VB für WINDOWS verlassen und in die Tiefen der WINDOWS-API abtauchen, werden Sie eine Anzahl von Überraschungen erleben. Sie werden mit seltsamen und umständlichen Datenstrukturen, unverständlichen Funktionsaufrufen und einem Haufen „allgemeiner Schutzverletzungen“ konfrontiert werden, die ohne langes Studium Ihre Produktivität eher hemmen als fördern.

Tonerzeugung

VB für WINDOWS erzeugt beim „BEEP“-Befehl das Windows-Standardtonsignal, das der Benutzer selbst über die Systemsteuerung auswählen kann. SOUND und PLAY werden nicht unterstützt; stattdessen verwendet VB für WINDOWS weitaus mächtigere Befehle zur Geräuscherzeugung, mit der Sie auch Soundkarten ansteuern und Samples (aufgenommene Text- oder Musiksequenzen) abspielen können.

Serielle Schnittstelle

Sofern Sie die serielle Schnittstelle zur Ansteuerung eines Druckers oder Plotters verwendet haben, läuft das jetzt über die Druckersteuerung (PRINTER-Objekt). Verwenden Sie die serielle Schnittstelle zum Datenaustausch, dann können Sie ab VB für WINDOWS Version 2.0 das Spezialobjekt MSCOMM nutzen, mit dem die Sache zum Kinderspiel wird.

24.3 Sonstige Unterschiede

Eine dritte Gruppe von Unterschieden zwischen VBDOS und VB für WINDOWS läßt sich nicht einfach mit der geänderten Oberfläche oder der WINDOWS-Struktur begründen, sondern muß (wohl oder übel) einfach akzeptiert werden:

Kein $INCLUDE

VB für WINDOWS arbeitet nicht mit dem Metabefehl $INCLUDE, sondern verwaltet für jedes Projekt ein sogenanntes „globales Modul“, in dem alle Deklarationen gespeichert werden, die für jedes Modul im Projekt gültig sein sollen.

VB für WINDOWS ist kein Compiler

Während VBDOS eigenständige EXE-Programme erzeugen kann, benötigen alle Programme, die mit VB für WINDOWS geschrieben werden, ein Laufzeitmodul (VBWIN200.DLL). In Wahrheit enthält ein von VB für WINDOWS erzeugtes EXE-Programm noch nicht einmal lauffähigen Code, sondern einen sogenannten „Threaded P-Code“, der zur Laufzeit durch VBWIN200.DLL gelesen und interpretiert wird.

Dadurch sind Programme, die mit VB für WINDOWS erzeugt werden, tendenziell langsamer als ihre DOS-Pendants. Da WINDOWS allerdings ein völlig anderes Speichermodell zugrunde liegt und außerdem konsequenter auf die erweiterten Möglichkeiten des 80286-Prozessors (auf 8086/8088 läuft WINDOWS 3.1 nicht mehr) zurückgegriffen wird, läßt sich allerdings keine pauschale Aussage über die Geschwindigkeit machen.

VB für WINDOWS erlaubt keinen Code auf Modulebene

Ebenso wie schon bei den Formmodulen in VBDOS dürfen in VB für WINDOWS weder Form- noch Codemodule ausführbare Befehle auf der Modulebene haben; das gesamte Programm muß also in Prozeduren und Funktionen „verpackt“ werden. VB für WINDOWS unterstützt nur noch ON LOCAL ERROR, nicht mehr ON ERROR.

Wenn in VB für WINDOWS die Startdatei ein Codemodul ist, wird dort eine Prozedur namens MAIN erwartet, die beim Start des Programms aufgerufen wird; ist die Startdatei ein Formmodul, dann wird – wie auch in VBDOS – sofort die darin gespeicherte Form geladen.

Fehlende ISAM-Unterstützung

VB für WINDOWS unterstützt ISAM auch in der professionellen Version nicht. Sie müssen, wenn Sie darauf Wert legen, von Drittanbietern Toolboxen zur ISAM-Unterstützung erwerben. Eine Alternative ist allerdings die sogenannte „ODBC“ (Open Database Connectivity), mit der Sie von Ihrem VB-Programm eine Verbindung zu einem WINDOWS-Datenbankprogramm aufbauen und dessen Fähigkeiten wie eingebaute Routinen nutzen können. Allerdings unterstützen bisher nur wenige WINDOWS-Datenbanken (allen voran Microsoft Access) die ODBC.

Mehr TYPE-Möglichkeiten

VB für WINDOWS erlaubt es, in einem selbstdefinierten Datentyp (TYPE...END TYPE) auch Strings mit variabler Länge unterzubringen. Dadurch haben selbstdefinierte Datentypen nicht mehr – wie in VBDOS – immer die gleiche Länge.

Anderes Vorgehen bei DIM und COMMON

In VB für WINDOWS wird für die Dimensionierung von Variablen das Schlüsselwort GLOBAL verwendet, wenn sie allen Modulen zugänglich sein sollen (bei VBDOS: COMMON SHARED evtl. in Verbindung mit DIM). Außerdem wird die Tatsache, ob ein Array dynamisch oder statisch ist, beim DIM- bzw. GLOBAL-Befehl direkt festgelegt. Statisch wird ein Array, dessen Grenzen in Form von Konstanten beim ersten DIM oder GLOBAL festgelegt sind; dynamisch wird es, wenn anstelle der Grenzen beim ersten DIM oder GLOBAL einfach eine leere Klammer angegeben wird.

Andere Speicherverwaltung

VB für WINDOWS verwaltet den Speicher etwas anders als VBDOS. Die wichtigsten Unterschiede sind die maximale Stringlänge von 64 KB (statt 32 KB in VBDOS) und die Obergrenze für Strings variabler Länge von ebenfalls 64 KB pro Modul (in VBDOS konnte man mit Tricks bis zu 256 KB verwenden, vgl. Kapitel 9). Allerdings erlaubt VB für WINDOWS dynamische Datenfelder, die den gesamten verfügbaren Speicher (also mehrere MB) ausfüllen, während VBDOS hier auf 640 KB abzüglich der Programmgröße beschränkt ist.

Diverse Sprachunterschiede

Die sonstigen Unterschiede in der BASIC-Sprache sind sehr gering. Die SWAP-Anweisung existiert nicht mehr; USING kann bei PRINT nicht mehr verwendet werden (dafür aber FORMAT$), SYSTEM muß durch END ersetzt werden, und SHELL hat eine etwas andere Funktionalität.

24.4 Automatisches Übertragen von Programmen

Im Lieferumfang von VBDOS befinden sich die Programme FT.EXE und TRNSLATE.EXE. Letzteres ist nur unter WINDOWS lauffähig und benötigt außerdem installierte Versionen von VBDOS und VB für WINDOWS.

TRNSLATE kann automatisch ein Projekt mit allen zugehörigen Dateien von VBDOS nach VB für WINDOWS oder umgekehrt übersetzen. Die Resultate müssen natürlich bei allen größeren Programmen nachgebessert werden; dennoch ist TRNSLATE zumindest als „Roh-Übersetzer“ durchaus brauchbar.

Alle oben beschriebenen Sprachunterschiede, die so klar sind, daß sie automatisch verarbeitet werden können – darunter z. B. die Umwandlung von DIM und COMMON, das Kopieren von INCLUDE-Dateien in ein globales Modul oder das Umwandeln des Modulcodes der Startdatei in eine Main-Prozedur – berücksichtigt TRNSLATE. Alle Problemfälle können dabei in eine Log-Datei eingetragen werden, so daß die Nachbearbeitung von Hand leichter fällt.

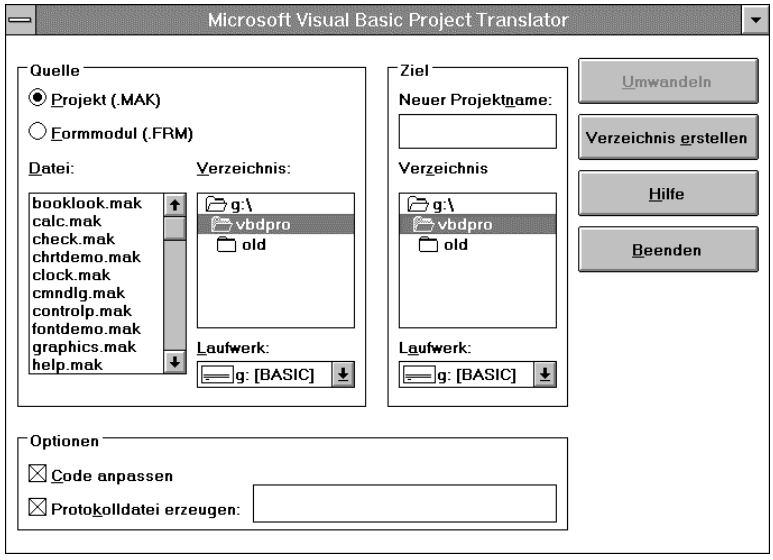


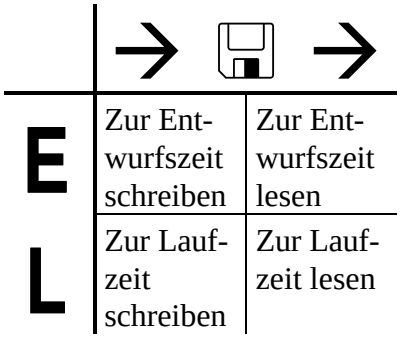
Abbildung 24–1: TRNSLATE.EXE

Das Programm FT.EXE übersetzt Formen von VBDOS nach VB für WINDOWS und umgekehrt. FT übersetzt nur das Aussehen der Form, nicht aber die Ereignisprozeduren, und wird von TRNSLATE aufgerufen.

Objekt-Referenz 25

Eigenschaften

Bei der Verwendung der Eigenschaften von Objekten ist es notwendig, zu wissen, ob diese zur Entwurfs- und Laufzeit oder nur in einer der beiden Situationen verfügbar sind und ob man die Werte jeweils nur lesen oder auch schreiben kann. Hierzu verwende ich eine kleine Symboltabelle:



Die in den vier Sektoren jeweils eingetragenen Symbole bedeuten:

Symbol	*Bedeutung*
⊕	ist möglich
⊖	ist nicht möglich
⊛	ist nur bei einigen Objekten oder nur eingeschränkt möglich (genaue Erläuterungen im Text)

Ereignisse

Die Ereignis-Einträge sprechen im wesentlichen für sich selbst; das Feld „Argumente“ benötigt möglicherweise eine kleine Erläuterung. Es listet auf, welche Argumente an eine Ereignisprozedur für das beschriebene Ereignis übergeben werden. Eine Angabe wie `Quelle AS CONTROL, X AS SINGLE, Y AS SINGLE` beim *DragDrop*-Ereignis bedeutet also, daß der zugehörige Prozedurkopf etwa so aussieht:

```
SUB Feld_DragDrop(Quelle AS CONTROL, X AS SINGLE, Y AS SINGLE),
```

wobei anstelle des Objektnamens *Feld* und der Variablennamen *Quelle*, *X* und *Y* natürlich beliebige andere Werte eingesetzt werden können.

Falls sich ein Objekt in einem Steuerelemente-Array befindet, wird vor allen hier angegebenen Argumenten noch ein Parameter `Index AS INTEGER` übergeben, der bezeichnet, um welches Element des Arrays es sich handelt. Auch da, wo ich unter „Argumente“ „keine“ angegeben habe, muß bei Steuerelemente-Arrays dieses eine Argument übergeben werden.

Aber keine Sorge: Wenn Sie in VBDOS mit der F12-Taste arbeiten, werden die Prozedurköpfe für Sie automatisch richtig gestaltet.

Methoden

Bei den Methoden ist unter „Verwendung“ die Aufrufsyntax in der Form `objekt.ADDITEM element$ [, position]` (Beispiel) angegeben.

Behalten Sie dabei im Hinterkopf, daß Sie, wenn Sie aus einem anderen Form- oder Codemodul heraus auf ein Objekt zugreifen wollen, den Namen der Form voranstellen müssen, also etwa `form.objekt.ADDITEM element$ [, position]`.

Ebenso können Sie die Bezeichnung *objekt* weglassen, wenn die Methode auf die Form des Formmoduls, in dem der Aufruf steht, angewendet werden soll: `MOVE links [, oben [, breite [, höhe]]]`.

Befehle, Funktionen, Sonstiges

Außer Eigenschaften, Ereignissen und Methoden braucht man wenig für die ereignisgesteuerte Programmierung. Was an gewöhnlichen Befehlen und Funktionen bzw. Metabefehlen hinzugekommen ist, läßt sich fast an einer Hand abzählen: Es sind DOEVENTS (Funktion), LOAD/UNLOAD (Befehle), $FORM (Metabefehl), MSGBOX (Befehl und Funktion), IF TYPEOF (Befehl) und INPUTBOX$ (Befehl und Funktion).

Verwendbarkeit

Die einzelnen Methoden, Eigenschaften und Ereignisse sind jeweils nur für einige der Objekte anwendbar. Die Tabelle auf den folgenden Seiten zeigt den Funktionsumfang der Objekte (● = wird unterstützt).

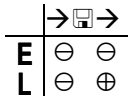
Objekt	Bezeichnung	Bildfeld	Bildlaufleisten	Dateiliste	Kombinationsfeld	Kontrollfeld	Laufwerksliste	Listenfeld	Menüeintrag	Optionsfeld	Schaltfläche	Rahmen	Textfeld	Timer	Verzeichnisliste	Form	CLIPBOARD	PRINTER	SCREEN
Methoden																			
ADDITEM					●			●											
CLEAR																	●		
DRAG	●	●	●	●	●	●	●	●		●	●	●	●		●				
ENDDOC																		●	
GETTEXT																	●		
HIDE																●			●
MOVE	●	●	●	●	●	●	●	●		●	●	●	●		●	●			
NEWPAGE																		●	
PRINT		●														●		●	
PRINTFORM																●			
REFRESH	●	●	●	●	●	●	●	●		●	●	●	●		●	●			
REMOVEITEM					●			●											
SETFOCUS		●	●	●	●	●	●	●		●	●		●		●	●			
SETTEXT																	●		
SHOW																●			●
TEXTHEIGHT		●														●			
TEXTWIDTH		●														●			
Eigenschaften																			
ActiveControl																			●
ActiveForm																			●
Alignment	●																		
Archive				●															
AutoRedraw		●														●			
AutoSize	●																		
BackColor	●	●		●	●	●	●	●		●	●	●	●		●	●			
BorderStyle	●	●											●			●			
Cancel											●								

Objekt	Bezeichnung	Bildfeld	Bildlaufleisten	Dateiliste	Kombinationsfeld	Kontrollfeld	Laufwerksliste	Listenfeld	Menüeintrag	Optionsfeld	Schaltfläche	Rahmen	Textfeld	Timer	Verzeichnisliste	Form	CLIPBOARD	PRINTER	SCREEN
Eigenschaften (Forts.)																			
Caption	●					●			●	●	●	●				●			
Checked									●										
ControlBox																●			
ControlPanel																			●
CtlName	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●				
CurrentX		●														●			
CurrentY		●														●			
Default											●								
DragMode	●	●	●	●	●	●	●	●		●	●	●	●		●				
Drive							●												
Enabled	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●			
FileName				●															
ForeColor	●	●		●	●	●	●	●		●	●		●		●	●			
FormName																●			
Height	●	●	●	●	●	●	●	●		●	●	●	●		●	●			●
Hidden				●															
Index	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●				
Interval														●					
LargeChange			●																
Left	●	●	●	●	●	●	●	●		●	●	●	●		●	●			
List				●	●		●	●							●				
ListCount				●	●		●	●							●				
ListIndex				●	●		●	●							●				
Max			●																
MaxButton																●			
Min			●																
MinButton																●			

Objekt	Bezeichnung	Bildfeld	Bildlaufleisten	Dateiliste	Kombinationsfeld	Kontrollfeld	Laufwerksliste	Listenfeld	Menüeintrag	Optionsfeld	Schaltfläche	Rahmen	Textfeld	Timer	Verzeichnisliste	Form	CLIPBOARD	PRINTER	SCREEN
Eigenschaften (Forts.)																			
MousePointer	●	●	●	●	●	●	●	●		●	●	●	●		●	●			●
MultiLine														●					
Normal				●															
Parent	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●			
Path				●											●				
Pattern				●															
PrintTarget																		●	
ReadOnly				●															
ScaleHeight		●														●			
ScaleWidth		●														●			
ScrollBars													●						
SelLength					●								●						
SelStart					●								●						
SelText					●								●						
Separator									●										
SmallChange			●																
Sorted					●			●											
Style					●														
System				●															
TabIndex	●	●	●	●	●	●	●	●		●	●	●	●		●				
TabStop		●	●	●	●	●	●	●		●	●		●		●				
Tag	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●			
Text					●			●					●						
Top	●	●	●	●	●	●	●	●		●	●	●	●		●	●			
Value			●			●				●	●								
Visible	●	●	●	●	●	●	●	●	●	●	●	●	●		●	●			
Width	●	●	●	●	●	●	●	●		●	●	●	●		●	●		●	●

Objekt	Bezeichnung	Bildfeld	Bildlaufleisten	Dateiliste	Kombinationsfeld	Kontrollfeld	Laufwerksliste	Listenfeld	Menüeintrag	Optionsfeld	Schaltfläche	Rahmen	Textfeld	Timer	Verzeichnisliste	Form	CLIPBOARD	PRINTER	SCREEN
Eigenschaften (Forts.)																			
WindowState																●			
Ereignisse																			
Change	●		●		●		●						●		●				
Click	●	●		●	●	●		●	●	●	●				●	●			
DblClick	●	●		●	●			●		●	●					●			
DragDrop	●	●	●	●	●	●	●	●		●	●	●	●		●	●			
DragOver	●	●	●	●	●	●	●	●		●	●	●	●		●	●			
DropDown					●														
GotFocus		●	●	●	●	●	●	●		●	●		●		●	●			
KeyDown		●	●	●	●	●	●	●		●	●		●		●	●			
KeyPress		●	●	●	●	●	●	●		●	●		●		●	●			
KeyUp		●	●	●	●	●	●	●		●	●		●		●	●			
Load																●			
LostFocus		●	●	●	●	●	●	●		●	●		●		●	●			
MouseDown	●	●		●				●							●	●			
MouseMove	●	●		●				●							●	●			
MouseUp	●	●		●				●							●	●			
Paint		●														●			
PathChange				●															
PatternChange				●															
Resize																●			
Timer														●					
UnLoad																●			

ActiveControl (Eigenschaft)



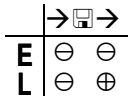
Objekte SCREEN

Datentyp CONTROL

Nutzen *ActiveControl* gibt das Steuerelement zurück, das gerade aktiv ist (also den Fokus hat). Ist kein Steuerelement aktiv, wird ein Fehler 430 generiert.

Siehe auch GotFocus (431), LostFocus (441).

ActiveForm (Eigenschaft)



Objekte SCREEN

Datentyp FORM

Nutzen *ActiveForm* gibt die Form zurück, die gerade aktiv ist (also entweder selbst den Fokus hat oder ein Steuerelement besitzt, das ihn gerade hat). Ist keine Form aktiv, wird ein Fehler 431 generiert.

Siehe auch GotFocus (431), LostFocus (441).

ADDITEM (Methode)

Objekte Kombinationsfeld, Listenfeld

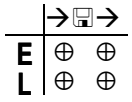
Verwendung `objekt.ADDITEM` *`element$`* `[,` *`position`*`]`

Nutzen Mit ADDITEM können Sie Elemente an eine Liste anfügen. Standardmäßig wird das neue Element *element$* am Ende der Liste angehängt oder, wenn die *Sorted*-Eigenschaft TRUE ist, alphabetisch einsortiert. Wenn Sie jedoch *position* angeben, wird das neue Element an die angegebene Stelle eingefügt (erste Stelle = 0).

Bemerkung • Verwenden Sie *position* nicht mit Listen, bei denen die *Sorted*-Eigenschaft TRUE ist; ADDITEM kann sonst evtl. in Zukunft keine Elemente mehr korrekt einsortieren.

Siehe auch List (438), REMOVEITEM (452).

Alignment (Eigenschaft)



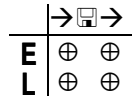
Objekte Bezeichnung

Datentyp INTEGER (0–2)

Nutzen *Alignment* gibt an, wie die Caption-Eigenschaft eines Bezeichnung-

Steuerfeldes in diesem Feld angezeigt wird: 0 = linksbündig (Standard), 1 = rechtsbündig, 2 = zentriert.

Archive (Eigenschaft)



Objekte Dateiliste

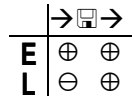
Datentyp INTEGER (True/False)

Nutzen Archive ist TRUE, wenn in einem Dateilistenfeld Dateien angezeigt werden (sollen), bei denen das Archiv-Attribut gesetzt ist. Zur Änderung dieses Attributs vgl. Kapitel 22.

Bemerkung • Beachten Sie hierzu die Ausführungen über Dateilistenfelder im Kapitel 7.

Siehe auch Hidden (432), Normal (447), ReadOnly (451), System (457); Refresh (452).

Attached (Eigenschaft)



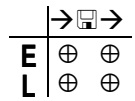
Objekte Horizontale und vertikale Bildlaufleisten

Datentyp INTEGER (True/False)

Nutzen *Attached* ist TRUE, wenn eine Bildlaufleiste auf dem Rand einer Form sitzt und ihre Größe mit dieser automatisch geändert werden soll. (Vgl. Ausführungen zu Bildlaufleisten im Kapitel 7.)

Die Standard-Einstellung ist FALSE.

AutoRedraw (Eigenschaft)



Objekte Form, Bildfeld

Datentyp INTEGER (True/False)

Nutzen Wenn *AutoRedraw* TRUE ist, wird eine Kopie des Bildfeldes bzw. der Form im Speicher gehalten, mittels derer es stets neu gezeichnet werden kann (z. B. wenn ein anderes Objekt, das das Bildfeld zuvor überlappte, fortbewegt wird). Das Bildfeld bzw. die Form erzeugt dann keine *Paint*-Ereignisse. Ist *AutoRedraw* FALSE, muß das Bildfeld bzw. die Form vom Programm selbst auf den neusten Stand gebracht werden, wenn sich das Aussehen ändert; in diesem Falle wird dazu ein Paint-Ereignis generiert.

Die folgende Skizze verdeutlicht die Wirkung von *AutoRedraw*:

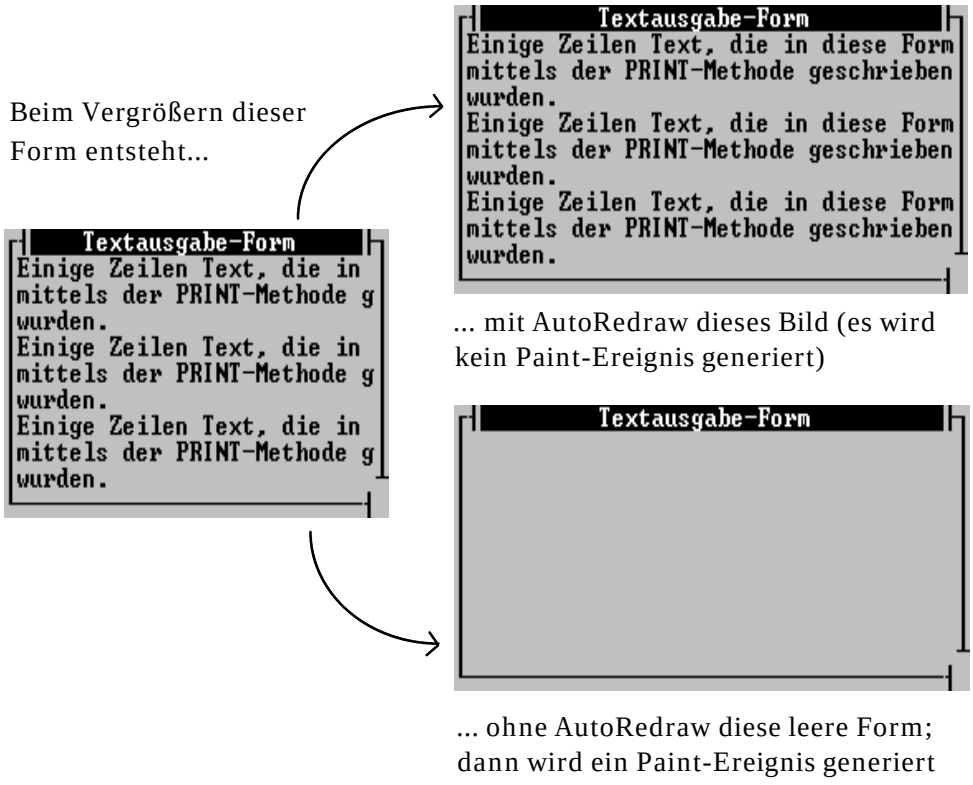


Abbildung 25–1: Die Wirkung von AutoRedraw

Siehe auch Paint (447).

AutoSize (Eigenschaft)

Objekte Bezeichnung

Datentyp INTEGER (True/False)

	→	🖫	→
E	⊕		⊕
L	⊕		⊕

Nutzen Durch die *AutoSize*-Eigenschaft wird festgelegt, ob die Breite eines Bezeichnungsfeldes sich automatisch ändert, wenn ein anderer Text in die *Caption*-Eigenschaft eingetragen wird. Bei *AutoSize* = TRUE findet diese Anpassung statt, sonst nicht.

Bemerkung • Die Breitenanpassung kann auch durch eine Änderung der *Width*-Eigenschaft „manuell“ vorgenommen werden.

Siehe auch Width (464).

BackColor (Eigenschaft)

Objekte Alle außer Bildlaufleiste, CLIPBOARD, Menüeintrag, PRINTER, SCREEN, Timer

	→	🖫	→
E	⊕		⊕
L	⊕		⊕

Datentyp INTEGER (0–15)

Nutzen Die Eigenschaft *BackColor* legt die Hintergrundfarbe eines Objektes fest. Die Voreinstellung ist 0. Gültige Werte sind die üblichen Farb-Codes:

Wert	*Farbe*	*Wert*	*Farbe*
0	Schwarz	8	Grau
1	Blau	9	Hellblau
2	Grün	10	Hellgrün
3	Zyan (himmelblau)	11	Hellzyan
4	Rot	12	Hellrot
5	Violett	13	Rosa
6	Braun	14	Gelb
7	Weiß	15	leuchtendes Weiß

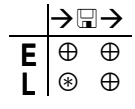
Bemerkung • Diese Zuordnung ist der Standard; sie kann jedoch mit dem PALETTE-Befehl (siehe Disketten-Referenzteil) bei den meisten Grafikkarten modifiziert werden.

• Wenn der Wert von *BackColor* bei Formen oder Bildfeldern geändert wird, wird alles, was mittels der PRINT-Methode hineingeschrieben wurde, gelöscht.

Siehe auch ForeColor (430).

BorderStyle (Eigenschaft)

Objekte Form, Bezeichnung, Bildfeld, Textfeld



Datentyp INTEGER (0–6 bei Formen, sonst 0–2)

Nutzen Die Eigenschaft *BorderStyle* bestimmt, ob kein Rahmen, ein einfacher oder ein doppelter Rahmen um ein Objekt gezogen wird. Bei Formen legt die Eigenschaft außerdem fest, ob die Form vergrößert/verkleinert und verschoben werden kann:

Wert	*bei Formen*	*bei Steuerelementen*
0	Kein Titel (*Caption* wird nicht angezeigt), kein Rahmen, feste Größe und Position	kein Rahmen
1	Titel, Menüfeld, einfacher Rahmen, feste Größe, Position änderbar; MinButton, MaxButton möglich	einfacher Rahmen
2	Titel, Menüfeld, einfacher Rahmen; Größe und Position mit Maus änderbar; MinButton, MaxButton möglich	doppelter Rahmen

Wert	*bei Formen*
3	wie 1, aber doppelter Rahmen
4	wie 2, aber doppelter Rahmen
5	wie 1, aber Rahmen aus CHR$(219)-Zeichen
6	wie 2, aber Rahmen aus CHR$(219)-Zeichen

Bemerkung • Ein Rahmen wird stets in den Farben gezeichnet, die durch die Felder 1 und 2 der *SCREEN.ControlPanel*-Eigenschaft festgelegt sind.

• Bei Formen und Textfeldern kann *BorderStyle* zur Laufzeit nicht verändert werden.

Siehe auch Caption (415), ControlBox (420), ControlPanel (420), MinButton (442), MaxButton (442).

Cancel (Eigenschaft)

Objekte Schaltfläche

Datentyp INTEGER (True/False)

	→🖫→
E	⊕ ⊕
L	⊕ ⊕

Nutzen Für eine Schaltfläche, deren *Cancel*-Eigenschaft auf TRUE gesetzt ist, wird automatisch ein *Click*-Ereignis erzeugt, wenn der Benutzer irgendwo auf der Form die ESC-Taste drückt, auch wenn das Steuerelement, das den Fokus hat, die ESC-Taste selbst mit einem der drei Key-Ereignisse verarbeiten würde. Der Tastendruck wird nicht mehr an das Steuerelement mit dem Fokus weitergeleitet.

Bemerkung • Nur bei einer Schaltfläche pro Form kann die *Cancel*-Eigenschaft TRUE sein.

Siehe auch Default (423), KeyDown, KeyUp (436), KeyPress (437).

Caption (Eigenschaft)

Objekte Schaltfläche, Bezeichnungsfeld, Form, Kontrollfeld, Menüeintrag, Optionsfeld, Rahmen

Datentyp STRING

	→🖫→
E	⊕ ⊕
L	⊕ ⊕

Nutzen *Caption* ist der Text, der als Name bzw. Überschrift eines Objektes angezeigt wird. Bei Formen wird *Caption* nur angezeigt, wenn *BorderStyle* nicht 0 ist (siehe dort); bei Bezeichnungsfeldern kann mit-

tels der *AutoSize*-Eigenschaft festgelegt werden, ob sich die Größe des Feldes dem Caption-Text anpaßt, und *Alignment* bestimmt, in welcher Ausrichtung Caption ausgegeben wird.

Bemerkung

- Wenn im *Caption*-Text ein kommerzielles Und („&“) vorkommt, wird das darauf folgende Zeichen zur „Zugriffstaste“ des Steuerelements, das heißt, es erhält den Fokus, wenn der Benutzer Alt und diese Taste drückt. Kann das Steuerelement selbst nicht den Fokus erhalten (z. B. weil es ein Bezeichnungsfeld ist), erhält das Objekt mit der nächsthöheren *TabIndex*-Eigenschaft den Fokus. Verwenden Sie „&&“, wenn Sie ein kommerzielles Und im Titel des Steuerelements anzeigen wollen, ohne eine Zugriffstaste zu definieren.
- Obwohl mehrere Objekte auf einer Form dieselbe Zugriffstaste haben dürfen, sollten Sie das der Bedienungsfreundlichkeit wegen vermeiden.

Siehe auch Alignment (411), AutoSize (413), TabIndex (458).

Change (Ereignis)

Objekte Bezeichnung, horizontale und vertikale Bildlaufleiste, Kombinationsfeld, Laufwerksliste, Textfeld, Verzeichnisliste

Argumente keine

Nutzen *Change* tritt ein, wenn sich der Inhalt eines Steuerelements ändert. Dabei spielt es keine Rolle, ob der Inhalt vom Benutzer selbst oder von einer Prozedur Ihres Programms geändert wurde; Bezeichnungsfelder sind nur vom Programm aus änderbar.

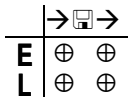
Steuerelement	*Change wird ausgelöst bei Änderung der Eigenschaft*
Bezeichnung	*Caption*
Bildlaufleiste	*Value*
Kombinationsfeld	*Text* oder *ListIndex*
Laufwerksliste	*Drive* oder *ListIndex*
Textfeld	*Text*
Verzeichnisliste	*Path* oder *ListIndex*

Bemerkung

- Bei Kontroll- und Optionsfeldern gibt es zwar kein *Change*-Ereignis, aber das *Click*-Ereignis kann ersatzweise verwendet werden.

Siehe auch Caption (415), Click (417), Drive (427), ListIndex (439), Path (448), Text (460), Value (463).

Checked (Eigenschaft)



Objekte Menüeintrag

Datentyp INTEGER (True/False)

Nutzen *Checked* ist TRUE, wenn neben dem betreffenden Menüeintrag eine Markierung angezeigt wird (wie z. B. in VBDOS neben „Ablauf verfolgen“ aus dem „Testen“-Menü, wenn man darauf klickt).

Bemerkung • Sie werden *Checked* üblicherweise verwenden, um Ein/Aus-Funktionen über Menüs zu realisieren. Sie müssen dann in der *Click*-Ereignisprozedur des Menüeintrags mit einer Zeile à la `Menuepunkt.Checked = NOT Menuepunkt.Checked` dafür sorgen, daß die Markierung bei jedem Klick gewechselt wird.

Siehe auch Menüeintrag-Steuerelement im Kapitel 7.

CLEAR (Methode)

Objekte CLIPBOARD

Verwendung `CLIPBOARD.CLEAR`

Nutzen Löscht den Inhalt der Zwischenablage.

Siehe auch GETTEXT (431), SETTEXT (456).

Click (Ereignis)

Objekte Bezeichnung, Bildfeld, Dateiliste, Form, Kombinationsfeld, Kontrollfeld, Listenfeld, Menüeintrag, Optionsfeld, Schaltfläche, Verzeichnisliste

Argumente keine

Nutzen *Click* tritt ein, wenn mit einer Maustaste auf ein Objekt geklickt wird. Wenn mehrere Objekte übereinanderliegen (z. B. Kontrollfelder auf einem Bildfeld), wird das *Click*-Ereignis für das oberste Objekt, dessen *Visible*-Eigenschaft auf TRUE gestellt ist, erzeugt. Hat dieses Objekt eine *Enabled*-Eigenschaft von FALSE, wird überhaupt kein *Click*-Ereignis – auch nicht für die darunterliegenden Objekte – generiert.

Objekt	*Click-Funktionsweise*
Bezeichnung	*Click* tritt ein, wenn mit einer der beiden Maustasten auf das Feld geklickt wird. *Click* tritt nicht ein, wenn die *Caption*-Eigenschaft verändert wird.
Bildfeld	*Click* tritt ein, wenn mit einer der beiden Maustasten auf das Feld geklickt wird.
Dateiliste	*Click* tritt ein, wenn mit der linken Maustaste auf das Feld geklickt oder die *ListIndex*-Eigenschaft vom Programm verändert wird, oder wenn der Benutzer die Pfeiltasten benutzt, während das Objekt den Fokus hat. *Click* tritt nicht ein, wenn die *FileName*-Eigenschaft vom Programm verändert wird oder der Benutzer durch die Pfeiltasten den Leuchtbalken nicht verschiebt (z. B. Pfeil-Ab, wenn der Balken am Ende der Liste steht).
Form	*Click* tritt ein, wenn mit einer der beiden Maustasten auf den Innenbereich der Form geklickt wird. *Click* tritt nicht ein, wenn auf ein deaktiviertes Steuerelement geklickt wird (obwohl's im Handbuch steht).
Kombinationsfeld	*Click* tritt ein, wenn mit der linken Maustaste in den Listenbereich des Feldes geklickt oder die *ListIndex*-Eigenschaft vom Programm verändert wird oder der Benutzer die Pfeiltasten benutzt, während das Objekt den Fokus hat. *Click* tritt nicht ein, wenn in den Textbereich geklickt oder die *Text*-Eigenschaft geändert wird oder der Benutzer durch die Pfeiltasten den Leuchtbalken nicht verschiebt (z. B. Pfeil-Ab, wenn der Balken am Ende der Liste steht)
Kontrollfeld	*Click* tritt ein, wenn - mit der linken Maustaste auf das Feld geklickt wird oder - der Benutzer die Zugriffstaste drückt oder - das Programm die *Value*-Eigenschft ändert oder - die Leertaste betätigt wird und das Objekt den Fokus hat.
Listenfeld	*Click* tritt ein, wenn mit der linken Maustaste in den Listenbereich geklickt oder die *ListIndex*-Eigenschaft vom Programm verändert wird, oder wenn der Benutzer mit den Pfeiltasten in der Liste manövriert, während das Objekt den Fokus hat. *Click* tritt nicht ein, wenn der Benutzer durch die Pfeiltasten den Leuchtbalken nicht verschiebt (z. B. Pfeil-Ab, wenn der Balken am Ende der Liste steht).

Objekt	*Click-Funktionsweise*
Menüeintrag	*Click* tritt ein, wenn mit der linken Maustaste auf den Eintrag geklickt wird oder der Benutzer - die Zugriffstaste drückt oder - das Element mit den Pfeiltasten wählt und Enter drückt oder - das Element mit den Pfeiltasten ansteuert und es untergeordnete Elemente hat oder - die Abkürzungstaste des Menüeintrags betätigt. *Click* tritt nicht ein, wenn der Benutzer das Element mit den Pfeiltasten ansteuert, es aber keine untergeordneten Elemente hat (dann muß er erst ENTER drücken).
Optionsfeld	*Click* tritt ein, wenn mit der linken Maustaste auf das Feld geklickt wird oder der Benutzer die Zugriffstaste drückt oder die Leertaste betätigt, solange das Objekt den Fokus hat, oder Ihr Programm die *Value*-Eigenschaft auf TRUE setzt. *Click* tritt nicht ein, wenn die *Value*-Eigenschaft schon vor dem Klicken oder Betätigen der Zugriffstaste TRUE war.
Schaltfläche	*Click* tritt ein, wenn - mit der linken Maustaste auf das Feld geklickt wird oder - Ihr Programm die *Value*-Eigenschaft auf TRUE setzt oder - ENTER betätigt wird und das Element die Eigenschaft *Default* = TRUE hat oder - ESC betätigt wird und das Element die Eigenschaft *Cancel* = TRUE hat oder - die Leertaste betätigt wird und das Element den Fokus hat.
Verzeichnisliste	wie Dateiliste (*FileName* durch *Path* ersetzen).

Bemerkung • Zwar können Mausereignisse bei den meisten Steuerelementen auch mit *MouseDown* und *MouseUp* abgefragt werden, aber Click ist bisweilen komfortabler, weil es auch die Tastatur mit einbezieht und selbsttätig zwischen „sinnvollem“ und „nicht sinnvollem“ Klicken unterscheidet, während die Mouse-Ereignisse stur jedes Klicken registrieren.

Siehe auch Change (416), DblClick (424), MouseDown, MouseUp (443).

CLS (Methode)

Objekte Bildfeld, Form

Verwendung *objekt*.CLS

Nutzen CLS löscht den mit PRINT ausgegebenen Inhalt eines Bildfeldes oder einer Form. Steuerelemente, die sich im Bildfeld oder in der

Form befinden, werden hiervon nicht betroffen. CLS setzt die *CurrentX*- und *CurrentY*-Eigenschaft auf 0.

Siehe auch CurrentX, CurrentY (423), PRINT (450).

ControlBox (Eigenschaft)

Objekte Form

Datentyp INTEGER (True/False)

	→ 🖫 →
E	⊕ ⊕
L	⊖ ⊕

Nutzen *ControlBox* wird auf TRUE gesetzt, wenn die Form ein „Systemmenüfeld“ haben soll, über das der Benutzer in WINDOWS-Manier einige Manipulationen am Fenster vornehmen kann:

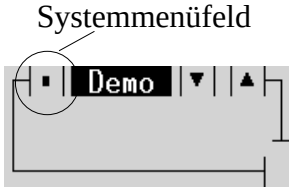


Abbildung 25–2: Form mit Systemmenüfeld

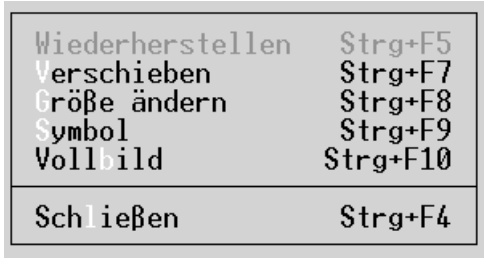


Abbildung 25–3: Systemmenü

Bemerkung • *ControlBox* ist unwirksam, wenn *BorderStyle* auf 0 gesetzt ist.

Siehe auch BorderStyle (414), MinButton (442), MaxButton (442).

ControlPanel (Eigenschaft)

Objekte SCREEN

Datentyp INTEGER-Array (diverse Wertebereiche)

	→ 🖫 →
E	⊖ ⊖
L	⊕ ⊕

Nutzen *ControlPanel* ist keine Eigenschaft wie die anderen, sondern steuert grundlegende, allgemeingültige Einstellungen für das gesamte System des ereignisgesteuerten Programmierens. Unter VBDOS dient *ControlPanel* fast ausschließlich zur Wahl der Farben, die alle Steuerelemente und Formen gleichermaßen verwenden. In der folgenden Tabelle sind die 17 Elemente des *ControlPanel* aufgelistet; verwenden Sie am besten die genannten, in CONSTANT.BI definierten Konstanten für den Zugriff, dann haben Sie bei einer

eventuellen Portierung nach WINDOWS keine Probleme (z. B. würden Sie den Befehl `SCREEN.ControlPanel(Desktop_Pattern) = 65` verwenden, um den Bildschirmhintergrund mit dem Buchstaben „A“ füllen zu lassen).

Index	*Bereich*	*vordefinierte Konstante*	*Bedeutung*
0	0–15	`Accesskey_ForeColor`	Die Vordergrundfarbe für die Zugriffstasten (mit & in der *Caption*-Eigenschaft hervorgehoben)
1	0–15	`Active_Border_BackColor`	Hintergrundfarbe des Rahmens der aktiven Form
2	0–15	`Active_Border_ForeColor`	Vordergrundfarbe des Rahmens der aktiven Form
3	–1/0	`Active_Window_Shadow`	TRUE, wenn aktive Form mit Schatten angezeigt werden soll
4	0–15	`Combutton_ForeColor`	Vordergrundfarbe für Schaltflächen (Hintergrund wird über *BackColor*-Eigenschaft einzeln eingestellt; mehrzeilige Schaltflächen werden anders gefärbt, wenn `Three_D` TRUE ist)
5	0–15	`Desktop_BackColor`	Hintergrundfarbe für den Bereich, der nicht von Formen belegt wird
6	0–15	`Desktop_ForeColor`	Vordergrundfarbe für den Bereich, der nicht von Formen belegt wird
7	0–255	`Desktop_Pattern`	ASCII-Zeichen, mit dem der nicht von Formen belegte Bereich gefüllt wird (0 = nicht füllen)
8	0–15	`Disabled_Item_ForeColor`	Vordergrundfarbe für Steuerelemente, deren *Enabled*-Eigenschaft auf FALSE gesetzt wurde
9	0–15	`Menu_BackColor`	Hintergrundfarbe für die Menüleiste und Pulldown-Menüs
10	0–15	`Menu_ForeColor`	Vordergrundfarbe für die Menüleiste und Pulldown-Menüs
11	0–15	`Menu_Selected_BackColor`	Hintergrundfarbe für Menüeinträge, auf denen der Leuchtbalken steht
12	0–15	`Menu_Selected_ForeColor`	Vordergrundfarbe für Menüeinträge, auf denen der Leuchtbalken steht

Index	*Bereich*	*vordefinierte Konstante*	*Bedeutung*
13	0–15	`Scrollbar_BackColor`	Hintergrundfarbe für Bildlaufleisten
14	0–15	`Scrollbar_ForeColor`	Vordergrundfarbe für Bildlaufleisten
15	–1/0	`Three_D`	TRUE, wenn Steuerelemente automatisch mit einem „3D-Effekt“ versehen werden sollen (der Rahmen wird dann in zwei verschiedenen Farben dargestellt)
16	0–15	`Titlebar_BackColor`	Hintergrundfarbe der Titelleiste der aktiven Form
17	0–15	`Titlebar_ForeColor`	Vordergrundfarbe der Titelleiste der aktiven Form

Bemerkung

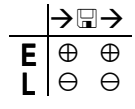
- Wie Sie sehen, ist hier für Vorder- und Hintergrundfarben der Wertebereich 0 bis 15 zulässig. Dadurch können Sie auch z.B. einen gelben Hintergrund anzeigen (was beim üblichen COLOR-Befehl nur mit Tricks möglich ist); die Möglichkeit blinkender Anzeigen durch höhere Vordergrund-Farbwerte als 15 geht jedoch verloren.
- Eine Liste der Farben finden Sie bei der *BackColor*-Eigenschaft.
- Bei Verwendung von `Desktop_Pattern` ist zu beachten, daß VBDOS Speicher reserviert und den gesamten aktuellen Bildschirminhalt hineinkopiert, wenn Sie dieses Feld auf 0 setzen. Das ist notwendig, um nach dem Verschieben von Formen wieder den ursprünglichen Bildschirmhintergrund anzeigen zu können. Setzen Sie dieses Feld auf einen anderen Wert als 0, muß kein Speicher reserviert werden, weil dann auf allen Bildschirmpositionen das gleiche Zeichen angezeigt wird.

Siehe auch BackColor (413), COLOR im Disketten-Referenzteil.

CtlName (Eigenschaft)

Objekte Alle außer Form, SCREEN, PRINTER, CLIPBOARD

Datentyp STRING



Nutzen *CtlName* wird im Form-Designer benutzt, um die Namen festzulegen, unter denen Ihre Steuerelemente später im Programm mit Ereignisprozeduren verbunden werden und unter denen auf die Eigenschaften der Steuerelemente zugegriffen wird. *CtlName* dient

nur als Programmierhilfe (damit Sie Ihre Schaltflächen zum Beispiel auch intern „Abbruch“, „OK“ und „Aendern“ nennen können und nicht auf „Befehl1“, „Befehl2“ usw. angewiesen sind). Deshalb ist *CtlName* zur Laufzeit nicht verfügbar. Um Objekte zur Laufzeit eindeutig zu identifizieren, können Sie die *Tag*-Eigenschaft verwenden.

Siehe auch FormName (430), Tag (459).

CurrentX, CurrentY (Eigenschaften)

Objekte Bildfeld, Form

Datentyp INTEGER (0–254)

	→ 🖫 →
E	⊖ ⊖
L	⊕ ⊕

Nutzen *CurrentX* und *CurrentY* beschreiben die Position, an die auf dem Bildfeld oder der Form die nächste Ausgabe mit PRINT ausgegeben werden wird. Durch die Verwendung von PRINT werden *CurrentX* und *CurrentY* verändert (je nach Länge des ausgegebenen Ausdrucks und je nachdem, ob am Ende der PRINT-Methode ein Komma, ein Semikolon oder keines dieser Zeichen stand). Die Anwendung der CLS-Methode für das betreffende Objekt setzt beide Eigenschaften auf 0.

Bemerkung • Diese beiden Eigenschaften ersetzen CSRLIN und POS(0), aber auch LOCATE im herkömmlichen System, da sie gelesen und verändert werden können.

Siehe auch PRINT (450).

Default (Eigenschaft)

Objekte Schaltfläche

Datentyp INTEGER (True/False)

	→ 🖫 →
E	⊕ ⊕
L	⊕ ⊕

Nutzen *Default* ist das Gegenstück zu *Cancel*. Wenn der Benutzer die Eingabetaste (Return, Enter) drückt, wird für die Schaltfläche auf der Form ein *Click*-Ereignis erzeugt, deren *Default*-Eigenschaft TRUE ist (falls es eine solche gibt).

Bemerkung • Nur für eine Schaltfläche pro Form kann *Default* auf TRUE gesetzt werden.

Siehe auch Cancel (415).

DblClick (Ereignis)

Objekte Bezeichnung, Bildfeld, Dateiliste, Form, Kombinationsfeld (*Style* = 1), Listenfeld, Optionsfeld, Schaltfläche

Argumente keine

Nutzen *DblClick* tritt ein, wenn der Benutzer innerhalb einer vom Maustreiber festgelegten Zeitspanne zweimal hintereinander den Mausknopf betätigt. Für den ersten Klick wird zunächst ein *Click*-Ereignis erzeugt (weil VBDOS ja nicht erst abwarten kann, ob vielleicht noch ein zweiter Klick folgt), und wenn innerhalb der gesetzten Zeit dann ein zweiter Klick folgt, wird kein zweites *Click*-, sondern nun ein *DblClick*-Ereignis erzeugt.

Das *DblClick*-Ereignis tritt außerdem ein, wenn Sie von Ihrem Programm aus die *FileName*-Eigenschaft eines Dateilistenfeldes auf den Namen einer vorhandenen Datei setzen (siehe dort).

Bemerkung • Wenn Sie sich nicht auf die Doppelklick-Zeitvorgabe des Systems verlassen wollen, schauen Sie auf die Diskette. Die Datei KLICZEIT.BAS enthält eine Routine, die den Augenblick, in dem auf ein Objekt geklickt wird, in seiner *Tag*-Eigenschaft speichert und so selbst erkennen kann, ob ein Doppelklick stattgefunden hat oder nicht (die Zeit ist frei wählbar). Damit kann ein Doppelklick auch für Schaltflächen, Kontrollfelder, Menüeinträge und Verzeichnislisten erkannt werden.

Siehe auch Click (417).

DOEVENTS (Funktion)

Anwendung `x% = DOEVENTS()`

Nutzen DOEVENTS gibt die Anzahl der zur Zeit geladenen (nicht die Anzahl der sichtbaren) Formen zurück.

Außerdem ermöglicht DOEVENTS für Sekundenbruchteile die Verarbeitung von Ereignissen und erlaubt es so, bei eigenen Programmen einen (zuweilen erstaunlichen) „Multitasking“-Effekt zu erzielen.

Während Ihr Programm irgendeine gewöhnliche Operation durchführt, die etwas länger dauert (z. B. auf der Festplatte nach einer Datei sucht, eine Liste ausdruckt oder eine komplizierte Funktion berechnet), können normalerweise keine Ereignisse verarbeitet

werden. (Sie werden begrenzt in einer Warteschlange gespeichert.)

Wenn Sie nun während des länger dauernden Vorgangs regelmäßig DOEVENTS aufrufen, können durch diesen Aufruf Ereignisse verarbeitet werden. Ein einzelner DOEVENTS-Aufruf kann in Millisekunden beendet sein, wenn der Benutzer tatenlos ist, er kann aber auch dazu führen, daß der Benutzer auf eine Schaltfläche klickt, in deren *Click*-Ereignisprozedur Sie ebenfalls eine länger dauernde Berechnung durchführen. Durch einen DOEVENTS-Aufruf kann also jedes denkbare Ereignis ausgelöst werden. Mehr zum Thema „Pseudo-Multitasking“ und den damit verbundenen Problemen finden Sie im Kapitel 7.

Kompatibel PDS ⊖ VBWIN ⊕ Formen ⊕ Mathe ⊖

DRAG (Methode)

Objekte Alle außer CLIPBOARD, Form, Menüeintrag, PRINTER, SCREEN, Timer

Verwendung `objekt.DRAG [modus]`

Nutzen DRAG startet oder beendet einen Drag-and-Drop-Vorgang für ein Objekt. Im Gegensatz zum automatischen „Ziehen“, das möglich wird, wenn die Eigenschaft *DragMode* eines Objektes 1 ist, muß der Benutzer beim Ziehen, das durch DRAG ausgelöst wird, nicht die Maustaste gedrückt halten.

modus ist 0 für das Abbrechen des Vorgangs, 1 für das Einleiten eines Ziehvorgangs und 2 für das ordnungsgemäße Beenden eines Ziehvorgangs. Im Unterschied zu *modus* = 0 wird bei *modus* = 2 ein *DragDrop*-Ereignis generiert.

Siehe auch DragDrop (425), DragMode (426).

DragDrop (Ereignis)

Objekte Alle bis auf CLIPBOARD, Menüeintrag, PRINTER, SCREEN und Timer

Argumente `Source AS CONTROL, X AS SINGLE, Y AS SINGLE`

Nutzen *DragDrop* tritt für ein Objekt A ein, wenn ein anderes Objekt B „gezogen“ und über A fallengelassen wurde. *Source* ist das gezogene Objekt; sie können *Source* wie ein Steuerelement verwenden

(beachten Sie aber, daß die Eigenschaft *Source.CtlName* nicht verfügbar ist).

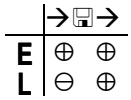
Ein Objekt kann nur gezogen werden, wenn seine *DragMode*-Eigenschaft auf 1 gesetzt ist oder Ihr Programm den „Ziehen- und Loslassen-Vorgang“, wie Microsoft die Drag-and-Drop-Technik nennt, durch Anwendung der DRAG-Methode selbst ausgelöst hat.

Die Variablen *X* und *Y* geben an, an welcher Stelle (linke obere Ekke innen = 0,0) des Steuerelements sich der Mauszeiger befand, als *Source* fallengelassen wurde. Diese Werte können –1 sein, wenn *Source* auf dem Rahmen fallengelassen wurde. Sie geben nicht an, an welcher Stelle sich die linke obere Ecke des gezogenen Steuerelements befand, als es fallengelassen wurde!

Siehe auch DRAG (425), DragMode (426), DragOver (427).

DragMode (Eigenschaft)

Objekte Alle außer Form, Menüeintrag, Timer, SCREEN, PRINTER, CLIPBOARD



Datentyp INTEGER (0, 1)

Nutzen Mittels *DragMode* können Sie bestimmen, ob ein Steuerelement ohne Zutun Ihres Programms vom Benutzer „gezogen“ werden kann oder nicht. Die Standardeinstellung ist 0, das heißt, daß kein Objekt gezogen werden kann, solange Ihr Programm das nicht mit einer DRAG-Methode einleitet. Setzen Sie *DragMode* auf 1, dann kann der Benutzer durch Festhalten des (rechten) Mausknopfes auf dem betreffenden Objekt und Bewegen der Maus das Objekt an einen beliebigen Platz auf dem Bildschirm „ziehen“. Dabei verschiebt er eine Kopie des Objektes, die in dem Moment gelöscht wird, da er die Maustaste losläßt. Am Bildschirm findet also keine dauerhafte Veränderung statt (etwa wie beim Verschieben einer Form, die ja dann am neuen Platz stehenbleibt). Es werden jedoch *DragDrop*- und *DragOver*-Ereignisse generiert, auf die Sie im Programm entsprechend reagieren können, so daß auch ein wirkliches Verschieben des Steuerelements denkbar ist.

Siehe auch DRAG (425), DragDrop (425), DragOver (427).

DragOver (Ereignis)

Objekte Alle außer CLIPBOARD, Menüeintrag, PRINTER, SCREEN und Timer

Argumente
```
Source AS CONTROL, X AS SINGLE, Y AS SINGLE,
State AS INTEGER
```

Nutzen *DragOver* tritt für ein Objekt A ein, wenn ein anderes Objekt B gerade gezogen wird (= die Maus bewegt wird) und sich der Mauszeiger dabei über dem Objekt A befindet.

DragOver wird verwendet, um zum Beispiel gültige Ziele für eine Drag-and-Drop-Aktion anzuzeigen; Sie können mittels *DragOver* dafür sorgen, daß sich der Mauszeiger in ein X verwandelt, wenn ein bestimmtes Objekt irgendwo „schwebt“, wo es nicht fallengelassen werden darf (tatsächlich kann der Benutzer ein Objekt, das er gerade zieht, überall fallenlassen; wenn das Objekt, über dem er die Maustaste losläßt, jedoch kein *DragDrop*-Ereignis verarbeitet, passiert nichts).

Source, *X* und *Y* werden wie bei *DragDrop* verwendet; die zusätzliche Variable *State* gibt an, ob das gezogene Objekt gerade in den Bereich des Objektes, für das das *DragOver*-Ereignis generiert wird, eintritt (0), den Bereich verläßt (1) oder sich innerhalb dieses Bereiches bewegt (2).

Siehe auch DRAG (425), DragDrop (425).

Drive (Eigenschaft)

Objekte Laufwerksliste

Datentyp STRING (zwei Zeichen in der Form „C:“)

	→ 🖫 →
E	⊖ ⊖
L	⊕ ⊕

Nutzen *Drive* gibt an (oder legt fest), welches das gerade ausgewählte Laufwerk eines Laufwerkslistenfeldes ist. Wenn Sie dieser Eigenschaft einen Wert zuweisen, ist nur das erste Zeichen relevant und muß ein gültiges Laufwerk repräsentieren. Außerdem wird dann ein *Change*-Ereignis ausgelöst, und ein REFRESH wird ausgeführt (die Laufwerksliste wird neu ermittelt).

Siehe auch Path (448), FileName (429).

DropDown (Ereignis)

Objekte Kombinationsfeld (*Style* = 0 oder 2)

Argumente keine

Nutzen Das *DropDown*-Ereignis tritt für die Kombinationsfelder, die eine „ausklappbare“ Liste enthalten, dann ein, wenn der Benutzer auf das ↓-Symbol des Kombinationsfeldes klickt oder die Tastenkombination Alt+Pfeil-Ab betätigt, um die Liste anzuzeigen.

Das *DropDown*-Ereignis tritt ein, bevor die Liste angezeigt wird. Sie können diese Eigenschaft verwenden, um die Liste erst dann (mit ADDITEM) aufzubauen, wenn sie wirklich angefordert wird.

Siehe auch ADDITEM (411).

Enabled (Eigenschaft)

Objekte Alle außer SCREEN, CLIPBOARD, PRINTER

	→🖫→	
E	⊖	⊖
L	⊕	⊕

Datentyp INTEGER (True, False)

Nutzen *Enabled* legt fest, ob ein Objekt „funktioniert“. Objekte, deren *Enabled*-Eigenschaft FALSE ist, sind zwar sichtbar, werden allerdings in einer anderen Farbe dargestellt (siehe *ControlPanel*) und können auf kein Ereignis reagieren. Ein Beispiel für solche „eingefrorenen“ Felder ist das „Drucken“-Dialogfeld aus VBDOS:

Abbildung 25–4: Das Dialogfeld „Drucken“ in VBDOS

Das Textfeld hinter dem Optionsfeld „Datei“ und die beiden Optionsfelder „Wenn Datei existiert:“ sind nur verfügbar, wenn das Optionsfeld „Datei“ gewählt ist. So sieht der Benutzer sofort, was für Möglichkeiten er hat, wenn er „Datei“ anwählt, ohne jedoch unsinnige Einstellungen vornehmen zu können.

Siehe auch Visible (463).

ENDDOC (Methode)

Objekte PRINTER

Verwendung `PRINTER.ENDDOC`

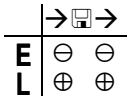
Nutzen Beendet ein Dokument, das an den Drucker geschickt wurde. Falls nicht unmittelbar zuvor die NEWPAGE-Methode aufgerufen wurde, wird ein Seitenvorschub an den Drucker geschickt.

Bemerkung • Diese Methode tut unter VBDOS nicht mehr, als ein CHR$(12)-Zeichen an den Drucker zu senden. Sie dient im wesentlichen der WINDOWS-Kompatibilität.

Siehe auch PRINT (450).

FileName (Eigenschaft)

Objekte Dateiliste



Datentyp STRING der Form „NAME.ERW“, maximal 12 Zeichen

Nutzen *FileName* gibt den Namen der gerade gewählten Datei aus einem Dateilistenfeld zurück. Ist keine gewählt (oder keine vorhanden), ist *FileName* leer.

Sie können *FileName* auch verwenden, um dem Dateilistenfeld eine neue Dateimaske oder ein neues Verzeichnis zuzuweisen, ohne dabei explizit auf die Eigenschaften *Path* oder *Pattern* zuzugreifen. Der Befehl `Liste.FileName = "C:\VBDOS\*.TXT"` hat dieselbe Wirkung wie `Liste.Pattern = "*.TXT": Liste.Path = "C:\VBDOS"`.

Bei solchen Zuweisungen werden, falls die betreffenden Eigenschaften sich ändern, *PathChange*- bzw. *PatternChange*-Ereignisse aufgerufen. Dadurch läßt sich leicht prüfen, ob ein bestimmter String eine Pfad- oder Musterangabe enthält: Weisen Sie den String der *FileName*-Eigenschaft zu, und reagieren Sie auf das *Path*- bzw. *PatternChange*-Ereignis.

Außerdem erzeugt eine Zuweisung an *FileName* ein DblClick-Ereignis, wenn es sich um den Namen einer existierenden Datei handelt (z. B. `Liste.FileName = "C:\VBDOS\VBDOS.EXE"`).

Bemerkung • Entgegen den Angaben im Microsoft-Handbuch gibt es bei Dateilistenfeldern keine *Drive*-Eigenschaft. Das Laufwerk ist in der *Path*-Eigenschaft mitgespeichert.

Siehe auch Path (448), Pattern (449).

ForeColor (Eigenschaft)

Objekte Alle außer Schaltfläche, Bildlaufleiste, CLIPBOARD, Menüeintrag, PRINTER, SCREEN, Timer

	→🖫→
E	⊕ ⊕
L	⊕ ⊕

Datentyp INTEGER (0–15)

Nutzen *ForeColor* legt die Vordergrundfarbe des betr. Objektes fest. Die Vordergrundfarben für die hier nicht unterstützten Objekte Schaltfläche, Bildlaufleiste, Menüeintrag und SCREEN können über die Eigenschaft *ControlPanel* des SCREEN gesteuert werden, sind dann aber über alle Objekte eines Programms gleich.

Bemerkung • Eine Liste der Farben finden Sie bei *BackColor*.

Siehe auch BackColor (413), ControlPanel (420).

$FORM (Metabefehl)

Anwendung `REM $FORM formname`

Nutzen Wenn Sie auf Eigenschaften einer Form (oder auf Eigenschaften ihrer Steuerelemente) aus einem anderen Modul als dem Formmodul, das diese Form beinhaltet, zugreifen möchten, müssen Sie in diesem Modul den Zugriff auf die Form mit dem $FORM-Metabefehl „anmelden“. Hinter $FORM wird der Name der Form (also der Inhalt der *FormName*-Eigenschaft), nicht der Dateiname des Formmoduls angegeben.

Der $FORM-Metabefehl ist vergleichbar mit dem DECLARE-Befehl für Funktionen; auch Funktionen müssen nur dann mit DECLARE deklariert werden, wenn sie sich in einem anderen Modul befinden als dem, in dem sie verwendet werden sollen.

Kompatibel PDS ⊖ VBWIN ⊖ Formen ⊕ Mathe ⊖

Siehe auch FormName (430).

FormName (Eigenschaft)

Objekte Form

	→🖫→
E	⊕ ⊕
L	⊖ ⊖

Datentyp STRING

Nutzen *FormName* ist das Gegenstück zu *CtlName* bei Steuerelementen. Mit dieser Eigenschaft benennen Sie eine Form zur weiteren Verwendung in Ihrem Programm; alle Referenzen auf Eigenschaften dieser Form müssen mit dem in *FormName* gespeicherten Namen

erfolgen. Dieser Name ist jedoch zur Laufzeit nicht verfügbar (vgl. Ausführungen bei *CtlName*).

Siehe auch CtlName (422), Tag (459).

FormType (Eigenschaft)

Objekte Form

	→🖫→
E	⊕ ⊕
L	⊖ ⊖

Datentyp INTEGER (0, 1)

Nutzen *FormType* ist 0 für gewöhnliche Formen und 1 für MDI-Formen. MDI-Formen nehmen immer den ganzen Bildschirm ein und sind die einzigen Formen, die selbst keine Steuerelemente (außer einem Menü), sondern nur untergeordnete Formen enthalten dürfen. Mehr dazu finden Sie im Kapitel 7.

Siehe auch CtlName (422), Tag (459).

GETTEXT (Methode)

Objekte CLIPBOARD

Verwendung
```
x$ = CLIPBOARD.GETTEXT()
```

Nutzen GETTEXT liefert den im Clipboard mit SETTEXT abgelieferten Text zurück. Die Funktionalität läßt sich zwar auch ohne das Clipboard erreichen – sogar einfacher – aber GETTEXT und das Clipboard sind kompatibel zu VB für WINDOWS.

Bemerkung • Falls Ihr VBDOS-Programm in einer DOS-Box unter WINDOWS läuft, können Sie über GETTEXT *nicht* auf den Inhalt der WINDOWS-Zwischenablage zugreifen.

Siehe auch CLEAR (417), SETTEXT(456).

GotFocus (Ereignis)

Objekte Alle außer Bezeichnung, CLIPBOARD, Menüeintrag, PRINTER, Rahmen, SCREEN, Timer

Argumente keine

Nutzen Das *GotFocus*-Ereignis tritt ein, wenn ein Objekt den Fokus erhält. Ein Objekt erhält den Fokus, wenn darauf geklickt wird, der Benutzer mit der Tab-Taste den Fokus dorthin manövriert (siehe *TabIndex*-Eigenschaft), der Benutzer die Zugriffstaste drückt oder das Programm die SETFOCUS-Methode anwendet.

Meistens tritt direkt nach dem *GotFocus*-Ereignis auch ein *Click*-Ereignis ein, da es bei vielen Objekten ebenfalls durch Drücken der Zugriffstaste oder Klicken ausgelöst wird (siehe dort).

Bemerkung • Die *GotFocus*-Eigenschaft läßt sich gut verwenden, um in einem Bezeichnungs- oder Textfeld stets einen Hilfstext anzeigen zu lassen, der dem Benutzer Erläuterungen zur aktuellen Eingabe gibt.

Siehe auch Click (417), LostFocus (441), SETFOCUS (455), TabIndex (458).

Height (Eigenschaft)

Objekte Alle außer CLIPBOARD, Menüeintrag, PRINTER, Timer

	→🖫→
E	⊛ ⊕
L	⊛ ⊕

Datentyp INTEGER (1–254)

Nutzen *Height* ist die „externe Höhe“ eines Objekts. Mit eingerechnet werden dabei Rahmen und Menüleisten, nicht berücksichtigt eventuelle Schatten.

Bemerkung • *Height* kann bei gewöhnlichen Objekten die Werte 1–254 annehmen; für Formen sind Werte von 4 (5, wenn Menüleiste angezeigt wird) bis Anzahl der Bildschirmzeilen erlaubt, und für SCREEN sind nur 25, 43, und 50 möglich. Die Werte für SCREEN sind immer schreibgeschützt (der WIDTH-Befehl muß verwendet werden, um die Bildschirmhöhe zu ändern); die Höhe eines Dateilistenfeldes, eines Kombinationsfeldes und einer MDI-Form sind zur Laufzeit nicht änderbar.

• Offenbar aufgrund eines Fehlers wird die *Height*-Eigenschaft einer Form bei jeder Änderung von *Top*, *Left*, *Width* oder *Height* auf 25 zurückgesetzt, wenn sie vorher einen größeren Wert hatte. Formen können zwar vom Benutzer auf eine Höhe von über 25 Zeilen vergrößert werden, vom Programm aus klappt das aber leider nicht.

Siehe auch WIDTH (Befehl) im Diskettenreferenzteil, Width (464).

Hidden (Eigenschaft)

Objekte Dateiliste

	→🖫→
E	⊕ ⊕
L	⊕ ⊕

Datentyp INTEGER (True/False)

Nutzen Gibt an bzw. legt fest, ob in der Dateiliste auch Dateien mit Hidden-Attribut angezeigt werden sollen („Attribute“ sind Dateieigenschaften, die von DOS zusätzlich zum sichtbaren Verzeichniseintrag gespeichert werden).

Bemerkung • Eine Änderung der *Hidden*-Eigenschaft verursacht ein erneutes Einlesen der Dateiliste.

Siehe auch Archive (412), Normal (447), System (457), ReadOnly (451).

HIDE (Methode)

Objekte Form, SCREEN

Verwendung *objekt*.HIDE

Nutzen Mit HIDE können Sie eine Form unsichtbar machen. `Form.HIDE` hat die gleiche Wirkung wie `Form.Visible = FALSE`; wenn Sie HIDE allerdings auf den Bildschirm (SCREEN) anwenden, werden alle Formen versteckt. Das ist sinnvoll, wenn Sie Operationen durchführen wollen, die während der Formen-Anzeige nicht funktionieren (z. B. Umschalten in den Grafikmodus).

Bemerkung • Eine versteckte Form benötigt weiterhin Speicherplatz. Verwenden Sie die UNLOAD-Methode, um die Form aus dem Speicher zu löschen.

Siehe auch SHOW (456), UNLOAD (462), Visible (463).

IF TYPEOF (Befehl)

Anwendung

```
IF TYPEOF objekt IS typbezeichnung THEN
   ...
[ELSEIF und ELSE möglich wie bei IF...THEN...ELSE]
   ...
END IF
```

Nutzen Ermöglicht es, festzustellen, von welchem Typ ein übergebenes Steuerelement ist.

Während bei der Übergabe von „normalen“ Variablen immer genau feststeht, welchen Variablentyp eine Prozedur als Argument erwartet, ist im Falle eines Parameters vom Typ CONTROL der genaue Datentyp unklar. Die *DragDrop*-Ereignisprozedur zum Beispiel erhält einen CONTROL-Typ; würde sie nun einfach versuchen, die *ForeColor*-Eigenschaft dieses übergebenen Steuerelements zu ändern, so könnte dabei – falls es sich um ein Element handelt, das

diese Eigenschaft nicht unterstützt – ein Fehler auftreten.

In solchen Fällen kann mit IF TYPEOF der genaue Typ eines CONTROL-Objektes festgestellt werden. *objekt* ist der Name der CONTROL-Variablen, und für *typbezeichnung* sind zulässig:

typbezeichnung	*steht für das Steuerelement*
`CheckBox`	Kontrollfeld
`ComboBox`	Kombinationsfeld
`CommandButton`	Schaltfläche
`Custom`	jedes benutzerdefinierte
`DirListBox`	Verzeichnisliste
`DriveListBox`	Laufwerksliste
`FileListBox`	Dateiliste
`Frame`	Rahmen
`Label`	Bezeichnung
`HScrollBar`	horizontale Bildlaufleiste
`ListBox`	Listenfeld
`Menu`	Menüeintrag
`OptionButton`	Optionsfeld
`PictureBox`	Bildfeld
`TextBox`	Textfeld
`Timer`	Timer („Zeitmesser“)
`VScrollBar`	vertikale Bildlaufleiste

Bemerkung • Sie können in Ihrem Programm nicht die deutschen Bezeichnungen verwenden (wie Sie irrtümlich dem Microsoft-Handbuch entnehmen könnten).

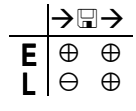
Kompatibel PDS ⊖ VBWIN ⊕ Formen ⊕ Mathe ⊖

Siehe auch IF...THEN...ELSE im Diskettenreferenzteil.

Index (Eigenschaft)

Objekte Alle außer CLIPBOARD, Form, SCREEN, PRINTER

Datentyp INTEGER oder leer



Nutzen *Index* ist bei gewöhnlichen Steuerelementen leer und wird nur verwendet, wenn ein Array von Steuerelementen erstellt wird. Ein Steuerelement-Array besteht aus beliebig vielen Steuerelementen gleichen Typs, die über den gleichen Namen angesprochen werden und auch die gleichen (gemeinsamen) Ereignisprozeduren verwenden; zur Unterscheidung zwischen den Elementen wird die Eigenschaft *Index* verwendet. Sie muß von 0 ab fortlaufend numeriert sein und kann zur Laufzeit nicht manuell verändert werden.

Siehe auch „Arrays von Steuerelementen“ im Kapitel 7, LOAD (440), UNLOAD (462).

INPUTBOX$ (Funktion)

Anwendung

```
x$ = INPUTBOX$(anzeige$ [, titel$ [, vorgabe$
[, spalte%, zeile%]]])
```

Nutzen Zeigt eine Eingabe-Box auf dem Bildschirm an und wartet, bis der Benutzer einen Text eingegeben hat. Es werden eine „OK“- und eine „Abbrechen“-Schaltfläche angezeigt; das Wählen von „OK“ oder Drücken von ENTER beendet die Eingabe und gibt den Text im Eingabefeld zurück, während das Betätigen von „Abbrechen“ oder der ESC-Taste einen Leerstring zurückgibt.

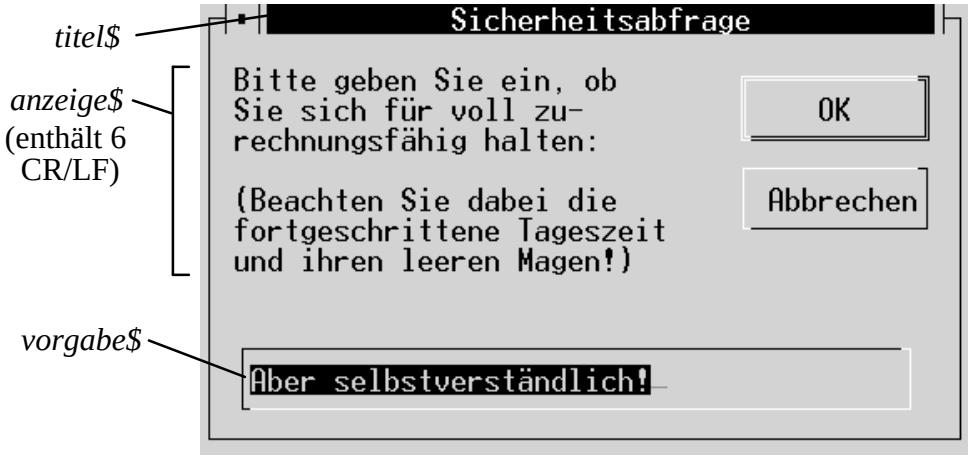


Abbildung 25–5: Typische INPUTBOX-Ausgabe

anzeige$* ist die Eingabeaufforderung an den Benutzer; sie kann mehrzeilig gestaltet werden, indem man CR/LF-Kombinationen einfügt. *titel$ ist der Titel des angezeigten Fensters. *vorgabe$* ist der Text, der von vornherein im Eingabebereich stehen soll, und *spalte%* und *zeile%* sind die Bildschirmspalte und -zeile, an der die Aufforderung erscheinen soll (sie wird zentriert, wenn diese Angaben weggelassen werden; *spalte%* und *zeile%* fangen bei 0 an zu zählen).

Bemerkung • Obwohl INPUTBOX$ auch verwendet werden kann, wenn man sonst keine Formen einsetzt, ist das wenig sinnvoll, da bei Anzeige einer solchen Box der Bildschirmhintergrund entsprechend der *SCREEN.ControlPanel*-Eigenschaft gefüllt wird, die Farben auch

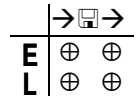
von *SCREEN.ControlPanel* abhängen und noch dazu ein kleiner Fehler in VBDOS dafür sorgt, daß nach Abbruch einer Eingabe mit ESC oder Klick auf „Abbruch“ kein PRINT-Befehl mehr funktioniert.

Kompatibel PDS ⊖ VBWIN ⊕ Formen ⊕ Mathe ⊖

Siehe auch ControlPanel (420), MSGBOX (445).

Interval (Eigenschaft)

Objekte Timer



Datentyp LONGINT (0–65535)

Nutzen *Interval* gibt die Anzahl Millisekunden an, nach der für das zugehörige *Timer*-Objekt ein *Timer*-Ereignis ausgelöst wird. Obwohl der Wert in Millisekunden angegeben wird, liegt die Auflösung eines Timers nur bei etwa 55 Millisekunden. Wenn *Interval* 0 ist, ist der Timer deaktiviert (wie *Enabled* = FALSE).

Siehe auch Enabled (428), TIMER im Disketten-Referenzteil.

KeyDown, KeyUp (Ereignisse)

Objekte Alle außer Bezeichnung, CLIPBOARD, Menüeintrag, PRINTER, Rahmen, SCREEN, Timer

Argumente `KeyCode AS INTEGER, Shift AS INTEGER`

Nutzen Das *KeyDown*-Ereignis tritt ein, wenn eine Taste gedrückt wird, das *KeyUp*-Ereignis, wenn sie losgelassen wird. Empfänger des Ereignisses ist das Objekt, das gerade den Fokus hat.

Als *KeyCode* wird eine spezielle Zahl übergeben, die die Taste identifiziert; es handelt sich aber weder um den Standard-ASCII-Code noch um den sogenannten „Scan-Code“, bei dem alle Tasten durchnumeriert sind. Buchstaben und Ziffern entsprechen dem ASCII-Code; in der Datei CONSTANT.BI sind Konstanten für die wichtigsten zusätzlichen Tasten definiert.

Shift ist eine Zahl im Bereich 0 bis 7, an der man ablesen kann, ob zum Zeitpunkt des Drückens oder Loslassens der Taste Shift (Code 1), Strg (Code 2) oder Alt (Code 4) gedrückt war. Addieren Sie die Werte, um die Codes für Tastenkombinationen zu erhalten (z. B. 5 für Shift und Alt).

Wenn der Benutzer die Taste länger gedrückt hält, so daß die Wie-

derholfunktion einsetzt, werden am laufenden Band abwechselnd *KeyUp*- und *KeyDown*-Ereignisse erzeugt.

Bemerkung
- *KeyUp*- und *KeyDown*-Ereignisse werden für alle Tasten (auch z. B. NumLock) erkannt – mit drei Ausnahmen: die Tab-Taste schaltet den Fokus weiter, anstatt erkannt zu werden, ENTER und ESC werden von Schaltflächen „weggeschnappt“, die mit der *Default*- oder *Cancel*-Eigenschaft versehen sind, und Tastenkombinationen mit Alt, die Zugriffstasten eines anderen Steuerelements sind, können auch nicht erkannt werden, weil sie sofort zu diesem Steuerelement schalten.
- Funktionstastendefinitionen mit dem KEY-Befehl sind nicht wirksam.

Siehe auch Cancel (415), Default (423), KeyPress (437).

KeyPress (Ereignis)

Objekte Alle außer Bezeichnung, CLIPBOARD, Menüeintrag, PRINTER, Rahmen, SCREEN, Timer

Argumente
```
KeyAscii AS INTEGER
```

Nutzen *KeyPress* funktioniert etwa wie INKEY$ im gewöhnlichen BASIC. Dieses Ereignis wird ausgelöst, wenn der Benutzer eine Taste drückt, die mit einem einfachen ASCII-Code wiedergegeben werden kann. *KeyAscii* enthält dann den ASCII-Wert des Zeichens. Sie können diesen Wert in der Prozedur manipulieren und so Tasten „umfunktionieren“. So ist es zum Beispiel möglich, jede gedrückte Taste sofort (bevor sie angezeigt wird) in einen Großbuchstaben oder (für Paßwort-Eingaben) in ein Sternchen zu verwandeln.

Wenn Sie in der Prozedur *KeyAscii* auf 0 setzen, wird der Tastendruck ignoriert, so daß man zum Beispiel ungültige Zeichen, die in ein Textfeld eingegeben werden, gleich abfangen kann.

Bemerkung
- Wenn der Benutzer eine Taste drückt, wird zuerst das *KeyDown*-, dann (falls es keine Sondertaste ist) das *KeyPress*- und schließlich das *KeyUp*-Ereignis erzeugt. Im Normalfall dürfte aber die Verarbeitung des *KeyPress*-Ereignisses völlig ausreichen.
- Funktionstastendefinitionen mit KEY sind unwirksam.

Siehe auch KeyDown, KeyUp (436), INKEY$ im Disketten-Referenzteil.

LargeChange (Eigenschaft)

Objekte Horizontale und vertikale Bildlaufleiste

	→🖫→	
E	⊕	⊕
L	⊕	⊕

Datentyp INTEGER

Nutzen Legt fest, um wieviel sich der Wert einer Bildlaufleiste verändert, wenn der Benutzer mit der Maus neben die Markierung auf der Leiste klickt bzw. die Tasten PgUp oder PgDn (Bild↑, Bild↓) betätigt.

Siehe auch SmallChange (457), Ausführungen zu Bildlaufleisten im Kapitel 7.

Left (Eigenschaft)

Objekte Alle außer CLIPBOARD, Menüeintrag, SCREEN, PRINTER, Timer

Datentyp INTEGER (0–254)

Nutzen *Left* gibt die Spaltenposition eines Objekts relativ zum übergeordneten Objekt an (in Zeichen). Als übergeordnetes Element gilt dabei für Steuerelemente die Form, in der sie sich befinden, oder das Bildfeld bzw. der Rahmen, in das sie gezeichnet wurden; Formen haben als übergeordnetes Element den Bildschirm oder (bei MDI-Formen) eine andere Form.

Bei *Left* = 0 beginnt das Objekt in der ersten inneren Spalte des übergeordneten Objekts („innere Spalte“ bedeutet, daß ein eventueller Rahmen berücksichtigt wird).

Bemerkung • *Left* kann bei MDI-Formen nicht verändert werden.

Siehe auch Height (432), Top (461), Width (464).

List (Eigenschaft)

Objekte Dateiliste, Kombinationsfeld, Laufwerksliste, Listenfeld, Verzeichnisliste

	→🖫→	
E	⊖	⊖
L	⊛	⊕

Datentyp STRING-Array

Nutzen *List* ist die Schnittstelle zwischen Ihrem Programm und dem Datenfeld, das jedes der oben genannten Steuerelemente intern verwaltet. Mit `List(index)` können Sie auf ein beliebiges Element einer solchen Liste zugreifen, vorausgesetzt, Sie beachten dabei die Feldgröße (Eigenschaft *ListCount*). Zur Entwurfszeit kann eine solche Liste nicht bearbeitet werden. Der kleinste Index ist 0 (Ausnahme:

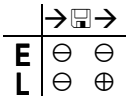
Verzeichnislisten; hier gibt es negative Indizes); der größte ist `ListCount 1`.

Sie können zur Laufzeit zwar per Zuweisung direkt in die List-Eigenschaft eingreifen; eigentlich sind dafür aber die ADDITEM- und REMOVEITEM-Methoden gedacht. Die Datei-, Laufwerks- und Verzeichnislisten sind immer schreibgeschützt.

Siehe auch ADDITEM (411), ListCount (439), ListIndex (439), REMOVEITEM (452).

ListCount (Eigenschaft)

Objekte Dateiliste, Kombinationsfeld, Laufwerksliste, Listenfeld, Verzeichnisliste



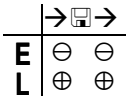
Datentyp INTEGER

Nutzen Gibt an, wieviele Elemente das zugehörige Listenfeld enthält. Die Indizes der Elemente reichen von 0 bis `ListCount 1`. Bei Verzeichnislistenfeldern gibt es darüber hinaus noch Einträge mit negativem Index, die bei *ListCount* jedoch nicht berücksichtigt werden.

Siehe auch List (438), ListIndex (439).

ListIndex (Eigenschaft)

Objekte Dateiliste, Kombinationsfeld, Laufwerksliste, Listenfeld, Verzeichnisliste



Datentyp INTEGER

Nutzen *ListIndex* gibt zurück, welches Element aus der Liste gerade gewählt ist. –1 wird zurückgegeben, wenn kein Element gewählt ist, oder (bei Kombinationsfeldern) der Benutzer eine Auswahl von Hand eingibt, anstatt aus der Liste zu wählen.

Wenn Sie ListIndex in einer Zuweisung verwenden, um ein Element zu wählen, wird automatisch auch ein Click-Ereignis für das zugehörige Steuerelement ausgelöst.

Siehe auch List (438), ListCount (439).

Load (Ereignis)

Objekte Form

Argumente keine

Nutzen Das *Load*-Ereignis tritt ein, wenn die Form in den Speicher geladen (nicht notwendigerweise angezeigt) wird. In der Load-Ereignisprozedur bringen Sie sinnvollerweise allen Code unter, der zur Initialisierung Ihrer Form und der zugehörigen Steuerelemente benötigt wird.

Da das *Load*-Ereignis generiert wird, bevor die Form angezeigt wird, können auch Größen- und Positionseinstellungen hier vorgenommen werden.

Siehe auch LOAD (Befehl) (440), Unload (462).

LOAD (Befehl)

Anwendung (1) `LOAD formname`
(2) `LOAD objekt(index)`

Nutzen Lädt eine Form oder ein neues Element eines Steuerelemente-Arrays.

Zum Laden von Formen muß LOAD nicht benutzt werden, da eine Form automatisch geladen wird, wenn auf eine ihrer Eigenschaften zugegriffen wird (so kann zum Beispiel die Methode HIDE auf eine ungeladene Form angewandt werden, um diese zu laden, ohne sie sofort anzuzeigen). Beim Laden einer Form (auch dann, wenn es automatisch geschieht) wird ein *Load*-Ereignis generiert.

Viel wichtiger ist LOAD zum Hinzufügen neuer Elemente eines Steuerelemente-Arrays. Dabei muß der Objektname des Arrays und ein *index* angegeben werden, der festlegt, an welche Stelle im Array das neue Element geladen werden soll (Sie müssen nicht unbedingt bei 0 anfangen und aufsteigend Elemente hinzuladen).

Während das erneute Laden einer bereits geladenen Form keine Auswirkungen hat, wird der Versuch, ein Steuerelement an eine Stelle im Array zu laden, die schon belegt ist, mit einem Fehler 360 (*Objekt bereits geladen*) quittiert.

Kompatibel PDS ⊖ VBWIN ⊕ Formen ⊕ Mathe ⊖

Siehe auch HIDE (433), Index (434), SHOW (456), UNLOAD (462).

LostFocus (Ereignis)

Objekte Alle außer Bezeichnung, CLIPBOARD, Menüeintrag, PRINTER, Rahmen, SCREEN, Timer

Argumente keine

Nutzen *LostFocus* tritt ein, wenn ein Objekt den Fokus verliert, weil der Benutzer die Zugriffstaste eines anderen Objektes oder die Tab-Taste gedrückt hat oder ein anderes Objekt durch Mausklick auswählt.

Ein Objekt kann auch durch eine Aktion Ihres Programms (Anwendung der UNLOAD-Methode auf die Form, Setzen der *Visible*- oder der *Enabled*-Eigenschaft auf FALSE, Anwendung der SETFOCUS-Methode auf ein anderes Objekt) den Fokus verlieren.

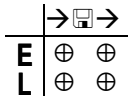
Bemerkung • LostFocus wird meist eingesetzt, um z. B. bei Textfeldern zu prüfen, ob die Eingabe des Benutzers Gültigkeit hat. Man erspart sich so, bei jedem Change-Ereignis die Eingabe zu überprüfen (und den Benutzer mit Fehlermeldungen zu nerven, obwohl er vielleicht mit der Eingabe noch gar nicht fertig ist) und prüft die Eingabe erst beim Verlassen des Feldes.

• Mit *ActiveControl* können Sie feststellen, welches Objekt den Fokus erhalten hat.

Siehe auch ActiveControl (411), ActiveForm (411), Enabled (428), GotFocus (431), SETFOCUS (455), Visible (463).

Max (Eigenschaft)

Objekte Horizontale und vertikale Bildlaufleiste



Datentyp INTEGER

Nutzen *Max* ist der maximale Wert einer Bildlaufleiste, der angenommen wird, wenn die Markierung sich ganz oben bzw. ganz rechts befindet. *Max* kann kleiner sein als *Min*; dadurch erreichen Sie, daß der höhere Wert ganz unten bzw. ganz links eingenommen wird. Es trifft nicht zu (wie im Microsoft-Handbuch beschrieben), daß *Max* dann auf den gleichen Wert wie *Min* gesetzt wird.

Siehe auch Min (442).

MaxButton (Eigenschaft)

Objekte Form

Datentyp INTEGER (True/False)

	→🖫→
E	⊕ ⊕
L	⊖ ⊕

Nutzen MaxButton gibt an, ob die Form in der rechten oberen Ecke einen Vollbildknopf hat, mit dem sie auf die volle Bildschirmgröße vergrößert werden kann.

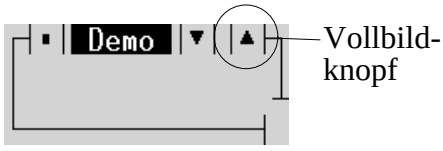


Abbildung 25–6: Form mit Vollbild-Knopf

Bemerkung
- Auch Formen ohne Vollbildknopf können vergrößert werden, und zwar einerseits über das Systemmenü (Eigenschaft *ControlBox*) oder indem man mit der Maus den Rahmen „zieht“ (geht nur bei bestimmten *BorderStyle*-Eigenschaften). Ein Vollbildknopf wird nicht bei *BorderStyle* = 0 angezeigt.
- *MaxButton* kann bei MDI-Formen nicht verändert werden.

Siehe auch BorderStyle (414), ControlBox (420), MinButton (442).

Min (Eigenschaft)

Objekte Horizontale und vertikale Bildlaufleiste

Datentyp INTEGER

	→🖫→
E	⊕ ⊕
L	⊕ ⊕

Nutzen *Min* legt den Wert fest, den eine Bildlaufleiste hat, wenn ihre Markierung sich ganz links bzw. ganz unten befindet. *Min* darf größer als *Max* sein (vgl. Ausführungen zu *Max*).

Siehe auch Max (441).

MinButton (Eigenschaft)

Objekte Form

Datentyp INTEGER (True/False)

	→🖫→
E	⊕ ⊕
L	⊖ ⊕

Nutzen MinButton gibt an, ob die Form in der rechten oberen Ecke einen Symbolknopf hat, mit dem sie vom Benutzer auf Symbolgröße verkleinert werden kann:

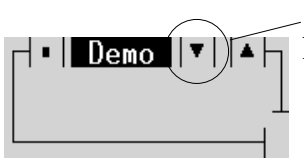


Abbildung 25–7: Form mit Symbolknopf

Bemerkung
- Auch über das Sytemmenü ist eine Verkleinerung möglich. Ein Symbolknopf wird nicht bei *BorderStyle* = 0 angezeigt.
- *MinButton* kann bei MDI-Formen nicht verändert werden.

Siehe auch BorderStyle (414), ControlBox (420), MaxButton (442).

MouseDown, MouseUp (Ereignisse)

Objekte Bezeichnung, Bildfeld, Dateiliste, Form, Listenfeld, Verzeichnisliste

Argumente
```
Button AS INTEGER, Shift AS INTEGER, X AS SINGLE,
Y AS SINGLE
```

Nutzen Die *MouseDown*- und *MouseUp*-Ereignisse treten ein, wenn ein Mausknopf gedrückt (*MouseDown*) oder losgelassen (*MouseUp*) wird. Sie treten für das Objekt ein, über dem der Mausknopf gedrückt wurde; selbst wenn die Maus bei gedrückter Maustaste vom Objekt wegbewegt wird, tritt das *MouseUp*-Ereignis noch für dieses Objekt ein (u. U. mit *X*- und *Y*-Koordinaten, die negativ oder größer als die Breite und Höhe des Objekts sind).

Bei Doppelklicks tritt (seltsamerweise) das *MouseDown*-Ereignis einmal, das *MouseUp*-Ereignis aber zweimal ein.

Button ist entweder 1 (linke Maustaste) oder 2 (rechte Maustaste).

Shift gibt an, ob die Shift-, Strg- oder Alt-Taste(n) während des Drückens oder Loslassens des Mausknopfes gedrückt wurden: 1 = Shift, 2 = Strg, 4 = Alt; Kombinationen sind möglich.

X und *Y* geben an, wo sich der Mauszeiger während des Drückens oder Loslassens des Mausknopfes befand (relativ zur oberen linken Ecke des Objektes, das das Ereignis empfängt).

Siehe auch Click (417), KeyDown, KeyUp (436), MouseMove (443).

MouseMove (Ereignis)

Objekte Bezeichnung, Bildfeld, Dateiliste, Form, Listenfeld, Verzeichnisliste

Argumente
```
Button AS INTEGER, Shift AS INTEGER, X AS SINGLE,
Y AS SINGLE
```

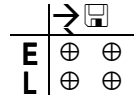
Nutzen *MouseMove* tritt ein, wenn die Maus über ein Objekt bewegt wird. Üblicherweise tritt *MouseMove* für ein Objekt nicht mehr ein, wenn der Mauszeiger den Bereich des Objektes verläßt; wird jedoch auf dem Objekt die Maustaste gedrückt und die Maus bei gedrückter Taste weiterbewegt, empfängt das Objekt weiter *MouseMove*-Ereignisse, auch dann, wenn die Maus aus dem Bereich des Objektes herausbewegt wird. Das trifft nicht zu, wenn die Eigenschaft *DragMode* des Objektes auf 1 gesetzt ist.

Button, *Shift*, *X* und *Y* enthalten die gleichen Werte wie bei den *MouseUp*- und *MouseDown*-Ereignissen, mit dem Unterschied, daß *Button* hier auch den Wert 3 (beide Knöpfe gedrückt) annehmen kann.

Siehe auch DragMode (476), MouseUp, MouseDown (493).

MousePointer (Eigenschaft)

Objekte Alle außer CLIPBOARD, Menüeintrag, PRINTER, Timer



Datentyp INTEGER (0–12)

Nutzen *MousePointer* legt fest, wie der Mauszeiger aussehen soll, während er sich über einem Objekt befindet. Folgende Einstellungen, für die in CONSTANT.BI Konstanten definiert werden, sind möglich:

Nummer	*Aussehen*	*vordefinierte Konstante*
0		DEFAULT
1		BLOCK
2		CROSSHAIR
3		IBEAM
4		ICON
5		SIZE_POINTER
6		LEFT_ARROW
7		SIZE_N_S
8		RIGHT_ARROW
9		SIZE_W_E
10		UP_ARROW
11		HOURGLASS
12		DOWN_ARROW

Bemerkung • Wenn Sie für SCREEN eine andere *MousePointer*-Eigenschaft als DEFAULT festlegen, sieht der Mauszeiger überall auf dem Schirm so aus. Andernfalls gelten die Einstellungen, die für die einzelnen Formen und Steuerelemente gemacht wurden, wobei ein Steuerelement mit *MousePointer* = DEFAULT den Mauszeiger des übergeordneten Steuerelements bzw. der Form übernimmt.

MOVE (Methode)

Objekte Alle außer CLIPBOARD, Menüeintrag, PRINTER, SCREEN und Timer

Verwendung *objekt*.MOVE *links* [, *oben* [, *breite* [, *höhe*]]]

Nutzen MOVE verändert die Position und/oder Größe eines Objektes. Der gleiche Effekt kann erreicht werden, indem Sie eine direkte Zuweisung an die Eigenschaften *Left*, *Top*, *Width* und *Height* vornehmen.

Falls Sie Ihr Programm einmal nach WINDOWS portieren sollten, unterscheidet sich die MOVE-Methode von der direkten Zuweisung, weil man unter WINDOWS am Bildschirm die Bewegung und Größenänderung des Objektes sehen kann (es werden also auch Zwischenstufen angezeigt).

Siehe auch Height (482), Left (488), Top (511), Width (514).

MSGBOX (Befehl und Funktion)

Anwendung (1) *x%* = MSGBOX(*text$* [, *typ%* [, *titel$*]])
(2) MSGBOX(*text$* [, *typ%* [, *titel$*]])

Nutzen Zeigt eine Meldung in einem eigenen Fenster an und bietet dem Benutzer verschiedene Schaltflächen zur Reaktion.

text$ ist die anzuzeigende Meldung (darf CR/LF-Kombinationen enthalten). Wenn *typ%* weggelassen wird, wird 0 verwendet; es gilt folgende Tabelle (mit Konstanten aus CONSTANT.BI):

typ%	*vordefinierte Konstante*	*Schaltflächen*
0	MB_OK	„OK“
1	MB_OKCANCEL	„OK“, „Abbrechen“
2	MB_ABORTRETRYIGNORE	„Abbrechen“, „Wiederholen“, „Ignorieren“
3	MB_YESNOCANCEL	„Ja“, „Nein“, „Abbrechen“
4	MB_YESNO	„Ja“, „Nein“
5	MB_RETRYCANCEL	„Wiederholen“, „Abbrechen“

Standardmäßig ist die erste Schaltfläche voreingestellt; durch Addition von 256 zum *typ%*-Wert erreichen Sie, daß die zweite voreingestellt wird, und mit 512 ist es die dritte.

titel$ ist der Titel, der in der Kopfzeile des Fensters angezeigt wird.

Wenn Sie MSGBOX in der zweiten Syntax als Befehl verwenden, gibt es keinen Rückgabewert; daher ist auch ein anderer *typ%* als 0 nicht sinnvoll.

Die MSGBOX-Funktion hat als Rückgabewert folgende Zahlen:

Wert	*vordefinierte Konstante*	*gedrückte Schaltfläche*
1	IDOK	„OK“
2	IDCANCEL	„Abbrechen“ (oder ESC)
3	IDABORT	„Abbrechen“ bei *typ%* = 2
4	IDRETRY	„Wiederholen“
5	IDIGNORE	„Ignorieren“
6	IDYES	„Ja“
7	IDNO	„Nein“

Bemerkung • Bitte lesen Sie die Bemerkung bei INPUTBOX$, die hier genauso gilt.

Kompatibel PDS ⊕ VBWIN ⊛ Formen ⊖ Mathe ⊕

Siehe auch INPUTBOX$ (483).

MultiLine (Eigenschaft)

Objekte Textfeld

Datentyp INTEGER (True/False)

	→🖫→
E	⊕ ⊕
L	⊖ ⊕

Nutzen *MultiLine* muß im Form-Designer auf TRUE gesetzt werden, wenn ein Textfeld mehrere Zeilen enthalten kann. Es reicht nicht, das Feld mehrere Zeilen groß zu zeichnen. Ein *MultiLine*-Textfeld bricht Text, der in der Breite nicht mehr darstellbar ist, automatisch in die nächste Zeile um, während ein Textfeld ohne *MultiLine* ihn abschneidet.

Siehe auch Height (482), TextHeight (510).

NEWPAGE (Methode)

Objekte PRINTER

Verwendung PRINTER.NEWPAGE

Nutzen NEWPAGE schickt ein Form-Feed-Zeichen (CHR$(12)) an den Drucker, um eine neue Seite zu beginnen.

Siehe auch ENDDOC (479).

Normal (Eigenschaft)

Objekte Dateiliste

Datentyp INTEGER (True/False)

	→🖫→
E	⊕ ⊕
L	⊕ ⊕

Nutzen Gibt an bzw. legt fest, ob in der Dateiliste „normale“ Dateien – das sind Dateien, für die weder das Hidden- noch das System-Attribut gesetzt sind – angezeigt werden sollen.

Bemerkung
- Eine Änderung der Normal-Eigenschaft verursacht ein erneutes Einlesen der Dateiliste.
- Beachten Sie den Abschnitt über Dateilistenfelder im Kapitel 7.

Siehe auch Archive (462), Hidden (482), System (508), ReadOnly (502).

Paint (Ereignis)

Objekte Bildfeld, Form

Argumente keine

Nutzen Das *Paint*-Ereignis tritt immer dann ein, wenn sich der Innenbereich eines Bildfeldes oder einer Form verändert und eventuell neu aufgebaut werden muß. Das kann der Fall sein, wenn die Form erstmals angezeigt, vergrößert, verkleinert oder verschoben wird, aber auch, wenn eine andere Form, die die betroffene Form bisher überdeckte, verschoben wird und dadurch den Blick freigibt.

Paint tritt außerdem ein, wenn die REFRESH-Methode angewandt wird. *Paint* tritt nicht ein, wenn die *AutoRedraw*-Eigenschaft eines Objektes auf TRUE gesetzt ist (siehe illustriertes Beispiel bei *AutoRedraw*).

Siehe auch AutoRedraw (462), Load (490), REFRESH (502), Resize (503).

Parent (Eigenschaft)

Objekte Alle außer CLIPBOARD, PRINTER, SCREEN

Datentyp FORM

	→🖫→
E	⊖ ⊖
L	⊖ ⊕

Nutzen *Parent* gibt die Form zurück, die dem betreffenden Steuerelement bzw. der Form übergeordnet ist. Formen können nur dann einer anderen Form untergeordnet sein, wenn dies eine MDI-Form ist.

Wenn Sie auf die *Parent*-Eigenschaft einer Form, die nicht einer anderen Form untergeordnet ist, zugreifen, wird Fehler 422 erzeugt.

Wenn ein Steuerelement einem anderen untergeordnet ist (z.B. Optionsfelder in einem Rahmen), enthält die *Parent*-Eigenschaft bei allen dennoch die Form und nicht etwa das übergeordnete Steuerelement. Sie haben zur Laufzeit keine Möglichkeit, festzustellen, ob ein Steuerelement einem anderen untergeordnet oder „solo“ auf der Form angebracht ist.

Siehe auch FormType (481).

Path (Eigenschaft)

Objekte Dateiliste, Verzeichnisliste

Datentyp STRING

	→🖫→
E	⊕ ⊕
L	⊕ ⊕

Nutzen *Path* gibt das aktuelle Verzeichnis einer Dateiliste oder einer Verzeichnisliste zurück. Die Funktionsweise unterscheidet sich dabei deutlich:

Verzeichnislisten

Hier gibt *Path* das Verzeichnis zurück, das gerade ausgewählt ist (komplette Bezeichnung inkl. Laufwerksbuchstabe). Per Zuweisung kann ein anderes Verzeichnis ausgewählt oder auch das Laufwerk gewechselt werden. Bei einer solchen Änderung wird ein *Change*-Ereignis für die Verzeichnisliste ausgelöst.

Dateilisten

Hier gibt *Path* das Verzeichnis zurück, dessen Dateien gerade angezeigt werden (komplette Bezeichnung inkl. Laufwerksbuchstabe). Per Zuweisung kann ein anderes Verzeichnis ausgewählt oder auch das Laufwerk gewechselt werden; dies ist aber auch mit der *FileName*-Eigenschaft möglich. Bei einer solchen Änderung wird ein *PathChange*-Ereignis für die Dateiliste ausgelöst.

Siehe auch FileName (479).

PathChange (Ereignis)

Objekte Dateiliste

Argumente keine

Nutzen *PathChange* tritt ein, wenn durch eine Zuweisung in Ihrem Programm die *Path*-Eigenschaft der Dateiliste verändert wird.

Wenn Sie per Zuweisung in *FileName* eine Verzeichnisangabe eintragen, wird die *Path*-Eigenschaft ebenfalls geändert, und ein *PathChange*-Ereignis wird erzeugt.

Siehe auch FileName (429), Path (448).

Pattern (Eigenschaft)

Objekte Dateiliste

Datentyp STRING

	→🖫→	
E	⊕	⊕
L	⊕	⊕

Nutzen *Pattern* gibt die Suchmaske für Dateien in der Dateiliste an. Es wird eine Maske der üblichen DOS-Form (auf Wunsch mit ? und * als Platzhalter) verwendet.

Bei einer Zuweisung an die *Pattern*-Eigenschaft wird die Dateiliste neu aufgebaut; wenn *Pattern* der Name einer existierenden Datei ist, wird ein *DblClick*-Ereignis für die Dateiliste generiert.

Siehe auch FileName (429), Path (448).

PatternChange (Ereignis)

Objekte Dateiliste

Argumente keine

Nutzen Das *PatternChange*-Ereignis tritt ein, wenn Sie durch eine Zuweisung in Ihrem Programm die *Pattern*-Eigenschaft einer Dateiliste ändern.

PatternChange tritt auch ein, wenn die *Pattern*-Eigenschaft indirekt durch eine Zuweisung an die *FileName*-Eigenschaft verändert wird (siehe dort).

Siehe auch FileName (429), Pattern (449).

PRINT (Methode)

Objekte Bildfeld, Form, PRINTER

Verwendung `objekt.PRINT [USING format$] ausdrucksliste`

Nutzen PRINT gibt Daten auf einer Form, einem Bildfeld oder dem Drukker aus. Die Funktionsweise entspricht dem bekannten PRINT-Befehl aus BASIC, mit dem Unterschied, daß die Ausgabe an der durch die Eigenschaften *CurrentX* und *CurrentY* bestimmten Stelle beginnt und diese Eigenschaften auch entsprechend „weitergezählt" werden.

Text, der über den rechten oder unteren Rand einer Form oder eines Bildfeldes gedruckt wird, wird nicht automatisch umgebrochen. Es findet auch kein „Scrollen" statt (wie üblicherweise auf dem Bilschirm). Text, der über den Rand gedruckt wurde, wird allenfalls dann sichtbar, wenn die Form oder das Bildfeld vergrößert wird und die *AutoRedraw*-Eigenschaft auf TRUE gesetzt ist.

Daß Text, der über den unteren Rand hinaus gedruckt wurde, nicht mehr angezeigt und auch manuell der Form-Inhalt nicht „nach oben geschoben" werden kann, ist ein schwerer Mangel an VBDOS. In den VBDOS-Handbüchern steht zwar „verwenden Sie die Bildlaufleisten, um Text anzuzeigen, der über die Grenzen des Bildfeldes bzw. der Form gedruckt wird"; aber wenn Sie das im Sinn haben, müssen Sie in einem eigenen Datenfeld alle Textzeilen speichern, die Sie in der Form/dem Bildfeld ausgegeben haben, und bei jedem *Change*-Ereignis der Bildlaufleiste die gesamte Form bzw. das gesamte Bildfeld neu zeichnen.

Um dieses Problem zu umgehen, können Sie versuchen, ein Textfeld einzusetzen und dieses gegen Veränderungen durch den Benutzer zu schützen (die Zeile `KeyAscii = 0` in der *KeyPress*-Ereignisprozedur reicht dafür aus). Das ist allerdings nicht zuletzt deshalb ein schwacher Trost, weil PRINT, *CurrentX* und *CurrentY* dann nicht zur Verfügung stehen.

Siehe auch CLS (419), CurrentX, CurrentY (423).

PRINTFORM (Methode)

Objekte Form

Verwendung `objekt.PRINTFORM`

Nutzen Druckt die Form auf dem Drucker aus (durch die *PrintTarget*-Eigenschaft ist das Ziel festgelegt). Die Form wird dabei mit allen Steuerelementen und dem Text, der auf ihr sichtbar ist, gedruckt, aber der Druckerunabhängigkeit werden alle Grafikzeichen geopfert, so daß von der Optik der Form nicht mehr viel übrigbleibt:

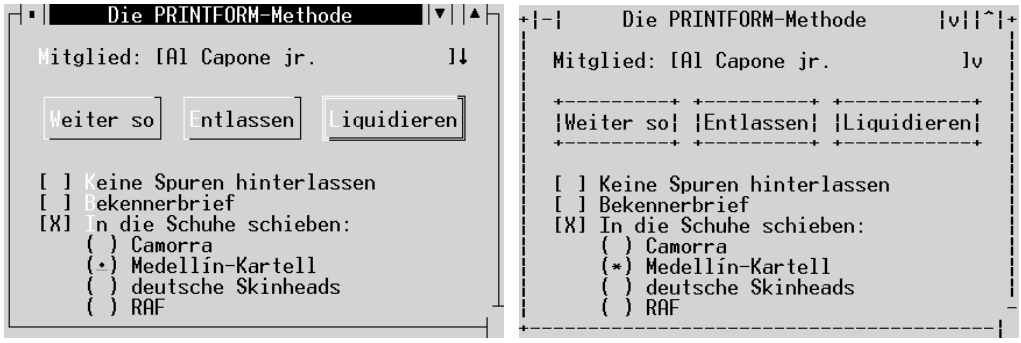


Abbildung 25–8: Eine Form und ihre Ausgabe durch PRINTFORM

Siehe auch PrintTarget (451).

PrintTarget (Eigenschaft)

Objekte PRINTER

Datentyp STRING

	→🖫→
E	⊖ ⊖
L	⊕ ⊕

Nutzen PrintTarget legt fest, wohin die Ausgaben an das PRINTER-Objekt (mit den Methoden ENDDOC, NEWPAGE, PRINT und PRINTFORM) gesendet werden. Standard ist „LPT1“, erlaubt sind aber auch „LPT2“, „LPT3“ und jeder (gültige) Dateiname.

Wenn Sie die *PrintTarget*-Eigenschaft auf einen schreibgeschützten, ungültigen oder sonstwie im Zugriff behinderten Dateinamen setzen, erzeugt die Verwendung der ENDDOC-, NEWPAGE-, PRINT- und PRINTFORM-Methoden einen Fehler, dessen Nummer die gleiche ist wie die, die auftreten würde, wenn man mit PRINT direkt in diese Datei zu schreiben versuchte.

Siehe auch ENDDOC (429), NEWPAGE (446), PRINT (450), PRINTFORM (450).

ReadOnly (Eigenschaft)

Objekte Dateiliste

Datentyp INTEGER (True/False)

	→🖫→
E	⊕ ⊕
L	⊕ ⊕

Nutzen Gibt an bzw. legt fest, ob in der Dateiliste auch Dateien mit Read-Only-Attribut angezeigt werden sollen („Attribute“ sind Dateieigenschaften, die von DOS zusätzlich zum sichtbaren Verzeichniseintrag gespeichert werden).

Bemerkung • Eine Änderung der *ReadOnly*-Eigenschaft verursacht ein erneutes Einlesen der Dateiliste.

Siehe auch Archive (412), Hidden (432), Normal (447), System (457).

REFRESH (Methode)

Objekte Alle außer CLIPBOARD, Menüeintrag, PRINTER, SCREEN und Timer

Verwendung *objekt*.REFRESH

Nutzen REFRESH zeichnet ein Steuerelement neu auf den Bildschirm; dabei wird bei Formen und Bildfeldern ein *Paint*-Ereignis ausgelöst. REFRESH veranlaßt Datei-, Laufwerks- und Verzeichnislisten, ihren Inhalt zu aktualisieren.

Bemerkung • Verwenden Sie REFRESH für die Dateilisten, wenn Sie Dateien erstellen und/oder löschen. Sonst werden Sie REFRESH nicht benötigen, es sei denn, sie müssen annehmen, daß (z. B. durch einen SHELL-Befehl) der Bildschirminhalt überschrieben wurde.

Siehe auch Paint (447).

REMOVEITEM (Methode)

Objekte Kombinationsfeld, Listenfeld

Verwendung *objekt*.REMOVEITEM *position*

Nutzen REMOVEITEM löscht ein Element aus einer Liste. *position* gibt an, welches Element entfernt werden soll (0 = erstes Element). Der Versuch, ein nicht existentes Element zu löschen, scheitert mit *Funktionsaufruf unzulässig* (Fehlercode 5).

Bemerkung • Wenn das gelöschte Element gerade ausgewählt war, ist danach kein Element ausgewählt. Es wird auch kein Ereignis für das Listenfeld erzeugt.

Siehe auch ADDITEM (411), ListIndex (439).

Resize (Ereignis)

Objekte Form

Argumente keine

Nutzen *Resize* tritt ein, wenn die Größe der Form durch Ändern der *Width*- oder der *Height*-Eigenschaft oder durch eine Aktion des Benutzers geändert wird. Der Benutzer kann die Größe der Form durch Ziehen der unteren rechten Ecke, durch Klicken auf den Vollbild- oder Symbolknopf oder mit dem Systemmenü verändern. Sie können dies unterbinden, indem Sie der Form eine feste Größe geben (*BorderStyle*-Eigenschaft) oder die Anzeige des Vollbild- und Symbolknopfes (*MaxButton*, *MinButton*) und des Systemmenüfeldes (*ControlBox*) abschalten.

Sie können *Resize* verwenden, um die Anordnung und Größe der Steuerelemente anzupassen, wenn sich die Größe einer Form ändert. Das ist jedoch mit einer Menge Tüftelei verbunden, wenn die Optik immer stimmen soll; also werden Sie meist die Möglichkeit der Größenänderung abschalten.

Siehe auch BorderStyle (414), ControlBox (420), Height (432), MaxButton (442), MinButton (442), Width (464).

ScaleHeight (Eigenschaft)

Objekte Bildfeld, Form

	→🖫→
E	⊖ ⊖
L	⊖ ⊕

Datentyp INTEGER (0–255)

Nutzen *ScaleHeight* gibt die „innere Höhe“ einer Form oder eines Bildfeldes zurück, also genau die Anzahl von Zeilen, die zur Ausgabe von Text zur Verfügung stehen. Die Differenz zwischen *ScaleHeight* und *Height* ist durch den Rahmen und eine eventuelle Menüzeile bei Formen bedingt.

Bemerkung • Obwohl Sie *ScaleHeight* leicht ausrechnen könnten, ist es sinnvoll, diese Eigenschaft zu verwenden; dann haben Sie es leichter, wenn Sie Ihr Programm nach WINDOWS portieren möchten, wo ein Rahmen nicht immer die Breite von 1 hat.

Siehe auch Height (432).

ScaleWidth (Eigenschaft)

	→🖫→
E	⊖ ⊖
L	⊖ ⊕

Objekte Bildfeld, Form

Datentyp INTEGER (0–255)

Nutzen *ScaleWidth* gibt die „innere Breite“ eines Objekts zurück, also die Anzahl von Spalten, die für die Ausgabe zur Verfügung stehen. Alles weitere siehe *ScaleHeight*.

Siehe auch Width (464).

ScrollBars (Eigenschaft)

	→🖫→
E	⊖ ⊖
L	⊖ ⊕

Objekte Textfeld

Datentyp INTEGER (0–3)

Nutzen Legt fest, ob ein Textfeld automatische Bildlaufleisten haben soll. Wählen Sie den Wert 0 für keine, 1 für horizontal, 2 für vertikal und 3 für beide.

Bemerkung • Weitere Informationen zu automatischen Bildlaufleisten finden Sie im Kapitel 7.

SelLength (Eigenschaft)

	→🖫→
E	⊖ ⊖
L	⊕ ⊕

Objekte Kombinationsfeld (*Style* 0 und 1), Textfeld

Datentyp INTEGER (≥ 0)

Nutzen *SelLength* gibt die Länge der Markierung in einem Texteingabefeld (in Zeichen) an oder stellt diese ein. Mit `SelLength = 0` entfernen Sie die Markierung (ohne jedoch den markierten Text zu löschen).

Der Benutzer kann die Markierung verändern, indem er die Pfeiltasten zusammen mit der Shift-Taste verwendet oder mit gedrückter Maustaste den Cursor in einem Textfeld bewegt.

Siehe auch SelStart (454), SelText (455).

SelStart (Eigenschaft)

	→🖫→
E	⊖ ⊖
L	⊕ ⊕

Objekte Kombinationsfeld (*Style* 0 und 1), Textfeld

Datentyp INTEGER (≥ 0)

Nutzen Gibt die Position innerhalb des Textes eines Kombinations- oder Textfeldes an, an der die Markierung beginnt bzw. der Cursor steht.

Sie können diese Eigenschaft auch durch eine Zuweisung verändern, zum Beispiel um zu verhindern, daß der Benutzer bestimmte feste Bestandteile des Eingabefeldes überschreibt (Datumseingabe mit festen Punkten).

Siehe auch SelLength (454), SelText (455).

SelText (Eigenschaft)

Objekte Kombinationsfeld (*Style* 0 und 1), Textfeld

Datentyp STRING

	→ 🖫 →
E	⊖ ⊖
L	⊕ ⊕

Nutzen Gibt den Teil der *Text*-Eigenschaft eines Steuerelements zurück, der gerade markiert ist. Besteht keine Markierung (*SelLength* = 0), ist *SelText* ein Leerstring.

SelText kann in einer Zuweisung verwendet werden, um den gerade markierten Textteil durch einen anderen zu ersetzen. Besteht zum Zeitpunkt der Ausführung einer `SelText = x$`-Zuweisung keine Markierung, wird `x$` an der Stelle, an der der Cursor steht, eingefügt.

Siehe auch SelStart (454), SelLength (454).

Separator (Eigenschaft)

Objekte Menüeintrag

Datentyp INTEGER (True/False)

	→ 🖫 →
E	⊕ ⊕
L	⊕ ⊕

Nutzen Ein Menüeintrag, dessen *Separator*-Eigenschaft auf TRUE gesetzt ist, wird nicht angezeigt; stattdessen wird an dieser Stelle das Menü durch eine Linie unterteilt. Menüeinträge auf der obersten Ebene können nicht als Trennlinie angezeigt werden. Wenn Sie zur Laufzeit die *Separator*-Eigenschaft eines solchen Menüelements auf TRUE setzen, wird Fehler 387 erzeugt.

SETFOCUS (Methode)

Objekte Alle bis auf CLIPBOARD, Bezeichnungsfeld, Menüeintrag, PRINTER, Rahmen, SCREEN und Timer

Verwendung *objekt*`.SETFOCUS`

Nutzen Setzt den Fokus auf das angegebene Objekt. Für das Objekt wird ein *GotFocus*-Ereignis ausgelöst, und die *SCREEN.ActiveControl*-Eigenschaft wird entsprechend geändert.

Nur Objekte, deren *Visible*- und *Enabled*-Eigenschaft TRUE ist, können den Fokus erhalten. Wenn Sie den Fokus auf eine Form mit untergeordneten Steuerelementen setzen, erhält stattdessen das Steuerelement mit der *TabIndex*-Eigenschaft 0 den Fokus.

Siehe auch Enabled (428), GotFocus (431), Visible (463).

SETTEXT (Methode)

Objekte CLIPBOARD

Verwendung `CLIPBOARD.SETTEXT text$`

Nutzen Schreibt den Text *text$* in das Clipboard. Siehe Anmerkungen bei GETTEXT.

Siehe auch GETTEXT (431).

SHOW (Methode)

Objekte Form, SCREEN

Verwendung `objekt.SHOW [gebunden]`

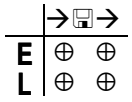
Nutzen SHOW zeigt eine Form am Bildschirm an. Die Wirkung ist dieselbe wie die Zuweisung von TRUE an die *Visible*-Eigenschaft der Form. Wenn Sie die SHOW-Methode auf den Bildschirm anwenden (SCREEN), werden alle Formen, deren *Visible*-Eigenschaft vor Anwendung der SCREEN.HIDE-Methode TRUE war, wieder sichtbar; andere bleiben unsichtbar.

Der optionale Parameter *gebunden* gibt an, ob es sich um eine ungebundene (FALSE) oder gebundene (TRUE) Form handelt. Die Anzeige einer gebundenen Form hält den Programmablauf nach der SHOW-Methode so lange an, bis der Benutzer die Form schließt oder sie vom Programm geschlossen wird. Während eine gebundene Form angezeigt wird, kann der Benutzer auch nicht durch einen Mausklick den Fokus auf eine andere (vielleicht im Hintergrund sichtbare) Form setzen. Im Gegensatz dazu wird bei Anzeige einer ungebundene Form die Programmausführung hinter dem SHOW-Aufruf sofort fortgesetzt.

Siehe auch HIDE (433), Visible (463).

SmallChange (Eigenschaft)

Objekte Horizontale und vertikale Bildlaufleisten



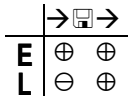
Datentyp INTEGER

Nutzen Legt die Schrittweite an, um die eine Bildlaufleiste bewegt wird, wenn der Benutzer auf die Pfeilsymbole an den Enden der Leiste klickt oder die Pfeiltasten betätigt.

Siehe auch LargeChange (438).

Sorted (Eigenschaft)

Objekte Kombinationsfeld, Listenfeld



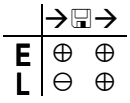
Datentyp INTEGER (True/False)

Nutzen Gibt an, ob die Liste eines Kombinations- oder Listenfelds sortiert werden soll. Wenn Sie *Sorted* auf TRUE setzen, sortiert ADDITEM neue Einträge alphabetisch in die Liste ein, wobei die Groß- und Kleinschreibung ignoriert wird; Umlaute werden exakt wie die zugehörigen Vokale behandelt, das ß wird genau wie s einsortiert.

Siehe auch ADDITEM (411).

Style (Eigenschaft)

Objekte Kombinationsfeld



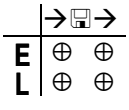
Datentyp INTEGER (0–2)

Nutzen Legt fest, wie sich ein Kombinationsfeld genau verhalten soll. Typ 0 ist ein Dropdown-Kombifeld (Liste klappt bei Bedarf aus, manuelle Eingabe möglich), 1 ist ein einfaches Kombifeld (Liste wird immer angezeigt, manuelle Eingabe möglich), und 2 ist eine Dropdown-Liste, die bei Bedarf ausklappt, aber keine manuelle Eingabe ermöglicht.

Ein illustriertes Beispiel finden Sie im Kapitel 7.

System (Eigenschaft)

Objekte Dateiliste



Datentyp INTEGER (True/False)

Nutzen Gibt an bzw. legt fest, ob in der Dateiliste auch Dateien mit System-Attribut angezeigt werden sollen.

Bemerkung
- Eine Änderung der *System*-Eigenschaft verursacht ein erneutes Einlesen der Dateiliste.
- Beachten Sie hierzu die Ausführungen im Kapitel 7.

Siehe auch Archive (412), Normal (447), Hidden (432), ReadOnly (451).

TabIndex (Eigenschaft)

Objekte Alle bis auf CLIPBOARD, Form, Menüeintrag, PRINTER, SCREEN und Timer

	→🖫→
E	⊛ ⊕
L	⊛ ⊕

Datentyp INTEGER (≥ 0)

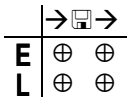
Nutzen Gibt die Stellung des Objektes in der Tabulator-Reihenfolge an oder legt sie fest. Alle Steuerelemente auf einer Form, die mit der Tab-Taste angesteuert werden können, werden mittels ihrer *TabIndex*-Eigenschaft von 0 beginnend durchnumeriert. Entlang dieser Reihenfolge wird dann der Fokus weitergeschaltet, wenn der Benutzer die Tab-Taste drückt. (Steuerelemente, deren *TabStop*-Eigenschaft auf FALSE gesetzt ist oder die gar keine *TabStop*-Eigenschaft besitzen, werden dabei übersprungen.)

Wenn Sie die *TabIndex*-Eigenschaft eines Steuerelements ändern, werden alle anderen so umnumeriert, daß sich eine durchlaufende Nummernfolge ergibt. Ist die von Ihnen eingegebene Nummer größer oder gleich der Anzahl der Steuerelemente, die die *TabIndex*-Eigenschaft besitzen, wird sie entsprechend reduziert.

Bemerkung
- Die *TabIndex*-Eigenschaft dient auch dazu, Objekten ohne eigene *Caption*-Eigenschaft eine Zugriffstaste zuzuordnen. Dazu ordnet man als Überschrift eines solchen Elements auf der Form ein Bezeichnungsfeld mit der gewünschten Zugriffstaste an und sorgt dafür, daß dessen *TabIndex*-Eigenschaft um eins kleiner ist als die des betreffenden Objekts. Wenn der Benutzer nun die Zugriffstaste des Bezeichnungsfeldes drückt, das ja selbst den Fokus nicht erhalten kann, wird der Fokus so lange in der *TabIndex*-Reihenfolge weitergereicht, bis ein Objekt ihn erhalten kann.
- Wenn eine Form erstmals angezeigt wird und Sie SETFOCUS nicht verwenden, hat das Element mit *TabIndex*=0 den Fokus (oder ein folgendes, wenn dieses nicht den Fokus haben kann).

Siehe auch SETFOCUS (455), TabStop (459).

TabStop (Eigenschaft)



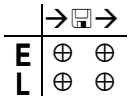
Objekte Alle bis auf CLIPBOARD, Bezeichnungsfeld, Form, Menüeintrag, PRINTER, Rahmen, SCREEN und Timer

Datentyp INTEGER (True/False)

Nutzen Gibt an, ob der Fokus mit der Tab-Taste auf dieses Steuerelement gesetzt werden kann oder nicht. Elemente mit *TabStop* = FALSE werden beim Verarbeiten der Tab-Taste ignoriert.

Siehe auch TabIndex (458).

Tag (Eigenschaft)



Objekte Alle außer CLIPBOARD, PRINTER, SCREEN

Datentyp STRING

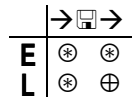
Nutzen *Tag* ist eine Joker-Eigenschaft. Sie können sie für beliebige Zwekke verwenden; VBDOS selbst benutzt diese Eigenschaft nicht. Über die *Tag*-Eigenschaft lassen sich beliebige Daten direkt bei einem Objekt speichern. Der Vorteil von direkt beim Objekt gespeicherten Daten gegenüber explizit (mit DIM) angelegten Datenstrukturen ist, daß die *Tag*-Daten immer zusammen mit dem Objekt verarbeitet werden; so kann zum Beispiel auch über die *ActiveControl*- oder die *Parent*-Eigenschaft eines Objektes auf die *Tag*-Daten eines anderen Objektes zugegriffen werden.

Sie sollten *Tag* immer dann verwenden, wenn ein Objekt seine eigenen, lokalen Daten zu verwalten hat und es nicht ohne weiteres möglich ist, im Formmodul ein Datenfeld zu deklarieren. Das ist zum Beispiel dann der Fall, wenn es sich um ein Steuerelemente-Array handelt, dessen Größe sich während der Programmlaufzeit ändern kann.

Auf der Diskette sind einige Programme, die die *Tag*-Eigenschaft verwenden, darunter auch KLICZEIT.BAS (siehe *DblClick*-Ereignis) und XPROF.FRM (siehe Kapitel 16).

Siehe auch CtlName (422), DblClick (424).

Text (Eigenschaft)



Objekte Kombinationsfeld, Listenfeld, Textfeld

Datentyp STRING

Nutzen *Text* ist bei Kombinations- und Textfeldern der Text im Eingabebereich; er kann per Zuweisung geändert werden. Bei Listenfeldern ist eine Zuweisung nicht erlaubt, und auch zur Entwurfszeit ist die *Text*-Eigenschaft hier nicht verfügbar. Stattdessen gibt *Text* bei Listenfeldern einfach das gerade gewählte Element zurück (entspricht `Feld.List(Feld.ListIndex)`).

Bei Textfeldern mit mehr als einer Zeile (*MultiLine* = TRUE) kann *Text* auch CR/LF-Kombinationen (`CHR$(13)+CHR$(10)`) enthalten, um den Anfang einer neuen Zeile zu erzwingen. Sonst wird eine neue Zeile da begonnen, wo aufgrund der Breite ein Umbruch erforderlich ist. Wenn das Textfeld horizontale automatische Bildlaufleisten besitzt (*ScrollBars*-Eigenschaft), erfolgt überhaupt kein automatischer Umbruch.

Siehe auch SelText (455).

TEXTHEIGHT, TEXTWIDTH (Methoden)

Objekte Bildfeld, Form

Verwendung
```
objekt.TEXTHEIGHT (text$)
objekt.TEXTWIDTH (text$)
```

Nutzen TEXTHEIGHT gibt die Höhe eines Textes (in Zeilen) zurück. Die Höhe ist 1, solange sich keine CR/LF-Kombination (`CHR$(13)+CHR$(10)`) im Text befindet, ansonsten ist sie um 1 größer als die Anzahl dieser Steuerzeichen.

TEXTWIDTH gibt die Breite eines Textes (in Zeichen) zurück. Diese Breite entspricht `LEN(text$)`, solange keine CR/LF-Kombinationen enthalten sind, und der Länge der längsten Zeile, wenn der *text$* diese Steuerzeichen enthält..

Bemerkung
- In VBDOS ist es völlig egal, auf welches Objekt Sie die Methoden anwenden, um die Höhe bzw. Breite eines Textes zu ermitteln. In VB für WINDOWS kommt es auf die jeweils eingestellte Schriftart an (die Größe wird in kleineren Einheiten als „Zeichen“ ermittelt), daher wird hier unterschieden.

Timer (Ereignis)

Objekte Timer („Zeitmesser“)

Argumente keine

Nutzen Das *Timer*-Ereignis tritt ein, wenn bei einem *Timer*-Objekt die in der Eigenschaft *Interval* angegebene Zeit abgelaufen und die *Enabled*-Eigenschaft des Objekts auf TRUE gesetzt ist.

Um zum Beispiel eine Uhr anzuzeigen, die jede Minute weitergeschaltet wird, würden Sie ein Timer-Objekt mit *Interval* = 60.000 (Millisekunden) erstellen und in der zugehörigen Timer-Ereignisprozedur die Zeit ausgeben.

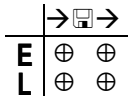
Bemerkung • Die Zeit läuft weiter, während die Timer-Ereignisprozedur arbeitet. Falls vor Ende der Ereignisprozedur die DOEVENTS-Funktion aufgerufen wird, kann es sein, daß ein weiteres *Timer*-Ereignis eintritt, obwohl das erste noch nicht vollständig verarbeitet ist. In solchen Fällen sollte die *Enabled*-Eigenschaft des Zeitmessers am Anfang der Prozedur auf FALSE und erst bei Verlassen wieder auf TRUE gesetzt werden, um endlose Schleifen zu verhindern.

• Ein *Timer*-Ereignis, das nicht eintreten kann, weil gerade ein länger andauernder anderer Prozeß läuft, wird „aufbewahrt“ und nach Abschluß dieses Prozesses (bzw. beim nächsten DOEVENTS-Aufruf) ausgeführt. Solange bereits ein *Timer*-Ereignis derart in der Warteschlange steht, werden keine weiteren ausgelöst.

Siehe auch DOEVENTS (424).

Top (Eigenschaft)

Objekte Alle außer CLIPBOARD, Menüeintrag, PRINTER, SCREEN, Timer



Datentyp INTEGER (0–254)

Nutzen Gibt die Zeilenposition eines Objektes im übergeordneten Objekt an. Ein Wert von 0 bedeutet, daß das Objekt in der ersten Zeile des Innenbereichs des übergeordneten Objekts angezeigt wird. Alles weitere siehe *Left*.

Bemerkung • *Top* kann bei MDI-Formen nicht verändert werden.

Siehe auch Left (438), Height (432).

UnLoad (Ereignis)

Objekte Form

Argumente `Cancel AS INTEGER`

Nutzen Dieses Ereignis tritt ein, bevor eine Form aus dem Speicher entfernt wird. Dies kann entweder durch Ihr Programm mit der UNLOAD-Anweisung oder durch den Benutzer geschehen, wenn er die Tastenkombination Strg+F4 drückt oder im Systemmenü „Schließen“ wählt.

Sie können das Ereignis *UnLoad* zum Beispiel verwenden, um die Daten, die in einer Form gespeichert sind, in eine Datei zu schreiben, bevor sie gelöscht wird. (Eine Form kann ja sehr viele Daten in Form von Textfeld- oder Listeninhalten beherbergen.)

Wenn Sie in Ihrer Prozedur die Variable *Cancel* auf TRUE setzen, wird die Form nicht aus dem Speicher entfernt. Das ist sinnvoll, wenn Sie z. B. beim Versuch, eine Form zu schließen, eine Meldung à la „Sind Sie sicher?“ anzeigen und der Benutzer dann „Nein“ wählt.

Siehe auch Load (440), UNLOAD (Befehl) (462).

UNLOAD (Befehl)

Anwendung (1) `UNLOAD formname`
(2) `UNLOAD objekt(index)`

Nutzen Entfernt eine Form oder ein Element eines Steuerelemente-Arrays aus dem Speicher. Elemente aus Steuerelement-Arrays können nur dann entfernt werden, wenn sie zur Laufzeit mit LOAD erzeugt wurden; der Versuch, ein zur Entwurfszeit erstelltes Element zu entfernen, resultiert in einem Fehler 362.

Im Gegensatz zur HIDE-Methode entfernt der UNLOAD-Befehl eine Form nicht nur vom Bildschirm, sondern auch aus dem Speicher. Alle Veränderungen der Form-Eigenschaften oder der Eigenschaften der Steuerelemente dieser Form werden verworfen. Zwar kann auch danach auf die Eigenschaften der Form zugegriffen werden, aber dann wird sie durch diesen Zugriff neu geladen, und alle Eigenschaften haben ihre zur Entwurfszeit definierten Standardeinstellungen (vgl. LOAD-Befehl).

Beim Löschen eines Steuerelements aus einem Steuerelemente-Array muß der Objektname und der *index* des zu löschenden Steuerelements angegeben werden.

Kompatibel PDS ⊖ VBWIN ⊕ Formen ⊕ Mathe ⊖

Siehe auch LOAD (440), Index (434), SHOW (456).

Value (Eigenschaft)

Objekte Horizontale und vertikale Bildlaufleiste, Schaltfläche, Kontrollfeld, Optionsfeld

	→🖫→
E	⊛ ⊛
L	⊕ ⊕

Datentyp INTEGER

Nutzen Gibt den Status eines „numerischen“ Steuerelements zurück. Schaltflächen und Kontrollfelder können TRUE („gerade gedrückt“ bzw. „markiert“) und FALSE („nicht gedrückt“ bzw. „nicht markiert“) zurückliefern; Bildlaufleisten können jeden Wert zwischen ihren *Min*- und *Max*-Eigenschaften haben, und Kontrollfelder haben einen *Value* von 0 („nicht angekreuzt“), 1 („angekreuzt“) oder 2 („durchgestrichen“). In den Status 2 können Sie ein Kontrollfeld nur vom Programm aus versetzen; der Benutzer kann dies nicht.

Value kann in einer Zuweisung verwendet werden; bei Bildlaufleisten tritt dann ein *Change*-Ereignis ein (sofern sich der Wert verändert), bei Schaltflächen tritt ein *Click*-Ereignis ein, wenn Sie *Value* TRUE zuweisen.

Siehe auch Change (416), Click (417).

Visible (Eigenschaft)

Objekte Alle bis auf CLIPBOARD, SCREEN, PRINTER, Timer

	→🖫→
E	⊕ ⊕
L	⊕ ⊕

Datentyp INTEGER (True/False)

Nutzen Gibt an, ob das betreffende Objekt sichtbar ist. Ein nicht sichtbares Objekt kann keinerlei Ereignisse empfangen. Sie können diese Eigenschaft zum Beispiel verwenden, wenn Sie mittels eines Laufwerkslistenfeldes eine Übersicht über alle Laufwerke einlesen wollen, diese jedoch nicht angezeigt werden soll. Außerdem können Sie mit dem Visible-Trick mehrere überlappende Steuerelemente auf einer Form anlegen, von denen Sie jeweils nur einige sichtbar machen, so daß dieselbe Form verschiedene Erscheinungsbilder haben kann. Ein gutes – fast schon unübersichtliches – Beispiel

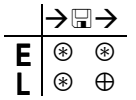
hierfür ist das Projekt CMNDLG, das im Lieferumfang von VBDOS enthalten ist.

Die *Visible*-Eigenschaft von Formen wird auch verändert, wenn Sie die HIDE- und SHOW-Methoden verwenden.

Siehe auch Enabled (428), HIDE (433), SHOW (456).

Width (Eigenschaft)

Objekte Alle außer CLIPBOARD, Menüeintrag, Timer



Datentyp INTEGER

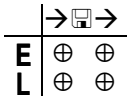
Nutzen Bestimmt die äußere Breite eines Objektes. Die Breite des PRINTER-Objektes ist die Zeilenlänge, nach der automatisch eine neue Zeile angefangen wird (vgl. WIDTH-Befehl im Diskettenreferenzteil). Die Breite des SCREEN-Objektes ist immer 80 und kann nicht verändert werden. SCREEN und PRINTER sind zur Entwurfszeit nicht verfügbar, alle anderen *Width*-Eigenschaften können bereits im Form-Designer festgelegt werden.

Bemerkung • *Width* kann bei MDI-Formen nicht verändert werden.

Siehe auch ScaleWidth (454), Height (432).

WindowState (Eigenschaft)

Objekte Form



Datentyp INTEGER (0–2)

Nutzen Gibt an, wie eine Form dargestellt wird: 0 für „normal“, 1 für „auf Symbolgröße verkleinert“ und 2 für „auf Vollbild vergrößert“. Ob eine solche Verkleinerung oder Vergrößerung möglich ist, wird durch die *BorderStyle*-, die *ControlBox*-, die *MinButton*- und die *MaxButton*-Eigenschaft determiniert.

Bemerkung • *WindowState* kann bei MDI-Formen nicht verändert werden.

Siehe auch BorderStyle (414), ControlBox (420), MinButton (442), MaxButton (442).

BEGINTRANS (Befehl)

Anwendung BEGINTRANS

Nutzen Mit BEGINTRANS beginnt eine ISAM-Transaktion. Eine Transaktion ist eine Gruppe von beliebig vielen Operationen mit beliebig vielen ISAM-Datenbanken. Transaktionen können nicht ineinander geschachtelt werden, das heißt, daß eine mit BEGINTRANS begonnene Transaktion erst durch CLOSE oder COMMITTRANS beendet werden muß, bevor eine neue beginnen kann.

Transaktionen sind deshalb sinnvoll, weil innerhalb ihrer bereits ausgeführte Befehle, die die Datenbank verändert haben, mittels SAVEPOINT und ROLLBACK wieder rückgängig gemacht werden können.

Siehe auch COMMITTRANS (464), SAVEPOINT (471).

BOF (Funktion)

Anwendung *x%* = BOF(*dateinummer*)

Nutzen Stellt fest, ob der Dateizeiger vor dem ersten Datensatz in einer ISAM-Datei steht. *dateinummer* ist die Nummer der geöffneten ISAM-Datenbank. Die Funktion gibt –1 (TRUE) zurück, wenn der Dateizeiger vor dem ersten Datensatz (nach der gerade aktiven Indexliste) steht, und 0, wenn das nicht der Fall ist.

Bemerkung • Vor den ersten Datensatz kann der Dateizeiger ausschließlich durch MOVEPREVIOUS gelangen. Für eine gerade neu erstellte ISAM-Datenbank, die noch keine Datensätze enthält, ist BOF immer TRUE.

Siehe auch EOF (467).

CHECKPOINT (Befehl)

Anwendung CHECKPOINT

Nutzen CHECKPOINT veranlaßt ISAM, alle im RAM gepufferten Daten sofort auf die Festplatte in die entsprechenden ISAM-Dateien zu speichern.

ISAM schreibt normalerweise nicht alle Änderungen, die an ISAM-Dateien gemacht werden, sofort auf die Platte. Unter Umständen werden – vor allem, wenn man EMS besitzt – sehr viele Daten im Speicher zwischengepuffert, um die Anzahl der Zugriffe auf die Festplatte zu minimieren und so die Geschwindigkeit zu erhöhen.

Bemerkung • Die Tatsache, daß mit ROLLBACK Operationen innerhalb einer Transaktion zurückgenommen werden können, wird von CHECKPOINT in keiner Weise beeinflußt.

CLOSE (Befehl)

Anwendung `CLOSE [[#]dateinummer [,[#]dateinummer]...]`

Nutzen Wenn CLOSE auf eine ISAM-Datenbank angewendet wird, wird diese geschlossen, und falls gerade eine Transaktion läuft, wird automatisch ein COMMITTRANS-Befehl ausgeführt.

Das geschieht übrigens *immer*, wenn irgendeine ISAM-Datei geschlossen wird, also auch bei den Befehlen RESET, SYSTEM und RUN oder bei verschiedenen Fehlern, die in Zusammenhang mit ISAM-Operationen auftreten können.

Bemerkung • Details zu CLOSE siehe im Diskettenreferenzteil.

Siehe auch COMMITTRANS (464).

COMMITTRANS (Befehl)

Anwendung `COMMITTRANS`

Nutzen Beendet die gerade laufende Transaktion. COMMITTRANS darf nicht benutzt werden, wenn keine Transaktion läuft; andernfalls wird ein *Funktionsaufruf unzulässig*-Fehler auftreten.

Während Datenänderungen innerhalb der laufenden Transaktion mit dem ROLLBACK-Befehl zurückgenommen werden können, sind Änderungen nach einem COMMITTRANS-Befehl endgültig.

COMMITTRANS wird automatisch immer dann ausgeführt, wenn irgendeine ISAM-Datei geschlossen wird.

Bemerkung • COMMITTRANS ist nicht dafür zuständig, Daten aus dem Puffer in die ISAM-Datei zu schreiben – dazu dient CHECKPOINT.

Siehe auch BEGINTRANS (463), CLOSE (464).

CREATEINDEX (Befehl)

Anwendung `CREATEINDEX [#]dateinummer, indexname$, universell, feldname$ [, feldname$]...`

Nutzen Erstellt zu der Datenbank, die unter der Nummer *dateinummer* geöffnet wurde, einen neuen Index. *indexname*$ ist der Name für den neuen Index (nur Buchstaben und Zahlen, erstes Zeichen ein Buchstabe, maximal 30 Zeichen lang). Der Name darf noch nicht vergeben sein.

Für *universell* müssen Sie 0 einsetzen, wenn die indizierten Felder einen Datensatz nicht eindeutig identifizieren. Wenn Sie also zum Beispiel in einer Adreßdatei nur Vor- und Nachnamen indizieren, wäre 0 angebracht, denn es können durchaus mehrere Personen den gleichen Namen haben. Indizieren Sie andererseits beispielsweise ein Feld namens „Gehaltskonto“, können Sie einen von 0 verschiedenen Wert (der Übersichtlichkeit halber am besten –1 für TRUE, obwohl es egal ist) für *universell* verwenden. Wenn dann jemals der Versuch gemacht wird, einen Datensatz in die Datenbank aufzunehmen, der gegen die Eindeutigkeit verstößt (zum Beispiel, indem er die gleiche Gehaltskontonummer hat wie ein bereits existierender), wird der Fehler 86 (*Doppelter Wert für eindeutigen Index)* erzeugt.

Schließlich können Sie beliebig viele *feldname$*-Strings angeben; jeder *feldname$* muß den Namen eines Elements des Datentyps, mit dem die Datenbank geöffnet wurde, enthalten. Nicht als Basis für einen Index benutzt werden dürfen Arrays und selbstdefinierte Datentypen. Außerdem darf die Summe der Längen aller angegebenen Felder 255 nicht überschreiten.

Pro Datenbank können maximal 28 Indexlisten erstellt werden.

Wenn der so erstellte Index mit SETINDEX aktiviert wird, ist damit eine Sortierfolge aktiv, die sich primär nach dem ersten genannten *feldname$*, sekundär nach dem zweiten etc. richtet.

Bemerkung • Je mehr Indexlisten einem Datenbestand zugeordnet sind, desto länger dauert es, einen UPDATE-, INSERT- oder DELETE-Befehl auszuführen, weil jedesmal sämtliche Indizes aktualisiert werden müssen. Die benötigte Zeit ist zwar gering, summiert sich aber und kann vor allem bei automatischen Prozessen so stark ins Gewicht fallen, daß es zuweilen besser ist, einen Index zunächst zu löschen

und nach einer umfangreichen Serie von Veränderungen neu zu erstellen.

Siehe auch DELETEINDEX (466), SETINDEX (472), GETINDEX$ (467).

DELETE (Befehl)

Anwendung `DELETE [#]dateinummer`

Nutzen DELETE löscht aus der ISAM-Datenbank, die unter der Nummer *dateinummer* geöffnet wurde, den Datensatz, auf den der Datenzeiger gerade zeigt. Zeigt er vor das erste oder hinter das letzte Element, wird der Fehler *Kein aktueller Datensatz* generiert.

Bemerkung • Nach dem Löschen eines Datensatzes zeigt der Datenzeiger für die betreffende Datei auf den Datensatz, der nach der gerade aktiven Sortierfolge (dem aktiven Index) auf den gelöschten folgte.

Siehe auch INSERT (468), UPDATE (473), ROLLBACK (470).

DELETEINDEX (Befehl)

Anwendung `DELETEINDEX [#]dateinummer, indexname$`

Nutzen Löscht einen Index aus einer ISAM-Datenbank. *dateinummer* ist die Nummer, unter der die Datenbank geöffnet wurde; *indexname$* ist der Name der zu löschenden Indexliste.

Bemerkung • Wenn Sie den aktuellen Index löschen, wird der Null-Index aktiv, der Index also, bei dem die Elemente in der Reihenfolge geordnet sind, in der sie aufgenommen wurden. Der Null-Index kann nicht gelöscht werden.

Siehe auch CREATEINDEX (465), SETINDEX (472).

DELETETABLE (Befehl)

Anwendung `DELETETABLE isamfile$, datenbank$`

Nutzen Löscht eine ganze Datenbank mit all ihren Datensätzen und Indizes aus einer ISAM-Datei. *isamfile$* ist der Name der Datei, *datenbank$* der Name der zu löschenden Datenbank.

Sie können zwar Datenbanken aus einer ISAM-Datei löschen, in der eine andere Datenbank gerade geöffnet ist, niemals aber eine Datenbank, die selbst gerade geöffnet ist.

Bemerkung • DELETETABLE-Befehle zählen nicht als Bestandteil einer Transaktion und können deshalb nicht wieder rückgängig gemacht werden.

• Wenn Sie die einzige Datenbank aus einer ISAM-Datei löschen, existiert die Datei trotzdem weiterhin.

Siehe auch OPEN (469).

EOF (Funktion)

Anwendung `x% = EOF(dateinummer)`

Nutzen Stellt fest, ob der Dateizeiger hinter dem letzten Datensatz einer ISAM-Datenbank steht. *dateinummer* ist die Nummer der geöffneten ISAM-Datenbank. Die Funktion gibt –1 (TRUE) zurück, wenn der Dateizeiger hinter dem letzten (nach der gerade aktiven Indexliste) Datensatz steht, und 0, wenn das nicht der Fall ist.

Bemerkung Hinter den letzten Datensatz kann der Dateizeiger ausschließlich durch MOVENEXT oder einen der SEEK-Befehle gelangen. Dann wird EOF benötigt, damit man nicht versehentlich versucht, einen Datensatz einzulesen, solange der Dateizeiger dort steht. Für eine gerade neu erstellte ISAM-Datenbank, die noch keine Datensätze enthält, ist EOF immer TRUE.

Siehe auch BOF (463).

FILEATTR (Funktion)

Anwendung `x = FILEATTR (dateinummer, typ)`

Nutzen In Ergänzung zu seiner üblichen Bedeutung gibt FILEATTR für ISAM-Dateien bei *typ* = 1 den Wert 64 und für *typ* = 2 den Wert 0 zurück.

Bemerkung Details zu FILEATTR siehe im Disketten-Referenzteil.

GETINDEX$ (Funktion)

Anwendung `x$ = GETINDEX$(dateinummer)`

Nutzen Gibt den Namen des gerade aktiven Index der ISAM-Datenbank zurück, die unter der Nummer *dateinummer* geöffnet wurde.

Wenn in der betreffenden Datenbank der Null-Index aktiv ist, gibt die Funktion GETINDEX$ einen Leerstring zurück.

Siehe auch SETINDEX (472), CREATEINDEX (465).

INSERT (Befehl)

Anwendung `INSERT [#]dateinummer, datensatz`

Nutzen Fügt einen neuen Datensatz an eine ISAM-Datenbank an. *dateinummer* ist die Nummer, unter der die Datenbank geöffnet wurde; *datensatz* ist eine Variable von dem selbstdefinierten Typ, mit dem die Datenbank auch geöffnet wurde.

Der Datensatz wird an die Datenbank angehängt und sofort in alle Indizes korrekt einsortiert. Falls es Konflikte mit einem Index gibt (siehe *universell*-Parameter bei CREATEINDEX), kann ein Fehler generiert werden.

Bemerkung • Der Dateizeiger wird durch den INSERT-Befehl nicht verändert, das heißt, er zeigt nach dem Befehl auf denselben Datensatz, auf den er schon vorher zeigte.

Siehe auch UPDATE (473), DELETE (466), CREATEINDEX (465).

LOF (Funktion)

Anwendung `x = LOF(dateinummer)`

Nutzen Auf ISAM-Datenbanken angewandt, gibt LOF die Anzahl der Datensätze in der Datenbank zurück, die unter der Nummer *dateinummer* geöffnet wurde.

Bemerkung • Details zu LOF siehe im Diskettenreferenzteil.

MOVE*xxxx* (Befehle)

Anwendung
```
MOVEFIRST [#]dateinummer
MOVELAST [#]dateinummer
MOVENEXT [#]dateinummer
MOVEPREVIOUS [#]dateinummer
```

Nutzen Die MOVE-Befehle verschieben den Dateizeiger in einer ISAM-Datenbank. *dateinummer* ist jeweils die Nummer, unter der die Datenbank geöffnet wurde.

MOVEFIRST setzt den Zeiger auf den ersten Datensatz nach der aktiven Sortierfolge (dem aktiven Index); MOVEFIRST wird bei der Ausführung von SETINDEX automatisch durchgeführt.
MOVELAST setzt den Zeiger auf den letzten Datensatz.
MOVEPREVIOUS setzt den Zeiger auf den Datensatz, der dem aktuellen Datensatz (dem, auf den der Zeiger gerade zeigt) vorangeht. Wenn der erste Datensatz der aktuelle ist, setzt MOVEPREVIOUS den Zeiger *vor* den ersten Datensatz, so daß BOF(*dateinummer*) TRUE wird. Wird MOVEPREVIOUS ausgeführt, wenn das schon der Fall ist, ignoriert ISAM den Befehl, ohne einen Fehler zu verursachen. MOVENEXT setzt den Zeiger auf den Datensatz, der dem aktuellen Datensatz folgt. Wenn der letzte Datensatz der aktuelle ist, setzt MOVENEXT den Zeiger *hinter* den ersten Datensatz, so daß EOF(*dateinummer*) TRUE wird. Wird MOVENEXT ausgeführt, wenn das schon der Fall ist, ignoriert ISAM den Befehl, ohne einen Fehler zu verursachen.

Siehe auch BOF (463), EOF (467), SEEK*xx* (471), SETINDEX (472).

OPEN (Befehl)

Anwendung
```
OPEN dateiname$ FOR ISAM datentyp datenbank$
     AS [#]dateinummer
```

Nutzen In Ergänzung zu seinen anderen Funktionen dient OPEN auch zum Öffnen einer ISAM-Datenbank.

dateiname$ ist ein gültiger Dateiname, der Laufwerk und Pfad enthalten darf. Wenn die Datei schon vorhanden ist, muß es eine ISAM-Datei sein. Ist sie noch nicht vorhanden, wird sie als ISAM-Datei neu erstellt. Beachten Sie, daß hierfür die erweiterten ISAM-Funktionen (PROISAMD.EXE) benötigt werden.

datenbank$ ist der Name der Datenbank innerhalb der ISAM-Datei, auf die zugegriffen werden soll. Wenn sie noch nicht existiert, wird sie neu erstellt, wozu ebenfalls die erweiterten ISAM-Funktionen gebraucht werden. Existiert die Datenbank schon, so muß der *datentyp* (siehe unten) entweder identisch mit dem sein, der bei OPEN angegeben wurde, als die Datenbank erstellt wurde, oder er muß eine Teilmenge der Felder enthalten, die dieser ursprüngliche Typ hatte. Wird nur ein Teil-Typ angegeben, kann auch nur auf die in ihm enthaltenen Felder zugegriffen werden.

Mit dieser Form des OPEN-Befehls wird strenggenommen keine Datei, sondern nur eine Datenbank geöffnet. Dieselbe Datei kann, wenn sie mehrere Datenbanken enthält, unter mehreren Dateinummern gleichzeitig geöffnet sein.

datentyp ist der Name eines mit TYPE...END TYPE selbstdefinierten Datentyps, über den der Zugriff auf die ISAM-Datenbank laufen soll. Von diesem Datentyp müssen auch die Variablen sein, die man beim Zugriff auf die Datei mit den INSERT-, UPDATE- und RETRIEVE-Befehlen verwendet.

dateinummer ist eine Nummer zwischen 1 und 255, unter der sich nachfolgende Befehle auf die geöffnete Datei beziehen können.

Bemerkung • Die ISAM-fremden Verwendungsweisen von OPEN finden Sie im Diskettenreferenzteil.

• Es können maximal 13 ISAM-Datenbanken zugleich geöffnet sein (siehe auch „Anzahl gleichzeitig geöffneter ISAM-Dateien und -Datenbanken“ in Kapitel 12).

Siehe auch CLOSE (464).

RETRIEVE (Befehl)

Anwendung
```
RETRIEVE [#]dateinummer, datensatz
```

Nutzen Liest den Datensatz, auf den der Dateizeiger zeigt, aus der ISAM-Datenbank, die unter der Nummer *dateinummer* geöffnet wurde, in die Variable *datensatz*. *datensatz* muß denselben Typ haben, mit dem die Datenbank geöffnet wurde.

Bemerkung • Der Versuch, einen Datensatz mit RETRIEVE zu lesen, wenn der Dateizeiger vor dem ersten oder hinter dem letzten Datensatz steht, führt zu einem Fehler 85 (*Kein aktueller Datensatz*).

Siehe auch UPDATE (473), INSERT (468).

ROLLBACK (Befehl)

Anwendung
```
ROLLBACK [markierung]
ROLLBACK ALL
```

Nutzen ROLLBACK ist nur im Rahmen einer Transaktion zulässig und hat die Aufgabe, bereits ausgeführte ISAM-Befehle rückgängig zu machen. ROLLBACK ohne Argument stellt den Zustand wieder her, der beim letzten Aufruf der SAVEPOINT-Funktion herrschte.

ROLLBACK gefolgt von einer Zahl *markierung* stellt den Zustand wieder her, in dem sich die ISAM-Datenbanken zum Zeitpunkt der SAVEPOINT-Markierung *markierung* befanden. ROLLBACK ALL schließlich macht alle ISAM-Befehle rückgängig, die seit BEGINTRANS ausgeführt wurden.

Befehle aus einer Transaktion, die mit COMMITTRANS (oder CLOSE) abgeschlossen ist, können nicht mehr mit ROLLBACK rückgängig gemacht werden.

Wenn ROLLBACK aufgerufen wird, obwohl bereits der älteste Zustand wiederhergestellt ist, wird kein Fehler erzeugt, sondern der Befehl ignoriert. Wurde SAVEPOINT überhaupt nicht aufgerufen, stellt ROLLBACK den Status wieder her, der bei BEGINTRANS herrschte.

Bemerkung • ROLLBACK wirkt auf alle geöffneten ISAM-Datenbanken gleichzeitig, nicht aber auf andere Dateien.

Siehe auch BEGINTRANS (463), COMMITTRANS (464), SAVEPOINT.

SAVEPOINT (Funktion)

Anwendung `x = SAVEPOINT`

Nutzen Setzt innerhalb einer Transaktion eine SAVEPOINT-Markierung, so daß der Status aller geöffneten ISAM-Datenbanken zum Zeitpunkt des Setzens der Markierung später, nach Veränderungen der Datenbank, mit ROLLBACK wiederhergestellt werden kann.

SAVEPOINT kann nur innerhalb einer laufenden Transaktion benutzt werden und gibt als Funktionswert die laufende Nummer der SAVEPOINT-Markierung zurück, die später als Argument zu ROLLBACK benutzt werden muß, wenn man einen bestimmten Zustand wiederherstellen möchte.

Siehe auch BEGINTRANS (463), COMMITTRANS (464), ROLLBACK (470).

SEEKxx (Befehle)

Anwendung
```
SEEKEQ [#]dateinummer, wert [, wert]...
SEEKGE [#]dateinummer, wert [, wert]...
SEEKGT [#]dateinummer, wert [, wert]...
```

Nutzen Den Dateizeiger einer ISAM-Datei auf den ersten Datensatz der Datei setzen, dessen indizierte Felder...
bei SEEKEQ: gleich den angegebenen *werten* sind.
bei SEEKGE: größer oder gleich den angegebenen *werten* sind.
bei SEEKGT: größer als die angegebenen *werte* sind.

dateinummer ist die Nummer, unter der die ISAM-Datenbank geöffnet wurde.

Es muß mindestens ein *wert* angegeben werden; maximal dürfen es so viele sein, wie Felder mit dem aktuellen Index indiziert sind. Wenn also ein mit `CREATEINDEX 1, "DemoIndex", 0, "Name", "Vorname", "PLZ", "Ort"` erstellter Index aktiv ist, dürfen maximal vier *werte* angegeben werden, wobei der erste vom Typ her zum Feld „Name“, der zweite zum Feld „Vorname“ usw. passen muß.

Bei SEEKEQ ist es nicht sinnvoll, weniger Werte anzugeben als mit dem aktiven Index Felder indiziert sind, weil SEEKEQ dann immer fehlschlägt. Die anderen beiden, SEEKGT und SEEKGE, verkraften das allerdings ohne weiteres.

Wenn eine SEEK*xx*-Operation fehlschlägt, wenn also kein Datensatz gefunden wird, der den Bedingungen genügt, wird der Dateizeiger hinter den letzten Datensatz in der Datenbank gesetzt, so daß EOF(*dateinummer*) TRUE wird.

Bemerkung • SEEK*xx*-Befehle können nicht benutzt werden, wenn der Null-Index aktiv ist. Das wäre auch gar nicht sinnvoll, denn wenn die Daten unsortiert sind, haben die SEEK*xx*-Befehle keinen praktischen Nutzen, weil sie immer nur den ersten passenden Datensatz aus der Datei suchen.

Siehe auch MOVE*xxxx* (468), CREATEINDEX (465), SETINDEX (472), TEXTCOMP (473).

SETINDEX (Befehl)

Anwendung `SETINDEX [#]dateinummer, indexname$`

Nutzen Macht den Index *indexname$* zum aktiven Index in der ISAM-Datenbank, die unter der Nummer *dateinummer* geöffnet wurde.

indexname$* muß der Name eines vorhandenen (mit CREATEINDEX erstellten) Index sein. Wenn Sie einen Leerstring ("") für *indexname$ verwenden, wird der Null-Index aktiv.

Siehe auch CREATEINDEX (465); GETINDEX$ (467).

TEXTCOMP (Funktion)

Anwendung `x% = TEXTCOMP(string1$, string2$)`

Nutzen TEXTCOMP vergleicht die beiden Strings *string1$* und *string2$* nach ISAM-Maßstäben (siehe „Wie ISAM Strings sortiert“ in Kapitel 12). TEXTCOMP gibt –1 zurück, wenn *string1$* kleiner als *string2$* ist, 0, wenn beide Strings gleich sind, und 1, wenn *string2$* kleiner *string1$* ist.

TEXTCOMP erzeugt zwar keinen Fehler, wenn die Strings länger als 255 Zeichen sind, vergleicht dann aber nur die ersten 255 Zeichen.

Bemerkung • TEXTCOMP findet auch Erwähnung im Kapitel 20 („Veränderungen an den Sortier- und Suchalgorithmen“).

• Auf der Diskette befindet ein Programm namens LONGCOMP.BAS, mit der es möglich ist, auch längere Strings mit TEXTCOMP zu vergleichen.

Siehe auch SEEK*xx* (471).

UPDATE (Befehl)

Anwendung `UPDATE [#]dateinummer, datensatz`

Nutzen Überschreibt den Datensatz, auf den in der ISAM-Datenbank der Dateizeiger zeigt, durch den Inhalt der angegebenen Variable *datensatz*.

dateinummer ist die Nummer, unter der die Datenbank geöffnet wurde; *datensatz* ist eine Variable von dem selbstdefinierten Typ, mit dem auch die Datenbank erstellt wurde.

Falls erforderlich, wird der Datensatz sofort in alle Indizes korrekt einsortiert. Wenn es dabei Konflikte mit einem Index gibt (siehe *universell*-Parameter bei CREATEINDEX), kann ein Fehler generiert werden.

Wenn der ISAM-Dateizeiger vor dem ersten oder hinter dem letzten Element steht, wird der Fehler *Kein aktueller Datensatz* erzeugt.

Siehe auch INSERT (468), DELETE (466), CREATEINDEX (465).

Referenzteil Toolboxen 27

27.1 Matrizenmathematik

Von fast allen Funktionen der Matrizenmathematik-Toolbox gibt es fünf Ausführungen, für jeden numerischen Datentyp eine. Alle Routinen sind als Funktionen ausgeführt, die als Funktionswert einen Fehlercode zurückgeben, dessen Bedeutung in den Abschnitten über die einzelnen Funktionen beschrieben ist.

FUNCTION MatAdd*x*

Anwendung
```
x% = MatAddI(matrix1%(), matrix2%())
x% = MatAddL(matrix1&(), matrix2&())
x% = MatAddC(matrix1@(), matrix2@())
x% = MatAddS(matrix1!(), matrix2!())
x% = MatAddD(matrix1#(), matrix2#())
```

Nutzen Addiert zwei Matrizen. *matrix1()* und *matrix2()* müssen Arrays mit gleichen Dimensionen sein; je nach Funktion werden INTEGER-, LONG-, CURRENCY-, SINGLE- oder DOUBLE-Arrays erwartet.

Die Ergebnismatrix wird in das Array *matrix1()* geschrieben und hat dieselben Dimensionen wie die beiden Ausgangsmatrizen. *matrix2()* wird im Verlauf der Operation zerstört.

Als Funktionswert gibt die Funktion einen Fehlercode zurück. 0 bedeutet, daß die Operation erfolgreich verlief, –4, daß *matrix1()* und *matrix2()* nicht dieselbe Dimension haben. Ein positiver Funktionswert bedeutet einen Standard-BASIC-Fehler.

FUNCTION MatSub*x*

Anwendung
```
x% = MatSubI(matrix1%(), matrix2%())
x% = MatSubL(matrix1&(), matrix2&())
x% = MatSubC(matrix1@(), matrix2@())
x% = MatSubS(matrix1!(), matrix2!())
x% = MatSubD(matrix1#(), matrix2#())
```

Nutzen Subtrahiert *matrix1* von *matrix2*. Das bei *MatAddx* Gesagte gilt entsprechend.

FUNCTION MatMult*x*

Anwendung
```
x% = MatMultI(matrix1%(), matrix2%(), matrix3%())
x% = MatMultL(matrix1&(), matrix2&(), matrix3&())
x% = MatMultC(matrix1@(), matrix2@(), matrix3@())
x% = MatMultS(matrix1!(), matrix2!(), matrix3!())
x% = MatMultD(matrix1#(), matrix2#(), matrix3#())
```

Nutzen Multipliziert zwei Matrizen miteinander. Alle drei Matrizen müssen denselben Datentyp haben: je nach verwendeter Funktion werden INTEGER-, LONG-, CURRENCY-, SINGLE- oder DOUBLE-Arrays erwartet. *matrix1( )* muß so viele Spalten haben wie *matrix2( )* Zeilen hat, und *matrix3( )* muß so viele Zeilen haben wie *matrix1( )* und so viele Spalten wie *matrix2( )*. Im mathematischen Klartext: *matrix1( )* ist eine Matrix mit *m·n* Elementen, *matrix2( )* hat *n·k* Elemente, und *matrix3()* enthält *m·k* Zahlen.

matrix1() und *matrix2()* werden im Verlauf der Operation zerstört. Das Ergebnis der Multiplikation wird in *matrix3()* geschrieben.

Als Funktionswert gibt die Funktion einen Fehlercode zurück. 0 bedeutet, daß die Operation erfolgreich verlief; –3 heißt, daß die Dimensionen der Matrizen *matrix1( )* und *matrix2( )* nicht korrekt aufeinander abgestimmt sind, und –5 wird zurückgegeben, wenn *matrix3( )* nicht die benötigte Anzahl von Zeilen und Spalten aufweist.

Jeder positive Funktionswert bedeutet einen gewöhnlichen BASIC-Fehler.

Siehe auch MatInvx (479).

FUNCTION MatDetx

Anwendung
```
x% = MatDetI(matrix%(), determinante%)
x% = MatDetL(matrix&(), determinante&)
x% = MatDetC(matrix@(), determinante@)
x% = MatDetS(matrix!(), determinante!)
x% = MatDetD(matrix#(), determinante#)
```

Nutzen Findet die Determinante einer quadratischen Matrix. *matrix()* muß genausoviele Zeilen wie Spalten enthalten und wird je nach verwendeter Funktion als INTEGER-, LONG-, CURRENCY-, SINGLE- oder DOUBLE-Array erwartet. Die *determinante* hat denselben Datentyp wie die Elemente der Matrix.

matrix() wird im Verlauf der Operation zerstört.

Als Funktionswert gibt die Funktion einen Fehlercode zurück. 0 bedeutet, daß die Operation erfolgreich verlief; –2 wird zurückgegeben, wenn die Matrix nicht quadratisch war.

Bemerkung • Theoretisch ist es möglich, daß die Determinante einer Matrix exakt 0 ist, und bei Ganzzahl- und CURRENCY-Berechnungen ist der Rechner da auch sehr genau. Arbeitet man aber mit SINGLE- oder DOUBLE-Werten, wird durch Rundungsfehler aus einer 0 schnell eine „sehr kleine Zahl“. Deshalb befinden sich in der Include-Datei MATH.BI zwei Konstantendefinitionen, `Seps!` und `Deps#`, die bestimmen, ab welcher Größe eine SINGLE-Zahl (`Seps!`) bzw. eine DOUBLE-Zahl (`Deps#`) als 0 angesehen wird. Durch Verkleinern dieser Konstanten erhöhen Sie die Genauigkeit der Funktionen, aber auch ihre Empfindlichkeit gegen Rundungsfehler. Wenn Sie die Werte ändern, müssen Sie alle Matrizenmathematik-Libraries neu erstellen. Die Konstanten haben auch Auswirkung auf die Funktionen MatInv*x* und MatSEqn*x*.

Siehe auch MatSub*x* (477).

FUNCTION MatInv*X*

Anwendung
```
x% = MatInvC(matrix@())
x% = MatInvS(matrix!())
x% = MatInvD(matrix#())
```

Nutzen Invertiert eine Matrix. Die Matrix wird, je nach verwendeter Funktion, als CURRENCY-, SINGLE- oder DOUBLE-Array erwartet und muß quadratisch sein.

Die Ergebnismatrix wird wieder in das Array *matrix()* geschrieben.

Wenn eine Matrix die Determinante 0 hat, kann sie nicht invertiert werden. Siehe dazu aber auch die Bemerkung zu MatDet*x*.

Als Funktionswert gibt die Funktion einen Fehlercode zurück. 0 bedeutet, daß die Operation erfolgreich verlief; –1 heißt, daß die Matrix nicht invertiert werden kann (Determinante ist 0), und –2 bedeutet, daß die Matrix nicht quadratisch ist.

Siehe auch MatMultx (478).

FUNCTION MatSEqn*x*

Anwendung
```
x% = MatSEqnC(matrix@(), vektor@())
x% = MatSEqnS(matrix!(), vektor!())
x% = MatSEqnD(matrix#(), vektor#())
```

Nutzen Löst ein lineares Gleichungssystem. *matrix( )* ist die quadratische Koeffizientenmatrix; *vektor( )* hat nur eine Spalte und enthält die Skalare, die rechts vom Gleichheitszeichen in den Gleichungen stehen.

Nach Ablauf der Funktion steht im ersten Element von *vektor()* die Lösung für die erste Variable, im zweiten Element die Lösung für die zweite Variable etc.

Die *matrix()* wird im Verlauf der Operation zerstört.

Als Funktionswert gibt die Funktion einen Fehlercode zurück. 0 bedeutet, daß die Operation erfolgreich verlief; –1 heißt, daß das Gleichungssystem nicht gelöst werden kann (Determinante ist 0), –2 bedeutet, daß die Matrix nicht quadratisch ist, und –3 tritt auf, wenn der *vektor()* nicht die richtige Anzahl von Zeilen hat.

Bemerkung Siehe Bemerkung zu MatDet*x*.

Beispiel Angenommen, Sie haben das folgende Gleichungssystem:

```
I. 7b   2a   8 + 13c = 12
II. a + b = 3   c
III. 8b + 15 = 5a
```

Bevor Sie es in MatSEqn*x* einspeisen können, müssen Sie es in eine simultane Form bringen, so daß die Reihenfolge der Koeffizienten auf der linken Seite identisch ist und auf der rechten Seite nur noch Konstanten stehen:

```
I*.  2a + 7b + 13c = 20
II*. 1a + 1b + 1c = 3
III*.  5a + 8b + 0c =  15
```

Jetzt können Sie MatSEqn aufrufen:

```
REM $INCLUDE: 'MATH.BI'
DIM Matrix(1 TO 3, 1 TO 3) AS SINGLE
DIM Ergebnis(1 TO 3) AS SINGLE

FOR a% = 1 TO 3 : FOR b% = 1 TO 3
READ Matrix(a%, b%): NEXT b%, a%
DATA  2, 7, 13, 1, 1, 1,  5, 8, 0
FOR a% = 1 TO 3 : READ Ergebnis(a%) : NEXT
```

```
DATA 20, 3,  15

x% = MatSEqn(Matrix(), Ergebnis())
IF x% <> 0 THEN PRINT "Fehler"; x% : END
PRINT " a = "; Ergebnis(1): PRINT " b = "; Ergebnis(2)
PRINT " c = "; Ergebnis(3)
```

Siehe auch MatDet*x* (478).

27.2 Font-Toolbox

SUB DefaultFont

Anwendung `DefaultFont` *segment%, offset%*

Nutzen Diese Routine ermittelt die Segment- und die Offsetadresse der internen Schriftart, die in das EXE-File eingebunden wurde (bzw. bei VBDOS sich in der Quick Library befindet). Diese Adressen werden benötigt, um die Schriftart mit RegisterMemFont zu registrieren.

Bemerkung • Zwar steht diese Eintragung im Font-Referenzteil, aber die Routine `DefaultFont` und die zugehörigen Daten der internen Schriftart sind in der Datei CHARTASM.OBJ enthalten, die ursprünglich nur in die Chart-Toolbox eingebaut wird. Wenn Sie mit der Chart-Toolbox arbeiten, können Sie diese Routine ohne weiteres benutzen. Wollen Sie allerdings die interne Schriftart benutzen, ohne die gesamte Chart-Toolbox laden zu müssen, so müssen Sie in Ihre Programme zusätzlich die Zeile

```
DECLARE SUB DefaultFont (SEG Segment%, SEG Offset%)
```

einbauen und außerdem dafür sorgen, daß die Datei CHARTASM.OBJ in die Font-Quick Library eingefügt wird. Darüberhinaus müssen Sie, wenn Sie ein EXE-File erstellen, beim Linken CHARTASM gesondert aufführen oder in die Font-Libraries einschließen.

Siehe auch RegisterMemFont (489).

SUB GetFontInfo

Anwendung `GetFontInfo` *beschreibung*

Nutzen GetFontInfo stellt Informationen über den gerade aktiven Font zur Verfügung. *beschreibung* ist eine Variable vom Typ `FontInfo`, der wie folgt vereinbart ist:

```
TYPE FontInfo
   FontNum AS INTEGER
   Ascent AS INTEGER
   Points AS INTEGER
   PixWidth AS INTEGER
```

```
    PixHeight AS INTEGER
    Leading AS INTEGER
    AvgWidth AS INTEGER
    MaxWidth AS INTEGER
    FileName AS STRING * cMaxFileName
    FaceName AS STRING * cMaxFaceName
END TYPE
```

Nach dem Aufruf von GetFontInfo sind die Felder von *beschreibung* mit den entsprechenden Daten gefüllt:

Variable	*Inhalt*
fontnum	die Nummer der Schriftart unter allen geladenen, also mindestens 1 und höchstens die Anzahl der geladenen Schriftarten.
ascent	ist die Höhe eines Zeichens von der Grundlinie aus bis zur Oberkante (*leading* ist dabei allerdings mitgerechnet – siehe unten).
points	die Größe der Schriftart in Punkt. Hierbei handelt es sich nicht um eine exakte Größe, sondern um einen angenäherten Wert, der die Vergleichbarkeit der Schriftgrößen in verschiedenen Screen-Modi erleichtern soll.
pixwidth, pixheight, maxwidth, avgwidth	Breite und Höhe eines Buchstabens bzw. Zeichens; *pixwidth* ist 0, wenn es sich um eine Proportionalschrift handelt. Dann steht in *avgwidth* die durchschnittliche und in *maxwidth* die maximale Größe eines Buchstabens; andernfalls, wenn eine nichtproportionale Schriftart geladen ist, enthalten *pixwidth*, *avgwidth* und *maxwidth* alle denselben Wert.
leading	Die Anzahl der Pixel, die am oberen Rand des Zeichens nicht belegt sind. Da – bis auf die ganz kleinen – alle Schriftarten hier mindestens 1 haben, können Sie mehrere Zeilen Text direkt untereinander schreiben, ohne daß die Buchstaben ineinanderlaufen.
filename	der Name der FON-Datei, aus der der Font gelesen wurde.
facename	der Name des Schrifttyps (bei den mitgelieferten Dateien „Helv“, „Tms Rmn“ oder „Courier“ bzw. „IBM“).

Bemerkung • *ascent–leading* ist die Höhe eines Großbuchstabens in Pixel.

Siehe auch GetRFontInfo (484), LoadFont (486).

FUNCTION GetGTextLen

Anwendung `x% = GetGTextLen(text$)`

Nutzen Diese Funktion gibt die Länge in Pixeln zurück, die der Text *text$* hätte, würde er mit der Funktion OutGText in der aktuellen Schrift-

art ausgegeben. Wenn die Länge des Textes nicht korrekt ermittelt werden kann (zum Beispiel, weil kein Font geladen ist), ist GetGTextLen –1.

Bemerkung • Die Höhe des Textes können Sie mit *PixHeight* aus GetFontInfo ermitteln.

Siehe auch OutGText (489).

SUB GetMaxFonts

Anwendung `GetMaxFonts register%, laden%`

Nutzen GetMaxFonts stellt fest, wieviele Fonts maximal registriert (*register%*) und geladen (*laden%*) werden können.

Bemerkung • Benutzen Sie GetTotalFonts, um festzustellen, wieviele Fonts geladen bzw. registriert sind.

• Wenn Sie SetMaxFonts nicht benutzen, sind maximal 10 Fonts registrier- und ladbar.

Siehe auch GetTotalFonts (485), SetMaxFonts (492), RegisterFonts (489), LoadFont (486).

SUB GetRFontInfo

Anwendung `GetRFontInfo nummer%, beschreibung`

Nutzen GetRFontInfo holt Informationen über einen beliebigen registrierten Font ein. *nummer%* ist seine Nummer unter allen registrierten (der erste bekommt die Nummer 1, Maximum ist die mit SetMaxFonts gesetzte Grenze). *beschreibung* ist eine Variable vom Datentyp `FontInfo`, die die Informationen aufnimmt.

Die Aufschlüsselung von *beschreibung* ist identisch mit der Liste bei GetFontInfo, bis auf das Element *fontnum*, das bei der Verwendung von GetRFontInfo denselben Wert wie *nummer%* enthält. Wenn Sie mit GetRFontInfo Informationen über eine Schriftart abgefragt haben, können Sie diese mit dem N-Befehl der Funktion LoadFont unter derselben Nummer, die Sie auch bei GetRFontInfo angegeben haben, laden.

Siehe auch GetFontInfo (482), LoadFont (486).

SUB GTextWindow

Anwendung `GTextWindow x1!, y1!, x2!, y2!, scrn%`

Nutzen GTextWindow ist das Äquivalent zum Befehl WINDOW. Mit WINDOW es möglich, ein eigenes Koordinatensystem (logische Koordinaten) anstatt der gewöhnlichen Pixel-Koordinaten zu verwenden. Damit sich auch die OutGText-Routine an die mit WINDOW vereinbarten logischen Koordinaten hält, muß diese Routine – am besten unmittelbar nach WINDOW-Aufruf – mit den logischen Bildschirmkoordinaten als Argumenten (also fast genauso wie der WINDOW-Befehl selbst) aufgerufen werden. *x1!*, *y1!* sind die logischen Koordinaten für die obere linke Ecke, *x2!*, *y2!* die für die untere rechte Ecke des Bildschirms. Wenn Sie WINDOW „SCREEN“ benutzt haben, müssen Sie zusätzlich bei GTextWindow den Parameter *scrn%* auf TRUE setzen.

Wenn Sie die Definition wieder löschen (also wieder auf Pixel-Koordinaten umschalten) wollen, rufen Sie GTextWindow mit *x1! = x2!* auf.

Bemerkung • Ein System mit logischen Koordinaten bezieht sich in der Font-Toolbox ausschließlich auf die Koordinatenangaben der Funktion OutGText. Alle anderen Daten wie Textlänge und -höhe werden weiterhin in Pixeln und nicht in logischen Einheiten ausgedrückt.

Siehe auch WINDOW (Disketten-Referenzteil), OutGText (489).

SUB GetTotalFonts

Anwendung `GetTotalFonts registered%, loaded%`

Nutzen GetTotalFonts ermittelt, wieviele Schriftarten zum Zeitpunkt des Aufrufs geladen (*loaded%*) und registriert (*registered%*) sind.

Bemerkung • Die Anzahl der registrierten Fonts wird durch das Registrieren von Fonts mit RegisterFonts erhöht und durch das Entfernen der Registrierungen mit UnRegisterFonts oder SetMaxFonts auf 0 gesetzt.

• Die Anzahl der geladenen Fonts ist identisch mit der Zahl, die die Funktion LoadFont als Funktionswert zurückgibt, da LoadFont jedesmal sämtliche bisher geladenen Fonts löscht.

Siehe auch GetMaxFonts (484), SetMaxFonts (492).

FUNCTION LoadFont

Anwendung `x% = LoadFont(`*`fontbezeichnung$`*`)`

Nutzen Mit dieser Funktion werden Fonts, die bereits registriert sind, in den Speicher geladen. Welche der registrierten Fonts geladen werden sollen, bestimmt *fontbezeichnung$*. LoadFont löscht alle bisher geladenen Fonts, so daß man mehrere Fonts in *fontbezeichnung$* angeben muß, wenn man über mehrere gleichzeitig verfügen will. Als Funktionswert gibt LoadFont die Anzahl der wirklich geladenen Fonts zurück, an der man erkennen kann, ob die Operation wie gewünscht verlief.

fontbezeichnung$ kann aus beliebig vielen Beschreibungen bestehen, die alle durch Schrägstriche (/) getrennt sein müssen.

Eine einzelne Fontbeschreibung setzt sich aus einem oder mehreren Schriftart-Auswahlkriterien – hier kurz Befehle genannt – zusammen. Diese sind:

Befehl	*Funktion*
N *nummer*	Lädt von allen registrierten Fonts den Font Nr. *nummer*. Keine weiteren Befehle werden benötigt.
R	Lädt nur den internen Font (kein anderer Befehl ist notwendig). Der interne Font muß natürlich auch registriert sein (siehe RegisterMemFont).
S *punkt*	Wählt nur eine Schriftart mit der Größe *punkt* Punkt. Wenn Sie keinen einschränkenden Befehl hinzusetzen, wird der nächstbeste Font mit der entsprechenden Größe genommen. Wenn Sie den S- und den H-Befehl gleichzeitig benutzen, wird H ignoriert. S ignoriert alle Fonts, die für den aktuellen oder mit dem M-Befehl angegebenen Screen-Modus nicht das richtige Seitenverhältnis haben. Dieses Seitenverhältnis läßt sich am letzten Buchstaben des FON-Files ablesen.
H *höhe*	Wählt nur eine Schriftart mit der Schrifthöhe *höhe* Pixel. Wenn Sie keinen einschränkenden Befehl hinzusetzen, wird der nächstbeste Font mit der entsprechenden Größe genommen.
B	(nur gleichzeitig mit S oder H) Wenn die bei S bzw. H gestellte Größenbedingung nicht erfüllt werden kann, wird normalerweise kein Font geladen. Geben Sie aber B dazu an, so wird ein Font geladen, der möglichst nahe an der angegebenen Größe liegt.

Befehl	*Funktion*
M *modus*	(nur gleichzeitig mit S) Der S-Befehl ignoriert normalerweise alle Fonts, die nicht das zum aktuellen Bildschirm passende Seitenverhältnis haben. Gibt man allerdings den M-Befehl gefolgt von einer BASIC-SCREEN-Nummer an, so ignoriert S alle Fonts, die nicht das zum angegebenen Bildschirm passende Seitenverhältnis haben.
F	Lädt nur einen nichtproportionalen Font.
P	Lädt nur einen proportionalen Font.
T*name*	(*kein* Leerzeichen zwischen T und *name*, auch wenn es im Handbuch anders steht!) Lädt nur einen Font vom Schrifttyp *name*. *name* entspricht dem Element *facename* aus dem Typ `FontInfo`, der bei GetFontInfo näher erklärt ist. Möglich sind bei den mitgelieferten Schriften „Courier“, „Tms Rmn“, „Helv“ und „IBM“ (für die interne Schriftart). Der T-Befehl ist sehr sensibel und funktioniert nur, wenn er der letzte in einer Font-Beschreibung ist (nach *name* müssen entweder ein Schrägstrich und die nächste Beschreibung kommen, oder *fontbezeichnung$* muß zu Ende sein) Außerdem müssen Sie für *name* die Groß- und Kleinschreibung exakt einhalten.

Wenn in der Liste der zu ladenden Fonts eine ungültige Bezeichnung enthalten ist, oder wenn ein geforderter Font nicht gefunden werden kann, wird stattdessen der allererste registrierte Font benutzt.

Die LoadFont-Routine scheut sich nicht, denselben Font mehrmals zu laden, wenn Sie die Beschreibungen so gestalten, daß derselbe Font auf mehrere zutrifft.

Da in jedem Falle so viele Fonts geladen werden, wie Sie Bezeichnungen angegeben haben (notfalls wird ja mit dem ersten gefüllt), können Sie, wenn Sie mit SelectFont einen der geladenen Fonts zum aktuellen machen wollen, sich auf die Positionen in *fontbezeichnung$* berufen. Nach `x% = LoadFont("n1/fbh12/pbh14")` würde also `SelectFont 2` bestimmt den Font aktiv machen, auf den die Beschreibung „fbh12“ zutrifft.

Bemerkung • Die Ausführungen zum T-Befehl gehen davon aus, daß Sie die im Abschnitt „Programmfehler in der Font-Toolbox“ in Kapitel 15 beschriebenen notwendigen Änderungen an der Font-Toolbox gemacht haben. Wenn nicht, können Sie den T-Befehl erstens nicht

mit der Schriftart „Tms Rmn“ benutzen, da diese ein Leerzeichen im Namen hat, und außerdem muß dann nach *name* der ganze *font-bezeichnung$* zu Ende sein.

• Vergessen Sie nicht, daß ein geladener Font sehr viel mehr Speicher belegt als ein nur registrierter. Außerdem dauert das Laden von Fonts natürlich länger als das Registrieren.

Beispiel (zur Anwendung der verschiedenen Spezifikationsbefehle)

```
REM $INCLUDE: 'FONTB.BI'

' Wir werden 15 Fonts registrieren, aber nur 5 gleichzeitig laden
SetMaxFonts 15, 5

SCREEN 11
a% = RegisterFonts("HELVE.FON"): a% = RegisterFonts("TMSRA.FON")
a% = RegisterFonts("COURB.FON")
' Registriert sind jetzt Times Roman 8, 10, 12, 14, 18 und 24 Punkt
' sowie Helvetica in denselben Größen; Times Roman jedoch in der Aus-
' führung für die SCREEN-Modi 2, 3, 4 und 8, Helvetica in der für die
' Modi 11 und 12. Beide sind proportional. Außerdem sind noch die
' nichtproportionalen Courier-Größen 8, 10 und 12 registriert, und zwar
' für SCREEN 1, 7, 9, 10 und 13.

a% = LoadFont("s12")
' Lädt die nächstbeste 12-Punkt-Schrift, die zum aktuellen SCREEN-Modus
' paßt, also Helvetica 12.

a% = LoadFont("m2 s12")
' Lädt die nächstbeste 12-Punkt-Schrift, die zum SCREEN 2 paßt, also
' Times Roman 12.

a% = LoadFont("f")
' Lädt die nächstbeste nichtproportionale Schrift, also Courier 8.

a% = LoadFont("b h16 p tHelv")
' Lädt die nächstbeste proportionale, 16 Pixel hohe Schrift bzw. eine,
' die in der Höhe möglichst nahe daran liegt, falls es 16 nicht
' gibt; es soll Helvetica sein.

a%= LoadFont("bs8 / bs10 / bs13 / bs17 / bs24")
' Lädt eine Palette von 5 Schriftgrößen, die zum aktuellen
' Bildschirmmodus passen; wenn die angegebenen Punktgrößen nicht
' verfügbar sind, sollen möglichst ähnliche Werte gewählt werden.
```

Siehe auch RegisterFonts (489), RegisterMemFont (489).

FUNCTION OutGText

Anwendung `z% = OutGText(x!, y!, text$)`

Nutzen Die Funktion gibt einen Text auf dem Bildschirm aus. Dabei werden der aktuelle Font und die mit SetGTextColor gesetzte Farbe benutzt. Der Text wird horizontal ausgegeben, falls nicht mit SetGTextDir eine andere Richtung festgelegt wurde. *x!* und *y!* sind – als SINGLE-Variablen – die Koordinaten der oberen linken Ecke des ersten Zeichens, das ausgegeben wird. *x!* und *y!* sind Pixelkoordinaten, wenn nicht mit GTextWindow logische Koordinaten vereinbart wurden.

Die Funktion gibt als Funktionswert die Textlänge in Pixeln zurück (genau wie GetGTextLen); außerdem verändert sie *x!* und *y!* dahingehend, daß diese jetzt die Position angeben, auf die das nächste Zeichen geschrieben worden wäre.

Siehe auch LoadFont (486), SelectFont (490), GetGTextLen (483), GTextWindow (485), SetGTextColor (491), SetGTextDir (491).

FUNCTION RegisterFonts

Anwendung `x% = RegisterFonts(dateiname$)`

Nutzen Registriert alle Schriftarten, die in der angegebenen Datei *dateiname$* enthalten sind. Die eventuell schon früher registrierten Schriftarten bleiben registriert. Wenn während des Registrierens der Speicher nicht mehr ausreicht oder das mit SetMaxFonts gesetzte Register-Limit überschritten wird, bricht der Registriervorgang ab. Die Funktion übergibt als Funktionswert die Anzahl der Fonts, die durch sie in diesem Aufruf registriert wurden. Die Gesamtzahl der registrierten Fonts erhalten Sie mit der Prozedur GetTotalFonts.

Bemerkung • Zum Registrieren der internen Schriftart müssen Sie die Funktion RegisterMemFont benutzen.

Siehe auch RegisterMemFont, LoadFont (486).

FUNCTION RegisterMemFont

Anwendung `x% = RegisterMemFont(segment%, offset%)`

Nutzen RegisterMemFont registriert die interne „IBM“-Schriftart. Der Funktionswert von RegisterMemFont ist entweder 0 (wenn etwas daneben ging, etwa die Adressen falsch waren) oder 1 (wenn es geklappt hat).

Der RegisterMemFont-Routine werden die Segment- und die Offset-Adresse der internen Schriftart übergeben, damit sie weiß, wo im Speicher der Font steht. Um diese beiden Adressen zu ermitteln, können Sie die Routine DefaultFont benutzen. Die interne „IBM“-Schriftart wird jedoch nur in das EXE-File eingebunden, wenn Sie die Chart-Library verwenden oder mit CHARTASM.OBJ linken.

Bemerkung • Eine andere Schriftart als die in CHARTASM.OBJ codierte kann nicht in das EXE-File eingebunden werden. Sie sind – ausgenommen, Sie ändern die Daten in CHARTASM.ASM von Hand und assemblieren die Datei neu – also für alle Schriften außer der „IBM“-Schriftart weiterhin auf die .FON-Dateien angewiesen.

Siehe auch DefaultFont (482), SetGCharSet (490).

SUB SelectFont

Anwendung `SelectFont` *`fontnummer%`*

Nutzen Macht einen der geladenen Fonts zum aktuellen. *fontnummer%* ist eine Zahl zwischen 1 und der Anzahl der geladenen Fonts.

Bemerkung • Nach einem LoadFont-Befehl wird automatisch ein `SelectFont 1` ausgeführt; dadurch ist bis zum nächsten SelectFont-Befehl der erste geladene Font aktiv.

Siehe auch LoadFont (486).

SUB SetGCharSet

Anwendung `SetGCharSet` *`zeichensatz%`*

Nutzen Microsoft WINDOWS benutzt einen Zeichensatz, der sich von dem üblichen IBM-Zeichensatz unterscheidet. Bis zum Zeichen Nr. 127 sind beide identisch; danach treten Unterschiede auf. Beim WINDOWS-Zeichensatz befinden sich die Umlaute an anderer Stelle, und es sind zusätzliche Zeichen enthalten (zum Beispiel das Copyright-Zeichen). Der IBM-Zeichensatz besitzt bei den meisten Schriften nicht die Blockgrafikzeichen, die ja bei Proportionalschriften ohnehin sinnlos sind.

Mit dem Befehl SetGCharSet können Sie für alle folgenden OutGText-Aufrufe den Zeichensatz bestimmen; die Konstante `cWindowsChars` steht für den Windows-Zeichensatz, `cIBMChars` bedeutet IBM-Zeichensatz. Standardmäßig ist der IBM-Zeichensatz eingestellt.

Bemerkung • Durch einen Programmierfehler sind die Zeichensätze bei der internen Schriftart, die in CHARTASM.OBJ enthalten ist, vertauscht. Dadurch wird der WINDOWS-Zeichensatz aktiviert, wenn Sie `SetGCharSet cIBMChars` aufrufen und umgekehrt, und der WINDOWS-Zeichensatz ist auch Standard. Das bedeutet, daß Umlaute in der Präsentationsgrafik-Toolbox (die die interne Schriftart benutzt, wenn Sie keine andere laden) unter Umständen nicht korrekt dargestellt werden. Laden Sie entweder eine andere Schriftart vor Benutzung der Grafikroutinen, oder stellen Sie mit `SetGCharSet cWindowsChars` auf den WINDOWS-Zeichensatz um, der bei der internen Schriftart ja der IBM-Zeichensatz ist.

Siehe auch OutGText (489), RegisterMemFont (489).

SUB SetGTextColor

Anwendung `SetGTextColor` *`farbe%`*

Nutzen Legt die Farbe fest, in der alle künftigen Textausgaben erfolgen sollen. *farbe%* ist ein Farbattribut, wie es auch mit dem PSET-Befehl oder anderen gewöhnlichen Grafikbefehlen verwendet wird.

Siehe auch OutGText (489).

SUB SetGTextDir

Anwendung `SetGTextDir` *`richtung%`*

Nutzen Legt fest, in welche Richtung die Zeichen ausgegeben werden. Gültig sind:

richtung%	*Ausgabe*
0	Von links nach rechts (0°)
1	Von unten nach oben (90°)
2	Von rechts nach links (180°)
3	Von oben nach unten (270°)

Siehe auch OutGText (489).

SUB SetMaxFonts

Anwendung `SetMaxFonts register%, laden%`

Nutzen Setzt die maximale Anzahl von ladbaren oder registrierbaren Fonts. Dabei werden sämtliche geladenen und registrierten Fonts gelöscht.

Siehe auch GetMaxFonts (484).

SUB UnRegisterFonts

Anwendung `UnRegisterFonts`

Nutzen Löscht alle Font-Registrierungen aus dem Speicher. Geladene Fonts bleiben geladen. Es ist also durchaus sinnvoll, wenn man alle gewünschten Fonts geladen hat, UnRegisterFonts aufzurufen.

Bemerkung • Bevor Sie neue Fonts mit RegisterFonts registrieren können, müssen Sie SetMaxFonts aufrufen und die maximale Anzahl der registrierbaren Fonts neu angeben, da diese bei UnRegisterFonts auf 1 reduziert wird.

Siehe auch SetMaxFonts (492), RegisterFonts (489).

27.3 Präsentationsgrafik

SUB AnalyzeChart
SUB AnalyzeChartMS
SUB AnalyzePie
SUB AnalyzeScatter
SUB AnalyzeScatterMS

Anwendung *wie die zugehörigen Chart-Routinen*

Nutzen Die Analyze-Routinen Chart funktionieren genauso wie die zugehörigen Chart-Routinen (gleicher Name ohne „Analyze“), mit dem Unterschied, daß sie keine Grafik auf den Bildschirm zeichnen. Sie berechnen nur aufgrund der gegebenen Werte verschiedene Teile der Variable *env* neu, zum Beispiel die obere und untere Grenze für die Zahlen an der Achse oder die Größe des Datenfensters.

Bemerkung
- Bevor Die Analyze-Routinen funktionieren können, muß ein Grafikmodus mit ChartScreen eingestellt werden.
- Die Analyze-Routinen sind für Standardgrafiken nicht erforderlich; verwenden Sie sie nur, wenn Sie Grafiken nach eigenem Geschmack manipulieren wollen.

SUB Chart

Anwendung `Chart env, name$(), wert!(), anzahl%`

Nutzen Zeichnet Balken- und Liniengrafiken mit einer Reihe.

env ist eine Variable vom Typ `ChartEnvironment`, die alle nötigen Informationen enthält und zuvor mit DefaultChart gesetzt werden sollte.

name$ ist ein eindimensionales Feld mit den Bezeichnungen der Datenpunkte (zum Beispiel Januar, Februar etc.), und *wert!()* enthält die Werte, die gezeichnet werden sollen. *anzahl%* ist die Anzahl der Werte in *wert!()*; die Felder *name$()* und *wert!()* sollten

eine untere Index-Grenze von 1 haben.

Bemerkung • Bevor Chart richtig funktioniert, muß ein Grafikmodus mit ChartScreen eingestellt werden.

• Die Chart-Routine erkennt fehlende Werte, wenn Sie ihnen die Konstante `cMissingValue` zuordnen. Ein Wert innerhalb des Feldes *wert!( )*, der `cMissingValue` enthält, wird nicht gezeichnet (auch nicht als 0).

• Beide Bemerkungen gelten für alle Chart-Routinen.

Siehe auch AnalyzeChart (493), ChartMS (494), ChartScreen (496), DefaultChart (496).

SUB ChartMS

Anwendung `ChartMS` *`env, name$(), wert!(), anzahl%, von%, bis%, reihe$()`*

Nutzen ChartMS hat die gleiche Aufgabe wie Chart, nämlich Balken- und Liniengrafiken zu produzieren. Allerdings kann ChartMS auch mehrere Datenreihen gleichzeitig in einer Grafik darstellen.

env ist wieder eine beschreibende Variable vom Typ `ChartEnvironment`; *name$* und *wert!* müssen hier zweidimensionale Felder sein, wobei in der ersten Dimension die Werte (von 1 bis *Anzahl%*) und in der zweiten Dimension die verschiedenen Reihen (von *von%* bis *bis%*) abgetragen werden. *von%* und *bis%* beschreiben also den Bereich der Reihen, der ausgegeben werden soll.

Das eindimensionale Feld *reihe$* enthält für jede der Datenreihen (wieder von *von%* bis *bis%*) eine Bezeichnung, die in der Legende aufgeführt werden soll.

Siehe auch AnalyzeChartMS (493), Chart (493), ChartScreen (496), DefaultChart (496).

SUB ChartPie

Anwendung `ChartPie` *`env, name$(), wert!(), ausrück%(), anzahl%`*

Nutzen ChartPie wird benutzt, um Kuchen-Diagramme zu erstellen. Bis auf das Array *ausrück%( )* sind alle Parameter bereits von Chart bekannt (*env* ist eine Grafikbeschreibung vom Typ `ChartEnvironment`, *name$()* ein eindimensionales Array mit den Bezeichnungen der einzelnen Werte und *wert!( )* ebenfalls ein eindimensionales Feld, das die Werte selbst enthält).

Die Elemente des Arrays *ausrück%()* – für jeden Wert im Kuchen gibt es hier eines – sind entweder `cYes` oder `cNo` und bestimmen, ob das betreffende Kuchenstück aus dem Kuchen herausgesetzt werden soll oder nicht. Setzt man *ausrück* für alle Werte auf `cYes`, dann ergibt sich ein Kuchen mit Lücken zwischen den einzelnen Stücken.

Siehe auch AnalyzePie (493), Chart (493), ChartScreen (496), DefaultChart (496).

SUB ChartScatter

Anwendung `ChartScatter` *`env, wertx!(), werty!(), anzahl%`*

Nutzen ChartScatter zeichnet eine Punktgrafik, ein Bild, in dem zwei zusammenhängende Werte gegeneinander abgetragen und als ein Punkt dargestellt werden. Solche Grafiken können bei statistischen Auswertungen eingesetzt werden, um mit dem bloßen Auge Zusammenhänge zwischen den beiden dargestellten Größen feststellen zu können.

Die Koordinaten der Punkte müssen in den Feldern *wertx!()* und *werty!()* enthalten sein, wobei es natürlich nicht auf den Bereich ankommt (es sind also nicht etwa Bildschirmkoordinaten, die angegeben werden, sondern Koordinaten eines beliebigen Systems, dessen Grenzen die Toolbox automatisch aufgrund der gegebenen Werte ermittelt). In *anzahl%* ist die Anzahl der insgesamt vorhandenen Koordinatenpaare einzutragen.

Siehe auch AnalyzeScatter (493), ChartScreen (496), DefaultChart (496).

SUB ChartScatterMS

Anwendung `ChartScatterMS` *`env, wertx!(), werty!(), anzahl%, von%, bis%, reihe$()`*

Nutzen Analog zu ChartMS ist ChartScatterMS dafür zuständig, mehrere Punktgrafiken gleichzeitig in einem Bild darzustellen. Zur Unterscheidung werden entsprechend der Farb- und Musterpalette (siehe Kapitel 15) verschiedene Symbole als Punkte benutzt. Die Parameter *env*, *wertx!()*, *werty!()* und *anzahl%* sind von ChartScatter her bekannt; *wertx!()* und *werty!()* müssen hier allerdings zweidimensionale Felder sein, in deren zweiter Dimension die Unterscheidung zwischen den verschiedenen Reihen stattfindet. Mit *von%* und

bis% geben Sie an, welche der Reihen ausgegeben werden sollen (*von%* und *bis%* beziehen sich auf die zweite Dimension von *wertx!* und *werty!*). *reihe$()* ist ein eindimensionales Feld, in dem für jede auszugebende Reihe ein Name enthalten sein muß, der in der Legende erscheint.

Bemerkung • Die Felder *wertx!* und *werty!* sollten in der ersten Dimension als untere Grenze 1 haben.

Siehe auch ChartScatter (495), AnalyzeScatterMS (493).

SUB ChartScreen

Anwendung `ChartScreen` *modus%*

Nutzen Setzt einen SCREEN-Modus für nachfolgende Grafikausgaben. Wenn Sie ChartScreen verwenden, wird die Benutzung des Befehls SCREEN überflüssig, es sei denn, Sie wollen einige der speziellen Optionen (aktive Seite, virtuelle Seite o. ä.) des SCREEN-Befehls benutzen. In diesem Fall müssen Sie aber auch zuvor einen ChartScreen-Befehl verwenden, da durch ChartScreen einige globale Variablen eingestellt werden, die die Analyze- und die Chart-Routinen unbedingt benötigen.

modus% ist eine gültige SCREEN-Modus-Bezeichnung (siehe bei SCREEN im Diskettenreferenzteil).

Bemerkung • Wenn Sie einen ungültigen *modus%* angeben, wird die globale Variable *ChartErr* auf `cBadScreen` gesetzt.

• ChartScreen setzt die interne Farb- und Musterpalette auf die Standardeinstellung für den angegebenen Grafikmodus.

Siehe auch SCREEN im Diskettenreferenzteil.

SUB DefaultChart

Anwendung `DefaultChart` *env*, *grafiktyp*, *variation*

Nutzen Setzt einige Elemente der Beschreibungsvariable *env* (vom Typ `ChartEnvironment`) entsprechend den angegebenen Parametern *grafiktyp* und *variation*. DefaultChart muß vor einer Chart- oder

Analyze-Routine aufgerufen werden. Die möglichen Kombinationen sind:

grafiktyp = `cBar`: horizontale Balken
  variation = `cPlain`: Balken nebeneinander
  variation = `cStacked`: Balken aufeinander

grafiktyp = `cColumn`: vertikale Balken
  variation = `cPlain`: Balken nebeneinander
  variation = `cStacked`: Balken aufeinander

grafiktyp = `cLine`: Liniengrafik
  variation = `cLines`: Punkte durch Linien verbinden
  variation = `cNoLines`: Nur Punkte zeichnen

grafiktyp = `cScatter`: Punktgrafik
  variation = `cLines`: Punkte durch Linien verbinden
  variation = `cNoLines`: Nur Punkte zeichnen

grafiktyp = `cPie`: Kuchengrafik
  variation = `cPercent`: Prozentzahlen an Kreissegmenten
  variation = `cNoPercent`: Kreissegmente unbeschriftet lassen

Bemerkung • Die Konstanten sind in CHART.BI definiert.

SUB GetPaletteDef

Anwendung `GetPaletteDef` *farbe%(), linie%(), muster$(), punkt%(), rand%()*

Nutzen Liest den Inhalt der Farb- und Musterpalette der Präsentationsgrafik-Toolbox aus. Alle angegebenen Arrays müssen den Definitionsbereich 0 bis `cPalLen` haben. In *farbe%( )* werden die Farbnummern, in *linie%( )* die Linientypen, in *muster$( )* die Füllmuster, in *punkt%( )* die ASCII-Codes der Zeichen, die zur Punktdarstellung bei Linien- und Punktgrafiken benutzt werden, und in *rand%( )* die Linientypen für Ränder und Gridlinien zurückgegeben.

Die Linientypen sind als INTEGER-Zahlen angegeben, so, wie man auch beim LINE-Befehl einen Linientyp angibt. Die Füllmuster sind Strings, wie sie für den PAINT-Befehl verwendet werden.

Bemerkung • Sie werden diese Funktion nur dann benötigen, wenn Sie einzelne, kleine Veränderungen an der Palette vornehmen möchten, ohne alles neu einstellen zu müssen. Dann nämlich können Sie den alten Inhalt der Palette mit GetPaletteDef auslesen, Ihre Änderungen machen und dann die fünf Arrays mittels SetPaletteDef wieder in die Palette eintragen.

Siehe auch SetPaletteDef (500), ResetPaletteDef (500).

FUNCTION GetPattern$

Anwendung `x$ = GetPattern$(bitspropixel%, musternummer%)`

Nutzen Eine korrekte Füllmusterdefinition, wie sie zum Beispiel beim PAINT-Befehl benutzt wird und in der Farb- und Musterpalette der Grafik-Toolbox steht, definiert nicht nur das eigentliche Füllmuster, sondern auch die beim Füllen zu verwendende Vorder- und Hintergrundfarbe.

Die Funktion GetPattern$ gibt eine Musterdefinition zurück, die nur ein Muster enthält und der zuerst noch mit Hilfe der Funktion MakeChartPattern$ die Farbdaten beigefügt werden müssen, ehe sie verwendbar wird. Damit GetPattern$ dieses „Mustergerüst“ liefern kann, müssen Sie erstens angeben, für welchen SCREEN-Modus Sie ein Muster haben wollen: Setzen Sie *bitspropixel%* auf 8 für SCREEN 13, auf 2 für SCREEN 1 und auf 1 für alle anderen Modi. Mit *musternummer%* können Sie eines aus 15 vordefinierten Mustern wählen (die Zahlen 1 bis 15 sind erlaubt).

Beispiel Das Programm MUSTER.BAS auf der Diskette benutzt die Funktionen GetPattern$ und MakeChartPattern$, um Füllmuster zu erzeugen, die dann als Argument zum PAINT-Befehl „mißbraucht“ werden. Es erzeugt die folgende Mustertabelle (für den SCREEN-Modus 1 mit *bitspropixel%* = 2 ist sie etwas anders):

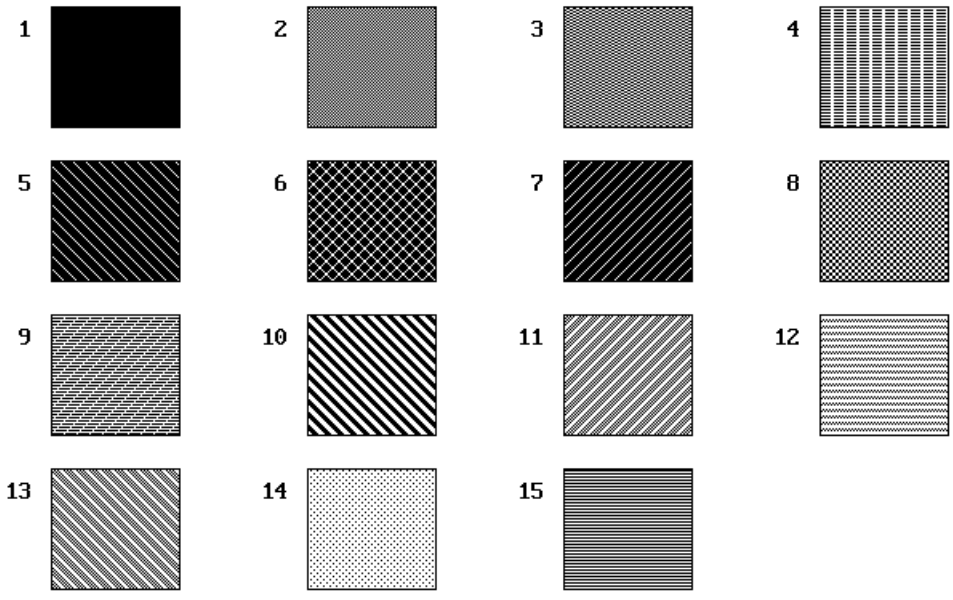


Abbildung 27–1: Die 16 Mustertypen bei GetPattern$

Siehe auch MakeChartPattern$ (499), SetPaletteDef (500).

SUB LabelChartH

Anwendung `LabelChartH env, x%, y%, font%, farbe%, text$`

Nutzen Gibt einen beliebigen Text horizontal entlang einer Grafik aus. *env* ist die übliche `ChartEnvironment`-Variable, die in diesem Falle schon alle Werte enthalten muß, die von den Grafik-Routinen dort eingetragen werden; *x%* und *y%* sind die Pixelkoordinaten des Textes, relativ zur oberen linken Ecke des Grafikfensters. *font%* ist die Nummer der geladenen Schriftart, die benutzt werden soll.

farbe% verweist auf den Eintrag in der Farb- und Musterpalette, mittels dessen dem Text eine Farbe zugeordnet werden soll. *text$* ist der Text, der ausgegeben wird.

Siehe auch LabelChartV, SetGCharSet (490).

SUB LabelChartV

Anwendung `LabelChartV env, x%, y%, font%, color%, text$`

Nutzen Gibt einen beliebigen Text vertikal entlang einer Grafik aus. Alles weitere gilt analog zu LabelChartH.

Siehe auch LabelChartH.

SUB MakeChartPattern$

Anwendung
```
x$ = MakeChartPattern$ (grundmuster$, vordergrund%,
                        hintergrund%)
```

Nutzen Erzeugt ein gültiges Grafikmuster, das mit dem PAINT-Befehl benutzt oder mit SetPaletteDef in die Farb- und Musterpalette der Präsentationsgrafik-Toolbox eingetragen werden kann. Das endgültige Muster wird als Funktionswert zurückgegeben; als Eingaben verwendet diese Funktion *grundmuster$*, ein Muster, das mit der Funktion GetPattern$ erzeugt wurde, sowie *vordergrund%* und *hintergrund%* als Vorder- und Hintergrundfarben. Diese Farben sollten natürlich für den Bildschirmmodus, in dem das Muster benutzt wird, gültige Farbattribute sein.

Siehe auch GetPattern$ (498), SetPaletteDef (500), GetPaletteDef (497).

SUB ResetPaletteDef

Anwendung `ResetPaletteDef`

Nutzen Re-initialisiert die Farb- und Musterpalette für den gerade mit ChartScreen eingestellten SCREEN-Modus. Alle Änderungen, die mit SetPaletteDef gemacht wurden, werden überschrieben.

Siehe auch SetPaletteDef (500), GetPaletteDef (497).

SUB SetPaletteDef

Anwendung `SetPaletteDef` *farbe%(), linie%(), muster$(), punkt%(), rand%()*

Nutzen Schreibt neue Daten in die Farb- und Musterpalette der Präsentationsgrafik-Toolbox. Alle angegebenen Arrays müssen den Definitionsbereich 0 bis `cPalLen` haben. In *farbe%( )* werden die Farbnummern, in *linie%()* die Linientypen, in *muster$()* die Füllmuster, in *punkt%( )* die ASCII-Codes der Zeichen, die zur Punktdarstellung bei Linien- und Punktgrafiken benutzt werden sollen, und in *rand%()* die Linientypen für Ränder und Gridlinien zu jedem Palleteneintrag angegeben.

Siehe auch GetPaletteDef (497), ResetPaletteDef (500).

27.4 Maus-Routinen

Alle Mausroutinen arbeiten mit Zeilen- und Spaltenangaben. Im Bildschirmmodus 0 sind dies Zeichenpositionen beginnend bei 1;1 (wie sie auch mit dem LOCATE-Befehl verwendet werden). In den Grafikmodi werden Koordinaten verwendet, die der jeweiligen Auflösung entsprechen; diese beginnen bei 0;0.

SUB MouseBorder

Anwendung `MouseBorder` *y1%, x1%, y2%, x2%*

Nutzen Schränkt den Maus-Bewegungsbereich ein. Nach Ausführung dieses Befehls kann der Mauszeiger nur noch in dem angegebenen Koordinatenbereich bewegt werden.

Siehe auch MouseInit (502).

SUB MouseDriver

Anwendung `MouseDriver` *ax%, bx%, cx%, dx%*

Nutzen Ruft den Maustreiber (Interrupt 33h) auf. Im Gegensatz zu einem einfachen Interrupt-Aufruf mit INTERRUPT prüft MouseDriver das Vorhandensein eines Maustreibers und führt zukünftige Interrupt-Aufrufe nur dann durch, wenn ein Maustreiber vorhanden ist. Eine globale Variable namens `MousePresent` wird auf TRUE gesetzt, wenn ein Maustreiber geladen ist.

Bemerkung
- Es gibt keine Möglichkeit, festzustellen, ob eine Maus angeschlossen ist; man kann nur prüfen, ob ein Maustreiber geladen ist.
- Verwenden Sie diese Funktion, um den Maustreiber direkt anzusprechen, wenn Ihnen die Funktionalität der anderen Mouse-Prozeduren nicht ausreicht.

Siehe auch INTERRUPT im Disketten-Referenzteil.

SUB MouseHide

Anwendung `MouseHide`

Nutzen Macht den Maus-Cursor unsichtbar. Sie müssen den Cursor jedesmal unsichtbar machen, bevor Sie etwas auf den Bildschirm schreiben.

Bemerkung • Der Maustreiber verwaltet einen Cursor-Zähler, der bei der Initialisierung auf –1 gesetzt wird. Die Prozedur MouseShow erhöht diesen Zähler um 1 (er kann aber nie größer 0 werden), während MouseHide ihn um 1 verkleinert. Wenn der Zähler kleiner als 0 ist, wird der Maus-Cursor nicht angezeigt. Diese etwas komplizierte Mimik hat den Vorteil, daß Sie verschiedene Routinen, die etwas auf den Bildschirm schreiben und zuvor den Maus-Cursor abschalten, ineinander schachteln können. Voraussetzung ist nur, daß jede Routine den Maus-Cursor wieder anschaltet, wenn sie ihn zuvor abgeschaltet hat.

Siehe auch MouseInit, MouseShow (503).

SUB MouseInit

Anwendung `MouseInit`

Nutzen Initialisiert den Maustreiber. Diese Routine muß vor jedem anderen Aufruf des Maustreibers aufgerufen werden.

Bemerkung • MouseInit tut nichts weiter, als die Funktion 0 des Maustreiber-Interrupts aufzurufen.

• Der Maus-Cursor-Zähler wird auf –1 gesetzt, so daß der Cursor erst nach einem MouseShow-Befehl sichtbar wird.

Siehe auch MouseHide (501), MouseShow (503).

SUB MousePoll

Anwendung `MousePoll` *y%, x%, links%, rechts%*

Nutzen Fragt die aktuelle Position und den Status der Knöpfe der Maus ab. *y%* und *x%* sind Zeile und Spalte, in der der Mauszeiger steht; *links%* und *rechts%* – entweder TRUE oder FALSE – geben an, ob die entsprechenden Mausknöpfe gedrückt sind.

Bemerkung • Den mittleren Mausknopf können Sie abfragen, indem Sie stattdessen den Befehl `MouseDriver 3,` *knopf%, x%, y%* verwenden. *knopf%* ist 1, wenn der linke, 2, wenn der rechte, und 4, wenn der mittlere Mausknopf gedrückt ist. Summen davon sind möglich (7 = alle Knöpfe gedrückt).

Siehe auch MouseDriver (501).

SUB MouseShow

Anwendung MouseShow

Nutzen Erhöht den Maus-Cursor-Flag um 1, so daß, wenn dieser 0 wird, der Maus-Cursor angezeigt wird.

Bemerkung • Siehe Bemerkung zu MouseHide.

Siehe auch MouseHide (501), MouseInit (502).

27.5 Finanzmathematik

In Ergänzung zu den Gepflogenheiten der anderen Referenzteile finden Sie hier zu jeder Funktion noch die Rubrik „Englisch“, in der der englische Name der Funktion aufgeführt ist, der oft hilft, die Kurzbezeichnung im Kopf zu behalten.

Wenn Sie eine Fachfrau oder ein Fachmann sind, werden Sie über die teilweise recht einfachen Beispiele lächeln; sie sind insbesondere für diejenigen gedacht, die mit Finanzmathematik, Betriebswirtschaft oder betrieblichem Rechnungswesen bisher nicht in Berührung kamen.

FUNCTION DDB

Anwendung `x# = DDB(awert#, rwert#, dauer%, periode%, fehler%)`

Englisch **D**ouble-**D**eclining **B**alance Method

Nutzen Errechnet die degressive Abschreibung eines Vermögenswertes für eine bestimmte Abschreibungsperiode.

awert# ist der Anschaffungswert, *rwert#* der Restwert, der nach der Abschreibungsdauer verbleibt. *dauer%* ist die Abschreibungsdauer in Perioden; *periode%* ist die Periode, für die die Abschreibung errechnet werden soll. Um welche Zeiteinheiten es sich bei den Perioden handelt, ist für die Berechnung nicht relevant.

fehler% wird von der Funktion auf 1 gesetzt, wenn die Berechnung nicht ausgeführt werden konnte; ansonsten ist diese Variable 0.

Bemerkung • Im Gegensatz zur linearen Abschreibung sinkt die degressive Abschreibung mit sinkendem Restwert. Die degressive Abschreibung stellt einen gewissen Prozentsatz des gegenwärtigen Wertes dar, während die lineare ein Prozentsatz des Anschaffungswertes ist.

• Offenbar wegen eines Fehlers in der Library berechnet die Funktion für größere Laufzeiten (8 Perioden und mehr) ungenaue Resultate; die Summe der durch DBB errechneten Abschreibungswerte für alle Perioden ist dann nicht mehr gleich *awert#–rwert#*.

Beispiel Ein Computer werde für DM 10.000,– eingekauft; nach 4 Jahren betrage sein Restwert DM 1.296,–. Um auszurechnen, wie hoch die degressive Abschreibung im dritten Jahr ist, schreibt man `PRINT DDB(10000, 1296, 4, 3, x%)` und erhält DM 1.204,–.

Siehe auch SLN (512), SYD (513).

FUNCTION FV

Anwendung `x# = FV(zins#, dauer%, rate#, anfang#, typ%, fehler%)`

Englisch **F**uture **V**alue

Nutzen Errechnet den künftigen Wert (Endwert) einer Reihe regelmäßiger, gleichbleibender Zahlungen (konstanter Zinssatz vorausgesetzt).

zins# ist der Zinssatz (als Faktor, also eine Zahl zwischen 0 und 1), *dauer%* die Anzahl der Perioden und damit die Anzahl der gezahlten Raten, *rate#* ist die Höhe einer Rate; *anfang#* ist der Gegenwartswert oder Pauschalbetrag (Barwert) für die Ratenzahlungen. *typ%* ist 0, wenn die Rate am Ende jeder Periode gezahlt wird, 1, wenn die Zahlung am Anfang jeder Periode stattfindet.

Sowohl für den Funktionswert als auch für *rate#* und *anfang#* gilt: Eingehende Beträge sind positive Zahlen, zahlbare Beträge sind negative Zahlen.

fehler% wird von der Funktion auf 1 gesetzt, wenn die Berechnung nicht ausgeführt werden konnte; ansonsten ist diese Variable 0.

Beispiel Ein Sparvertrag sehe einen Zinssatz von 6,5% p.a. vor. Die Laufzeit sei 10 Jahre, die Einzahlung am Anfang jedes Jahres DM 1.000,–. Bei einem Sparvertrag gibt es keinen Gegenwartswert. Die Höhe des Guthabens nach Ablauf der 10 Jahre ist `FV(0.065, 10, 1000, 0, 1, x%)` = DM 14.371,56. Oder:

Beim Kauf eines Computers im Wert von DM 10.000,– werde die Zahlung in 10 Jahresraten à 1.150,– DM, fällig am Ende des Jahres, ohne Anzahlung, vereinbart. Der Kreditzins, den man bei einer Bank hätte zahlen müssen, betrage 8% p.a. Aus `FV(0.08, 10, 1150, 10000, 1, x%)` = DM –3.596,94 ergibt sich, daß Sie bei der Sache DM 3.596,94 sparen, weil Sie, wenn Sie sich die benötigten DM 10.000,– bei der Bank geliehen hätten, höhere jährliche Beträge hätten zahlen müssen.

Siehe auch NPV (509), PV (511), Rate (512).

FUNCTION IPmt

Anwendung `x# = IPmt(zins#, periode%, dauer%, anfang#, ende#, typ%, fehler%)`

Englisch **I**nterest **P**ayment

Nutzen Errechnet die Zinszahlung für eine bestimmte Periode einer Serie regelmäßiger, gleich hoher Zahlungen mit gleichbleibendem Zinssatz und einem bestimmten Anfangs- und Endwert.

zins# ist der Zinssatz ($0 \le zins\# < 1$); *periode%* ist die Periode, für die Sie die Zinszahlung wissen möchten; *dauer%* die Anzahl der Perioden, über die sich die Zahlungen hinziehen, *anfang#* ist der Anfangs- (Bar-) und ende# der End- (zukünftige) Wert der Annuität. *typ%* ist 1, wenn die Zahlungen am Anfang jeder Periode getätigt werden, und 0, wenn am Ende einer Periode gezahlt wird.

fehler% wird von der Funktion auf 1 gesetzt, wenn die Berechnung aus irgendeinem Grunde fehlschlug.

Verwenden Sie positive Zahlen für Geld, das eingenommen, und negative für Geld, das ausgegeben wird.

Für die Abzahlung eines Kredites müßte beispielsweise *anfang#* auf die Höhe des Kredits und *ende#* auf 0 gesetzt werden, denn der Kredit soll ja nach der angegebenen Zeit getilgt sein. Für das Ansparen einer Summe würde umgekehrt *anfang#* auf 0 und *ende#* auf den gewünschten Wert gesetzt.

Beispiel Ein Kredit von DM 30.000,– wird mit einer Laufzeit von 5 Jahren, monatlicher Zahlung am letzten Tag des Monats und einem Zinssatz von ½% pro Monat vereinbart. Um zu errechnen, wieviel Zinsen im 24. Monat gezahlt werden, rufen Sie auf: `PRINT IPmt(.005, 24, 60, 30000, 0, 0, x%)` und erhalten als Ergebnis etwa –97,73, was einer Zinszahlung von DM 97,73 entspricht. Um zu errechnen, wieviel in diesem Monat insgesamt gezahlt werden muß, müssen Sie entweder noch die Funktion PPmt mit denselben Parametern aufrufen, die Ihnen die Tilgungszahlung liefert, und diese zur Zinszahlung addieren, oder Sie benutzen einfach die Funktion Pmt, die sofort das gewünschte Ergebnis ausgibt. Sie benötigt nicht die Angabe, für welche Periode Sie die Zahlung wissen möchten, da alle drei Funktionen von gleichbleibender Gesamtzahlung in jeder Periode ausgehen.

Siehe auch Pmt (510), PPmt (510).

FUNCTION IRR

Anwendung `x# = IRR(cashflow#(), anzahl%, schaetzung#, fehler%)`

Englisch **I**nternal **R**ate of **R**eturn

Nutzen Errechnet den internen Zinsfuß einer Reihe von Cash Flows (Einnahmen und Ausgaben). Im Feld *cashflow#()* sind diese Einnahmen und Ausgaben enthalten, in *anzahl%* ihre Anzahl. *cashflow#()* muß mindestens einen positiven Wert (eine Einnahme) und einen negativen Wert (eine Ausgabe) enthalten. Die Funktion geht davon aus, daß die in *cashflow#()* enthaltenen Cash Flows alle den gleichen zeitlichen Abstand voneinander haben. Außerdem wird für Einnahmen und Ausgaben derselbe Zinssatz angenommen (siehe dazu aber Funktion MIRR).

In *schaetzung#* müssen Sie das ungefähre Ergebnis der Berechnung angeben, von dem ausgehend IRR dann das genaue Ergebnis iteriert. Liegt *schaetzung#* zu weit neben dem richtigen Ergebnis, wird nach 20 Iterationen *fehler%* auf 1 gesetzt und die Berechnung abgebrochen. Außerdem wird *fehler%* auch auf 1 gesetzt, wenn Ihre Zahlungsreihe keinen internen Zinsfuß besitzt.

Bemerkung • Achten Sie darauf, daß das erste Element im *cashflow#*-Feld den Index 1 hat. Dimensionieren Sie also entweder mit `DIM cashflow#(1 TO ...)`, oder benutzen Sie OPTION BASE.

Beispiel Siehe vergleichbares Beispiel zu MIRR.

Siehe auch MIRR, NPV (509), Rate (512).

FUNCTION MIRR

Anwendung *x#* = MIRR(*cashflow#()*, *anzahl%*, *fzins#*, *izins#*, *fehler%*)

Englisch **M**odified **I**nternal **R**ate of **R**eturn

Nutzen Errechnet den „modifizierten“ internen Zinsfuß einer Reihe periodischer Cash Flows. MIRR funktioniert genauso wie IRR, mit dem Unterschied, daß sich bei MIRR verschiedene Zinssätze für die Finanzierung (*fzins#*) und für den Ertrag aus der Reinvestition von Einnahmen (*izins#*) festlegen lassen. IRR geht davon aus, daß beide Zinssätze gleich sind, daher muß bei IRR keiner von beiden angegeben werden. Weil MIRR nach einem anderen Prinzip funktioniert als IRR, entfällt hier die Angabe eines Schätzwertes.

Bemerkung • Achten Sie darauf, daß das erste Element im *cashflow#*-Feld den Index 1 hat. Dimensionieren Sie also entweder mit `DIM cashflow#(1 TO ...)`, oder benutzen Sie OPTION BASE.

Beispiel Sie eröffnen einen Radiosender. Dafür müssen Sie anfangs einen Kredit in Höhe von DM 500.000,– aufnehmen, für den Ihre halsabschneiderische Bank 14 Prozent Zinsen pro Jahr haben will. Im er-

sten Jahr machen Sie einen Verlust von DM 20.000,–, den Sie ebenfalls per Kredit finanzieren. In den folgenden acht Jahren aber floriert das Geschäft. Sie machen Gewinne von 90, 130, 160, 170, 140, 150, 145 und 130 TDM. Diese können, wenn sie nicht zur Tilgung verwendet werden, mit einem Ertrag von 11 Prozent reinvestiert bzw. angelegt werden. Mit diesem Programm...

```
REM $INCLUDE: 'finance.bi'
DIM CFlow#(1 TO 10)
DATA  500, 20,90,130,160, 170,140,150,145,130
FOR a% = 1 TO 10: READ x&: CFlow#(a%) = 1000 * x&: NEXT
PRINT MIRR(CFlow#%(), 10, .14, .11, x%)
```

... können Sie feststellen, daß der interne Zinsfuß für das ganze Unternehmen in diesen zehn Jahren 13,56% ist (die Funktion gibt 0,1356 aus). Falls Sie für Ihr Geld irgendwo mehr als diesen internen Zins des Projekts hätten bekommen können, war es eine schlechte Investition – sonst hat es sich gelohnt.

Siehe auch IRR (506), NPV (509), Rate (512).

FUNCTION NPer

Anwendung `x% = NPer(zins#, zahlung#, anfang#, ende#, typ%, fehler%)`

Englisch **N**umber of **Per**iods

Nutzen NPer errechnet die Anzahl der Zahlungen für eine Annuität auf der Grundlage periodischer, gleichhoher Zahlungen und eines konstanten Zinssatzes.

zins# ist der Zinssatz, *zahlung#* die Zahlung in jeder Periode (Ratenzahlung inkl. Zinsen). *anfang#* ist der Anfangswert (vor Beginn der Zahlungen), *ende#* der Endwert (nach Ende der Zahlungen). *typ%* ist 0, wenn am Ende der Periode gezahlt wird, und 1, wenn die Zahlung am Anfang einer Periode fällig ist. *fehler%* wird von der Funktion auf 1 gesetzt, wenn die Berechnung nicht korrekt durchgeführt werden konnte. Ausgaben werden als negative, Einnahmen als positive Werte dargestellt.

Beispiel Sie können pro Jahr DM 10.000,– auf die hohe Kante legen. Die Bank bietet Ihnen einen Sparplan mit 5% Zinsen p.a. Sie wollen wissen, wie lange es dauert, bis Sie die DM 250.000,– für ihren kleinen Hubschrauber zusammengespart haben. Die Zahlungen erfolgen am Ende jedes Jahres; Sie fangen bei 0 an. Mit dem Funktionsaufruf `PRINT NPer(.05, 10000, 0, 250000, 0, x%)` erhalten

Sie das Ergebnis 16,62 und wissen damit, daß Sie noch 17 Jahre auf Ihren Hubschrauber warten müssen. Als *zahlung#* muß hier eine negative Zahl angegeben werden, da Sie ja Geld ausgeben.

Siehe auch FV (505), IPmt (505), Pmt, PPmt (510), PV (511), Rate (512).

FUNCTION NPV

Anwendung `x# = NPV(zins#, cashflow#(), anzahl%, fehler%)`

Englisch **N**et **P**resent **V**alue

Nutzen Die Funktion errechnet den Gegenwartswert (Nettobarwert) einer Reihe periodischer Cash Flows, die in der Zukunft auftreten werden, auf der Basis eines festen Zinssatzes.

zins# ist dieser Zinssatz, *cashflow#( )* ist ein Array mit den Cash Flows, die den gleichen zeitlichen Abstand voneinander haben müssen. *anzahl%* ist die Anzahl der Cash Flows, und *fehler%* ist – wie üblich – eine Variable, durch die die Funktion mitteilen kann, wenn etwas bei der Berechnung danebenging.

Bemerkung • Achten Sie darauf, daß das erste Element im *cashflow#*-Feld den Index 1 hat. Dimensionieren Sie also entweder mit `DIM cashflow#(1 TO ...)`, oder benutzen Sie OPTION BASE.

• NPV geht davon aus, daß der erste Cash Flow am Ende der ersten Periode der gesamten Laufzeit auftritt. Der erste Cash Flow wird also schon in die Abzinsung einbezogen. Wenn Sie einen Cash Flow berücksichtigen wollen, der ganz am Anfang der Laufzeit steht, können sie diesen einfach zum Ergebnis addieren, da der Zinssatz auf ihn ja gar keinen Einfluß hat.

Beispiel Sie haben sich verpflichtet, an eine Firma im nächten Jahr DM 1.000,–, im darauffolgenden Jahr DM 2.500,– und im dritten Jahr DM 5.000,– zu zahlen. Sie wollen wissen, wieviel Geld Sie heute – also ein Jahr vor der ersten Zahlung – auf einem Sparkonto mit 3,5% Zinsen anlegen müssen, um die künftigen Zahlungen bestreiten zu können. Das folgende Programm...

```
DIM Zahlung(1 TO 3) AS DOUBLE
DATA  1000, 2500, 5000
FOR i% = 1 TO 3 : READ Zahlung(i%) : NEXT
PRINT NPV(0.035, Zahlung(), 3, x%)
```

...errechnet für Sie, daß es ausreicht, heute DM 7.809,68 auf das Sparkonto einzuzahlen, um die insgesamt 8.500,– DM über die Jahre hinweg zahlen zu können.

Siehe auch FV (505), IRR (506), PV (511).

FUNCTION Pmt

Anwendung `x# = Pmt(zins#, dauer%, anfang#, ende#, typ%, fehler%)`

Englisch **P**ay**m**en**t**

Nutzen Ermittelt die Annuitätszahlung (Summe aus Zahlung auf die Kapitalsumme und Zinszahlung einer Annuität pro Periode).

zins# ist der Zinssatz, *dauer%* ist die Laufzeit der Annuität, *anfang#* ist der Anfangs- und *ende#* der Endwert der Annuität. *typ%* ist 1, wenn die Zahlungen am Anfang jeder Periode getätigt werden, und 0, wenn am Ende einer Periode gezahlt wird.

fehler% wird von der Funktion auf 1 gesetzt, wenn die Berechnung aus irgendeinem Grunde fehlschlug. Verwenden Sie positive Zahlen für Geld, das eingenommen, und negative für Geld, das ausgegeben wird.

Bemerkung • Im Gegensatz zu den verwendeten Funktionen IPmt (Zinszahlung) und PPmt (Zahlung auf die Kapitalsumme) ist Pmt über die ganze Laufzeit der Annuität hinweg konstant. Die Summe von PPmt und IPmt für eine bestimmte Periode wird immer dasselbe Ergebnis wie Pmt haben.

Beispiel Siehe Beispiel zu IPmt.

Siehe auch IPmt (505), PPmt (510).

FUNCTION PPmt

Anwendung `x# = PPmt(zins#, periode%, dauer%, anfang#, ende#, typ%, fehler%)`

Englisch **P**rincipal **P**ay**m**en**t**

Nutzen Gibt die Zahlung auf die Kapitalsumme für eine bestimmte Periode einer Annuität zurück (auf der Basis konstanter periodischer Zahlungen und eines konstanten Zinssatzes).

zins# ist der Zinssatz ($0 \leq zins\# < 1$); *periode%* ist die Periode, für die die Zahlung ermittelt werden soll; *dauer%* die Anzahl der Perioden, über die sich die Zahlungen hinziehen, *anfang#* ist der An-

fangs- und *ende#* der Endwert der Annuität. *typ%* ist 1, wenn die Zahlungen am Anfang jeder Periode getätigt werden, und 0, wenn am Ende einer Periode gezahlt wird.

fehler% wird von der Funktion auf 1 gesetzt, wenn die Berechnung aus irgendeinem Grunde fehlschlug.

Beispiel Siehe Beispiel zu IPmt.

Siehe auch IPmt (505), Pmt (510).

FUNCTION PV

Anwendung `x# = PV(zins#, dauer%, zahlung#, ende#, typ%, fehler%)`

Englisch **P**resent **V**alue

Nutzen Die Funktion PV errechnet den Gegenwartswert (Barwert) einer Reihe periodischer, konstanter Zahlungen, die in der Zukunft gemacht werden, auf der Basis eines festen Abzinsungssatzes.

zins# ist dieser Abzinsungssatz pro Periode, *dauer%* ist die Anzahl der Perioden, über die sich die Zahlungen hinziehen, *zahlung#* die Höhe einer einzelnen Zahlung, *ende#* ist der Endwert (bei Kreditzahlungen wäre *zahlung#* negativ und *ende#* null; bei einem Sparvertrag wären die Zahlungen auch negativ, aber *ende#* positiv). *typ%* bestimmt, ob die Zahlungen am Ende (*typ*% = 0) oder am Anfang (*typ%* = 1) jeder Periode gemacht werden. *fehler%* wird auf 1 gesetzt, wenn die Berechnung nicht gelang.

Der Unterschied zur Funktion NPV ist, daß Sie bei NPV verschieden hohe Zahlungen angeben können, während PV von konstanten Zahlungen ausgeht, und daß NPV die Zahlungen grundsätzlich am Ende jeder Periode annimmt, während man bei PV wählen kann.

Beispiel Sie fragen sich, wieviel Geld Sie heute auf einem mit 3½% verzinsten Sparkonto anlegen müssen, um zwanzig Jahre lang eine „Rente“ von DM 8.000,– pro Jahr zur Verfügung zu haben. Nach Ablauf dieser zwanzig Jahre möchten Sie noch DM 20.000,– übrig haben. Der Funktionsaufruf `PRINT PV(0.035, 20, 8000, 20000, 0, x%)` fördert es zutage: DM 123.750,54 müssen auf das Konto eingezahlt werden. Die Funktion spuckt natürlich –123750.54 aus, da es sich um eine Zahlung handelt.

Siehe auch NPV (509), FV (505).

FUNCTION Rate

Anwendung `x# = Rate(dauer%, zahlung#, anfang#, ende#, typ%, schaetzung#, fehler%)`

Nutzen Errechnet den effektiven Zinssatz einer Annuität aus gegebenem Anfangs- und Endwert, gegebener Laufzeit und gegebener periodischer Zahlung.

dauer% ist die Anzahl der Perioden, über die sich die Zahlungen hinziehen (und somit die Anzahl der Zahlungen). *zahlung#* ist die Höhe einer einzelnen Zahlung; *anfang#* der Anfangs- und *ende#* der Endwert der Annuität. *typ%* legt fest, ob die Zahlungen am Ende (*typ%* = 0) oder am Anfang (*typ%* = 1) jeder Periode gemacht werden. *schaetzung#* ist das ungefähre (von Ihnen geschätzte) Ergebnis, auf der Basis dessen die Funktion das genaue Ergebnis iteriert. *fehler%* wird auf 1 gesetzt, wenn die Berechnung nicht gelang oder der Wert für *schaetzung#* so weit vom korrekten Ergebnis entfernt war, daß nach 20 Iterationen keine ausreichende Genauigkeit erzielt werden konnte.

Die Funktion gibt den Zinssatz als Faktor, also als Zahl zwischen 0 und 1 (einschl.), zurück. Um Prozent zu erhalten, müssen Sie mit 100 multiplizieren.

Beispiel Ihre Bank bietet Ihnen einen Sparplan an, nach dem Sie zwölf Jahre lang am Anfang jeden Jahres DM 2.000,– einzahlen. Nach zwölf Jahren garantiert Ihnen die Bank inklusive aller Bonuszahlungen eine Summe von DM 38.000,–. Sie möchten nun den effektiven Jahreszinssatz dieses Angebots wissen. Sie schreiben: `PRINT Rate (12, 2000, 0, 38000, 1, 0.1, x%)` – mit einer Schätzung von 10% liegt man meistens nah genug am Ergebnis – und erhalten etwa 6,9%.

Siehe auch FV (505), PV (511), NPV (509), Pmt (510).

FUNCTION SLN

Anwendung `x# = SLN(awert#, rwert#, dauer%, fehler%)`

Englisch **S**traight-**l**ine depreciatio**n**

Nutzen Errechnet die lineare Abschreibung eines Vermögenswertes für eine einzelne Periode der Abschreibungsdauer.

awert# sind die Anschaffungskosten, *rwert#* ist der Restwert, der

nach der Laufzeit von *dauer%* Perioden bleibt. *fehler%* wird von der Funktion auf 1 gesetzt, wenn die Berechnung nicht korrekt durchgeführt werden konnte.

Beispiel Ein Computer wird für DM 10.000,– eingekauft und hat nach fünf Jahren einen Restwert von DM 2.000,–. Die lineare Abschreibung pro Periode ist `SLN(10000, 2000, 5, x%)` = DM 1.600,-.

Siehe auch DDB (504), SYD (513).

FUNCTION SYD

Anwendung `x# = SYD(awert#, rwert#, dauer%, periode%, fehler%)`

Englisch **S**um-of-**y**ear's digits **d**epreciation

Nutzen Errechnet die arithmetisch-degressive (digitale) Abschreibung eines Vermögenswertes für eine bestimmte Periode seiner Abschreibungsdauer.

awert# ist der Anschaffungswert, *rwert#* der Restwert, der nach der Abschreibungsdauer verbleibt. *dauer%* ist die Abschreibungsdauer in Perioden; *periode%* ist die Periode, für die die Abschreibung errechnet werden soll. Um welche Zeiteinheiten es sich bei den Perioden handelt, ist für die Berechnung nicht relevant.

fehler% wird von der Funktion auf 1 gesetzt, wenn die Berechnung nicht ausgeführt werden konnte; ansonsten ist diese Variable 0.

Beispiel vgl. das Beispiel zu DDB; aufgrund des anderen Berechnungsverfahrens hat SYD natürlich ein anderes Ergebnis.

Siehe auch DDB (504), SLN (512).

Limits Anhang A

Variablenbereiche

Typ	*erlaubter Bereich*
INTEGER	–32.768 bis 32.768, nur ganze Zahlen.
LONG	–2.147.483.648 bis 2.147.483.647, nur ganze Zahlen.
CURRENCY	–922.337.203.685.477,4808 bis 922.337.203.685.477,4807 (etwa –9 E14 bis 9 E14), in der kleinsten Auflösung 0,001.
SINGLE	in VBDOS und mit der Emulator-Library: –3,402823 E38 bis –1,40129 E–45; 0; 1,40129 E–45 bis 3,402823 E38 mit der Alternate Math-Library: –3,402823 E38 bis –1,175494 E–38; 0; 1,175494 E–38 bis 3,402823 E38
DOUBLE	in VBDOS und mit der Emulator-Library: –1,797693134862315 E308 bis –4,94065 E–324; 0; 4,94065 E–324 bis 1,797693134862315 E308 mit der Alternate Math-Library: –1,79769313486232 E308 bis –2,2250738585072 E–308; 0; 2,2250738585072 bis 1,79769313486232 E308
STRING	Maximal 32.767 Zeichen
TYPE	Ein selbstdefinierter Typ darf nicht größer als 65.535 Bytes sein.
Eigenschaften	je nach Typ; eine Liste finden Sie in Kapitel 17.

Variablennamen

Variablennamen sind maximal 40 Zeichen lang und müssen mit einem Buchstaben beginnen. Sie dürfen nicht mit FN anfangen. Die Groß- und Kleinschreibung spielt keine Rolle. Erlaubte Zeichen sind Buchstaben (nur A-Z, keine Umlaute), Ziffern (0–9) und der Unterstrich (_). Das letzte Zeichen kann ein Typenbezeichner sein (% für INTEGER, & für LONG, @ für CURRENCY, ! für SINGLE und # für DOUBLE). Außerdem ist der Punkt in Variablennamen erlaubt, sofern es sich nicht um Elemente eines selbstdefinierten Typs handelt. Es wird jedoch empfohlen, den Punkt nur für den Zugriff auf Eigenschaften von Objekten und Elemente selbstdefinierte Typen zu verwenden.

Für Prozeduren und Funktionen gelten dieselben Einschränkungen. Typenbezeichner sind nur bei Funktionen erlaubt, nicht bei Prozeduren.

ISAM-Namen, also Namen für Datenbanken, Indizes und Elemente eines Datensatzes, dürfen nur 30 Zeichen lang sein und nicht den Dezimalpunkt enthalten; alles andere wie oben.

Variablenanzahl

Eine Prozedur oder Funktion darf nicht mehr als 255 lokale Stringvariablen haben. Ein Programm darf nicht mehr als 126 benannte COMMON-Blocks und maximal 240 TYPE-Definitionen enthalten.

Einer Prozedur können als Parameter nicht mehr als 60 Variablen übergeben werden.

Arrays

Die Gesamtgröße eines statischen Arrays darf 65.535 Bytes nicht überschreiten. Dynamische Arrays dürfen größer als 65.535 Bytes sein, wenn beim Kompilieren bzw. beim Aufruf von VBDOS der Switch /Ah angegeben wird. Größer als 131.071 Bytes dürfen auch sie nur dann werden, wenn die Größe eines einzelnen Elements eine Potenz von 2 ist.

Ein Array darf maximal 60 Dimensionen haben, in jeder Dimension maximal 32.767 Elemente. Der Array-Index muß im INTEGER-Bereich liegen.

„Impliziert deklarierte“ Arrays, für die es keinen DIM-Befehl gibt und die strukturell gesehen eine Unsitte sind, dürfen maximal 8 Dimensionen haben.

Dateien

Die Dateinummern müssen zwischen 1 und 255 (inkl.) liegen. Es können nicht mehr als 16 Dateien gleichzeitig geöffnet werden, es sei denn, man ändert diese Maximalzahl durch einen Interruptaufruf (vgl. Kpaitel 22). Die absolute Obergrenze ist um fünf kleiner als die in FILES=xxx angegebene Zahl in CONFIG.SYS (bei Arbeit mit Overlays ist sie um sechs kleiner).

Eine RANDOM-Datei kann maximal 2.147.483.647 Datensätze enthalten (eine BINARY-Datei ebensoviele Bytes*). Die Satzlänge einer RANDOM-Datei darf 32.767 nicht überschreiten. Die Gesamtgröße einer Datei ist allenfalls durch das Betriebssystem begrenzt.

ISAM

Eine Datenbank darf maximal 300 Indexlisten haben; wenn diese Grenze nicht durch den Switch /Ii beim Kompilieren bzw. bei PROISAM oder PRO-

---

* Sie kann länger sein, aber jenseits der genannten Grenze liegende Daten können nicht gelesen oder geschrieben werden.

ISAMD.EXE vergrößert wird, sind jedoch 28 Indizes das Maximum. Indiziert werden können nur Felder oder Gruppen von Feldern, deren Gesamtlänge 255 nicht überschreitet. Maximal 512 ISAM-Buffer werden benutzt; das Minimum sind 6 (für PROISAM.EXE) bzw. 9 (für PROISAMD.EXE). In den Typdefinitionen aller zu einem Zeitpunkt gleichzeitig geöffneten ISAM-Datenbanken dürfen zusammen nicht mehr als 60 nicht indizierbare Felder vorkommen. Nicht indizierbare Felder sind Strings mit einer Länge von mehr als 255 Zeichen, Arrays und selbstdefinierte Typen.

Es können maximal vier ISAM-Dateien gleichzeitig geöffnet sein; die maximale Anzahl der gleichzeitig geöffneten *Datenbanken* ist `1+3*(5-n)`, wobei *n* die Anzahl der verschiedenen ISAM-Dateien ist.

Programmgröße

Eine Prozedur darf, um in VBDOS lauffähig zu sein, nicht größer als 65.535 Bytes sein. Dokumente und Include-Dateien, die in VBDOS geladen werden, dürfen ebenfalls diese Grenze nicht überschreiten.

Eine Programmzeile darf nicht länger als 255 Zeichen sein; in BC können zwar längere logische Zeilen erzeugt werden, indem man am Ende einer Zeile das Unterstrich-Zeichen (_) angibt, das die Anknüpfung der Folgezeile bedeutet; trotzdem sollte man das nicht tun, da zusätzlicher Speicherplatz benötigt wird.

Die maximale Verschachtelungstiefe für $INCLUDE-Dateien ist 5.

LINK, Libraries und Overlays

Bei LINK dürfen maximal 32 Libraries angegeben werden. Eine Library kann nur das 65,536fache ihrer Seitenlänge lang sein (bei der Standard-Seitenlänge 16 also genau 1 MB). Die maximale Größe eines Overlays ist 65,636 Bytes. Die maximale Anzahl von Overlays in einem Programm ist 2,047. Overlays dürfen auf bis zu 64 Ebenen verschachtelt sein (Prozedur aus Overlay A ruft Prozedur auf Overlay B auf, diese dann eine Prozedur aus Overlay C usw.).

Formen und Steuerelemente

Eine Form kann maximal 254 Steuerelemente enthalten. Die Anzahl der Formen in einem Projekt ist unbeschränkt. Es dürfen maximal 16 Timer-Steuerelemente gleichzeitig in einem Programm *aktiv* sein (die Zahl der *vorhandenen* Timer ist nicht beschränkt).

Switches Anhang B

VBDOS

Aufruf:

`VBDOS [switches] [/RUN] programmname [/CMD command$]`

Switches:

/Ah	Huge Arrays
/B	VBDOS-Bildschirmanzeige in schwarz/weiß
/C:*b*	Standard-Kommunikations-Buffer auf *b* Bytes setzen
/CMD:*command$*	COMMAND$ setzen
/Ea	Arrays in EMS auslagern
/E:*x*,*y*	Nur *x*+*y* KB vom EMS benutzen
/Es	Expanded Memory für VBDOS und Routinen anderer Sprachen
/G	Beschleunigt den Bildschirmaufbau bei CGA
/H	Maximale Bildschirmauflösung benutzen
/L [*quicklibrary*]	Quick Library laden
/MBF	Funktionen *MKS$, MKD$, CVS* und *CVD* durch *MBF*-Funktionen ersetzen
/NOHI	Keine Highlight-Zeichen
/RUN	Programm sofort starten
/S:*x*	Nur *x* KB konventionellen Speicher nutzen
/X:*x*	Nur *x* KB XMS nutzen

BC

Aufruf:

```
BC [modulname [, [objektname] [, listname]]] [switches] [;]
```

Switches:

/A	Compiler erzeugt ein Listing des Assembler-Codes
/Ah	Huge Arrays
/C:*b*	Standard-Kommunikations-Buffer auf *b* Bytes setzen
/D	Produziert „debugging“-Code
/E	Muß angegeben werden, wenn das Modul die Befehle ON ERROR und RESUME mit Zeilennummer enthält
/Es	Teilt Expanded Memory zwischen BASIC und Routinen anderer Sprachen auf (nur für gemischtsprachliches Programmieren)
/FPa	Benutzt die „Alternate Math“-Libraries
/FPi	Benutzt die „Emulator“-Libraries
/G2	Produziert Code, der nur ab 80286 aufwärts funktioniert
/G3	Produziert Code, der nur ab 80386 aufwärts funktioniert
/Ib:*x*	Spezifiziert die Anzahl von ISAM-Puffern
/Ie:*x*	Setzt die Menge an EMS (in KB), die für nicht-ISAM-Anwendungen freigelassen werden soll
/Ii:*x*	Setzt die maximale Anzahl von ISAM-Indizes
/MBF	Ersetzt die Funktionen *MKS$*, *MKD$*, *CVS* und *CVD* durch ihre Pendants mit dem Anhängsel *MBF*
/O	Sorgt dafür, daß beim Linken ein Stand-Alone-EXE-Programm erzeugt wird
/R	Speichert Arrays nach Zeilen und nicht, wie üblich, nach Spalten
/S	Schreibt Strings direkt in das Object-File und nicht in die Symbol-Tabelle (vgl. Kapitel 23)
/T	Unterdrückt Warnungen
/V	Ermöglicht das Event Trapping
/W	Wie /V, prüft aber nur an jeder Zeilennummer bzw. an jedem Zeilenlabel
/X	Erlaubt ON ERROR und RESUME NEXT
/Zd	Erzeugt Object-File mit Zeilennummern
/Zi	Erzeugt Object-File mit kompletter Debug-Information für CodeView

LINK

Aufruf:

```
LINK objectname [+objectname...] [,[exename] [,[listname]
     [,[libname] [+libname...] [, definition]]]] [switches] [;]

LINK @steuerungsdatei
```

Steuerungsdatei:

Enthält in jeder Zeile den Inhalt eines der Felder. Das erste Feld kann mit + auf mehrere Zeilen ausgedehnt werden. OBJ-Dateien in Klammern bilden Overlays.

Switches (ausgewählte):

/?, /HE	Alle LINK-Switches als Tabelle zeigen
/BA	Batch-Modus
/CO	Codeview-Informationen einbinden
/DY:*x*	Anzahl der Aufrufe zwischen Overlay setzen
/E	EXE-File komprimieren
/F	Far Calls, wenn möglich, in Near Calls umwandeln
/INF	Detaillierte LINK-Informationen während des Ablaufs anzeigen
/LI	In die Listendatei die Adressen von Zeilennummern einschließen
/M	In die Listendatei eine sortierte Liste aller globalen Symbole einschließen
/NOD	Keine Standard-Libraries benutzen
/NOE	Kein Extended Dictionary beim Linken anwenden
/NOF	Far Calls nicht umwandeln (Gegenstück zu /F)
/NOL	Logo nicht anzeigen
/NON	Keine NUL-Zeichen vor _TEXT-Segment einfügen
/NOP	Code nicht packen (Gegenstück zu /PACKC)
/O:*nr*	Overlay-Interrupt setzen.
/PACKC	Programmcode packen
/PAU	Vor Schreiben des EXE-Files pausieren
/Q	Quick Library erstellen
/SE:*x*	Maximale Segmentanzahl setzen
/ST:*x*	Stackgröße setzen

LIB

Aufruf:

```
LIB libraryname[switches] [befehle] [, [listfile] [, neulib] [;]

LIB @steuerungsdatei
```

Befehle:

Jeder Befehl wird vom Namen eines OBJ-Files gefolgt; bei + kann auch der Name einer Library angegeben werden.

+	OBJ-File an Library anfügen
–	OBJ-File aus Library löschen
–+	OBJ-File in Library durch neues ersetzen
*	OBJ-File aus Library herauskopieren
–*	OBJ-File aus Library herauskopieren und entfernen

Switches:

/PA:*n*	Seitengröße setzen
/NOE	Ohne Extended Dictionary arbeiten
/NOLOGO	Logo nicht anzeigen

BUILDRTM

Aufruf:

```
BUILDRTM switches rtm exportliste

BUILDRTM switches /DEFAULT
```

Exportliste:

Enthält die drei Kategorien #OBJECTS (mit den Namen der OBJ-Files), #EXPORTS (mit den Namen der Routinen, die das RTM zur Verfügung stellen soll) und #LIBRARIES (optional, enthält die Namen von Libraries, die von den unter #OBJECTS genannten OBJ-Dateien benötigt werden).

Switches:

/FPa	RTM benutzt die „Alternate Math“-Libraries
/FPi	RTM Benutzt die „Emulator“-Libraries
/FPI87	RTM benutzt die „Emulator“-Libraries; läuft ausschließlich auf Rechnern mit Coprozessor
/DEFAULT	Standard-RTM zu den angegebenen Switches erstellen (kein RTM-Name und keine Exportliste werden benötigt)

CodeView

Aufruf:

```
CV [switches] [exename] [befzeile]
CV @steuerungsdatei
```

Switches:

/2	Anzeige des Programms auf einem, des CodeView-Schirms auf anderem Monitor
/8	auf PS/2-Systemen 8514a- und VGA-Bildschirme gleichzeitig verwenden
/*n*	wobei *n*=25, 43, 50: Anzeige im *n*-Zeilen-Modus
/C*text*	Die CodeView-Befehle in *text* sofort ausführen (durch Semikolon trennen; wenn einer der angegebenen Befehle Leerzeichen enthält, muß *text* in Anführungszeichen eingeschlossen werden)
/F	„Screen Flipping“ aktivieren
/I*n*	Bei *n*=0 wird das Abfangen von 8259-Interrupts und NMIs aktiviert, bei 1 deaktiviert
/M	Sperrt die Mausbenutzung in CodeView (falls Konflikte mit Ihrem Programm auftreten)
/N*n*	wie /I, wirkt aber nur auf NMIs und nicht auf 8259-Interrupts
/S	„Swap Screens“ aktivieren

Fehlermeldungen Anhang C

BASIC-Fehlermeldungen (nach Nummern)

Die Tabelle enthält die Fehlerbezeichnungen, wie sie von BASIC angezeigt und von ERROR$ zurückgegeben werden. Wenn Sie mit SMALLERR.OBJ linken, erhalten Sie statt der hier angegebenen Meldungen nur „Fehler xx aufgetreten“. Die exakte Bedeutung der Fehler können Sie im nächsten Abschnitt nachschlagen.

Alle hier nicht aufgeführten Fehlernummern resultieren in der Fehlermeldung „Fehler kann nicht ausgegeben werden“.

Nr.	*Fehler*
1	NEXT ohne FOR
2	Syntaxfehler
3	RETURN ohne GOSUB
4	Keine Daten mehr in DATA-Zeile vorhanden
5	Funktionsaufruf unzulässig
6	Überlauf
7	Nicht genügend Speicher
8	Marke nicht definiert
9	Index außerhalb des definierten Bereichs
10	Doppelt vorhandene Definition
11	Division durch Null
12	Unzulässig im Direktmodus
13	Datentypen unverträglich
14	Speicher für Zeichenfolge nicht ausreichend
16	Zeichenfolgenausdruck zu komplex
18	Funktion nicht definiert
19	RESUME fehlt
20	RESUME ohne Fehler
24	Zeitüberschreitung am Gerät aufgetreten
25	Gerätefehler
27	Papier zu Ende
29	WHILE ohne WEND
30	WEND ohne WHILE
33	Doppelt vorhandene Marke

Nr.	*Fehler*
35	Unterprogramm nicht definiert
37	Falsche Anzahl an Argumenten
38	Datenfeld nicht definiert
40	Variable erforderlich
50	FIELD-Überlauf
51	Interner Fehler
52	Dateiname oder -nummer falsch
53	Datei nicht gefunden
54	Dateizugriffsmodus falsch
55	Datei bereits geöffnet
56	FIELD-Anweisung aktiv
57	Geräte-E/A-Fehler
58	Datei existiert bereits
59	Falsche Datensatzlänge
61	Diskette/Festplatte voll
62	Eingabe über das Ende der Datei hinaus
63	Datensatznummer falsch
64	Dateiname falsch
67	Zu viele Dateien
68	Gerät nicht verfügbar
69	Überlauf des Kommunikationspuffers
70	Zugriff verweigert
71	Diskette nicht bereit
72	Datenträgerfehler
73	Leistungsmerkmal nicht verfügbar
74	Zwischen Laufwerken umbenennen
75	Pfad/Datei-Zugriffsfehler
76	Pfad nicht gefunden
80	Leistungsmerkmal entfernt
81	Name ungültig
82	Tabelle nicht gefunden
83	Index nicht gefunden
84	Spalte ungültig
85	Kein aktueller Datensatz
86	Doppelter Wert für eindeutigen Index

Nr.	*Fehler*
87	Operation auf Index NULL ungültig
88	Datenbank inkonsistent
89	Nicht genügend ISAM-Puffer
260	Zeitmesser nicht verfügbar
271	Bildschirmmodus ungültig
272	Ungültig, wenn Formen angezeigt werden
340	Element eines Steuerelementefeldes nicht vorhanden
341	Index für Objektdatenfeld unzulässig
342	Nicht genügend Speicher für Steuerelementefeld
343	Objekt ist kein Datenfeld
344	Objektdatenfeld erfordert Index
345	Grenze erreicht: Kein weiteres Steuerelement für diese Form möglich
360	Objekt bereits geladen
361	Objekt kann nicht geladen oder entfernt werden
362	Zur Entwurfszeit erzeugte Steuerelemente können nicht entfernt werden
364	Objekt wurde entfernt
365	Objekt kann in diesem Zusammenhang nicht entfernt werden
380	Eigenschaftswert ungültig
381	Index für Eigenschaftenfeld ungültig
382	Eigenschaft kann zur Laufzeit nicht festgelegt werden
383	Eigenschaft ist schreibgeschützt
384	Eigenschaftsänderung unmöglich bei Symbol- oder Vollbildgröße
385	Eigenschaftenfeld erfordert Index
386	Eigenschaft zur Laufzeit nicht verfügbar
387	Eigenschaft für Steuerelement kann nicht festgelegt werden
400	Form wird bereits gezeigt; Anzeige als gebundene Form unmöglich
401	Anzeige von ungebundenen Formen unmöglich, wenn gebundene Form angezeigt
402	Oberste gebundene Form muß zuerst geschlossen oder verborgen werden
403	MDI-Form kann nicht als gebundene Form angezeigt werden
410	Eigenschaft kann bei einer MDI-Form nicht geändert werden
420	Objektverweis ungültig
421	Methode ist auf dieses Objekt nicht anwendbar
422	Eigenschaft nicht gefunden
423	Eigenschaft oder Steuerelement nicht gefunden
424	Objekt erforderlich

Nr.	*Fehler*
425	Objektverwendung ungültig
430	Zur Zeit kein aktives Steuerelement vorhanden
431	Zur Zeit keine aktive Form vorhanden
480	AutoRedraw-Bild kann nicht erzeugt werden

BASIC-Fehlermeldungen (alphabetisch)

Das Kürzel VB steht bei VBDOS-Fehlermeldungen, EZ bei Fehlermeldungen, die während des Kompilierens („Entwurfszeit"), und LZ bei Fehlern, die in einem laufenden Programm („Laufzeit") auftreten können. Bei den LZ-Fehlern ist, sofern bekannt, die Fehlernummer angegeben.

Ich habe hier alle Fehler aufgeführt, von denen ich meine, daß sie einer Erläuterung bedürfen. So einfache Dinge wie „Semikolon erwartet" habe ich nicht berücksichtigt, es sei denn, es gibt einen nicht offensichtlichen Grund für das Auftreten dieser Fehler.

$FORM, COMMON und DECLARE müssen vor ausführbaren Anweisungen stehen (EZ)

Die drei Befehle müssen im Programm vor jedem ausführbaren Befehl erscheinen. Nicht ausführbare Befehle sind alle die, die nur Vereinbarungen darstellen: Alle Metabefehle, COMMON, CONST, DATA, DECLARE, DEF*xxx*, alle DIM-Befehle bis auf Vereinbarungen von dynamischen Arrays, EVENT ON/OFF, OPTION BASE, REM, SHARED, STATIC und TYPE.

$Metabefehl-Fehler (EZ)

Sie haben einen Fehler beim $INCLUDE-Befehl gemacht. Die Syntax ist $INCLUDE – Doppelpunkt – Apostroph – Dateiname – Apostroph – neue Zeile.

/C: Puffergröße zu groß (EZ)

Die maximale Größe für den Kommunikationspuffer ist 32.767. Der Compiler bricht nicht ab, sondern ignoriert den Switch.

ALIAS erfordert Zeichenfolgenkonstante (EZ)

In einer DECLARE-Anweisung haben Sie hinter ALIAS keinen Alias-Namen (in Anführungszeichen) angegeben.

Anweisungen/Marken zwischen SELECT CASE und CASE unzulässig (VB)

Sie dürfen auf den ersten Befehl nach einer CASE-Anweisung keinen Haltepunkt setzen. Setzen Sie ihn stattdessen weiter vorne, und benutzen Sie dann die Einzelschritt-Funktionen, um den Verlauf des Programms zu beobachten.

AS fehlt (EZ)

Sie haben beim OPEN-Befehl das AS vergessen oder falsch angebracht.

AS-Abschnitt erforderlich (EZ)
AS-Abschnitt bei erster Deklaration erforderlich (EZ)

Wenn eine Variable einmal mit einer AS-Formulierung deklariert wurde, muß jeder weitere DIM-, REDIM-, COMMON-, STATIC- oder SHARED-Befehl, der sich auf sie bezieht, dieselbe AS-Formulierung haben. Andersherum darf eine Variable, die zuerst ohne AS-Formulierung deklariert wurde, nicht später noch einmal mit AS in einem der genannten Befehle auftauchen.

Ausdruck zu komplex (EZ)

Ein Ausdruck ist zu umfangreich. Stringausdrücke sollten nicht zu viele temporäre Strings enthalten; generell können auch zu viele Klammern und Funktionsaufrufe innerhalb eines Ausdrucks diesen Fehler hervorrufen. Teilen Sie den komplizierten Ausdruck in mehrere Teilausdrücke auf.

Außerdem kann ein Überlauf auftreten, wenn Sie in einer ISAM-Datenbank, die schon 28 Indizes enthält, einen neuen Index erstellen (ohne /Ii verwendet zu haben), oder wenn Sie einen Index erstellen, bei dem die Summe der Längen der indizierten Felder 255 übersteigt.

Außerhalb von SUB, FUNCTION, oder DEF FN Anweisung ungültig (EZ)

Sie haben einen Befehl im Modulcode benutzt, der nur im Prozedurcode erlaubt ist. Dazu zählen beispielsweise EXIT SUB, STATIC und ON LOCAL ERROR.

AutoRedraw-Bild kann nicht erzeugt werden (LZ 480)

Bei Formen oder Bildfeldern mit AutoRedraw=TRUE muß VBDOS den Inhalt in einem eigenen Datenbereich speichern. Ist der Speicher knapp, kann dieser Fehler auftreten.

BASE fehlt (EZ)

Keine Variable darf OPTION heißen, denn der Compiler erwartet danach das Wort BASE oder EXPLICIT.

Bezeichner kann nicht mit %, &, !, #, $, oder @ enden (EZ)

Prozedurnamen dürfen keinen Typenbezeichner besitzen.

Bezeichner ungültig (EZ)

Sie haben einen ungültigen Namen für einen Typen, eine Variable, eine Konstante oder eine Prozedur benutzt. Erlaubt sind in solchen Namen nur die Buchstaben A-Z, Zahlen, der Dezimalpunkt (mit Einschränkungen) und der Unterstrich (_); sie müssen mit einem Buchstaben anfangen und dürfen maximal 40 Zeichen lang sein.

Bildschirmmodus ungültig (LZ 271)

Sie haben versucht, eine Form anzuzeigen, während der aktuelle Modus nicht 0 oder aber die 40-Spalten-Anzeige aktiv war.

Binäre Quelldatei (EZ)

Ein BASIC-Programm, das Sie kompilieren wollten, ist weder im ASCII- bzw. Text-Format noch im komprimierten VBDOS-Format gespeichert (das komprimierte Format von älteren QuickBASIC-Versionen ist nicht mehr kompatibel!). Diese Meldung kann auch als bloße Warnung auftreten, wenn Sie das gültige VBDOS-Format benutzen, aber die Compiler-Optionen /Zi oder /Zd angegeben haben, weil CodeView mit den komprimierten Source-Code-Zeilen nichts anfangen kann.

Block-IF ohne END IF (EZ)

Zu Ihrem IF fehlt ein END IF. Es kann vorkommen, daß diese Meldung ohne ersichtlichen Grund an einem Strukturbefehl auftritt; dann sollten Sie alle Strukturen (wie DO...LOOP, SELECT CASE...END SELECT, FOR...NEXT usw.) überprüfen, ob sie auch korrekt beendet werden. Siehe auch „Unerklärliche Strukturfehler“ weiter unten in diesem Anhang.

BYVAL nur mit numerischen Argumenten erlaubt (EZ)

BYVAL darf nur mit den Datentypen INTEGER, LONG, CURRENCY, SINGLE und DOUBLE benutzt werden.

CASE ohne SELECT (EZ)

Einem CASE kann kein vorangehendes SELECT CASE zugeordnet werden. Das mag daran liegen, daß es fehlt, oder daran, daß zwischen dem letzten SELECT CASE und dem betroffenen CASE eine unvollständig ausgeführte Strukturanweisung liegt (siehe auch „Unerklärliche Strukturfehler“ weiter unten in diesem Anhang).

COMMON in Quick Library zu klein (EZ in VBDOS)

In der Quick Library sind weniger Variablen mit COMMON deklariert als in der gerade aktiven Startdatei. Ändern Sie das Hauptprogramm (Sie können einen COMMON-Block mit Namen verwenden, um nicht mit der Quick Library ins Gehege zu kommen), oder erstellen Sie die Quick Library neu.

COMMON-Name unzulässig (EZ)

Sie haben einen ungültigen Namen benutzt, um einen COMMON-Block zu betiteln.

CONST/DIM SHARED folgt auf SUB/FUNCTION (EZ)

Bevor in einem Programm die erste Subroutine definiert wird, sollte die Vereinbarung der globalen Variablen und Konstanten abgeschlossen sein. Natürlich können trotzdem lokale Konstanten für Subroutinen definiert werden, dies muß jedoch innerhalb des SUB...END SUB-Blocks geschehen.

Datei bereits geladen (VB)

Es kann in VBDOS keine Datei geladen werden, die sich schon im Speicher befindet. Wahrscheinlich haben Sie übersehen, daß VBDOS die Datei, die Sie laden wollten, durch ein .MAK-File bereits automatisch geladen hat.

Datei bereits geöffnet (LZ 55)

Dieser Fehler tritt auf, wenn Sie mit dem OPEN-Befehl eine Datei öffnen, die bereits geöffnet ist, oder eine noch nicht geöffnete Datei unter einer Dateinummer öffnen, die bereits in Verwendung ist. Der Fehler tritt auch auf, wenn Sie mit KILL eine Datei zu löschen versuchen, die gerade geöffnet ist, nicht jedoch, wenn Sie eine RANDOM-, BINARY- oder ISAM-Datei erneut öffnen.

Datei existiert bereits (LZ 58)

Es wurde versucht, mit NAME einer Datei einen neuen Namen zuzuordnen, unter dem bereits eine andere Datei existiert.

Datei nicht gefunden (LZ 53, VB)

Ein RUN-, CHAIN-, KILL-, NAME- oder OPEN FOR INPUT-Befehl konnte die genannte Datei nicht finden, oder es wurde versucht, eine Datei in VBDOS zu laden, die nicht vorhanden ist (auch mit $INCLUDE).

Dateiname falsch (VB, LZ 64)

Sowohl in VBDOS als auch als Runtime-Fehler taucht diese Meldung auf, wenn Sie eine Datei öffnen, laden, speichern oder löschen wollen oder sonst ir-

gendetwas tun, wobei der Dateiname angegeben wird. Die Meldung bedeutet, daß der Name ungültig ist. Er darf zum Beispiel – außer bei KILL – nicht die Zeichen * und ? enthalten, keine zwei Punkte (es sei denn als Directory-Angabe) usw.

Dateiname oder -nummer falsch (LZ 52)

Ein OPEN FOR ISAM-Befehl konnte nicht richtig ausgeführt werden, weil der Dateiname ungültig war (siehe *Dateiname falsch*) oder die angegebene Datei zwar existierte, aber keine ISAM-Datei war; dieser Fehler tritt allerdings auch bei jedem Befehl auf, der sich auf eine Dateinummer bezieht, die noch nicht mit OPEN geöffnet (oder schon wieder geschlossen) wurde. Auch bei einem OPEN-Befehl kann der Fehler auftreten, wenn die angegebene Dateinummer außerhalb des gültigen Bereichs ist (siehe FREEFILE im Disketten-Referenzteil).

Dateizugriffsmodus falsch (VB, LZ 54)

In VBDOS erhalten Sie diese Meldung, wenn Sie versuchen, ein BASIC-File, das im komprimierten und nicht im Text-Format gespeichert ist, als Include-File zu laden oder mit „Textdatei einfügen“ Ihr Programm einzubinden. Die entsprechende Datei muß im Text-Format vorliegen (laden Sie sie normal, und speichern Sie sie mit „Speichern unter“ als Textdatei). Zur Laufzeit tritt diese Meldung bei verschiedenen Dateibefehlen auf, wenn man sie zweckentfremdet, also auf Dateien anwendet, auf die sie nicht angewendet werden dürfen. Das kann zum Beispiel mit BLOAD, GET, PUT, FIELD, PRINT#, INPUT#, INPUT$ und sämtlichen ISAM-Befehlen passieren, wenn Sie in eine zum Lesen geöffnete Datei schreiben oder aus einer zum Schreiben geöffneten Datei lesen wollen oder versuchen, mit gewöhnlichen Befehlen eine ISAM-Datei zu bearbeiten und umgekehrt.

Datenbank inkonsistent (LZ)

Sie haben mit einem OPEN FOR ISAM-Befehl eine Datei geöffnet, die zwar für eine ISAM-Datei gehalten wird, aber nicht korrekt gelesen werden kann. Das Programm ISAMREPR (siehe Kapitel 12) kann benutzt werden, um die Datenbank wieder in Ordnung zu bringen.

Datenfeld nicht dimensioniert (EZ)

Sie haben ein Array verwendet, zu dem kein DIM existiert. BC gibt in einem solchen Falle nur eine Warnung aus, und das Array ist von da an „implizit deklariert“, das heißt, der Bereich von 0 bis 10 (oder 1 bis 10, je nach OPTION BASE) ist erlaubt.

Datenfeld zu groß (EZ bei BC)

Versuchen Sie, das Array zu verkleinern (vielleicht, indem Sie bei einem selbstdefinierten Typ einige Bytes sparen). Benutzen Sie dynamische Arrays ($DYNAMIC) und notfalls den Compiler-Switch /Ah. Lesen Sie in Kapitel 8 unter „Huge Arrays“ nach.

Datensatz- oder Zeichenfolgenzuweisung erforderlich (EZ)

Sie haben mit LSET eine Variable von numerischem Datentyp angegeben.

Datensatznummer falsch (LZ 53)

In einem GET-, PUT- oder SEEK-Befehl haben Sie eine Satznummer angegeben, die kleiner als 1 oder größer als $2^{31} - 1$ ist.

Datensatzvariable erforderlich (EZ)

Sie haben einen INSERT-, RETRIEVE- oder UPDATE-Befehl benutzt, ohne die dazugehörige Zugriffsvariable anzugeben.

Datenträgerfehler (LZ 62 und VB)

Auf der Diskette oder Festplatte ist ein Hardware-Fehler beim Lesen oder Schreiben von Daten aufgetreten.

Datentypen unverträglich (EZ oder LZ 13)

Als LZ-Fehler tritt diese Meldung im Umgang mit ISAM auf, wenn Sie entweder versuchen, eine Datenbank mit einem ungeeigneten Typ zu öffnen (siehe OPEN im ISAM-Referenzteil), oder wenn ein UPDATE-, RETRIEVE- oder INSERT-Befehl mit einer Variable benutzt wird, die nicht den Typ hat, mit dem die Datenbank geöffnet wurde. Außerdem kann der Grund ein SEEK-Befehl sein, der mit Argumenten aufgerufen wurde, die nicht den Typen der indizierten Felder entsprechen.

Als EZ-Fehler tritt der Fehler dann auf, wenn Sie eine Variable ungeeigneten Typs einsetzen, zum Beispiel, wenn Sie einer Stringvariablen eine numerische Variable zuweisen oder umgekehrt, oder wenn Sie eine eingebaute Funktion aufrufen und ihr statt eines numerischen Arguments ein Stringargument oder umgekehrt übergeben. Numerische Variablen sind kompatibel, das heißt, daß Sie bei Zuweisungen durchaus einer INTEGER-Variablen den Inhalt einer SINGLE-Variablen zuweisen können.

DECLARE erforderlich (EZ)

Diese Fehlermeldung sollte beim Kompilieren auftreten, wenn ein SUB ohne CALL aufgerufen wird, bevor es mit DECLARE vereinbart wurde, oder wenn eine FUNCTION benutzt wird, ohne mit DECLARE vereinbart worden zu sein. In Wirklichkeit erzeugt der erstgenannte Fall jedoch einen „Gleichheitszeichen fehlt“, der zweite führt zu „Datenfeld nicht dimensioniert“, beide eventuell in Kombination mit „Syntaxfehler“.

DEF ohne END DEF (EZ)

Zu einem DEF FN fehlt das zugehörige END DEF. Siehe auch „Unerklärliche Strukturfehler“ weiter unten in diesem Anhang.

DEFxxx Zeichenangabe unzulässig (EZ)

Sie haben einen DEF*xxx*-Befehl (DEFLNG, DEFDBL, DEFINT, DEFSNG, DEFCUR, DEFSTR) falsch angewandt.

Diskette nicht bereit (LZ 71 und VB)

Eine Lese-/Schreiboperation auf eine Diskette kann nicht durchgeführt werden, weil die Laufwerksklappe offen ist. Einige Anti-Viren-Programme erzeugen diesen Fehler, wenn man versucht, auf die Festplatte zu schreiben, und schützen so die Festplatte vor Schreibzugriffen.

Diskette/Festplatte voll (LZ 61 und VB)

Das Schreiben von Daten in eine Datei ist nicht möglich, weil die Diskette/Platte voll ist.

Division durch Null (EZ und LZ)

Tritt bei Verwendung von /, \ und MOD auf, wenn der Divisor 0 ist. Handelt es sich beim Divisor um eine Konstante oder eine symbolische Konstante, wird der Fehler bereits beim Kompilieren erkannt, sonst erst bei der Ausführung des Programms.

DO ohne LOOP (EZ)

Zu einem DO fehlt das LOOP. Siehe auch „Unerklärliche Strukturfehler“ weiter unten in diesem Anhang.

Doppelpunkt nach /C erwartet (EZ)

Der Compiler-Switch /C erfordert die Angabe eines Doppelpunktes und der Kommunikationspufferlänge.

Doppelt vorhandene Definition (EZ oder LZ 10)

- Sie haben ein bereits vereinbartes Symbol erneut definiert, zum Beispiel auf ein bereits mit DIM erstelltes Array erneut den DIM-Befehl angewandt. Wenn Sie ein dynamisches Array mit ERASE löschen, kann es wieder mit DIM dimensioniert werden; bei einem statischen Array führt das zu dem genannten Fehler. Außerdem können statische Arrays nicht mit REDIM erneut dimensioniert werden. Der Fehler tritt auch auf, wenn eine Variable mit DIM vereinbart wird, die zugleich Name einer Funktion ist, oder wenn eine Konstante denselben Namen wie eine Variable oder eine Funktion hat.
- Beim CREATEINDEX-Befehl (ISAM) tritt der Fehler auf, wenn Sie einen bereits existenten Index erneut zu erzeugen versuchen.

Doppelt vorhandene Marke (EZ)

Ein Zeilenlabel kommt innerhalb desselben Moduls doppelt vor. Zeilenlabels sind nicht lokal, das heißt, daß auch zwei verschiedene Prozeduren kein gleichnamiges Label besitzen dürfen. Wenn Sie in verschiedenen Prozeduren gleiche Labels benutzen möchten (zum Beispiel „Fehler“), können Sie entweder jede Prozedur in ein eigenes .BAS-File stecken (dann werden die Labels unterschieden), oder Sie geben den Plan auf und setzen vor jedes Label noch den Namen der Subroutine (zum Beispiel „EingabeFehler“, „MenueFehler“ usw).

Doppelter Wert für eindeutigen Index (LZ 86)

Sie haben versucht, mit UPDATE oder APPEND einen Datensatz in eine ISAM-Datenbank einzutragen, der in einem eindeutigen Feld einen Wert enthält, der bereits in einem anderen Datensatz enthalten ist. Ein eindeutiges Feld ist ein Feld, das einer Indexliste zugrundeliegt, die als universeller Index erstellt wurde (siehe CREATEINDEX).

Auch im umgekehrten Falle wird dieser Fehler erzeugt, wenn Sie also einen eindeutigen Index neu erstellen, die Felder, auf die er sich bezieht, aber bereits Werte doppelt enthalten.

DOS 2.1 oder höher erforderlich (LZ)

Kompilierte Programme laufen erst ab DOS-Version 2.1 aufwärts. Dies ist ein *fataler* Fehler, das heißt, daß das Programm in jedem Falle abgebrochen wird, wenn er auftritt, egal, wieviele Error-Handling-Routinen Sie programmiert haben.

Dynamisches Datenfeld unzulässig (EZ)

Innerhalb der Definition eines selbstdefinierten Typs (TYPE...END TYPE) dürfen nur symbolische Konstanten oder Konstanten, nicht aber Variablen als Ar-

ray-Dimensionsvereinbarung benutzt werden, weil einem selbstdefinierten Typen schon beim Kompilieren eine feste Länge zugeordnet werden muß.

Eingabe über das Ende der Datei hinaus (LZ 62)

Sie haben versucht, mit INPUT$ oder INPUT aus einer sequentiellen Datei Daten zu lesen, obwohl aus dieser Datei bereits alles gelesen wurde.

Element eines Steuerelementefeldes nicht vorhanden (LZ 340)

Sie versuchen, auf ein Element eines Steuerelemente-Arrays mit UNLOAD durch Verwendung einer seiner Eigenschaften zuzugreifen; das Element ist jedoch nicht vorhanden.

Element nicht definiert (EZ)

Dieser Fehler tritt auf, wenn Sie ein Element einer Variablen selbstdefinierten Typs benutzen, das in der TYPE...END TYPE-Definition nicht vorkommt. Normalerweise können Sie z. B. die Variable `Person.Name` ganz normal benutzen; wenn aber `Person` eine Variable von irgendeinem selbstdefinierten Typ ist, und die Typdefinition nicht das Feld `Name` enthält, gibt es diesen Fehler.

ELSE ohne IF (EZ)

Sie benutzen ein ELSE, ohne zuvor den IF...THEN...ELSE-Block mit IF eingeleitet zu haben (vielleicht meinten Sie CASE ELSE und haben nur das CASE vergessen?), oder Sie haben andere Kontrollstrukturen innerhalb des IF...THEN...ELSE-Blocks offen gelassen.

EMS fehlerhaft (LZ)

Es wurden Overlays ins EMS geladen, die nun nicht mehr korrekt abgerufen werden können, weil ein Fehler im EMS vorliegt oder andere Programme in verbotener Weise am EMS herumgepfuscht haben. Dies ist ein *fataler* Fehler, das heißt, daß das Programm in jedem Falle abgebrochen wird, wenn er auftritt, egal, wieviele Error-Handling-Routinen Sie programmiert haben.

END DEF ohne DEF (EZ)

Sie haben END DEF benutzt, ohne vorher eine Funktionsdefinition mit DEF FN eingeleitet zu haben.

END IF ohne Block-IF (EZ)

Es fehlt zu einer IF-Zeile, die mit THEN aufhört und daher einen IF...THEN...ELSE-Block einleitet, das END IF. Siehe auch „Unerklärliche Strukturfehler“ weiter unten in diesem Anhang.

END SELECT ohne SELECT

Sie haben END SELECT benutzt, ohne daß zuvor ein zugehöriges SELECT aufgetreten wäre. Siehe auch „Unerklärliche Strukturfehler“ weiter unten in diesem Anhang.

END SUB oder END FUNCTION muß die letzte Zeile im Fenster sein (EZ)

Sie können keine Zeilen hinter END SUB oder END FUNCTION in einem Prozeduren- oder Funktionsfenster anfügen. Wenn Sie eine Prozedur aus ihrer Mitte heraus verlassen wollen, müssen Sie EXIT SUB bzw. EXIT FUNCTION benutzen.

EVENT ON without /v or /w on command line (EZ)

Sie haben in Ihrem Programm EVENT ON benutzt, ohne beim Kompilieren die Switches /v oder /w anzugeben. Löschen Sie das EVENT ON, oder geben Sie einen der Switches an.

Falsche Anzahl an Argumenten (EZ)

Sie versuchen, eine Prozedur oder Funktion aufzurufen, und haben die falsche Anzahl von Parametern angegeben, oder der DECLARE-Befehl stimmt nicht mit der wirklichen Prozedurdefinition überein.

Falsche Datensatzlänge (LZ 59)

Wenn Sie auf eine mit OPEN FOR RANDOM geöffnete Datei mit den Befehlen GET oder PUT zugreifen, dürfen die dabei angegebenen Record-Variablen keine größere Länge haben als die Satzlänge, mit der die Datei geöffnet wurde (Zusatz LEN = *länge* beim OPEN-Befehl). Wenn beim Öffnen keine Länge angegeben wird, ist die Länge der Standardwert 128.

Fehler kann nicht ausgegeben werden (LZ)

Tritt auf, wenn Sie mittels des ERROR-Befehls einen Fehler erzeugen, der dann nicht korrekt von einer Fehlerbehandlungsroutine abgefangen wird, und für den BASIC keine Meldung zur Verfügung hat.

FIELD-Anweisung aktiv (LZ)

Wenn auf eine RANDOM-Datei ein FIELD-Befehl angewandt wurde, dürfen von da an bei GET- und PUT-Anweisungen keine Satzvariablen mehr, sondern nur noch Satznummern angegeben werden.

FIELD-Überlauf (LZ 56)

Sie können mit einem FIELD-Befehl nicht mehr Bytes zu Puffervariablen zuordnen als die Satzlänge der Datei (die bei OPEN mit LEN=*x* angegeben wurde).

Dieser Fehler tritt auch auf, wenn Sie mit einer LSET- oder RSET-Anweisung mehr Zeichen in eine Puffervariable schreiben wollen als hineinpassen, oder wenn Sie mit einem PRINT- oder WRITE-Befehl mehr Zeichen in eine RANDOM-Datei schreiben, als es die Satzlänge zuläßt. Sie müssen dann zwischendurch einen PUT-Befehl ausführen, der den Puffer auf den Datenträger schreibt, um wieder PRINT und WRITE benutzen zu können.

FOR ohne NEXT (EZ)

Zu einem FOR fehlt das NEXT, oder Sie haben bei NEXT nicht dieselbe Zählervariable wie bei FOR angegeben. Siehe auch „Unerklärliche Strukturfehler“ weiter unten in diesem Anhang.

FOR-Indexvariable bereits verwendet (EZ)

In verschachtelten FOR...NEXT-Schleifen haben Sie mehrmals bei FOR dieselbe Zähler-Variable benutzt.

FOR-Indexvariable nicht zulässig (EZ)

Sie haben eine Variable selbstdefinierten Typs oder einen String als Zählervariable bei FOR...NEXT benutzt.

Formale Parameter nicht eindeutig (EZ)

In einer DECLARE-, SUB- oder FUNCTION-Zeile kommt derselbe Variablenname mehrfach vor.

Funktion nicht definiert (EZ oder LZ bei VB)

Eine DEF FN-Funktion wird benutzt, bevor sie definiert ist, oder Sie vergaßen das Laden einer Quick Library.

Funktionsaufruf unzulässig (LZ 5)

Sie benutzen einen BASIC-Befehl oder eine Funktion mit einem ungültigen (zu großen, zu kleinen) Argument, Sie wenden Funktionen auf Dateien oder Geräte an, auf die sie nicht angewendet werden können, oder Sie benutzen Funktionen oder Befehle, die aufgrund der Hardware des Systems oder wegen bestimmter Compilier-Methoden etc. nicht anwendbar sind.

SCREEN 3 erzeugt einen *Funktionsaufruf unzulässig*, wenn MSHERC.COM nicht geladen ist.

Sehen Sie in jedem Fall im Disketten-Referenzteil nach, und prüfen Sie, welche Nutzungsbeschränkungen für die verwendete Funktion/den verwendeten Befehl gelten.

Gerät nicht verfügbar (LZ 68 und VB)

Das Programm bzw. VBDOS versucht, eine Datei auf einem Laufwerk zu öffnen, das nicht existiert.

Geräte-E/A-Fehler (LZ 57 und VB)

Innerhalb VBDOS tritt dieser Fehler auf, wenn Sie ein Programm ausdrucken wollen und der Drucker nicht bereit oder nicht angeschlossen ist. Innerhalb eines Programms kann dieser oder ein „Zeitüberschreitung“-Fehler auftreten, wenn eine der Drucker- oder Kommunikationsschnittstellen nicht oder nicht in der erwarteten Weise reagiert.

Gleichheitszeichen fehlt (EZ)

Dieser Fehler tritt häufig auf, wenn Sie ein SUB ohne CALL aufrufen, das aber nicht mit DECLARE SUB am Programmanfang deklariert ist. Innerhalb desselben Moduls fügt VBDOS diese DECLARE-Zeilen automatisch ein, aber wenn Routinen aus anderen Modulen benutzt werden sollen, müssen die DECLARE-Zeilen von Hand oder per Include-File eingebunden werden.

Natürlich kann es auch sein, daß Sie wirklich ein Gleichheitszeichen vergessen haben.

GOSUB fehlt (EZ)

Sie haben ON *event* ohne GOSUB benutzt.

GOTO fehlt (EZ)

Sie haben ON ERROR ohne RESUME oder GOTO benutzt.

GOTO oder GOSUB erwartet (EZ)

Nach einer Formulierung ON *variable* fehlt das GOTO oder GOSUB.

Include-Datei zu groß (VB)

Eine Include-Datei darf maximal 64 KB haben.

Index außerhalb des definierten Bereichs (LZ 9 oder EZ)

Als Fehler beim Kompilieren tritt diese Meldung auf, wenn Sie ein Array deklarieren, das die erlaubten Maximalgrößen überschreitet (siehe Anhang A).

Während des Programmablaufs tritt der Fehler auf, wenn ein Array-Element benutzt wird, das außerhalb der mit DIM vereinbarten Grenzen liegt (oder wenn der DIM-Befehl – bei dynamischen Arrays – noch gar nicht ausgeführt wurde). VBDOS prüft die Array-Grenzen immer, während ein mit BC kompiliertes Programm die Array-Grenzen nur dann prüft, wenn beim Kompilieren der Switch /D angegeben wurde.

Index nicht gefunden (LZ 83)

Sie haben mit SETINDEX oder DELETEINDEX einen Index benannt, der nicht in der angegebenen ISAM-Datenbank existiert.

Innerhalb von SUB, FUNCTION oder DEF FN Anweisung ungültig (EZ)

Sie haben im Prozedurcode einen Befehl benutzt, der nur im Modulcode erlaubt ist (zum Beispiel SHARED, COMMON, CLEAR).

Interner Fehler (LZ 51)
Internal error near *xxxx* (EZ bei BC)

Böse Sache. Sie sind auf einen der Fehler in VBDOS gestoßen. Er kann beim Kompilieren oder beim Programmablauf auftreten; beim Programmablauf läßt er sich manchmal mit einer Trapping-Routine abfangen, manchmal nicht. Das nutzt Ihnen nicht viel, denn verbessern können Sie diesen Fehler nicht, genau herausfinden auch nicht. Sie können ihn an Microsoft melden, aber mindestens ein halbes Jahr müssen Sie schon Geduld haben, bis er verbessert ist.

Anderer Tip: Je durchschnittlicher Ihr Programm, desto kleiner die Wahrscheinlichkeit, daß dieser Fehler auftritt. Ändern Sie auf gut Glück an Ihrem Programm herum. Oft ist der Fehler schon durch das Vertauschen von zwei Zeilen zu bezwingen.

Kein aktueller Datensatz (LZ 85)

Sie haben den DELETE-, den RETRIEVE- oder den UPDATE-Befehl angewandt, obwohl der Dateizeiger bereits vor dem ersten bzw. hinter dem letzten Datensatz steht.

Keine Daten mehr in DATA-Zeile vorhanden (LZ 4)

Ein READ-Befehl wurde ausgeführt, obwohl die letzte DATA-Konstante bereits gelesen worden ist.

Komma fehlt (EZ)

Sie haben ein Komma und möglicherweise nachfolgende Parameter vergessen, zum Beispiel PCOPY (0). In VBDOS heißt diese Meldung „Erwartet: ,“.

Konstante ungültig (EZ)

Sie haben versucht, mit CONST einer symbolischen Konstanten einen Wert zuzuordnen, der einen Funktionsaufruf oder eine andere für Konstanten unzulässige Operation enthält (zum Beispiel `CONST a$ = CHR$(5)`).

Leistungsmerkmal nicht verfügbar (LZ 73)

Das Programm benutzt einen Befehl, der durch ein Verzicht-File beim Linken ausgeschlossen wurde (zum Beispiel SCREEN 9, wenn Sie mit NOGRAPH, NOEGA oder TSCNIO*xx* gelinkt haben).

Leistungsmerkmal entfernt (LZ 80)

Dieser Fehler tritt in VBDOS auf, wenn Sie ISAM-Befehle verwenden, ohne vorher PROISAM.EXE oder PROISAMD.EXE geladen zu haben, sowie in kompilierten Programmen, denen Sie durch Einbinden von NOISAM.OBJ in das Programm oder das Runtime-Modul die ISAM-Routinen vorenthalten.

Listing für binäre BASIC-Quelldateien kann nicht erzeugt werden (EZ)

Sie können kein Source-Listing erstellen (Compiler-Switch /A), wenn das Programm im komprimierten VBDOS-Format abgespeichert ist. Lassen Sie /A weg, oder laden Sie das Programm mit VBDOS, und speichern Sie es als Textdatei.

Marke nicht definiert (EZ)

Ein Befehl bezieht sich auf eine Zeilennummer oder ein Zeilenlabel, das nicht existiert. Der Fehler kann auch auftreten, wenn das betreffende Label nicht in derselben Code-Einheit (zum Beispiel im gleichen SUB) steht.

Modulcode zu groß (VB)

Ihr Modulcode ist zu groß für VBDOS. Modulcode kann nicht ins EMS verschoben werden. Versuchen Sie, einen Teil davon zu Prozedurcode zu machen, indem Sie neue SUBs und FUNCTIONs erstellen, und dadurch den EMs nutzen.

Muß die erste Anweisung in der Zeile sein (EZ)

In den IF-, ELSEIF-, ELSE- und END IF-Zeilen einer mehrzeiligen IF...THEN...ELSE-Konstruktion muß das Strukturwort am Anfang der Zeile stehen. Sie können es nicht mit Doppelpunkten zu einem anderen Befehl in die gleiche Zeile schreiben.

Name ungültig (LZ 81)

Sie haben bei ISAM einen Namen für einen Typen, ein Feld innerhalb eines Typs, eine Datenbank oder einen Index angegeben, der länger als 30 Zeichen ist, nicht mit einem Buchstaben beginnt oder andere Zeichen als die Buchstaben A-Z und die Ziffern 0 bis 9 enthält.

Nicht genügend ISAM-Puffer (LZ 89)

Für eine ISAM-Operation sind nicht genügend ISAM-Puffer vereinbart worden. Siehe Kapitel 12.

Nicht genügend Speicher (LZ 7 oder EZ)

Dieser Fehler kann immer auftreten – als Fehler beim Aufruf des Compilers oder von VBDOS, als Fehler beim Laden von Dateien, beim Kompilieren oder beim Ausführen eines Programms. Ihr Programm braucht mehr Speicher. Entfernen Sie unnötige speicherresidente Programme, Gerätetreiber oder Dateien, die in VBDOS geladen sind. Wenn Sie ein Programm mit SHELL aufrufen, kann der Fehler leicht auftreten, weil dann zwei Programme zugleich im Speicher sind. Wenn Sie EMS haben: Haben Sie einen EMS-Treiber geladen? Ohne ihn ist die Benutzung von EMS durch VBDOS nicht möglich. Rufen Sie VBDOS mit der Option /S:1 auf, dann haben Sie etwas mehr Speicher zur Verfügung. Sparen Sie mit Arrays; benutzen Sie nur die kleinsten geeigneten Datentypen.

Nur Variablen einfacher Datentypen erlaubt (EZ)

Mit READ und INPUT können nur einfache Variablen, keine ganzen Arrays oder Variablen selbstdefinierten Typs, eingelesen werden.

ON Ereignis ohne /V oder /W in der Befehlszeile (EZ)

Ihr Programm benutzt den ON *event*-Befehl, ohne daß Sie beim Aufruf von BC /V oder /W angegeben haben.

ON ERROR ohne /E in der Befehlszeile (EZ)

Das Programm enthält ON ERROR-Befehle, ohne daß Sie /X oder /E auf der BC-Befehlszeile angegeben haben.

Operation auf Index NULL ungültig (LZ)

SEEK kann nicht benutzt werden, wenn der Null-Index aktiv ist.

Parametertypen unverträglich (EZ)

Eine SUB- oder FUNCTION-Zeile stimmt in Anzahl und Typ der genannten Parameter nicht mit der zugehörigen DECLARE-Zeile überein. Dieser Fehler tritt meistens auf, wenn Sie in VBDOS den Aufruf einer Prozedur oder Funktion verändern, VBDOS aber bereits eine DECLARE-Zeile eingefügt hat. In VBDOS können Sie dann einfach die betreffende DECLARE-Zeile löschen, da sie ohnehin neu erstellt wird.

Pfad nicht gefunden (LZ 76)

In einer MKDIR-, CHDIR-, RMDIR- oder OPEN-Anweisung wurde ein ungültiger Pfad angegeben. DIR$ verursacht keinen Fehler, wenn man einen ungültigen Pfad angibt, sondern gibt einfach einen Leerstring zurück.

Pfad/Datei-Zugriffsfehler (LZ 75 oder VB)

Sie versuchen, eine Datei zu speichern, aber dadurch würde eine bestehende Read Only-Datei überschrieben. Dieser Fehler tritt auch auf, wenn eine Dateiangabe eine ungültige oder fehlerhafte Pfadangabe enthält.

Prozedur bereits in Quick Library definiert (EZ)

Sie können in VBDOS keine Prozedur eingeben, wenn unter gleichem Namen bereits eine Prozedur in einer geladenen Quick Library existiert.

Puffergröße nach /C: erwartet (EZ)

Wenn Sie den Switch /C beim Aufruf des Compilers benutzen, müssen Sie dahinter – ohne Leerzeichen – die Größe des Kommunikationspuffers angeben. 0 ist nicht erlaubt. Der Compiler bricht nicht ab, sondern ignoriert den Switch.

RESUME fehlt (LZ 19)

Das Programm ist zu Ende, ohne daß eine Error-Trapping-Routine mit RESUME verlassen wurde. Bauen Sie entweder einen RESUME-Befehl ein oder einen STOP-, END- oder SYSTEM-Befehl, der das Programm explizit verläßt.

RESUME ohne /X in der Befehlszeile (EZ)

Für ein Programm, das den RESUME-Befehl in der Form RESUME oder RESUME NEXT enthält, muß beim Aufruf von BC der Switch /X angegeben werden. Für RESUME *zeilennummer/-label* reicht ein /E aus.

Spalte ungültig (LZ 84)

Sie können eine ISAM-Datei, die schon existiert, nicht mit einem Variablentyp öffnen, der ein Feld enthält, das der Typ, mit dem die Datei erstellt wurde, nicht besaß. Außerdem können Sie keinen nichtexistenten Feldnamen und keinen Namen eines nicht indizierbaren Feldes in einer CREATEINDEX-Anweisung benutzen.

Speicherinhalt für Zeichenfolge verändert/ungültig (LZ oder VB)

Mein persönlicher Lieblingsfehler – der berüchtigte „String space corrupt“ in deutschem Gewand. Er tritt immer dann auf, wenn das Programm so perfekt ist, daß es keine anderen Fehler enthält.

Nein, im Ernst: Dies ist einer der Fehler, deren Ursache meist schwer zu finden ist. Wenn er in VBDOS auftritt, verabschiedet sich VBDOS mitsamt allen geladenen Programmen ins DOS, und man sollte neu booten, bevor man VBDOS erneut aufruft. Tritt er in einem kompilierten Programm auf, stürzt der Rechner entweder postwendend ab, oder es erfolgt die Rückkehr ins Betriebssystem.

Dieser Fehler bedeutet, daß BASIC den Stringspeicher nicht so vorfindet, wie es ihn erwartet. Gründe dafür können Programme oder Routinen anderer Sprachen sein, die unbefugt darin herumgepfuscht haben (z. B. unsachgemäße Anwendung von CopyData aus Kapitel 18); ein POKE-Befehl kann ebenfalls daran schuld sein, wenn er wichtige Bytes im Stringspeicher getroffen hat. Häufige Ursache sind auch COMMON-Blocks, die nicht genau übereinstimmen.

Dies ist ein *fataler* Fehler, das heißt, daß das Programm in jedem Falle abgebrochen wird, wenn er auftritt, egal, wieviele Error-Handling-Routinen Sie programmiert haben. Siehe auch „Unerklärliche Systemfehler“ weiter unten in diesem Anhang.

Sternchen fehlt (EZ)

In selbstdefinierten Typen dürfen nur Strings mit fester Länge benutzt werden. DIM x AS STRING ist dort also nicht zulässig, es muß heißen: DIM x AS STRING * *länge*, wobei Sie für *länge* eine symbolische Konstante oder eine Zahl einsetzen müssen.

Steuerungsstruktur in IF...THEN...ELSE unvollständig (EZ)

Sie dürfen das Ende einer Kontrollstruktur nicht von einer Bedingung mit IF abhängig machen. Das war im Interpreter-BASIC noch möglich, hier jedoch führt eine Zeile wie `IF a% = 0 THEN NEXT` unweigerlich zu diesem Fehler. Zu jedem Kontrollstrukturanfang muß ein und nur ein entsprechender Ende-Befehl im Programm vorhanden sein.

SUB/FUNCTION ohne END SUB/END FUNCTION (EZ)

Bei einer Prozedur oder Funktion fehlt der END SUB- beziehungsweise END FUNCTION-Befehl. In VBDOS tritt diese Meldung manchmal ohne ersichtlichen Grund auf (d. h., das Programm ist völlig in Ordnung, trotzdem beharrt VBDOS darauf, es fehle ein END SUB). Dann ist es ratsam, das Programm als Textdatei zu speichern, VBDOS zu verlassen und neu aufzurufen.

Syntaxfehler (EZ und LZ 2)

Als EZ-Fehler wird diese Meldung von falsch eingetippten BASIC-Befehlen oder von Prozeduraufrufen, denen die zugehörige Prozedur abhanden gekommen ist, verursacht.

Als LZ-Fehler wird er generiert, wenn Sie mit DATA in eine numerische Variable eine Stringkonstante einzulesen versuchen, oder wenn die Anzahl der Variablen bei einem SEEK-Befehl die Anzahl der indizierten Felder übersteigt.

Tabelle nicht gefunden (LZ 82)

Sie haben mittels DELETETABLE eine Datenbank löschen wollen, die in der genannten ISAM-Datei gar nicht vorhanden ist.

TYPE nicht definiert (EZ)

Für eine AS *typ*-Klausel (zum Beispiel in COMMON, DIM oder in einer TYPE...END TYPE-Anweisung) ist der angegebene Typ weder INTEGER, LONG, CURRENCY, SINGLE, LONG, STRING, STRING * *n* noch mit TYPE...END TYPE definiert.

Überlauf (LZ 6)

Das Ergebnis einer Berechnung ist zu groß für den angegebenen Datentyp. Der Befehl `a! = b% + c%` verursacht zum Beispiel einen Überlauf, wenn `b%` und `c%` beide 20.000 sind. Die Summe 40.000 paßt zwar ohne weiteres in den Datentyp SINGLE (`a!`), jedoch wird die Berechnung als INTEGER-Berechnung durchgeführt, weil sie nur INTEGER-Zahlen enthält, und ihr Ergebnis übersteigt dann den INTEGER-Datenbereich.

Überlauf bei arithmetischer Operation (EZ)

Eine Berechnung, die schon beim Kompilieren ausgeführt wird, hat ein zu großes Ergebnis, oder eine Konstantenzuweisung überschreitet den erlaubten Bereich (z. B. `CONST a% = 40000`).

Überlauf des Datenspeichers (EZ)

Der zur Verfügung stehende Speicherplatz reicht nicht aus, um das Programm zu kompilieren. Es ist möglich, daß ein Programm in VBDOS einwandfrei läuft und beim Kompilieren dann diese Meldung erzeugt, wenn es zu groß ist (VBDOS hat nicht so strenge Speicher-Restriktionen wie der Compiler BC).

- Teilen Sie Ihr Programm in mehrere Module (verschiedene .BAS-Programme, die mit LINK dennoch zu einem EXE-File verarbeitet werden können), oder
- verkürzen Sie das Programm, indem Sie weniger Konstanten verwenden, Strings aus einer Datei einlesen oder die Größe statischer Arrays verkleinern, oder
- versuchen Sie einmal, mit dem Switch /S zu kompilieren.

Überlauf des Kommunikationspuffers (LZ 69)

Beim Umgang mit einer seriellen Schnittstelle haben Sie zu selten oder zu langsam (ein langsamer Rechner und eine hohe Baudrate?) den Kommunikationspuffer abgefragt, so daß dieser nicht mehr alle ankommenden Zeichen halten kann. Vergrößern Sie den Buffer mit dem Compiler-Switch /C oder einem Zusatz beim OPEN-Befehl, oder fragen Sie den Buffer häufiger ab.

Überlauf des Programmspeichers (EZ bei BC)

Ein einzelnes Modul darf nicht größer als 64 KB sein (VBDOS akzeptiert größere Module, nicht aber der Compiler BC). Wenn Sie Subroutinen in dem Modul haben, ist es ein Leichtes, ein zweites Modul zu eröffnen und einige Subroutinen dorthinein zu verschieben. Dann müssen Sie unter Umständen noch einige COMMON-, DECLARE- oder CONST-Befehle in das neue Modul kopieren, können es dann getrennt kompilieren, und mit LINK können beide Module wieder zu einem Programm vereint werden.

Unerwartetes Dateiende in TYPE-Deklaration (EZ)

Die .BAS-Datei ist mitten innerhalb einer TYPE...END TYPE-Definition plötzlich zu Ende.

Ungültig, wenn Formen angezeigt werden (LZ 272)

Sie haben einen Befehl verwendet, der nicht funktioniert, während eine Form auf dem Bildschirm angezeigt wird (z. B. PRINT oder SCREEN). Entfernen Sie vorher alle Formen (`SCREEN.HIDE`).

Untere Grenze überschreitet obere Grenze (EZ)

Sie haben in einem DIM-, REDIM- oder TYPE-Befehl ein Array spezifiziert, dessen untere Grenze größer als seine obere ist. Wenn das während des Programmablaufs vorkommt, wird ein *Funktionsaufruf unzulässig* erzeugt; die Meldung tritt schon beim Kompilieren auf, wenn Sie Konstanten beim Dimensionieren verwenden.

Unterprogrammfehler (EZ)

Sie benutzen einen Befehl im Prozedurcode, der nur im Modulcode angewandt werden darf oder umgekehrt, oder Sie versuchen, mit GOTO, GOSUB oder RETURN *zeilennummer/-label* zu einer Zeile zu springen, die in einer anderen Code-Einheit als der Befehl liegt (zum Beispiel vom SUB in den Modulcode etc.).

Unzulässig im Direktmodus (VB)

Sie haben im Direkt-Fenster einen Befehl eingegeben, der nur in Programmen verwendet werden darf.

Variable erforderlich (EZ oder LZ 40)

Ein INPUT-, LET-, READ- oder SHARED-Befehl wird nicht von einem Variablennamen gefolgt, oder Sie haben bei einem GET- oder PUT-Befehl keine Satzvariable, sondern eine Konstante oder Funktion (zum Beispiel `PUT #1, , CHR$(26)`) angegeben.

Als LZ-Fehler tritt der Fehler auf, wenn bei einer GET- oder PUT-Anweisung die Satzvariable völlig weggelassen wurde und die Datei im BINARY-Modus geöffnet ist.

Variable nicht deklariert (EZ)

Sie verwenden OPTION EXPLICIT in Ihrem Programm und müssen daher jede Variable, die Sie verwenden möchten, mit DIM deklarieren.

Verschachtelte Funktionsdefinition (EZ)

Innerhalb einer SUB...END SUB oder FUNCTION...END FUNCTION Konstruktion darf kein zweiter solcher Block auftreten. Dasselbe gilt auch für DEF FN.

Währungsdatentyp im alternativen Mathematikpaket unzulässig (EZ)

Die Alternate Math-Library unterstützt den CURRENCY-Datentyp nicht. Deshalb dürfen Sie keine Currency-Variablen und auch nicht den Befehl DEFCUR oder die Funktionen CVC und MKC$ benutzen, wenn Sie mit /FPa kompilieren. Ändern Sie das Programm, oder benutzen Sie /FPi.

Zeichen ungültig (EZ)

Ihr Programm enthält ein Steuerzeichen oder ein anderes nicht erlaubtes Zeichen (zum Beispiel einen Umlaut außerhalb von Stringkonstanten).

Zeichenfolge fester Länge unzulässig (EZ)

Sie haben einen String mit fester Länge in einer SUB-, FUNCTION- oder DECLARE-Zeile oder in einem SSEG- oder SADD-Funktionsaufruf benutzt. Nur Strings mit variabler Länge sind hier erlaubt.

Zeichenfolge variabler Länge erforderlich (EZ)

Sie haben in einem FIELD-Befehl einen String fester Länge angegeben.

Zeichenfolgenausdruck zu komplex (LZ 16)

Mit einer INPUT-Anweisung können maximal 15 Stringvariablen eingelesen werden. Ein anderer Grund für diesen Fehler kann eine Verknüpfung von Strings sein, die so komplex ist, daß BASIC Schwierigkeiten mit seinen temporären Strings bekommt (Speicherplatz?). Splitten Sie den Ausdruck in mehrere Teilausdrücke auf, um das Problem zu umgehen.

Zeichenfolgenvariable erforderlich (EZ)

Sie haben einen RSET-Befehl falsch angewendet. Er kann nur mit Strings benutzt werden.

Zeile zu lang (EZ)

Die Zeilenlänge darf 255 Zeichen nicht überschreiten.

Zeitmesser nicht verfügbar (LZ 260)

Es sind bereits 16 Zeitmesser aktiv. Auch selbstdefinierte Steuerelemente, die das Timer-Ereignis verwenden, zählen mit.

Zeitüberschreitung am Gerät aufgetreten (LZ 24 und VB)

Innerhalb VBDOS bedeutet dieser Fehler dasselbe wie „Geräte-E/A-Fehler“. Innerhalb eines Programms kommt er zustande, wenn eine Schnittstelle nicht in der vorgegebenen Zeit reagiert (beim Öffnen der Kommunikationsschnittstellen können verschiedene Timeout-Zeiten angegeben werden).

Zu viele Argumente im Funktionsaufruf (EZ)

Sie haben versucht, einer Prozedur oder Funktion mehr als 60 verschiedene Variablen als Parameter zu übergeben.

Zu viele benannte COMMON-Blocks (EZ)

Ein Programm darf nur maximal 126 verschiedene COMMON-Blocks, die mit Blocknamen versehen sind, enthalten.

Zu viele Dateien (LZ 67 oder EZ)

Als Fehler beim Kompilieren tritt diese Meldung nur auf, wenn Sie Include-Files auf mehr als fünf Ebenen verschachteln.

Als LZ-Fehler wird er generiert, wenn Sie eine Datei öffnen wollen und schon 15 andere offen sind (oder auch schon früher; wenn das Limit in CONFIG.SYS zu klein ist – siehe FREEFILE im Disketten-Referenzteil), oder wenn Sie zu viele ISAM-Datenbanken gleichzeitig öffnen wollen (siehe „Anzahl gleichzeitig geöffneter ISAM-Dateien und -Datenbanken“ in Kapitel 12). Außerdem kann dieser Fehler auftreten, wenn Sie versuchen, im Hauptverzeichnis eines Datenträgers eine Datei anzulegen und das Hauptverzeichnis schon voll ist. Die Anzahl der Einträge im Hauptverzeichnis eines Datenträgers wird vom Betriebssystem begrenzt.

Zu viele Dimensionen (EZ)

Ein Array darf maximal 60 Dimensionen haben.

Zu viele lokale Zeichenfolgenkonstanten in Unterprogramm (EZ)

Eine Prozedur oder Funktion darf maximal 255 lokale Stringvariablen haben. Sie können das Problem umgehen, indem Sie die Prozedur globale Variablen benutzen lassen.

Zu viele Marken (EZ)

In einer ON...GOTO- oder ON...GOSUB-Konstruktion können nicht mehr als 60 Zeilennummern oder -labels angegeben werden.

Zu viele TYPE-Definitionen (EZ)

Es dürfen maximal 240 TYPE-Definitionen in einem Programm enthalten sein.

Zu viele Variablen für INPUT (EZ)

Eine INPUT-Anweisung darf nicht mehr als 60 Variablen einlesen (darunter maximal 15 Stringvariablen, aber wenn dieses Limit überschritten wird, tritt während des Programmablaufs ein *Zeichenfolgenausdruck zu komplex*-Fehler auf).

Zugriff verweigert (LZ 70)

Sie haben versucht, eine Read Only-Datei zu überschreiben oder eine Datei zu benutzen, die vom Netzwerksystem gesperrt wurde (z. B. weil ein anderer Prozeß sie gerade benutzt).

Zur Entwurfszeit erzeugte Steuerelemente können nicht entfernt werden (LZ 362)

Sie haben versucht, mit UNLOAD ein Steuerelement zu entfernen, das Sie im Form-Designer gezeichnet haben. Es können jedoch nur die Steuerelemente entfernt werden, die zur Laufzeit mit LOAD erzeugt wurden. Verstecken Sie stattdessen das Steuerelement (*Visible* = FALSE).

Zwischen Disketten/Festplatten umbenennen (LZ 74)

Die beiden Dateinamen, die mit dem NAME-Befehl angegeben werden, müssen auf dem gleichen Laufwerk liegen.

Unerklärliche Systemfehler

Wenn Fehler auftreten, deren Ursache Sie nicht feststellen können, und wenn es sich dann auch noch um Fehler wie „Speicherinhalt für Zeichenfolge verändert/ ungültig“ handelt, ist es sehr wahrscheinlich, daß der Fehler schon verursacht wird, lange bevor er ans Licht tritt. Grundsätzlich kann ziemlich jeder LZ-Fehler auftreten, wenn der Speicherbereich „zerschossen“ wurde, weil irgendwelche Befehle oder Funktionen unbefugt in den Speicher geschrieben haben (POKE?). Schuld daran können beispielsweise Routinen sein, die in anderen Sprachen geschrieben sind (benutzen Sie vielleicht eine alte Library, die für QuickBASIC 4 geschrieben wurde und von Far Strings keine Ahnung hat?).

Häufig passiert das auch, wenn falsche Array-Indizes benutzt werden. Es könnte zum Beispiel sein, daß das Array `Wuerfel` mit den Dimensionen 3, 3, 3 vereinbart wurde, daß aber – aufgrund eines logischen Fehlers o. ä. – an irgendeiner Stelle auf das Element (–5, 1, 2) des Arrays zugegriffen wird. Wenn Ihr Programm außerhalb VBDOS läuft und nicht mit /D kompiliert wurde, wird

dieser Fehler nicht bemerkt, aber es kann durchaus vorkommen, daß durch Zugriff auf dieses ungültige Element andere wichtige Daten überschrieben werden, was sich dann in einem ganz anderen Programmteil eine halbe Stunde später erst bemerkbar macht.

Ebenfalls gerngesehener Kandidat für solche Fehler sind unpassende COMMON-Blocks, also COMMONs in zwei zusammengelinkten oder sich mit CHAIN aufrufenden Programmen, die nicht übereinstimmen. Wenn Sie Glück haben, meldet der Linker einen Fehler, wenn nicht, dann bemerken Sie es irgendwann im Programm – in Form eines dieser „Unerklärlichen Systemfehler".

Außerdem können auch in Zusammenhang mit Overlays seltsame Dinge geschehen – insbesondere dann, wenn Sie versehentlich das erste beim Linken angegebene Modul einklammern oder mehr als 64 verschachtelte Overlay-Aufrufe im Programm stattfinden.

Unerklärliche Strukturfehler

Strukturfehler wie „NEXT ohne FOR" usw. haben Ihre Ursache häufig nicht darin, daß wirklich ein FOR vergessen wurde, sondern daß irgendeine Konstruktion (zum Beispiel IF...END IF oder SELECT CASE) unvollständig abgeschlossen ist oder Konstruktionen auf nicht erlaubte Weise verschachtelt werden. Die häufigste Ursache ist ein fehlendes (oder überflüssiges – einzeilige Konstruktionen benötigen ja keines) END IF irgendwo, und selten ist die Ursache wirklich die, die der Compiler nennt.

LINK-Fehler (Auswahl)

Hier folgt eine Auswahl von LINK-Fehlermeldungen, die mit BASIC auftreten können und *nicht* für sich selbst sprechen.

/CO sperrt /EXEPACK

Die Option /CO schaltet die Komprimierung von EXE-Dateien, die Sie mit dem Switch /E angegeben haben, aus.

/DYNAMIC sperrt /EXEPACK

Sie arbeiten mit Overlays. Der Switch /E ist hierbei nicht zugelassen bzw. hat keine Wirkung, wenn Sie ihn angeben.

Antwortdatei kann nicht geöffnet werden

Die Datei, die auf der Befehlszeile hinter dem @-Zeichen steht (die Steuerungsdatei), kann nicht geöffnet werden. Namen von OBJ-Dateien und Libraries dürfen nicht mit @ beginnen, um LINK nicht zu verwirren.

Ausführbare Datei kann nicht geöffnet werden

Das EXE-File kann nicht erzeugt werden, weil die Platte voll ist oder ein bestehendes EXE-File, das dadurch überschrieben würde, das Read-Only-Attribut hat und nicht überschrieben werden darf.

Bezeichnung der Ausgabedatei ist...

Diese Meldung tritt auf, wenn Sie Quick Libraries erzeugen, ohne die Extension QLB explizit anzugeben. Dann nimmt LINK an, daß Sie vielleicht denken, daß die entstehende Datei wie immer die Extension EXE bekommt, und warnt Sie, daß das nicht der Fall ist, sondern daß die entstehende Datei QLB heißen wird. Eine Meldung, die ignoriert werden kann.

Datei für /EXEPACK ungeeignet; erneut binden ohne /EXEPACK

Das mit /E komprimierte File ist länger, als es das nichtkomprimierte wäre.

Geforderte Segmentgrenze zu hoch

Es ist nicht genügend Speicherplatz vorhanden, um die mit /SE angegebene maximale Segmentzahl zu verwalten.

Modul für Microsoft Overlay-Manager nicht gefunden

Sie Schelm haben versucht, mit der Standardversion von VBDOS Overlays zu benutzen. Das ist nicht möglich, weil die Library VBDRT10.LIB bzw. VBDCL10.LIB nicht die für Overlaytechnik benötigten Routinen enthält.

Modul zur Unterstützung von Quick-Bibliotheken fehlt

Wenn Sie Quick Libraries erzeugen wollen, müssen Sie die Library VBDOS-QLB.LIB mit angeben.

Moduldefinitionsdatei kann nicht geöffnet werden

Die angegebene Definitionsdatei konnte nicht geöffnet werden. Vielleicht haben Sie sich in einer Steuerungsdatei mit den Zeilen verzählt, so daß der Linker Ihren EXE-Dateinamen als Namen einer Definitionsdatei versteht?

Nicht aufgelöste externe Symbole

Eine Prozedur oder Funktion, die in einem der OBJ-Files oder in einer Routine, die aus einer Library eingebunden werden mußte, aufgerufen wird, ist nicht zu finden. Sie haben vermutlich vergessen, ein OBJ-File oder eine Library anzugeben, das/die die genannten Routinen enthält.

Nicht genügend Speicher für ausführbare Datei

Das EXE-File konnte nicht erzeugt werden, weil die Diskette/Platte voll ist.

Objektmodul ungültig

Sie haben eine Datei als OBJ-Datei angegeben, die keine ist, oder Sie sollten die OBJ-Datei neu kompilieren, weil sie durch einen Fehler zerstört wurde.

Stapel und Daten überschreiten 64K

Ihr EXE-Programm ist zu groß. Stellen Sie den Stack kleiner ein (evtl. Option /ST verwenden), modularisieren Sie das Programm (SUBs/FUNCTIONs erzeugen und getrennt kompilieren), oder probieren Sie es einmal mit Overlays.

Symbol mehrfach definiert

In verschiedenen angegebenen OBJ-Files oder Libraries ist dieselbe Prozedur oder Funktion mehrfach definiert. Dieser Fehler kann auch auftreten, wenn Sie falsche Libraries benutzen. Versuchen Sie es in jedem Fall mit dem Switch /NOE; erst, wenn das keine Wirkung zeigt, müssen Sie versuchen, den Fehler anderweitig zu eliminieren. Die Verwendung von /NOD kann unter Umständen ebenfalls das Problem lösen, nämlich dann, wenn die OBJ-Dateien, die Sie verwenden, zum Teil mit /FPi und zum Teil mit /FPa kompiliert wurden.

Temporäre Datei kann nicht erzeugt werden
Temporäre Datei kann nicht geöffnet werden

Für die temporäre Datei, die LINK anlegen wollte, war nicht genügend Platz auf der Platte.

Überlauf in Referenz-Festlegung

Eventuell ist eine Ihrer OBJ-Dateien oder Libraries unbrauchbar. Sie sollten alle neu erstellen. Hilft alles nichts, sollten Sie versuchen, Ihre Programme zu verkleinern (auf jeden Fall beim Kompilieren den /D-Switch weglassen, wenn möglich, auch auf /X und /V verzichten).

Unerwartetes Dateiende

Eine Library hat ein ungültiges Format. Erzeugen Sie sie neu, oder prüfen Sie sie zunächst mit LIB.

Zeile in Antwortdatei zu lang

Eine Zeile in der mit @ eingeleiteten Steuerungsdatei war länger als 255 Zeichen. Die OBJ- und die LIB-Zeile können Sie problemlos aufteilen, indem Sie am Ende der Zeile ein + schreiben.

Zu viele Bibliotheken

Es dürfen nicht mehr als 32 Libraries angegeben werden. Notfalls können Sie aus mehreren Libraries mittels des Programms LIB.EXE eine einzige Library erstellen.

Zu viele Overlays

Keine Sorge: Dieser Fehler tritt nicht auf, wenn Sie mehr als 127 Overlays verwenden (wie es im Handbuch steht). Ich habe nicht ausprobiert, wieviele Overlays man verwenden kann; lt. Microsoft (an anderer Stelle) sind es 2.047.

Zu viele Segmente

Benutzen Sie den Switch /SE, um die erlaubte Segmentzahl höher als den Defaultwert 128 einzustellen.

LIB-Fehler (Auswahl)

Antwortdatei kann nicht geöffnet werden

Die Datei, die auf der Befehlszeile hinter dem @-Zeichen steht (die Steuerungsdatei), kann nicht geöffnet werden. Namen von OBJ-Dateien und Libraries dürfen nicht mit @ beginnen, um LIB nicht zu verwirren.

Modul nicht in Bibliothek enthalten; wird ignoriert

Sie haben den Befehl –+ benutzt, um ein in der Library existierendes OBJ-File durch eine neue Version, die auf der Platte steht, zu ersetzen. Das genannte OBJ-File war jedoch in der Library überhaupt nicht enthalten, und so wird nur der Befehl + ausgeführt.

Modulname bereits vorhanden; wird ignoriert

Wenn Sie eine OBJ-Datei in die Library aufnehmen wollen, die bereits darin enthalten ist, wird der Befehl ignoriert.

Seitengröße ist ungültig und wird ignoriert

Sie haben mit /P eine ungültige Seitengröße angegeben (Minimum ist 16).

Symbol bereits im Modul *xxx* definiert; Neudefinition wird ignoriert

Eines der OBJ-Files, die Sie in die Library aufnehmen, enthält eine Routine, die in einem anderen OBJ-File, das in der Library steht, bereits enthalten ist. Die neue Version wird ignoriert.

Fehlercodes der Toolboxen

Die Präsentationsgrafik- und die Font-Toolbox haben eine eigene globale Fehlervariable, die benutzt wird, um Fehler, die bei der Grafikerstellung oder Textausgabe auftreten, dem aufrufenden Programm zu übermitteln. Für jeden Fehler, der auftreten kann, sind Konstanten in den jeweiligen Include-Files definiert, die es einfacher machen sollen, die Fehler abzufragen.

Präsentationsgrafiken

Die Fehlervariable heißt hier *ChartErr*.

Wert	*Konstante*	*Fehlerbeschreibung*
15	`cBadLogBase`	Die Basis für den Logarithmus in der ChartEnvironment-Variable ist kleiner oder gleich Null.
20	`cBadScaleFactor`	Der Skalierungsfaktor in der ChartEnvironment-Variablen, durch den alle Werte dividiert werden sollen, ist Null.
25	`cBadScreen`	Ein ungültiges Argument zu ChartScreen wurde benutzt.
30	`cBadStyle`	Sie haben mit DefaultChart oder von Hand eine ungültige Grafik-Ausführung (cLines, cNoLines usw.) in die ChartEnvironment-Variable eingetragen.
105	`cBadDataWindow`	Das Datenfenster in ChartEnvironment ist zu klein.
110	`cBadLegendWindow`	Das Legendenfenster in ChartEnvironment ist zu klein.
135	`cBadType`	Sie haben mit DefaultChart oder von Hand einen ungültigen Grafiktyp in die ChartEnvironment-Variable eingetragen.
155	`cTooFewSeries`	In einer Grafik mit mehreren Reihen (ChartMS, ChartScatterMS) haben Sie weniger als eine Reihe zeichnen lassen wollen.
160	`cTooSmallN`	Sie versuchen, eine Grafik mit weniger als einem Wert zu erzeugen.
165	`cBadPalette`	Die Farb- und Musterpalette ist falsch dimensioniert.
170	`cPalettesNotSet`	Die Palette ist nicht eingerichtet (seltener Fall, geschieht gewöhnlich automatisch).
175	`cNoFontSpace`	Sie haben keinen Font geladen, und der Speicherplatz ist so knapp, daß nicht einmal der interne Font geladen werden kann.
> 200		Ein gewöhnlicher BASIC-Fehler (beispielsweise 7 = Nicht genügend Speicher) ist aufgetreten; zu der Fehlernummer wird 200 addiert, also wäre *ChartErr* bei „Nicht genügend Speicher“ 207.

Font-Toolbox

Die Fehlervariable heißt hier *FontErr*.

Wert	*Konstante*	*Fehlerbeschreibung*
1	cFileNotFound	Die angegebene Datei wurde nicht gefunden (bei RegisterFonts).
2	cBadFontSpec	Bei LoadFont wurde eine ungültige Font-Bezeichnung angegeben.
5	cBadFontFile	Ungültige Font-Datei.
6	cBadFontLimit	Mit SetMaxFonts wurden ungültige Grenzwerte angegeben.
7	cTooManyFonts	Es wurde versucht, mehr Fonts zu registieren oder zu laden, als per SetMaxFonts eingestellt war.
8	cNoFonts	Es wurde versucht, auf einen Font zuzugreifen, obwohl keiner geladen war.
10	cBadFontType	Die Font-Datei ist zwar nicht fehlerhaft, enthält aber keinen Bitmap-Font, und deshalb kann die Toolbox damit nichts anfangen.
11	cBadFontNumber	Es wurde eine ungültige Fontnummer angegeben.
12	cNoFontMem	Der Speicherplatz reicht nicht aus, um die angeforderten Fonts zu laden oder zu registrieren.
> 200		Ein gewöhnlicher BASIC-Fehler (beispielsweise 7 = Nicht genügend Speicher) ist aufgetreten; zu der Fehlernummer wird 200 addiert, also wäre *FontErr* bei „Nicht genügend Speicher“ 207.

Code-Tabellen Anhang D

Scancodes

Die hier angegebenen Scancodes beziehen sich auf die deutschen Tastaturbeschriftungen. Da Scancodes – im Gegensatz zu (erweiterten) ASCII-Codes – jedoch bestimmte *Tasten* bezeichnen und nicht bestimmte *Zeichen*, kann es hier durchaus zu Unstimmigkeiten kommen. Der Scancode 44 gehört zum Beispiel bei einer deutschen Tastatur zur Taste Y, während er bei einer amerikanischen Tastatur die Taste Z meint. Es ist *dieselbe* Taste, deshalb auch derselbe Scancode, aber der Tastaturtreiber ordnet der Taste verschiedene Zeichen zu, und die Hersteller schreiben einen anderen Buchstaben drauf.

Taste	*Code*
ESC	1
F1	59
F2	60
F3	61
F4	62
F5	63
F6	64
F7	65
F8	66
F9	67
F10	68
F11	87
F12	88
° ^	41
1 ! ²	2
2 " ³	3
3 §	4
4 $	5
5 %	6
6 &	7
7 /	8
8 (	9
9)	10
0 =	11
? ß \	12
' `	13

Taste	*Code*
Q q @	16
W w	17
E e	18
R r	19
T t	20
Z z	21
U u	22
I i	23
O o	24
P p	25
Ü ü	26
* + ~	27
A a	30
S s	31
D d	32
F f	33
G g	34
H h	35
J j	36
K k	37
L l	38
Ö ö	39
Ä ä	40
# '	41

Taste	*Code*
< > |	43
Y y	44
X x	45
C c	46
V v	47
B b	48
N n	49
M m µ	50
, ;	51
. :	52
– _	53
Leertaste	57
Tab	15
Caps Lk	58
L Shift	42
STRG	29
ALT (GR)	56
R Shift	54
Backsp.	14
Enter	28

Taste	*Code*
Einfg 0	82
Entf ,	83
Pos1 7	71
End 1	79
Bild ↑ 9	73
Bild ↓ 3	81
↑ 8	72
↓ 2	80
← 4	75
→ 6	77
5 (Num)	76
– (Num)	74
+ (Num)	78
÷ (Num)	53
× (Num)	55
* Druck	55
Num Lk	69
Scroll Lk	70

(Num) = Tasten auf dem Nummernblock

ASCII-Codes der verschiedenen Tastenkombinationen

Diese Codes werden zurückgegeben, wenn man mit INKEY$ eine Taste einliest. Die kursiven Zahlen bezeichnen sogenannte „erweiterte Codes“, die als zweistelliger String (erstes Zeichen = CHR$(0)) zurückgeliefert werden. Die grau schattierten Codes stehen für die zusätzlichen, grauen Tasten auf den modernen 101er Tastaturen.

Taste	*pur*	*Shift*	*Strg*	*Alt*
F1	*59*	*84*	*94*	*104*
F2	*60*	*85*	*95*	*105*
F3	*61*	*86*	*96*	*106*
F4	*62*	*87*	*97*	*107*
F5	*63*	*88*	*98*	*108*
F6	*64*	*89*	*99*	*109*
F7	*65*	*90*	*100*	*110*
F8	*66*	*91*	*101*	*111*
F9	*67*	*92*	*102*	*112*
F10	*68*	*93*	*103*	*113*
F11	*133*	*135*	*137*	*140*
F12	*134*	*136*	*138*	*141*
A	97	65	1	*30*
B	98	66	2	*48*
C	99	67	3	*46*
D	100	68	4	*32*
E	101	69	5	*18*
F	102	70	6	*33*
G	103	71	7	*34*
H	104	72	8	*35*
I	105	73	9	*23*
J	106	74	10	*36*
K	107	75	11	*37*
L	108	76	12	*38*
M	109	77	13	*50*
N	110	78	14	*49*
O	111	79	15	*24*
P	112	80	16	*25*
Q	113	81	17	*16*
R	114	82	18	*19*
S	115	83	19	*31*
T	116	84	20	*20*
U	117	85	21	*22*

Taste	*pur*	*Shift*	*Strg*	*Alt*
V	118	86	22	*47*
W	119	87	23	*17*
X	120	88	24	*45*
Y	121	89	25	*21*
Z	122	90	26	*44*
ESC	27	27	27	-
Tab	9	*15*	*148*	*165*
Enter	13	13	10	*28*
Leert.	32	32	32	*32*
Backsp.	8	8	*127*	*14*
Pos1	*71*	55	*119*	-
↑	*72*	56	*141*	-
Bild ↑	*73*	57	*132*	-
←	*75*	52	*115*	-
5	-	53	*143*	-
→	*77*	54	*116*	-
Ende	*79*	49	*117*	-
↓	*80*	50	*145*	-
Bild ↓	*81*	51	*118*	-
Einfg.	*82*	48	*146*	-
Entf.	*83*	44	*147*	-
Pos1	*71*	*71*	*119*	*151*
↑	*72*	*72*	*141*	*152*
Bild ↑	*73*	*73*	*132*	*153*
←	*75*	*75*	*115*	*155*
→	*77*	*77*	*116*	*156*
Ende	*79*	*79*	*117*	*59*
↓	*80*	*80*	*145*	*160*
Bild ↓	*81*	*81*	*118*	*161*
Einfg.	*82*	*82*	*146*	*162*
Entf.	*83*	*83*	*147*	*163*
Enter	13	13	10	*166*
Druck	-	-	*114*	-

Taste	*pur*	*Shift*	*Strg*	*Alt*
+	43	43	*144*	*78*
–	45	45	*142*	*74*
×	42	42	*150*	*55*
÷	47	47	*149*	*164*
^ °	94	248	-	*41*
1 !	49	33	-	*120*
2 "	50	34	*3*	*121*
3 §	51	21	-	*122*
4 $	52	36	-	*123*
5 %	53	37	-	*124*
6 &	54	38	30	*125*
7 /	55	47	-	*126*
8 (	56	40	-	*127*
9)	57	41	-	*128*
0 =	48	61	-	*129*
ß ?	225	63	28	-
' `	39	96	-	-

Taste	*pur*	*Shift*	*Strg*	*Alt*
< >	60	62	-	-
* +	43	42	29	*27*
Ü	129	154	27	*26*
Ö	148	153	-	*39*
Ä	132	142	-	*40*
# '	35	39	-	*43*
, ;	44	59	-	*51*
. :	46	58	-	*52*
– _	45	95	31	*130*

Die Codes der Spalten „pur“ und „Shift“ sind für die Tasten auf dem Nummernblock vertauscht, wenn NumLock eingeschaltet ist.

Beachten Sie, daß bei den *KeyDown*- und *KeyUp*-Ereignissen in VBDOS hiervon abweichende Codes erzeugt werden; Konstanten, die diese Codes beschreiben, finden Sie in der Datei CONSTANT.BI.

Standard-ASCII-Codes

Auf den nächsten Seiten folgt eine Tabelle aller ASCII-Zeichen. Sie können diese Zeichen mit der Tastatur erzeugen, indem Sie die ALT-Taste gedrückt halten, die Nummer auf dem Nummernblock eintippen und die ALT-Taste dann erst wieder loslassen. In BASIC kann die Funktion CHR$ benutzt werden, um beliebige ASCII-Zeichen zu erzeugen. Die Codes *32 bis einschließlich 126* können überall problemlos verwendet werden. Lediglich ein auf „Deutscher Zeichensatz“ eingestellter Drucker gibt statt [\]{|}~ die Zeichen ÄÖÜäöüß aus; das kann behoben werden, indem man den Drucker auf „Internationaler Zeichensatz“ umstellt. Die meisten Drucker können inzwischen Umlaute als Umlaute ausgeben und produzieren keine kursiven Buchstaben bei dem Versuch (schönen Gruß an Epson). Die Codes *0 bis 31* und *127* sind sogenannte Steuerzeichen, die nicht ohne weiteres auf dem Bildschirm und dem Drucker ausgegeben werden können. Auf dem Bildschirm ausgegeben, bedeutet zum Beispiel CHR$(12) das Löschen des Bildschirms, und CHR$(27) leitet einen ANSI-Befehl ein (die ANSI-Befehle finden Sie in Ihrem DOS-Handbuch; sie werden fast alle durch BASIC-Befehle ersetzt). Siehe dazu die Bemerkung zu OPEN im Disketten-Referenzteil.

Auf dem Drucker ausgegeben, ist CHR$(12) meist ein Seitenvorschub, und mit CHR$(27) beginnt eine sogenannte Escape-Sequenz, ein Steuerbefehl an den Drucker.

Die Codes *128 bis 255* zeigt jeder Bildschirm im Textmodus korrekt an; im Grafikmodus einer CGA-Karte kann es Schwierigkeiten geben, wenn das speicherresidente Programm GRAFTABL nicht geladen ist.

Bei Druckern kommt es darauf an, wie sie eingestellt sind. Viele Matrixdrucker geben doppelte Linien (zum Beispiel Zeichen Nr. 186) als einfache Linien aus. Bei den meisten Druckern lassen sich verschiedene Zeichensätze wählen, die nicht immer die Standard-ASCII-Zeichen enthalten. Wenn Sie einen Hewlett Packard- (oder kompatiblen) Laser- oder Tintenstrahldrucker mit diesem Zeichensatz verwenden wollen, stellen Sie den Zeichensatz „PC-8“ ein (`LPRINT CHR$(27); "(10U"`), um diese Zeichen so drucken zu können, wie sie hier in der Tabelle stehen.

Code	*Zeichen*	*Code*	*Zeichen*	*Code*	*Zeichen*	*Code*	*Zeichen*
0		42	*	84	T	126	~
1		43	+	85	U	127	⌂
2		44	,	86	V	128	Ç
3	♥	45	-	87	W	129	ü
4	♦	46	.	88	X	131	â
5	♣	47	/	89	Y	132	ä
6	♠	48	0	90	Z	133	à
7	●	49	1	91	[	134	å
8	◘	50	2	92	\	135	ç
9	<ht>	51	3	93	]	136	ê
10	<lf>	52	4	94	^	137	ë
11	<vt>	53	5	95	_	138	è
12	<ff>	54	6	96	`	139	ï
13	<cr>	55	7	97	a	140	î
14	<so>	56	8	98	b	141	ì
15	<si>	57	9	99	c	142	Ä
16		58	:	100	d	143	Å
17		59	;	101	e	144	É
18		60	<	102	f	145	æ
19		61	=	103	g	146	Æ
20	¶	62	>	104	h	147	ô
21	§	63	?	105	i	148	ö
22		64	@	106	j	149	ò
23		65	A	107	k	150	û
24	↑	66	B	108	l	151	ù
25	↓	67	C	109	m	152	ÿ
26	→	68	D	110	n	153	Ö
27	←	69	E	111	o	154	Ü
28		70	F	112	p	155	¢
29		71	G	113	q	156	£
30		72	H	114	r	157	¥
31	<us>	73	I	115	s	158	₧
32	Space	74	J	116	t	159	*f*
33	!	75	K	117	u	160	á
34	"	76	L	118	v	161	í
35	#	77	M	119	w	162	ó
36	$	78	N	120	x	163	ú
37	%	79	O	121	y	164	ñ
38	&	80	P	122	z	165	Ñ
39	'	81	Q	123	{	166	ª
40	(	82	R	124	|	167	º
41	)	83	S	125	}	168	¿

Code	Zeichen
169	⌐
170	¬
171	½
172	¼
173	¡
174	«
175	»
176	°
177	±
178	²
179	³
180	´
181	µ
182	¶
183	·
184	¸
185	¹
186	º
187	»
188	¼
189	½
190	¾
191	¿
192	À
193	Á
194	Â
195	Ã
196	Ä
197	Å
198	Æ
199	Ç
200	È
201	É
202	Ê
203	Ë
204	Ì
205	Í
206	Î
207	Ï
208	Ð
209	Ñ
210	Ò

Code	Zeichen
211	Ó
212	Ô
213	Õ
214	Ö
215	×
216	Ø
217	Ù
218	Ú
219	Û
220	Ü
221	Ý
222	Þ
223	ß
224	α
225	β
226	Γ
227	π
228	Σ
229	σ
230	µ
231	τ
232	Φ
233	Θ
234	Ω
235	δ
236	∞
237	∅
238	∈
239	∩
240	≡
241	±
242	≥
243	≤
244	⌠
245	⌡
246	÷
247	≈
248	°
249	•
250	·
251	√
252	^{n}

Code	Zeichen
253	²
254	þ
255	

ISAM-Sortiertabellen

Tabelle für Deutsch, Französisch, Englisch, Italienisch und Portugiesisch	Tabelle für Finnisch, Isländisch, Dänisch, Norwegisch und Schwedisch
A, Ä, Å, a, à, á, â, ä, å	A, a, à, á, â
Æ und æ werden wie ae einsortiert	B, b
B, b	C, Ç, c, ç
C, Ç, c, ç	D, d
D, d	E, É, e, è, é, ê, ë
E, É, e, è, é, ê, ë	F, f
F, f	G, g
G, g	H, h
H, h	I, i, ì, í, î
I, i, ì, í, î	J, j
J, j	K, k
K, k	L, l
L, l	M, m
M, m	N, Ñ, n, ñ
N, Ñ, n, ñ	O, o, ò, ó, ô
O, Ö, o, ò, ó, ô, ö	P, p
P, p	Q, q
Q, q	R, r
R, r	S, s
S, s	ß wird wie ss einsortiert
ß wird wie ss einsortiert	T, t
T, t	U, Ü, u, ù, ú, û, ü
U, Ü, u, ù, ú, û, ü	V, v, W, w
V, v	X, x
W, w	Y, y, ÿ
X, x	Z, z
Y, y, ÿ	Æ, æ
Z, z	Ý
	Å, å
	Ä, ä
	Ö, ö

Tabelle für Spanisch	Tabelle für Holländisch
A, Ä, Å, a, à, á, â, ä, å Æ und æ werden wie ae einsortiert B, b C, Ç, c, ç Ch, ch (werden als ein Zeichen zwischen C und D einsortiert) D, d E, É, e, è, é, ê, ë F, f G, g H, h I, i, ì, í, î J, j K, k L, l LL, ll (werden als ein Zeichen zwischen L und M einsortiert) M, m N, n Ñ, ñ O, Ö, o, ò, ó, ô, ö P, p Q, q R, r S, s ß wird wie ss einsortiert T, t U, Ü, u, ù, ú, û, ü V, v W, w X, x Y, y, ÿ Z, z	Entspricht der Tabelle für Deutsch, Französisch, Englisch, Italienisch und Portugiesisch, mit dem einzigen Unterschied, daß ÿ nicht nach y, sondern wie ij einsortiert wird.

Referenzteil Standard-BASIC Anhang E

Übersicht und zusätzliche Online-Hilfe

In diesem Anhang ist der Inhalt der "zusätzlichen Online-Hilfe" abgedruckt, die auf der Begleitdiskette zum Buch enthalten ist. Sie bietet zu sämtlichen Standard-BASIC-Befehlen Informationen, die häufig über das hinausgehen, was die VBDOS-Onlinehilfe anbietet. Die zusätzliche Onlinehilfe besteht aus den beiden Dateien *referenz.hlp* und *referenz.bat*. Die Installation der zusätzlichen Onlinehilfe ist im Vorwort dieses Buches beschrieben. Nach der Installation können Sie in der VBDOS-Entwicklungsumgebung zu jedem BASIC-Befehl schnell eine zusätzliche deutschsprachige Hilfe abrufen. Tippen Sie dazu einfach ein "r" gefolgt vom Befehlsnamen ein und drücken F1, zum Beispiel: `rOPEN <F1>`. Die Befehlsübersicht erreichen Sie immer mit ramm `<F1>`.

Informationen über die Objekte, Methoden und Eigenschaften sowie über ISAM und die Toolboxen finden sich in den Kapiteln 25 bis 27 des Buches.

Es folgt eine alphabetische Liste aller in diesem Anhang aufgeführten Standard-BASIC-Befehle und -Funktionen.

SADD (Funktion)
SCREEN (Befehl)
SEEK (Funktion)
SELECT CASE (Befehl)
SGN (Funktion)
SHELL (Befehl)
SLEEP (Befehl)
SPACE$ (Funktion)
SQR (Funktion)
SSEGADD (Funktion)
$STATIC (Metabefehl)
STICK (Funktion)
STR$ (Funktion)
STRIG (Befehl)
SUB/FUNCTION (Befehl)
SYSTEM (Befehl)
TAN (Funktion)
TIMER (Funktion)
TIMESERIAL (Funktion)
TRON, TROFF (Befehle)
UBOUND (Funktion)
UEVENT (Befehle)
VARPTR (Funktion)
VARSEG (Funktion)
VIEW PRINT (Befehl)
WEEKDAY (Funktion)
WIDTH (Befehl)
WRITE (Befehl)
SCREEN (Funktion)
SECOND (Funktion)
SEEK (Befehl)
SETMEM (Funktion)
SHARED (Befehl)
SIN (Funktion)
SOUND (Befehl)
SPC (Funktion)
SSEG (Funktion)
STACK (Funktion und Befehl)
STATIC (Befehl)
STOP (Befehl)
STRIG (Funktion)
STRING$ (Funktion)
SWAP (Befehl)
TAB (Funktion)
TIME$ (Systemvariable)
TIMER (Befehle)
TIMEVALUE (Funktion)
TYPE (Befehl)
UCASE$ (Funktion)
VAL (Funktion)
VARPTR$ (Funktion)
VIEW (Befehl)
WAIT (Befehl)
WHILE...WEND (Befehl)
WINDOW (Befehl)
YEAR (Funktion)

Struktur der Referenz-Einträge

Anwendung Hier wird angegeben, wie der Befehl oder die Funktion im Programm eingesetzt werden. Die kleingeschriebenen Variablennamen tauchen - ebenfalls kleingeschrieben - oft in der Beschreibung wieder auf.

Nutzen Hier wird kurz beschrieben, wozu der Befehl oder die Funktion eingesetzt wird. Wenn nötig, finden sich Tabellen für die einzelnen Parameter.

Bemerkung Hier finden Sie einige Tips zur Verwendung der Funktion oder des Befehls, oftmals mit Abgrenzungen, wann besser eine andere Methode eingesetzt werden sollte.

Kompatibel In diesem Abschnitt ist angegeben, ob der Befehl oder die Funktion mit VB für WINDOWS und dem BASIC PDS kompatibel ist und ob er/sie eingesetzt werden darf, während Formen angezeigt werden (+ = ja, - = nein, * = beschränkt). In der Rubrik Mathe ist angegeben, ob dieser Befehl oder

diese Funktion die Mathematik-Libraries benötigt (Programme, die mindestens einen solchen Befehl enthalten, sind etwa 10 KB länger als Programme ohne Fließkommarithmetik, deshalb werden Sie vielleicht bei kleinen Programmen darauf achten - vgl. Kapitel 15 und 23 im Buch.)

Siehe auch Hier finden Sie Querverweise auf andere relevante Befehle und Funktionen. Die betreffenden Seiten im Buch sind in Klammern angegeben. Wenn Sie in der zusätzlichen Online-Hilfe Querverweise auf die Original-Hilfstexte wünschen, klicken Sie auf MS-Hilfe hierzu rechts oben in der Ecke jeder Hilfeseite.

ABS (Funktion)

Anwendung `x = ABS(y)`

Nutzen Ermittelt den Betrag seines Arguments. `ABS(y)` ist äquivalent zu `y * SGN(y)`.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch SGN (672).

ASC (Funktion)

Anwendung `x = ASC(y$)`

Nutzen Gibt den ASCII-Wert (0 bis 255) des ersten Zeichens des String-Arguments zurück. *Funktionsaufruf unzulässig* bei Leerstring. Eine Tabelle aller ASCII-Zeichen finden Sie im Anhang D.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch CHR$ (575).

ATN (Funktion)

Anwendung `x = ATN(y)`

Nutzen Gibt den Arkustangens seines Argumentes im Bogenmaß zurück. Multiplizieren Sie mit 57.2957795130824, um Grad zu erhalten.

Ist y eine INTEGER- oder SINGLE-Zahl, wird das Ergebnis mit einfacher Genauigkeit berechnet, sonst mit doppelter.

Kompatibel PDS + VBWIN + Formen + Mathe +

Siehe auch SIN (673), COS (583), TAN (682).

BEEP (Befehl)

Anwendung BEEP

Nutzen Erzeugt einen Warnton, dessen Dauer und Frequenz von Computer zu Computer unterschiedlich sind. BEEP ist äquivalent zu PRINT CHR$(7).

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch SOUND (674), PLAY (647).

BLOAD (Befehl)

Anwendung BLOAD *dateiname$* [, *offset*]

Nutzen Lädt eine Datei an eine bestimmte Stelle im Speicher. *dateiname$* ist der Name der Datei, die geladen wird; wird kein *offset* angegeben, dann wird die Datei an dieselbe Stelle geladen, von der sie mit BSAVE gespeichert wurde. Gibt man *offset* an, dann wird die Datei in das zuletzt mit DEF SEG definierte Segment ab der angegebenen Offset-Adresse geladen. Wenn *offset* eine negative Zahl ist, addiert BASIC 65.536.

Bemerkung • Vorsicht bei der Anwendung: BASIC kann Variablen im Speicher verschieben.

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch BSAVE (571).

BSAVE (Befehl)

Anwendung BSAVE *dateiname$*, *offset*, *laenge*

Nutzen Kopiert einen bestimmten Speicherbereich in eine Datei. *dateiname$* ist der Name der Datei, *offset* ist die Stelle im aktuellen (durch DEF SEG gesetzten) Segment, ab der gespeichert werden soll, und *laenge* gibt an, wieviele Bytes kopiert werden sollen (maximal 65.536).

BLOAD und BSAVE werden hauptsächlich angewandt, um ganze Arrays schnell zu laden oder zu speichern. Insbesondere sind sie – zusammen mit den GET- und PUT-Befehlen für Grafik – von Nutzen, wenn man grafische Symbole speichern und abrufen will.

Bemerkung
- VBDOS speichert Arrays unter Umständen im Expanded Memory, wenn sie kleiner als 16 KB sind. Solche im Expanded Memory befindlichen Arrays können nicht mit BSAVE und BLOAD gespeichert und geladen werden. Im Zweifelsfalle starten Sie VBDOS ohne den /Ea-Switch, dann werden keine Arrays in das Expanded Memory ausgelagert.

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch BLOAD (571), DEF SEG (589), VARPTR (688), VARSEG (689).

CALL (Befehl)

Anwendung
(1) CALL prozedurname [(*argumente*)]
(2) prozedurname [*argumente*]

Nutzen Ruft eine mittels SUB...END SUB in BASIC programmierte Prozedur oder eine Prozedur einer anderen Programmiersprache auf und übergibt ihr die angegebenen *argumente*, das sind Variablen, Konstanten oder Ausdrücke, die durch Kommata getrennt sein müssen.

Die Verwendung der Syntax (2) erfordert, daß die aufgerufenen Prozeduren zuvor mit DECLARE-Anweisungen deklariert wurden.

argumente ist eine Liste von durch Kommata voneinander getrennten Variablennamen oder Ausdrücken. Diese können mit dem Schlüsselwort BYVAL oder SEG eingeleitet werden. BYVAL sorgt dafür, daß anstatt der Adresse der Variablen nur ihr Wert übergeben wird (siehe Bemerkung). SEG ist nur beim Aufruf von Routinen brauchbar, die in anderen Sprachen geschrieben sind, und sorgt dafür, daß die Adresse der Variablen auf jeden Fall als Far-Adresse übergeben wird. Beim Aufruf von Prozeduren, die in anderen Sprachen geschrieben sind, kann auch statt CALL der Befehl CALLS gewählt werden, der *alle* Adressen als Far-Adressen übergibt, so, als wäre vor jede Variable SEG geschrieben worden. CALLS und SEG können nicht benutzt werden, um ganze Arrays zu übergeben.

Als Argumente zu Prozeduren können ganze Arrays übergeben werden, indem man den Arraynamen gefolgt von offener und geschlossener runder Klammer angibt. Einzelne Array-Elemente können wie normale Variablen übergeben werden. Nicht möglich ist es jedoch, aus einem mehrdimensionalen Array zum Beispiel eine einzelne Spalte o.*. zu übergeben.

Bemerkung

• Variablen werden üblicherweise als Variablenparameter (*by reference*) und nicht als Werteparameter (*by value*) übergeben. Das heißt, daß Änderungen, die die Prozedur an der Variable vornimmt, auch nach Beendigung der Prozedur wirksam bleiben.

Ausnahmen von dieser Regel sind die Übergabe von Konstanten und von Ausdrücken. Als Ausdrücke versteht man Verknüpfungen wie `3 + TotalNumber` oder Funktionsaufrufe wie `LTRIM$(a$)`. Werden diese einer Funktion übergeben, so muß zunächst eine temporäre Variable erzeugt werden, die nach dem Ende der Prozedur nicht mehr gebraucht wird. Um auch bei normalen Variablen zu erzwingen, daß zunächst eine temporäre Variable als Kopie angelegt wird und die Prozedur dann nur die Kopie manipulieren kann, kann man die Variable in runde Klammern einschließen – das macht sie zu einem Ausdruck. Alternativ kann man bei der Prozedurdeklaration das Schlüsselwort BYVAL benutzen (siehe SUB...END SUB).

• Prozeduren und Funktionen können zwar keine Strings mit fester Länge als Parameter fordern, man kann aber einer Prozedur, die einen gewöhnlichen String (mit variabler Länge also) erwartet, auch stattdessen einen String von fester Länge übergeben. Falls die Prozedur an den String Zeichen anhängt, was ihr ja durchaus möglich ist, werden diese unter Umständen beim Zurückverwandeln abgeschnitten.

• Vorsicht beim Aufruf BASIC-fremder Prozeduren (beispielsweise C oder Assembler). BASIC verschiebt zuweilen Variablen im Speicher. Wenn irgendwo in der Argumentenliste eine Funktion aufgerufen wird (zum Beispiel CHR$ im Befehl CALL Alfred(Laenge, CHR$(Ende%)) können unter Umständen dadurch die Variablen, deren Adressen bereits ermittelt wurden, um sie bei CALL zu übergeben, noch verschoben werden. Dadurch würden falsche Adressen übergeben. Benutzen Sie nur Array-Variablen, Konstanten, arithmetische Ausdrücke oder gewöhnliche Variablen in CALL-Aufrufen an BASIC-fremde Prozeduren, niemals Funktionen.

• Eine Prozedur, die in Assembler programmiert und von BASIC aufgerufen wird, muß in Assembler als PUBLIC deklariert sein.

Beispiel Siehe Beispiel bei SUB/FUNCTION.

Kompatibel PDS + VBWIN * Formen + Mathe -

Siehe auch DECLARE (586), SUB/FUNCTION (680), ON *event* GOSUB (634).

CCUR (Funktion)

Anwendung x@ = CCUR(*y*)

Nutzen Wandelt einen beliebigen numerischen Wert in einen CURRENCY-Wert um. Der erlaubte Bereich ist –922.337.203.685.477,5808 bis 922.337.203.685.477,5807.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch CINT (576), CDBL (574), CLNG (577), CSNG (583).

CDBL (Funktion)

Anwendung x# = CDBL(*y*)

Nutzen Wandelt einen beliebigen numerischen Wert in einen DOUBLE-Wert um. Der erlaubte Bereich ist etwa –1,79E308 bis 1,79E308 (genauere Angaben siehe Anhang).

Kompatibel PDS + VBWIN + Formen + Mathe +

Siehe auch CINT (576), CCUR (574), CLNG (577), CSNG (583).

CHAIN (Befehl)

Anwendung CHAIN *dateiname$*

Nutzen Startet ein anderes Programm. Dabei werden geöffnete Dateien nicht geschlossen, und es besteht die Möglichkeit, durch COMMON-Befehle Variablen an das aufgerufene Programm zu übergeben. (Siehe auch Kapitel 10.)

Bemerkung • Die Übergabe von Variablen durch COMMON ist nur mit unbenannten COMMON-Blocks möglich, und nur, wenn man nicht mit Stand-Alone-Programmen, sondern mit Runtime-Modulen arbeitet.

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch RUN (661), COMMON (581).

CHDIR (Befehl)

Anwendung CHDIR *directoryName$*

Nutzen Der Befehl wechselt das aktuelle Directory, identisch mit dem DOS-Befehl CHDIR oder CD.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch CURDIR$ (583), CHDRIVE (575), MKDIR (629), RMDIR (660).

CHDRIVE (Befehl)

Anwendung CHDRIVE *laufwerk$*

Nutzen Wechselt das aktuelle Laufwerk. Nur der erste Buchstabe von *laufwerk$* ist relevant.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch CHDIR (575), CURDIR$ (583).

CHR$ (Funktion)

Anwendung *x$* = CHR$(*y*)

Nutzen Als Umkehrfunktion zu ASC liefert diese Funktion das ASCII-Zeichen zu einer gegebenen Zahl. Diese darf im Bereich von 0 bis 255 liegen. Eine Tabelle der ASCII-Zeichen finden Sie in Anhang D.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch ASC (570).

CINT (Funktion)

Anwendung *x%* = CINT(*y*)

Nutzen Wandelt einen beliebigen numerischen Wert in einen INTEGER-Wert um. Der erlaubte Bereich ist –32.768 bis 32.767.

Bemerkung • CINT, FIX und INT sind ähnliche Funktionen. Die Unterschiede sind im Abschnitt über INT zusammengefaßt.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch CCUR (574), CDBL (574), CLNG (577), CSNG (583), FIX (603), INT (616).

CIRCLE (Befehl)

Anwendung CIRCLE [STEP] (*x*, *y*), *radius*, [,[*farbe*] [,[*von*] [,[*bis*] [,*verhaelt*]]]]

Nutzen Zeichnet einen Kreis bzw. eine Ellipse oder ein Segment derselben auf den Bildschirm.

STEP gibt wie bei allen Grafikbefehlen an, daß die Koordinaten relativ vom momentanen Grafikcursor aus zu verstehen sind.

(*x, y*) sind die Koordinaten des Mittelpunkts. Auch für Ellipsen wird nur ein Mittelpunkt angegeben.

radius ist der Radius für den Kreis bzw. die Ellipse. Er wird in derselben Einheit gemessen wie die Koordinaten; diese ist gewöhnlich ein Pixel, kann aber mit dem WINDOW-Befehl geändert werden.

farbe ist das Farbattribut, das dem Kreis bzw. der Ellipse zugeordnet werden soll (siehe auch SCREEN, COLOR und PALETTE).

von, bis sind der Start- und der Endwinkel (im Bogenmaß). Bei Weglassen der Angaben werden 0 und 2 π eingesetzt, so daß ein Vollkreis (bzw. eine Vollellipse) entsteht. Alle Werte zwischen –2π und 2π sind erlaubt. Negative Werte haben zur Folge, daß der Radius als Linie zum Start- bzw. Endpunkt abgetragen wird. Wählt man für beide Winkelangaben negative Werte, entsteht ein Kreissegment, das leicht mit PAINT gefüllt werden kann.

verhaelt Das Seitenverhältnis zwischen y- und x-Radius. Normalerweise wird es automatisch auf einen Wert gesetzt, der einen Kreis entstehen läßt; dieser Wert errechnet sich nach der Formel `v = 4/3 * ypixel / xpixel`, wobei für *ypixel* und *xpixel* die Auflösung des verwendeten Grafikmodus eingetragen wird. Bei *verhaelt* < 1 ist *radius* der x-Radius, anderenfalls der y-Radius.

Kompatibel PDS + VBWIN * Formen - Mathe +

Siehe auch PAINT (642), SCREEN (663), COLOR (579), PALETTE (643), VIEW (689), WINDOW (692), LINE (623).

CLEAR (Befehl)

Anwendung `CLEAR [,, stack]`

Nutzen CLEAR schließt alle Dateien, setzt alle Variablen auf 0 bzw. Leerstrings, setzt im Grafikmodus den Grafik-Viewport wieder auf die volle Bildschirmgröße und kann auch benutzt werden, um die Stack-Größe zu setzen. Dazu wird als *stack* die gewünschte Größe des Stacks angegeben.

CLEAR führt die Funktion SETMEM so aus, daß wieder aller verfügbarer Far-Speicher für BASIC reserviert ist.

Bemerkung • CLEAR wird automatisch bei einem RUN-Befehl aufgeführt, und die Stack-Manipulationen können in VBDOS eleganter mit dem STACK-Befehl und der gleichnamigen Funktion erledigt werden.

• CLEAR darf keinesfalls innerhalb von SUBs oder FUNCTIONS (*Funktionsaufruf unzulässig*) und GOSUB-Routinen (kein RETURN mehr möglich) verwendet werden.

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch STACK (676), SETMEM (671), ERASE (597).

CLNG (Funktion)

Anwendung `x& = CLNG(y)`

Nutzen Wandelt einen beliebigen numerischen Wert in einen LONG-Wert um. Der erlaubte Bereich ist –2.147.483.648 bis 2.147.483.647.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch CINT (576), CDBL (574), CCUR (574), CSNG (583).

CLOSE (Befehl)

Anwendung CLOSE [[#]*dateinummer* [,[#]*dateinummer*]] ...

Nutzen Schließt Dateien. Ohne Argumente benutzt, werden alle Dateien geschlossen, ansonsten nur die angegebenen.

Bemerkung • CLEAR, END, RESET, RUN und SYSTEM schließen ebenfalls alle Dateien.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch OPEN (632), RESET (657).

CLS (Befehl)

Anwendung CLS [0|1|2]

Nutzen Löscht den Bildschirm oder Teile davon. Die Wirkung des CLS-Befehls hängt davon ab, welche Zahl man angibt, in welchem SCREEN-Modus der Rechner sich befindet (Grafik oder Text) und ob ein Grafik- oder Text-Viewport (mit VIEW bzw. VIEW PRINT) definiert ist. Außerdem ist relevant, ob mit KEY ON die Funktionstasten-Anzeige eingeschaltet wurde oder nicht.

CLS im Grafikmodus ohne Grafik-Viewport: Ganzer Bildschirm wird gelöscht; Funktionstastenanzeige wird, falls eingeschaltet, regeneriert; Cursor wird auf Zeile 1, Spalte 1 gesetzt.

CLS im Grafikmodus mit Grafik-Viewport: Definierter Grafik-Viewport wird gelöscht.

CLS im Textmodus (SCREEN 0): Der definierte Text-Viewport wird gelöscht (wenn kein VIEW PRINT aktiv, von Zeile 1 bis zur vorletzten); Funktionstastenanzeige wird, falls eingeschaltet, regeneriert; Cursor wird auf Zeile 1, Spalte 1 gesetzt.

CLS 0 löscht in jedem Falle den ganzen Bildschirm bis auf die unterste Zeile, regeneriert die Funktionstastenanzeige, wenn sie eingeschaltet ist, und setzt den Cursor auf Zeile 1, Spalte 1.

CLS 1 funktioniert genauso wie CLS ohne Argument, mit dem Unterschied, daß ein eventueller Text-Viewport unberührt bleibt, so daß im Textmodus, der nur einen Text-Viewport besitzt, überhaupt keine Wirkung sichtbar ist.

CLS 2 Löscht nur den Text-Viewport, läßt in jedem Falle die unterste Zeile unberührt und setzt den Cursor auf Zeile 1, Spalte 1.

Kompatibel PDS + VBWIN - Formen - Mathe -

Siehe auch VIEW (689), VIEW PRINT (651), KEY ON (618), SCREEN (663), WIDTH (691).

COLOR (Befehl)

Anwendung (1) COLOR *vordergrund, hintergrund, rahmen*
(2) COLOR *hintergrund, palette*
(3) COLOR *vordergrund*
(4) COLOR vordergrund, hintergrund

Nutzen Mit dem COLOR-Befehl können Farben ausgewählt werden. Welche Syntax zu verwenden ist und welche Möglichkeiten damit zur Verfügung stehen, hängt vom aktuellen SCREEN-Modus ab: Syntax (1) für Modus 0; Syntax (2) für Modus 1; Syntax (3) für Modi 4, 12 und 13; Syntax (4) für Modi 7, 8, 9 und 10.

In den Modi 2, 3 und 11 verursacht der COLOR-Befehl einen *Funktionsaufruf unzulässig*-Fehler.

vordergrund ist die Text- und im Grafikmodus zugleich auch die Grafikfarbe. *hintergrund* ist im Textmodus die Hintergrundfarbe für die von nun an auszugebenden Zeichen, während *hintergrund* im Grafikmodus sofort dem gesamten Bildschirmhintergrund eine Farbe zuordnet. *palette* im SCREEN-Modus 1 wählt eine der zwei CGA-Paletten (0 = grün/rot/braun, 1 = hellblau/violett/weiß) aus. *rahmen* hat nur im Textmodus und auch nur bei der CGA-Grafikkarte und der EGA-Karte, wenn an sie ein CGA-Bildschirm angeschlossen ist, Wirkung; damit kann die Farbe des Bildschirms außerhalb des beschreibbaren Bereichs festgelegt werden.

Bemerkung • Mit COLOR stellen Sie genaugenommen keine Farbe, sondern nur ein Farbattribut ein. Welche Farbe zu diesem Farbattribut gehört, kann bei EGA-, MCGA- und VGA-Karten mit dem PALETTE-Befehl festgelegt werden.

• Im Textmodus sind für hintergrund die Werte 0-7, für vordergrund 0-31 zulässig; Werte über 15 erzeugen die gleiche Farbe wie der um 16 kleinere Wert, allerdings blinkend.

Kompatibel PDS + VBWIN - Formen - Mathe -

Siehe auch PALETTE (643), SCREEN (663).

COM (Befehle)

Anwendung
```
COM (nummer) ON
COM (nummer) OFF
COM (nummer) STOP
```

Nutzen Diese drei Befehle dienen zur Überwachung des Trappings für die Kommunikationsschnittstellen. *nummer* ist die Nummer der COM-Schnittstelle; nur 1 oder 2 sind erlaubt. Ist ein COM ON-Befehl aktiv, dann wird, sobald am betreffenden Port ein Zeichen ankommt, in die mit ON COM(*nummer*) GOSUB angegebene Routine verzweigt. Diese Verzweigung kann mit COM STOP suspendiert oder mit COM OFF völlig abgestellt werden. Solange COM STOP wirksam ist, werden alle auftretenden Aufrufe unmittelbar nach dem nächsten COM ON ausgeführt.

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch EVENT ON/OFF (600), ON *event* GOSUB (634).

COMMAND$ (Funktion)

Anwendung
```
x$ = COMMAND$
```

Nutzen Diese Funktion gibt als Funktionswert die Zeichenkette zurück, die hinter dem Programmnamen beim Aufruf von DOS aus angegeben wurde (Parameterzeile oder *command line*). Alle Leerzeichen am Anfang werden gestrichen, und der String wird in Großbuchstaben umgewandelt (unter Vernachlässigung von Umlauten).

Bemerkung • Um Programme, die mit COMMAND$ arbeiten, in VBDOS laufen zu lassen, kann der Wert für COMMAND$ entweder mit der VBDOS-Option /CMD oder mit »COMMAND$ ändern« aus dem »Ausführen«-Menü gesetzt werden.

• Unter Verwendung des Interrupt 21h läßt sich auf die unverfälschte Befehlszeile zugreifen (vgl. Programm DETCMD.BAS auf der beiliegenden Diskette).

COMMON (Befehl)

Anwendung COMMON [SHARED] [/*blockname*/] *variablenliste*

Nutzen Der COMMON-Befehl definiert Variablen zur gemeinsamen Nutzung in mehreren Modulen. Das ist entweder notwendig, wenn man mit CHAIN ein anderes Programm aufruft und diesem Variablen übergeben möchte, oder wenn verschiedene Module als ein großes Programm gemeinsame Variablen haben sollen. Letzteres kann auch ausgedehnt werden auf Libraries und Quick Libraries; in den BI-Files zu den Toolboxen zum Beispiel befinden sich zahlreiche COMMON-Befehle, damit die Source-Routinen auf Variablen aus den Quick Libraries zugreifen können.

Gibt man einen (in zwei Schrägstriche eingeschlossenen) *blocknamen* an, dann werden die genannten Variablen nicht mit allen Programmen geteilt, sondern nur mit denen, die einen gleichnamigen COMMON-Befehl besitzen. Solche benannten COMMON-Blocks werden jedoch bei CHAIN nicht berücksichtigt.

Der Zusatz SHARED bedeutet, daß nicht nur der Modulcode anderer Programme, sondern auch jeglicher Prozedurcode diese Variablen benutzen darf (globale Variablen), und ist insofern weitgehend identisch mit SHARED-Befehlen in Prozeduren oder dem SHARED-Zusatz bei DIM.

In der *variablenliste* sind beliebig viele Variablen aufgeführt, entweder mit Typenbezeichner (zum Beispiel *Parameter$*) oder mit einer AS-Bezeichnung (*Parameter AS STRING*). Arrays, egal welcher Dimension, werden einfach durch eine geöffnete und eine geschlossene runde Klammer gekennzeichnet; die exakte Dimension muß in einem DIM-Befehl festgelegt werden.

Für einfache Variablen oder Variablen eines benutzerdefinierten Typs reicht eine Definition in COMMON aus, sie müssen (und dürfen) danach nicht mehr mit einem DIM-Befehl vereinbart werden.

Bemerkung • Bei Verwendung von COMMON muß penibel darauf geachtet werden, daß die Reihenfolge und die Datentypen in allen COMMON-Zeilen der verschiedenen Programme übereinstimmen, da der Compiler hier nicht in dem Maße Fehler aufdecken kann, wie das bei SUB- und FUNCTION-Aufrufen möglich ist. Nicht übereinstimmende COMMON-Blocks führen leicht zu schweren Fehlern. Benannte COMMON-Blocks dürfen problemlos vertauscht werden.

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch DIM (590), SHARED (672), CHAIN (574).

CONST (Befehl)

Anwendung CONST *name* = *ausdruck* [, *name* = *ausdruck*...]

Nutzen Vereinbart symbolische Konstanten. Diese Konstanten werden im Programm wie Variablen angewandt, deren Wert jedoch nicht verändert werden kann.

Bei der Konstantendefinition mit CONST erhält die Konstante automatisch den kleinstmöglichen Typ, der in der Lage ist, den angegebenen Ausdruck zu repräsentieren. Diese Standard-Typvergabe kann jedoch verhindert werden, indem man hinter *name* einen Typenbezeichner (wie %, &, ! etc.) setzt; dann hat die Konstante den angegebenen Typ. Im Programmtext kann die Konstante in jedem Falle ohne Typenbezeichner verwendet werden.

In dem *ausdruck*, der der Konstanten zugewiesen wird, dürfen (bei numerischen Konstanten) nur Zahlen, arithmetische und logische Operatoren (bis auf den Potenz-Operator ^) sowie andere, bereits vorher vereinbarte Konstanten auftreten. Bei String-Konstanten ist die Verknüpfung mit + nicht erlaubt. Auch eingebaute Funktionen (zum Beispiel CHR$) dürfen nicht in Konstantendefinitionen verwendet werden.

Kompatibel PDS + VBWIN * Formen + Mathe -

Bemerkung • Konstanten, die im Hauptprogramm definiert werden, sind global für das ganze Programm (nicht jedoch für andere, gleichzeitig geladene Programme in VBDOS oder für dazugelinkte Module außerhalb VBDOS). Konstanten, die in Prozeduren definiert werden, sind lokal für die betreffenden Prozeduren.

• Mit VBDOS wird eine Datei namens CONSTANT.BI geliefert, die viele oft benötigte Werte als Konstanten definiert.

COS (Funktion)

Anwendung *x* = COS(*y*)

Nutzen Gibt den Cosinus seines Argumentes zurück. Ist *y* INTEGER oder SINGLE, wird COS mit einfacher Genauigkeit berechnet, sonst mit doppelter.

Der Winkel *y* wird im Bogenmaß angegeben (ein Grad-Winkel muß zunächst mit 0,0174532925199433 multipliziert werden).

Kompatibel PDS + VBWIN + Formen + Mathe +

Siehe auch ATN (570), SIN (673), TAN (682).

CSNG (Funktion)

Anwendung *x!* = CSNG(*y*)

Nutzen Wandelt einen beliebigen numerischen Wert in einen SINGLE-Wert um. Der erlaubte Bereich ist etwa –3,4E38 bis 3,4E38 (genauere Angaben siehe Anhang A).

Kompatibel PDS + VBWIN + Formen + Mathe +

Siehe auch CINT (576), CDBL (574), CLNG (577), CCUR (574).

CSRLIN (Funktion)

Anwendung *x* = CSRLIN

Nutzen Gibt zurück, in welcher Bildschirmzeile sich der Textcursor gerade befindet.

Kompatibel PDS + VBWIN - Formen - Mathe -

In VBWIN und im Zusammenhang mit Formen verwenden Sie die *CurrentY*-Eigenschaft.

Siehe auch VIEW PRINT (651), POS (650), LOCATE (625).

CURDIR$ (Funktion)

Anwendung *x$* = CURDIR$ [(*laufwerk$*)]

Nutzen Gibt das aktuelle Verzeichnis zum aktuellen Laufwerk oder, wenn *laufwerk$* angegeben wurde, zu diesem Laufwerk zurück. Nur das erste Zeichen von *laufwerk$* ist relevant.

Da das erste Zeichen der Ergebnis-Zeichenkette immer das betreffende Laufwerk ist, kann CURDIR$ (ohne Argument) auch dazu benutzt werden, das aktuelle Laufwerk festzustellen.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch CHDIR (575), CHDRIVE (575).

CVx (Funktionen)

Anwendung
```
x% = CVI(y$)
x& = CVL(y$) (seit QB 4.0)
x! = CVS(y$)
x# = CVD(y$)
x@ = CVC(y$) (seit PDS 7.0)
```

Nutzen Diese Funktionen dienen dazu, codierte Zahlen zu entschlüsseln. Wenn BASIC Zahlen in Random-Access-Dateien schreibt (ob mit FIELD oder innerhalb eines TYPE-Records), werden sie so verschlüsselt (INTEGER als 2-Byte-String, SINGLE und LONG als 4 Bytes, DOUBLE und CURRENCY mit 8 Bytes); für die Fließkommazahlen wird die IEEE-Codierung benutzt.

Bemerkung • Der Compiler-Switch /MBF (vgl. Kapitel 7) beeinflußt diese Funktionen.

Kompatibel PDS + VBWIN - Formen + Mathe +

Siehe auch CV*x*MBF (584), MK*x*$ (630), FIELD (602).

CVxMBF (Funktionen)

Anwendung
```
x! = CVSMBF(y$)
x# = CVDMBF(y$)
```

Nutzen Diese beiden Funktionen entschlüsseln wie ihre Nachfolger CVS und CVD codierte SINGLE- bzw. DOUBLE-Zahlen, allerdings nicht im IEEE-Verfahren, sondern mit dem Microsoft-Binärformat.

Bemerkung • Von QuickBASIC 2 auf QuickBASIC 3 wurde die interne Darstellung der Fließkommazahlen geändert, um coprozessorkompatibel zu werden. Die Zahlen wurden nicht mehr im Microsoft-eigenen Binärformat, sondern im weitverbreiteten IEEE-Format gespeichert. Diese Funktionen werden zur Abwärtskompatibilität unterstützt.

Kompatibel PDS + VBWIN - Formen + Mathe +

Siehe auch MK*x*MBF$ (630), CV*x* (584), FIELD (602).

DATA (Befehl)

Anwendung DATA *wert* [, *wert*]...

Nutzen DATA wird benutzt, um für die READ-Anweisungen innerhalb eines Programms die entsprechenden Konstanten zur Verfügung zu stellen. *wert* steht für eine numerische oder eine String-Konstante. Wichtig ist, daß die Variablentypen in den READ-Anweisungen mit denen in den DATA-Zeilen übereinstimmen. Stringkonstanten können in Anführungszeichen stehen; enthalten sie keine Kommata und Doppelpunkte, ist das jedoch nicht notwendig.

DATA-Zeilen dürfen nur auf Modulebene verwendet werden.

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch READ (656), RESTORE (658).

DATE$ (Systemvariable)

Anwendung *x$* = DATE$

DATE$ = *x$*

Nutzen DATE$ enthält stets das Datum der Systemuhr im Format *mm-tt-jjjj*. Man kann in diese Variable ein neues Systemdatum eintragen, das dann entweder die Form *mm-tt-jj* oder *mm-tt-jjjj* (auch Schrägstriche als Trennzeichen) haben muß.

Bemerkung • Da DATE$ einen *Funktionsaufruf unzulässig* produziert, wenn man versucht, ein ungültiges Datum (zum Beispiel den 29. Februar eines Nicht-Schaltjahres) hineinzuschreiben, kann man, wenn man den alten Wert vorher sichert, mittels DATE$ leicht die Gültigkeit eines vom Benutzer eingegebenen Datums prüfen lassen.

• VBDOS stellt mit den Zeit- und Datumsroutinen viele Möglichkeiten zur Datumsverarbeitung zur Verfügung. Die Funktion *NOW* kann ebenfalls benutzt werden, um auf das aktuelle Datum zuzugreifen.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch FORMAT$ (604), NOW (631), TIME$ (683), TIMER (683).

DATESERIAL (Funktion)

Anwendung *x#* = DATESERIAL(*jahr%, monat%, tag%*)

Nutzen Gibt den Zeitcode des angegebenen Datums zurück. Für *jahr%* sind die Zahlen 0 bis 178 und 1753 bis 2078 erlaubt. Zu den erstgenannten wird jeweils 1900 addiert. *monat%* und *tag%* dürfen praktisch jeden beliebigen Wert annehmen; der 32. Tag eines Monats, der nur 29 Tage hat, entspräche zum Beispiel dem 3. Tag des Folgemonats, während der 0. Tag eines Monats der letzte des Vormonats ist usw. Dabei darf jedoch der angegebene Bereich für Jahre nicht überschritten werden. Ungültige Angaben verursachen einen *Funktionsaufruf unzulässig*-Fehler.

Bemerkung • Die von dieser Funktion errechneten reinen Datums-Zeitcodes sind immer ganzzahlig, denn der Datums-Teil wird, wie in Kapitel 9 beschrieben, immer im Vorkomma-Teil eines Zeitcodes abgespeichert. Um einen vollständigen Zeitcode zu erhalten, müßte ein reiner Uhrzeit-Zeitcode addiert werden, wie er zum Beispiel mit der Funktion TIMESERIAL ermittelt werden kann.

Kompatibel PDS * VBWIN + Formen + Mathe +

Siehe auch DATEVALUE (586), TIMESERIAL (684).

DATEVALUE (Funktion)

Anwendung *x#* = DATEVALUE(*text$*)

Nutzen Diese Funktion arbeitet wie DATESERIAL, mit dem Unterschied, daß ihr nicht drei INTEGER-Zahlen für Jahr, Monat und Tag übergeben werden, sondern eine beliebig formatierte Datumsangabe. DATEVALUE versucht dann, diese korrekt zu erkennen und in einen Zeitcode umzusetzen. Die Funktion hat leider den Haken, daß sie zwar verschiedenste amerikanische Datumsangaben problemlos erkennt, bei deutscher Formatierung aber versagt. "30.12.1999" als *text$* wäre schon Grund genug für die Funktion, einen *Funktionsaufruf unzulässig* zu generieren und die Flinte ins Korn zu werfen. In *text$* enthaltene Zeitangaben werden ignoriert, wenn sie gültig sind, und führen zu einem *Funktionsaufruf unzulässig*, wenn sie ungültig sind.

Bemerkung • Wie auch DATESERIAL gibt diese Funktion nur ganzzahlige Funktionswerte zurück (siehe dort).

Kompatibel PDS * VBWIN + Formen + Mathe +
Siehe auch DATESERIAL (586), TIMEVALUE (684).

DAY (Funktion)

Anwendung *x%* = DAY(*zeitcode#*)
Nutzen Ermittelt den zum angegebenen *zeitcode#* gehörigen Tag des Monats (1–31). Es spielt keine Rolle, ob *zeitcode#* ein reiner Datums-Zeitcode oder ein vollständiger Zeitcode ist.
Kompatibel PDS * VBWIN + Formen + Mathe +
Siehe auch MONTH (631), YEAR (694).

DECLARE (Befehl)

Anwendung DECLARE {FUNCTION|SUB} *name* [CDECL]
[ALIAS "*aliasname*"] [([*parameterliste*])]
Nutzen Vereinbart Prozeduren und Funktionen. Prozeduren müssen nur vereinbart werden, wenn man sie ohne CALL aufrufen will, Funktionen aber immer, weil ihre Aufrufe sonst für gewöhnliche Variablenreferenzen gehalten werden.

In der Deklaration müssen als *parameterliste* nicht dieselben Variablen angegeben werden wie im Prozedur- oder Funktionskopf selbst, es kommt nur darauf an, daß die Anzahl, die Reihenfolge und die Variablentypen identisch sind.

Die *parameterliste* besteht aus einer beliebigen Anzahl von durch Kommata getrennten Variablenvereinbarungen wie »Winkel AS INTEGER«, genauso wie im Prozedurkopf. Den Variablennamen kann hier ein BYVAL oder SEG vorangehen (siehe dazu CALL). Arrays werden durch eine offene und eine geschlossene runde Klammer angezeigt.

Der DECLARE-Befehl für eine Prozedur, die keine Argumente hat, muß als *parameterliste* eine offene und eine geschlossene runde Klammer haben. Völlig weglassen dürfen Sie diese Klammern nur dann, wenn die Prozedur separat kompiliert wurde (zum Beispiel als Bestandteil einer Library oder eines anderen Moduls); dann werden Typ und Anzahl der Argumente nicht vom Compiler geprüft.

Die Optionen CDECL und ALIAS sind nur für externe Prozeduren (getrennt kompiliert bzw. in anderen Sprachen programmiert) verfügbar. CDECL bedeutet, daß die Prozedur die Argumente nach den C-Regeln übergibt (nämlich von hinten nach vorne) und daß der Name der Routine in der OBJ-Datei oder Library nicht genau der angegebene ist, sondern daß an den in BASIC benutzten Namen vorne noch ein Underscore-Zeichen (_) angefügt werden soll. Mit ALIAS, gefolgt von einem beliebigen, in Anführungszeichen gekleideten Namen, können Sie selbst festlegen, unter welchem Namen die Prozedur in OBJ-Files oder Libraries gesucht werden soll. So können Sie zum Beispiel in BASIC eine Prozedur namens ZeigeErgebnis benutzen, die in Wirklichkeit DisplayResult heißt.

Bemerkung
- Sie können den Typ der Variablen – genau wie im SUB- bzw. FUNCTION-Kopf – entweder mit einem Typenbezeichner oder mit einer AS-Formulierung festlegen. Fehlt beides, wird der Standard-Typ benutzt (üblicherweise SINGLE, siehe aber DEF*xxx*-Befehle). Nur in DECLARE-Befehlen ist auch die Typvereinbarung AS ANY erlaubt, die für den Parameter, bei dem sie steht, die Typenprüfung durch den Compiler verhindert.

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch CALL (572), SUB/FUNCTION (680), DEF*xxx* (589).

DEF FN (Befehl)

Anwendung
(1) DEF FN*funktionsname* [(*parameterliste*)] = *ausdruck*
(2) DEF FN*funktionsname* [(*parameterliste*)]
...
END DEF

Nutzen DEF FN ist eine altertümliche Methode zur Funktionsdefinition, die inzwischen von der weitaus mächtigeren FUNCTION...END FUNCTION-Kontruktion ersetzt wurde und nur noch in Ausnahmefällen ihre Berechtigung hat, da sie ein bißchen schneller als die Standard-Funktionen ist. Ich verzichte daher auf eine ausführliche Darstellung.

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch SUB/FUNCTION (680), STATIC (677).

DEFxxx (Befehle)

Anwendung
```
DEFCUR bereich [,bereich]... (Seit PDS 7.0)
DEFDBL bereich [,bereich]...
DEFINT bereich [,bereich]...
DEFLNG bereich [,bereich]... (Seit 4)
DEFSNG bereich [,bereich]...
DEFSTR bereich [,bereich]...
```

Nutzen Diese Befehle verändern für bestimmte Variablenbereiche den Standardtyp. Der Standardtyp ist der Datentyp, den Variablen annehmen, die weder von einem Typenbezeichner gefolgt noch in einer DIM-, DECLARE- oder ähnlichen Anweisung mit einer AS-Formel typisiert werden. Ohne Verwendung eines DEF*xxx*-Befehls ist der Standardtyp für alle Variablen SINGLE.

Als *bereich* wird entweder ein einzelner Buchstabe oder ein Buchstabenbereich in der Form A-Z angegeben. Alle Variablen, deren erster Buchstabe identisch mit einem der angegebenen oder einem der eventuell dazwischenliegenden Buchstaben ist, haben von diesem Moment an den entsprechenden neuen Standardtyp (CUR = CURRENCY, DBL = DOUBLE, INT = INTEGER, LNG = LONG, SNG = SINGLE, STR = STRING). Groß- und Kleinschreibung spielt keine Rolle.

Bemerkung
- Die DEF*xxx*-Befehle heben sich gegenseitig auf. Wird beispielsweise ein DEFINT A-Z gefolgt von einem DEFDBL D, E, X, so wird der Standardtyp für D, E und X DOUBLE.
- Wenn Sie OPTION EXPLICIT verwenden, sind die DEF*xxx*-Befehle sinnlos, da dann alle Variablen ohnehin explizit definiert werden müssen.

Kompatibel PDS + VBWIN * Formen + Mathe -

Siehe auch DIM (590).

DEF SEG (Befehl)

Anwendung
```
DEF SEG [= segment]
```

Nutzen Setzt das Speichersegment, auf das sich alle folgenden BLOAD-, BSAVE-, PEEK-, POKE- und CALL ABSOLUTE-Befehle beziehen. Benutzt man DEF SEG ohne Parameter, wird das Standard-Datensegment DGROUP gesetzt. Der erlaubte Bereich für *segment* ist 0 bis 65.535.

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch BLOAD (571), BSAVE (571), PEEK (645), POKE (649), VARSEG (689), VARPTR (688), SSEG (675), SADD (662), SSEGADD (676).

DIM (Befehl)

Anwendung DIM [SHARED] *variable*[(*array-bereich*)] [AS *typ*]
[, *variable*[(*array-bereich*)] [AS *typ*]]...

Nutzen In erster Linie und ursprünglich ist DIM ein Befehl, der Arrays vereinbart. Hier dient er allerdings auch dazu, globale Variablen zu vereinbaren und Variablen ohne Typenbezeichner einen Typ zuzuordnen.

Arrays

Um ein Array zu vereinbaren (zu dimensionieren), wird der Name des Arrays angegeben; dahinter folgen runde Klammern, die den Array-Bereich enthalten. Eine Array-Bereich-Angabe enthält pro Dimension eine einzelne Zahl (dann ist der Bereich für diese Dimension 0 bis Zahl oder 1 bis Zahl, je nach OPTION BASE) oder eine Klausel der Form *Zahl1* TO *Zahl2*, die den Bereich für die betreffende Dimension auf *Zahl1* bis *Zahl2* setzt. Die Bereichsangaben für die einzelnen Dimensionen müssen durch Kommata getrennt sein. Der Datentyp des Arrays kann durch einen Typenbezeichner im Namen oder durch eine Typzuordnung mit AS festgelegt werden. Arrays können mit SHARED global dimensioniert werden.

Wenn statt Zahlen oder Konstanten bei der Dimensionierung von Arrays Variablen benutzt werden, werden dynamische Felder erzeugt (vgl. Kapitel 9).

Globale Variablen mit SHARED

Wenn Sie hinter DIM das Wort SHARED angeben (es gilt dann für alle Variablen, die in dieser Zeile vereinbart werden), sind die Variablen global. DIM SHARED kann nur im Modulcode benutzt werden.

Typzuordnung mit AS

Einem beliebigen Variablennamen kann mit DIM ein Datentyp zugeordnet werden, indem man AS *datentyp* anhängt. Zwingend notwendig ist diese Klausel nur dann, wenn man Variablen von einem selbstdefinierten Datentyp oder vom Typ »String mit fester Länge« vereinbaren will, denn für die gibt es keinen Typenbezeichner. Bei gewöhnlichen Datentypen ist das nicht notwendig, es sei denn, man verwendet OPTION EXPLICIT.

Als *typ* sind folgende Datentypen erlaubt: INTEGER, LONG, CURRENCY, SINGLE, DOUBLE, STRING, STRING * x (String mit fester Länge), sowie alle Typen, die mit TYPE...END TYPE vereinbart sind.

Kompatibel PDS + VBWIN * Formen + Mathe -

Siehe auch COMMON (581), OPTION BASE (642), OPTION EXPLICIT (642), STATIC (677), SHARED (672), REDIM (657), ERASE (597).

DIR$ (Funktion)

Anwendung *x$* = DIR$ [(*directorymaske$*)]

Nutzen DIR$ wird benutzt, um ein Directory als Liste von Dateinamen einzulesen. Dazu *muß* beim ersten Aufruf von DIR$ eine *directorymaske$* (zum Beispiel C:\DOS*.COM oder C:*.*) angegeben werden; der Funktionswert ist dann der Name der ersten passenden Datei oder ein Leerstring, wenn es keine passende gibt. Ein erneuter Aufruf von DIR$ ohne Angabe von *directorymaske$* gibt dann den zweiten passenden Dateinamen zurück usw. – bis DIR$ einen Leerstring ergibt, dann gibt es keine passenden Namen mehr.

Wenn man DIR$ mit einer neuen *directorymaske$* benutzt, noch bevor ein Directory zu Ende gelesen wurde, beginnt DIR$, das neue Directory zu lesen. Ruft man aber DIR$ ohne *directorymaske$* auf, wenn es vorher noch gar nicht benutzt wurde, führt das zu einem *Funktionsaufruf unzulässig*.

Bemerkung • Die Dateilisten-Steuerelemente (Datei-, Verzeichnis- und Laufwerkslistenfeld) von VBDOS lassen sich ebenfalls zum Einlesen von Dateiverzeichnissen verwenden (vgl. @@).

• Werden in *directorymaske$* Laufwerk und Pfad nicht angegeben, so wird jeweils das aktuelle Laufwerk bzw. das aktuelle Directory verwendet.

• DIR$ gibt nur den Dateinamen und seine Extension (zum Beispiel CHKDSK.EXE) zurück, nicht aber Laufwerk und Directory, die ja durchaus vom aktuellen Laufwerk bzw. Directory verschieden sein können. Um vollständige Dateinamen zu erhalten, müßte also die Laufwerks- und Pfadangabe (falls vorhanden) aus *directorymaske$* noch dem erhaltenen Dateinamen vorangestellt werden.

• DIR$ findet weder Subdirectories noch versteckte Dateien oder Systemdateien. Benutzen Sie dazu die in diesem Buch vorgestellten Interruptroutinen (Kapitel 24) oder aber die Datei-Steuerelemente von VBDOS.

• Wenn Sie wissen, daß ein gegebener Dateiname *n$* nicht ungültig ist, können Sie die Existenz der Datei *n$* leicht prüfen mit IF LEN(DIR$(n$)) = 0 THEN...

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch FILES (603), CURDIR$ (583).

DO...LOOP (Befehl)

Anwendung
```
DO [WHILE bedingung|UNTIL bedingung]
...
LOOP [WHILE bedingung|UNTIL bedingung]
```

Nutzen DO...LOOP ist ein universeller Strukturbefehl, der das ältere WHILE...WEND völlig ersetzt und darüber hinaus noch andere Möglichkeiten besitzt. Man kann frei wählen, ob man die Abbruchbedingung am Anfang der Schleife (hinter DO) oder am Ende (hinter LOOP) testen lassen will. Außerdem kann man diese Bedingung entweder mit WHILE (»solange wie«) oder mit UNTIL (»solange bis«) formulieren. WHILE *bedingung* ist identisch mit UNTIL NOT *bedingung*, und UNTIL *bedingung* läßt sich dementsprechend auch als WHILE NOT *bedingung* schreiben.

Ein EXIT DO innerhalb der Schleife sorgt für ihren sofortigen Abbruch, die Programmausführung wird dann hinter dem LOOP-Befehl fortgesetzt. Es können in einer Schleife beliebig viele EXIT DO-Befehle enthalten sein.

Es ist erlaubt, völlig auf eine Abbruchbedingung zu verzichten, also nur ein einfaches DO und ein einfaches LOOP zu verwenden. Die Schleife sollte dann aber mit EXIT DO verlassen werden, da sie sonst endlos wäre.

Nicht erlaubt ist hingegen, zwei Abbruchbedingungen (zum Beispiel DO WHILE...LOOP UNTIL) zu benutzen.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch WHILE...WEND (417), FOR...NEXT (314).

DRAW (Befehl)

Anwendung DRAW *zeichenbefehle$*

Nutzen DRAW wird zum Zeichnen von Linien im Grafikmodus benutzt. Es hat die Funktion eines Mini-Befehlsinterpreters: *zeichenbefehle$* kann beliebig viele Zeichenbefehle enthalten (die Befehle werden mit einem einzelnen Buchstaben abgekürzt), die bei der Ausführung des DRAW-Befehls umgesetzt werden. Wer schon einmal mit der Lernsprache LOGO zu tun gehabt hat, wird sich bei DRAW leicht heimisch fühlen.

Zeichenbefehle

Gezeichnet wird immer von der Position des Grafikcursors aus. Der Grafikcursor wird dann auf die neue Position gesetzt. Die meisten anderen Grafikbefehle setzen ihn ebenfalls auf die von ihnen zuletzt gezeichnete Position. Nach dem Einschalten des Grafikmodus steht der Grafikcursor in der Mitte des Bildschirms.

Befehl	*Bedeutung*
U [*n*]	Zeichne *n* Einheiten aufwärts
D [*n*]	Zeichne *n* Einheiten abwärts
L [*n*]	Zeichne *n* Einheiten nach links
R [*n*]	Zeichne *n* Einheiten nach rechts
E [*n*]	Zeichne diagonal *n* Einheiten auf/rechts
F [*n*]	Zeichne diagonal *n* Einheiten ab/rechts
G [*n*]	Zeichne diagonal *n* Einheiten ab/links
H [*n*]	Zeichne diagonal *n* Einheiten auf/links
M *x,y*	Zeichne zum Punkt *x, y* (absolut)
B	Kann einem der bisher genannten Befehle vorangehen und sorgt dafür, daß der Grafikcursor zwar bewegt wird, aber nichts zeichnet.

N	Kann einem der bisher genannten Befehle vorangehen und sorgt dafür, daß der Grafikcursor nach Ausführung des Zeichenbefehls wieder zurückgesetzt, also durch den Zeichenbefehl praktisch nicht verschoben wird.
M+|– *x,y*	Zeichne relativ vom aktuellen Punkt *x* Einheiten in *x*- und *y* Einheiten in *y*-Richtung (je nachdem, ob – oder + angegeben wird, werden *x* und *y* von den aktuellen Koordinaten abgezogen oder zu ihnen addiert).
A *n*	Zeichnung um *n* * 90° drehen (0 ≤ *n* ≤ 3)
TA *n*	Zeichnung um *n*° drehen (–360 ≤ *n* ≤ 360)
C *n*	Vordergrundfarbe (Zeichenfarbe) auf *n* setzen (siehe SCREEN für gültige Werte)
P *f, r*	Fläche ab aktuellem Punkt in Farbe *f* füllen; Füllgrenzen sind Linien der Farbe *r*
S *n*	Skalierung *n* setzen. *n* = 1 ist normal (1 DRAW-Einheit = 1 Pixel bzw. 1 Einheit gemäß WINDOW-Definition; bei Rotationen wird das Bildschirm-Seitenverhältnis 4:3 jedoch automatisch mit eingerechnet, so daß möglichst keine Verzerrungen auftreten), eine größere Zahl für n bedeutet mehr Pixel pro Einheit.
X*adr$*	Führt einen weiteren DRAW-String aus, dessen Adreßcode mit VARPTR$ ermittelt und in *adr$* eingetragen werden muß (siehe Bemerkung).
=*adr$*	Kann anstelle einer Zahl (*n, x, y, f, g* in obigen Beispielen) gesetzt werden; *adr$* muß die mit VARPTR$ ermittelte Adresse einer Zahl im Speicher sein, die dann hier eingesetzt wird.

Bemerkung • Leerzeichen sind überhaupt nicht erforderlich, aber an jeder Position und in beliebiger Anzahl im DRAW-String erlaubt – mit einer Ausnahme: Zwischen X bzw. = und dem dazugehörigen VARPTR$ darf sich *kein* Leerzeichen befinden.

Kompatibel PDS + VBWIN * Formen - Mathe +

Siehe auch LINE (623), PSET (653), CIRCLE (576), PAINT (642), SCREEN (663), COLOR (579).

$DYNAMIC (Metabefehl)

Anwendung REM $DYNAMIC

Nutzen Alle DIM-Befehle, die auf diesen Metabefehl folgen, erzeugen dynamische Arrays (Arrays, deren Speicherplatz erst bei der Ausführung des DIM-Befehls zugeordnet wird).

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch $STATIC (676), DIM (590).

END (Befehl)

Anwendung END [*errorlevel%*]

Nutzen Beendet ein Programm (END DEF siehe DEF FN, END FUNCTION und END SUB siehe SUB/FUNCTION, END IF siehe IF, END SELECT siehe SELECT CASE, END TYPE siehe TYPE) und schließt alle Dateien.

Die optionale Variable *errorlevel%* kann benutzt werden, um einen Beendigungs-Code an das aufrufende Programm (zumeist das Betriebssystem) zurückzugeben. In Batch-Dateien unter DOS kann dieser Code dann mit IF ERRORLEVEL... abgefragt werden. Schwere Fehler, die nicht vom Programm abgefangen werden, setzen diese Variable auf –1.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch STOP (678), SYSTEM (681), DEF FN (588), IF (611), SELECT CASE (670), SUB/FUNCTION (680), TYPE (685).

ENVIRON (Befehl)

Anwendung ENVIRON *variablenzuordnung$*

Nutzen Verändert bzw. löscht eine bestehende oder erzeugt neue Betriebssystemvariablen. Der String *variablenzuordnung$* muß die Form »VARIABLE=Zuordnung« haben, wobei »VARIABLE« in Großbuchstaben angegeben werden muß und dahinter kein Leerzeichen stehen darf. Läßt man »*Zuordnung*« weg oder schreibt an ihrer Stelle nur ein Semikolon, dann wird die genannte Variable gelöscht. Ordnet man einer bestehenden Variablen einen neuen Inhalt zu, wird der alte überschrieben, und wenn man eine bisher unbekannte Variable benutzt, wird sie neu erzeugt.

Bemerkung
- BASIC kann nur temporäre Änderungen an den Betriebssystemvariablen vornehmen; alle Änderungen gehen bei Programmende verloren. Die Änderungen können sich also nur auf Programme auswirken, die mit RUN oder CHAIN direkt oder mit SHELL als Tochterprozeß aufgerufen werden.
- BASIC kann die Gesamtgröße der Variablentabelle nicht erhöhen, so daß man, will man eine neue Variable einführen oder eine bestehende vergrößern, zuerst eine bestehende Variable löschen oder verkleinern muß, da anderenfalls ein Fehler auftritt.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch ENVIRON$ (596), SHELL (673).

ENVIRON$ (Funktion)

Anwendung
(1) *x$* = ENVIRON$(*variablenname$*)
(2) *x$* = ENVIRON$(*variablennummer%*)

Nutzen Liest den Inhalt einer Betriebssystemvariablen aus. Solche Variablen werden unter DOS mit dem SET-Befehl gesetzt. Die Befehle PATH und PROMPT etablieren selbst eine gleichnamige Betriebssystemvariable. Die erste Syntax wird benutzt, um den Inhalt einer bestimmten Variable zu ermitteln; *variablenname$* muß dabei in Großbuchstaben angegeben werden. ENVIRON$("PATH") gibt zum Beispiel den eingestellten PATH zurück (oder einen Leerstring, wenn keiner eingestellt ist). Wenn Sie in DOS zum Beispiel mit SET DRUCKER=FX80 die Betriebssystemvariable DRUCKER erfinden und belegen, können Sie mit ENVIRON$("DRUCKER") deren Inhalt abfragen.

Die zweite Syntax wird üblicherweise dazu benutzt, alle vorhandenen Variablen einzulesen. Man beginnt mit der Variable Nummer 1 (*variablennummer%* = 1) und erhöht sie so lange, bis ein Leerstring zurückgegeben wird.

Bemerkung
- Mittels ENVIRON$ können Sie prüfen, ob Ihr Programm von WINDOWS aus aufgerufen wurde. Wenn der Ausdruck ENVIRON$("windir") keinen Leerstring zurückgibt – ausnahmsweise müssen Kleinbuchstaben verwendet werden – ist WINDOWS aktiv, und der zurückgegebene String bezeichnet das WINDOWS-Verzeichnis.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch ENVIRON (595), SHELL (673).

EOF (Funktion)

Anwendung *x* = EOF(*dateinummer*)

Nutzen Zeigt das Ende einer Datei oder den Status einer COM-Schnittstelle an.

Bei allen Dateien außer ISAM-Dateien ist EOF –1 (TRUE), wenn das letzte Zeichen der Datei gelesen wurde. COM-Schnittstellen im ASCII-Modus setzen EOF auf TRUE, sobald sie ein CTRL-Z (ASCII #26) empfangen; es bleibt TRUE, bis die Schnittstelle geschlossen wird. Im Binär-Modus ist EOF für eine Kommunikationsschnittstelle TRUE, wenn kein Zeichen anliegt, und wird wieder FALSE, sobald ein Zeichen empfangen wurde.

Bemerkung • EOF wird auch für ISAM-Datenbanken verwendet (siehe ISAM-Referenzteil).

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch LOC (624), LOF (626), OPEN (632), CLOSE (578), DO...LOOP (592).

ERASE (Befehl)

Anwendung ERASE *arrayname* [, *arrayname*]...

Nutzen Löscht ein mit DIM oder REDIM definiertes Feld aus dem Speicher. Es muß nur der Name des Feldes angegeben werden, keine Dimension. Handelt es sich um ein dynamisches Feld, wird es völlig gelöscht und der von ihm belegte Speicher wieder freigegeben; bei statischen Feldern werden lediglich alle Variablen zurückgesetzt (numerische auf 0, Strings auf ""), der Speicherplatz bleibt aber belegt, und das Array kann weiterhin benutzt werden.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch DIM (590), REDIM (657).

ERDEV$, ERDEV$ (Funktionen)

Anwendung
```
x = ERDEV
x$ = ERDEV$
```

Nutzen Nach dem Auftreten eines externen Fehlers (nur Fehler, die von DOS oder einer Kommunikationsschnittstelle verursacht werden) kann über diese Funktionen genauer festgestellt werden, welches Gerät den Fehler verursachte und welcher Fehler es genau war.

Nach Auftreten eines Fehlers enthält ERDEV$ den Namen des Geräts (zum Beispiel COM1, A:, LPT1: usw.), und `ERDEV AND 255` (das niederwertige Byte von ERDEV) enthält einen Fehlercode, der entweder von DOS oder vom OPEN COM-Befehl gesetzt wurde. `ERDEV \ 256` (das höherwertige Byte von ERDEV) enthält bei Blocktreibern zusätzliche Geräte-Attributinformation.

Kompatibel PDS + VBWIN - Formen + Mathe +

Siehe auch ERR, ERL (598), ON ERROR (599)

ERR, ERL (Systemvariablen)

Anwendung
```
x = ERR
x = ERL
ERR = x
```

Nutzen Mit Hilfe dieser Funktionen läßt sich abfragen, welcher Fehler zuletzt auftrat (ERR), und in welcher Zeile das passierte (ERL). Die Variable ERR läßt sich darüberhinaus auch auf einen beliebigen Wert setzen, um programminterne Fehlerinformationen weiterzugeben (ERR verhält sich wie eine globale Variable, die mit COMMON SHARED vereinbart wurde – sie ist überall zugänglich).

ERR enthält nach dem Auftreten eines Fehlers dessen Fehlercode (siehe Anhang), während in ERL die Nummer der Zeile steht, in der der Fehler auftrat. Hat die betreffende Zeile selbst keine Nummer, so wird die im Source-File als letzte vor dieser Zeile vorkommende Nummer gewählt.

Bemerkung • Die Variable ERR wird nicht nur gesetzt, wenn ein Fehler auftritt, sondern auch wieder auf 0 zurückgestellt bei folgenden Befehlen: RESUME, ON ERROR, ON LOCAL ERROR; EXIT DEF, EXIT SUB und EXIT FUNCTION setzen ERR auf 0, wenn damit eine lokale Fehlerbehandlungsroutine verlassen wird.

• In VBDOS kann ERR Werte bis zu 32767 annehmen. Zuweisungen der Art ERR = x sind jedoch nur bis 255 möglich; um ERR auf einen höheren Wert zu setzen, muß ERROR x verwendet werden.

Kompatibel PDS + VBWIN + Formen + Mathe +

Siehe auch ERDEV (598), ERDEV$ (598), ERROR (599), ERROR$ (599), ON ERROR (632).

ERROR (Befehl)

Anwendung ERROR *fehlernummer*

Nutzen Simuliert das Auftreten eines Fehlers. Der erlaubte Bereich für *fehlernummer* ist 1 bis 32767.

Bemerkung • ERROR kann benutzt werden, um die Fehlerbehandlungsroutinen auf Dinge wie zum Beispiel falsche Eingaben o.*., die keine Fehler im Sinne des Compilers sind, auszudehnen. Dazu bedient man sich am besten eigener, von BASIC ungenutzter Fehler-Codes. Microsoft empfiehlt, eigene Fehler von 32767 abwärts durchzunumerieren. VBDOS selbst benutzt nur die Codes bis 480 (vgl. Anhang).

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch ERR (598), ERL (598), ON ERROR (632), RESUME (658).

ERROR$ (Funktion)

Anwendung x$ = ERROR$ [(*fehlernummer*)]

Nutzen Gibt den Klartext zu einem gültigen VBDOS-Fehlercode zurück. Dieser Klartext ist genau der, den VBDOS selbst bei Auftreten des fehlers anzeigt. Wird keine *fehlernummer* angegeben, so gibt ERROR$ den Klartext zum zuletzt aufgetretenen Fehler (dessen Nummer in ERR gespeichert ist) zurück.

Kompatibel PDS - VBWIN + Formen + Mathe -

Siehe auch ERR (598), ERL (598).

EVENT ON, EVENT OFF (Metabefehle)

Anwendung
EVENT ON
EVENT OFF

Nutzen
Diese Befehle schalten das Event-Trapping aus bzw. wieder ein. Diese Befehle haben, obwohl sie nicht mit einem $-Zeichen beginnen und auch nicht von einem REM eingeleitet werden müssen, den Charakter von Metabefehlen, denn sie wirken sich bereits während des Kompiliervorgangs aus.

Wenn mit den Compiler-Schaltern /W oder /V gearbeitet wird, erzeugt der Compiler für jedes Label bzw. jeden Befehl einen sogenannten Event-Test, das ist der Aufruf einer Routine, die prüft, ob irgendeines der zu überwachenden Ereignisse (eine bestimmte Taste wurde gedrückt [ON KEY], ein Zeichen liegt am Kommunikationsanschluß an [ON COM] usw.) eingetreten ist. Das benötigt viel Zeit und viel Platz.

Mit EVENT OFF kann diese zusätzliche Code-Erzeugung abgeschaltet werden. Dann werden bis zum nächsten EVENT ON keine Event-Tests eingefügt. Tritt später, während das Programm abläuft, innerhalb einer solchen EVENT-OFF-Sektion ein Ereignis ein, das eigentlich überwacht werden sollte, reagiert das Programm erst darauf, wenn es wieder in einen EVENT-ON-Block kommt.

Bemerkung
- EVENT OFF/ON ist natürlich nur bei Programmen sinnvoll, die überhaupt mit Event Trapping (/V oder /W) arbeiten. Dann sollte man es aber möglichst häufig einsetzen, zumindest bei zeitkritischen Routinen.

- Bedenken Sie, daß die Teilung in EVENT ON- und EVENT OFF-Blocks schon während des Kompilierens geschieht. Es spielt keine Rolle, in welcher Reihenfolge Ihr Programm später abläuft, sondern EVENT OFF und EVENT ON wirken auf alle Befehle, die dazwischen stehen (man könnte sie also in einem Listing-Ausdruck leicht markieren, indem man den Bereich zwischen EVENT OFF und EVENT ON ausfüllt).

Kompatibel PDS + VBWIN - Formen + Mathe +
Siehe auch ON...GOSUB (634).

EXIT (Befehl)

Anwendung EXIT *konstruktion*

Nutzen Beendet eine bestimmte Struktur vorzeitig; EXIT DEF siehe DEF FN; EXIT DO siehe DO...LOOP; EXIT FOR siehe FOR...NEXT; EXIT FUNCTION und EXIT SUB siehe SUB/FUNCTION.

Bemerkung • Mit EXIT kann nicht nur aus der tiefsten Struktur, sondern unter Umständen auch aus übergeordneten Strukturen gesprungen werden. Wenn innerhalb eines SUB...END SUB-Blocks, einer Subroutine also, sich ein FOR...NEXT-Block befindet und innerhalb desselben wiederum eine DO...LOOP-Konstruktion, so beendet ein EXIT SUB-Befehl innerhalb der DO-LOOP-Konstruktion sowohl DO...LOOP als auch FOR...NEXT als auch SUB...END SUB. Es ist jedoch nicht möglich, zum Beispiel bei zwei ineinander verschachtelten FOR...NEXT-Schleifen mit einem EXIT-Befehl beide gleichzeitig zu verlassen. Dazu müßte man um beide herum noch ein DO...LOOP konstruieren, um dann mit EXIT DO aus diesem springen zu können.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch DEF FN (588), DO...LOOP (592), FOR...NEXT (603), SUB/FUNCTION (680).

EXP (Funktion)

Anwendung x = EXP(y)

Nutzen EXP(y) gibt die Exponentialfunktion e^y zurück. Bei SINGLE-Genauigkeit (wenn y INTEGER oder SINGLE ist) darf y 88,02969, bei DOUBLE-Genauigkeit (wenn y einen anderen numerischen Typ hat) 709,782712893 nicht überschreiten.

Kompatibel PDS + VBWIN + Formen + Mathe +

Siehe auch LOG (626).

FIELD (Befehl)

Anwendung FIELD [#] dateinummer, byte AS buffer$
[, byte AS buffer$]...

Nutzen FIELD wird benutzt, um bei einer Random-Access-Datei (OPEN FOR RANDOM) die Ein-/Ausgabepuffer zu definieren, die Variablen, über die Daten aus der Datei gelesen und in sie geschrieben werden. Da FIELD für moderne Programme, die mit selbstdefinierten Typen arbeiten, nicht mehr erforderlich ist, wird von einer detaillierten Beschreibugn abgesehen.

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch GET (609), PUT (654), LSET (628), RSET (660), OPEN (632).

FILEATTR (Funktion)

Anwendung x = FILEATTR (dateinummer, typ)

Nutzen Mit FILEATTR kann man Informationen über eine geöffnete Datei abfragen. Ist *typ* = 1, dann gibt FILEATTR den Modus zurück, der beim OPEN-Befehl angegeben wurde: 1 = INPUT; 2 = OUTPUT, 4 = RANDOM, 8 = APPEND, 32 = BINARY. Mit *typ* = 2 erhält man die Nummer des File-Handles, den DOS dieser Datei beim Öffnen zugeordnet hat. Diese Information kann bei der Arbeit mit einigen Interrupts oder Assembler-Unterroutinen von Nutzen sein, wird in BASIC aber sonst nicht benötigt.

Bemerkung • FILEATTR kann auch auf ISAM-Dateien angewandt werden. Siehe ISAM-Referenzteil.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch OPEN (632).

FILES (Befehl)

Anwendung FILES [*directorymaske*]

Nutzen Gibt ein Verzeichnis der Dateien, die auf die angegebene *directorymaske* (zum Beispiel C:\SPIELE*.EXE) passen, aus. Läßt man *directorymaske* weg, werden alle Dateien des aktuellen Verzeichnisses ausgegeben. Die Ausgabe kann nicht beeinflußt werden. FILES wird in modernen Programmen durch DIR$ oder die Datei-Steuerelemente ersetzt, da es die Bildschirmausgabe empfindlich stört.

Kompatibel PDS + VBWIN - Formen - Mathe -

Siehe auch DIR$ (591).

FIX (Funktion)

Anwendung *x%* = FIX(*y*)

Nutzen Schneidet den Nachkommateil einer Zahl ab.

Bemerkung • FIX, CINT und INT sind ähnliche Funktionen. Die Unterschiede sind im Abschnitt über INT zusammengefaßt.

Kompatibel PDS + VBWIN + Formen + Mathe +

Siehe auch CINT (576), INT (616).

FOR...NEXT (Befehl)

Anwendung FOR *zähler* = *anfang* TO *ende* [STEP *schrittweite*]
...
NEXT [*zähler*] [, *zähler*]...

Nutzen FOR...NEXT ist ein Strukturbefehl, der für jeden Durchlauf des Befehlsblocks einen Zähler erhöht. Wenn die *schrittweite* weggelassen wird, wird *zähler* immer um 1 erhöht, ansonsten um die angegebene Schrittweite. Auch negative Schrittweiten sind möglich; dann muß allerdings *ende* kleiner als *anfang* sein.

zähler kann eine Variable beliebigen numerischen Typs sein. Beim NEXT-Befehl muß *zähler* nicht angegeben werden. Ein NEXT-Befehl kann verwendet werden, um mehrere Schleifen zu beenden: NEXT A, B ist identisch mit NEXT A: NEXT B.

Ein EXIT FOR-Befehl kann an beliebigen Stellen innerhalb der Schleife untergebracht werden. Nach seiner Ausführung bricht die Schleife ab, und das Programm wird hinter dem NEXT-Befehl fortgesetzt. Es ist jedoch nicht möglich, mehrere Schleifen gleichzeitig zu verlassen.

Bemerkung

- Die Schleife wird so lange ausgeführt, bis *zähler* entweder größer oder kleiner (je nach *schrittweite*) als *ende* ist. Sind *anfang* und *ende* also beim Aufruf der Schleife gleich, wird die Schleife trotzdem einmal durchlaufen.
- *ende* + *schrittweite* darf nie größer sein als der maximal zulässige Wert für den Typ von *zähler* (bzw., bei negativer *schrittweite*, nie kleiner als der minimale Wert).
- *anfang* und *ende* werden, wenn es sich um Variablen oder Ausdrücke handelt, nur am Anfang der Schleife einmal berechnet. Eine Änderung dieser Werte innerhalb der Schleife bleibt also ohne Auswirkungen.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch DO...LOOP (592), EXIT (601), WHILE...WEND (691).

FORMAT$ (Funktion)

Anwendung *x$* = FORMAT$(*zahl* [, *aussehen$*])

Nutzen Wandelt eine beliebige Zahl in einen String um; kann dabei, entgegen dem älteren STR$, in weiten Grenzen durch eine Formatangabe (*aussehen$*) beeinflußt werden.

Alle Zeichen, die keine besondere Bedeutung gemäß den folgenden Tabellen haben, können in *aussehen$* als gewöhnlicher Text verwendet werden und werden unverändert angezeigt. Wenn Sie Zeichen, die eine besondere Bedeutung haben, im Text benutzen wollen, müssen diese entweder einzeln von einem Backslash (\) eingeleitet werden oder in Anführungszeichen (") stehen.

Zeichen	*Bedeutung*
0	Ziffern-Platzhalter. Hat die Zahl mehr Kommastellen, als sich Nullen hinter dem Dezimalpunkt in *aussehen$* befinden, wird gerundet. Hat die Zahl mehr Vorkommastellen, als sich Nullen vor dem Dezimalpunkt in *aussehen$* befinden, arbeitet die Funktion so, als seien genug Nullen vorhanden gewesen. Hat die Zahl vor oder hinter dem Dezimalpunkt weniger Stellen, als sich Nullen in *aussehen$* befinden, werden die freien Plätze mit 0 aufgefüllt.
#	wie 0, füllt nicht besetzte Stellen aber nicht mit 0 auf. Im Gegensatz zum #-Zeichen in einer PRINT USING-Anweisung werden nicht besetzte Vorkommastellen aber nicht mit Leerzeichen gefüllt, so daß der String, der sich ergibt, wenn man die Zahl 9 mit "####" formatiert, nur ein Zeichen (anstatt wie bei PRINT USING vier Zeichen) lang ist.
.	Dezimalpunkt. Wenn mit SetFormatCC ein Land eingestellt wird, in dem üblicherweise ein Komma zur Abtrennung von Dezimalstellen benutzt wird (zum Beispiel 49 für Deutschland), müssen Sie statt des Punktes auch ein Komma in *format$* benutzen.
%	Der Wert wird mit 100 multipliziert, und ein Prozentzeichen wird an der Stelle eingefügt, an der es im *aussehen$* steht.
,	Erwirkt, daß an der Stelle, an der es in *aussehen$* steht, ein Komma als Tausender-Trennzeichen ausgegeben wird, falls noch Ziffern da sind, die es umrunden. Wenn zwei Kommata oder ein Komma und der Dezimalpunkt direkt aufeinanderfolgen, eine Trennzeichen-Ausgabe an dieser Stelle; zwei Kommata hintereinander oder ein Komma direkt vor dem Dezimalpunkt sorgen dafür, daß die drei umschlossenen Ziffern nicht angezeigt werden. Dadurch wird die Zahl faktisch durch 1000 dividiert. Je nachdem, welches Land mit SetFormatCC eingestellt wurde, müssen Sie den Punkt als Tausender-Trennzeichen benutzen.
E	Erwirkt wissenschaftliche Darstellung. Die Anzahl der #- oder 0-Zeichen ist die Anzahl der Stellen im Exponenten. E– und e– zeigen – an, wenn der Exponent negativ ist; E+ und e+ zeigen – oder + je nach Vorzeichen des Exponenten an.

;	Trennt verschiedene Format-Sektionen voneinander.

Die Formatangabe kann bis zu drei Sektionen enthalten. Ist nur eine angegeben, wird diese für alle Zahlen verwendet. Sind es zwei, wird die erste für Zahlen ≥ 0 verwendet, die zweite für negative. Bei drei Sektionen ist die erste für positive, die zweite für negative und die dritte für Zahlen vom Wert 0.

Bei einer leeren Formatangabe wird das Standard-STR$-Format gewählt, allerdings ohne das führende Leerzeichen bei positiven Zahlen.

FORMAT$ ist außerdem geeignet, Zeitcodes zu formatieren:

Zeichen	*Bedeutung*
d	Tag als Nummer ohne führende Null anzeigen
dd	Tag als Nummer mit führender Null anzeigen
ddd	Tag als 3-Buchstaben-Abkürzung anzeigen
dddd	Tag als vollen Namen anzeigen
m-mmmm	Für m gilt dasselbe wie für d; m steht für Monate.
yy	Jahr zweistellig anzeigen
yyyy	Jahr vierstellig anzeigen
h	Stunde ohne führende Null anzeigen
hh	Stunde mit führender Null anzeigen
m, s	Wie h für Stunden werden auch m (Minuten) und s (Sekunden) verwendet. Wenn m jedoch nicht direkt nach h auftritt, wird es als »Monat« fehlinterpretiert.
am/pm	Wenn im *aussehen$* irgendwo entweder a/p, A/P, am/pm oder AM/PM vorkommt, wird die Zeit im Zwölf-Stunden-Format angezeigt, und an der betreffenden Stelle steht dann die jeweils korrekte Bezeichnung (bei am/pm also am oder pm, bei A/P A oder P usw.)

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch STR$ (678).

FRE (Funktion)

Anwendung *x* = FRE(*argument*)

Nutzen Stellt fest, wieviel Speicher einer bestimmten Kategorie zur Verfügung steht. Die folgende Tabelle beschreibt die Möglichkeiten:

argument	*Ergebnis*
–1	Der verbleibende Platz im Far-Speicher in Bytes
–2	Noch nie genutzter Stack-Speicher in Bytes (Stack-Speicher, den das Programm einmal benutzt, aber inzwischen wieder freigegeben hat, wird nicht mehr eingerechnet)
–3	Freier Platz im Expanded Memory in KB (Feature unavailable-Fehler tritt auf, wenn kein EMS vorhanden ist)
–4	Freier Platz im Segment, in dem die gerade aktive Form gespeichert ist
x$	freier Platz in dem Segment, in dem sich x$ befindet
""	freier Platz für temporäre Strings (DGROUP-Segment)

Jedes *argument*, das hier nicht genannt ist, führt zu einem *Funktionsaufruf unzulässig*.

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch SETMEM (671), STACK (676).

FREEFILE (Funktion)

Anwendung *x%* = FREEFILE

Nutzen Gibt die kleinste noch unbenutzte Dateinummer (eine Nummer, unter der noch keine Datei mit OPEN geöffnet ist) zurück.

Bemerkung • Unabhängig von der FILES-Zahl in CONFIG.SYS darf die höchste Dateinummer nicht größer als 255 sein, und es dürfen unter DOS nicht mehr als 16 Dateien gleichzeitig geöffnet sein. Ein Trick, das letztgenannte Limit zu umgehen, findet sich auf Seite 359.

Kompatibel PDS + VBWIN + Formen + Mathe +

Siehe auch OPEN (632).

GET (für Dateien) (Befehl)

Anwendung GET [#]*dateinummer*[,[*satznummer*], [*satzvariable*]]

Nutzen Liest Daten aus einer Datei, die FOR RANDOM oder FOR BINARY geöffnet wurde. Der Unterschied zwischen beiden Dateiarten beim GET-Befehl ist, daß (a) bei RANDOM-Dateien *satznummer* die Nummer des Datensatzes ist (Länge des Datensatzes wird beim Öffnen angegeben), während *satznummer* bei BINARY-Dateien die absolute Byte-Position innerhalb der Datei ist, ab der gelesen werden soll, und daß (b) für BINARY-Dateien die Angabe einer *satzvariable* unerläßlich ist, während RANDOM-Dateien stattdessen auch mit dem FIELD-Befehl bearbeitet werden können.

Läßt man die *satznummer* weg, so wird anstelle dessen LOC(*dateinummer*) + 1 benutzt, die Stelle nach der zuletzt eingelesenen Position in der Datei.

Bei RANDOM-Dateien wird eine Satzlänge an Bytes gelesen; die Daten werden in die entsprechenden FIELD-Variablen geschrieben, wenn keine *satzvariable* angegeben ist, sonst in die angegebene *satzvariable*. Es wird keine Typenprüfung durchgeführt. Die Variable darf aber nicht mehr Bytes umfassen als die Satzlänge, mit der die Datei geöffnet wurde. Bei BINARY-Dateien werden soviele Bytes gelesen, wie benötigt werden, um die angegebene *satzvariable* zu füllen (also LEN(*satzvariable*)). Wird eine *satzvariable* von numerischem Typ angegeben, so werden die eingelesenen Daten sofort entsprechend den Funktionen CV*x* konvertiert.

Bemerkung
- Arrays sind nicht als *satzvariable* zugelassen.
- Strings mit variabler Länge als *satzvariable* bei BINARY-Dateien lesen so viele Bytes ein, wie sie lang sind.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch PUT (für Dateien) (654), OPEN (632), LOC (624), SEEK (669).

GET (für Grafik) (Befehl)

Anwendung GET [STEP](*x1,y1*)–[STEP](*x2,y2*), feldname[(*index*)]

Nutzen Liest einen rechteckigen Bereich vom Grafikbildschirm in ein Datenfeld. Die Koordinatenpaare (*x1,y1*) und (*x2,y2*) sind die Eckpunkte des Rechtecks; gibt man STEP mit an, werden sie relativ von der aktuellen Position (für das erste Paar) bzw. relativ von der ersten Position (für das zweite Paar) bestimmt.

feldname ist der Name des Datenfeldes, in das der Grafikbereich zu kopieren ist. Jeder numerische Typ ist erlaubt, aber man sollte sich an die drei Integer-Typen INTEGER, LONG und CURRENCY halten. *index* steht für einen oder (je nach Dimension des angegebenen Feldes) mehrere Indizes, die, wenn sie angegeben werden, das Array-Element bezeichnen, bei dem die Übertragung aus dem Bildschirmspeicher beginnen soll. Auf diese Weise lassen sich mehrere Bildteile in einem einzigen Array hintereinander abspeichern. Dabei ist darauf zu achten, daß die Array-Elemente eines mehrdimensionalen Arrays mit den erstgenannten Indizes zuerst gezählt werden. Die Elemente eines zweidimensionalen Arrays sind also in der Reihenfolge (0,0), (1,0), (2,0)... (0,1), (1,1) im Speicher abgelegt.

Bemerkung • Wenn das Array nicht genügend Platz bereithält, um den angegebenen Bereich zu speichern, gibt es einen *Funktionsaufruf unzulässig*-Fehler. Die benötigte Anzahl an Bytes errechnet sich wie folgt: *bytes* = 4 + INT ((*xpixel*) * *bpp* + 7) / 8) * *p* * *ypixel*. Dabei sind *xpixel* und *ypixel* die Seitenlängen des Rechtecks in Pixel; *bpp* ist 1 für alle Screen-Modi außer 1 (*bpp* = 2) und 13 (*bpp* = 8); *p* ist immer 1 mit den Ausnahmen *p* = 2 für die Modi 9 (bei alten EGA-Karten mit nur 64 KB Bildschirmspeicher) und 10 und *p* = 4 für die Modi 9 (EGA-Karte mit mehr als 64 KB), 7, 8 und 12. Die Anzahl *Byte*, die man durch diese Rechnung erhält, kann durch die Anzahl der Bytes pro Element des verwendeten Arrays geteilt werden (2 für INTEGER, 4 für LONG, 8 für CURRENCY), um die Anzahl der Array-Elemente zu errechnen, die der Grafikbereich belegt.

Kompatibel PDS + VBWIN - Formen - Mathe -

Siehe auch PUT (für Grafik) (654), DIM (590).

GOSUB (Befehl)

Anwendung GOSUB *zeilennummer/-label*

Nutzen GOSUB verzweigt zu der angegebenen Stelle im Programm. Die aktuelle Position wird auf dem Stack gespeichert, so daß beim nächsten RETURN-Befehl dorthin zurückgekehrt werden kann.

Sie können mit GOSUB nur zu einer Zeilennummer bzw. einem Zeilenlabel verzweigen, das in derselben Code-Einheit ist, also nicht vom Hauptprogramm in ein SUB oder umgekehrt, nicht von einem SUB in eine anderes und auch nicht von einem Modul in ein anderes. Ausgenommen sind die ON *event* GOSUB-Befehle (siehe dort).

Bemerkung • Der Befehl CLEAR löscht den Stack, so daß danach auch kein RETURN zu einem früher ausgeführten GOSUB mehr durchgeführt werden kann.

• GOSUB hat nichts zu tun mit Prozeduren, die durch SUB...END SUB definiert werden. SUB bzw. FUNCTION sind – zumeist bessere – Alternativen zu GOSUB. Trotzdem können GOSUB und RETURN zum Beispiel gut eingesetzt werden, um eine lange Prozedur weiter zu modularisieren. Würde man dafür SUB...END SUB einsetzen, stiege die Gesamtzahl der Prozeduren und Funktionen womöglich auf ein unübersichtliches Maß.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch CLEAR (577), ON *event* GOSUB (634), SUB/FUNCTION (680).

GOTO (Befehl)

Anwendung GOTO *zeilennummer/-label*

Nutzen GOTO verzweigt zu der angegebenen Stelle im Programm. Sie können mit GOTO nur zu einer Stelle in derselben Code-Einheit verzweigen, also nicht vom Hauptprogramm in ein SUB und umgekehrt, nicht zwischen verschiedenen SUBs oder FUNCTIONs und auch nicht von einem Modul in ein anderes.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch GOSUB (610).

HEX$ (Funktion)

Anwendung *x$* = HEX$(*y*)

Nutzen Wandelt eine beliebige Zahl in ihre Hexadezimaldarstellung um. Diese Darstellung wird als String zurückgegeben. Nur ganze Zahlen können umgewandelt werden, Dezimalstellen werden gerundet.

Bemerkung • Bedingt durch die automatische Typkonvertierung ist das Verhalten von HEX$ etwas außergewöhnlich. Positive Werte bis $2^{31}-1$ werden korrekt dargestellt. Negative Werte, die größer oder gleich -2^{15} sind, werden um 2^{16} vergrößert und dann hexadezimal dargestellt, und negative Zahlen, die kleiner als -2^{15} sind, werden um 2^{32} vergrößert und dann umgewandelt. Zahlen außerhalb des Bereichs $-2^{31} \leq x < 2^{31}$ führen zu einem *Überlauf*.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch OCT$ (631).

HOUR (Funktion)

Anwendung *x%* = HOUR(*zeitcode#*)

Nutzen Ermittelt die zum angegebenen *zeitcode#* gehörige Stunde (0–23).

Kompatibel PDS * VBWIN + Formen + Mathe +

Siehe auch MINUTE (629), SECOND (669).

IF...THEN...ELSE (Befehl)

Anwendung (1) IF *bedingung* THEN *befehle* [ELSE *befehle*]
(2) IF *bedingung* THEN
befehle
[ELSEIF *bedingung* THEN
befehle]...
[ELSE
befehle]
END IF

Nutzen Führt einen Befehl oder eine Gruppe von Befehlen aus, wenn eine gegebene Bedingung zutrifft.

In der einzeiligen Syntax (1) werden, wenn *bedingung* erfüllt ist, alle Befehle bis zum Zeilenende oder bis zum ELSE ausgeführt, wenn *bedingung* nicht erfüllt ist, alle Befehle nach dem ELSE. Die Syntax (1) benötigt kein separates Ende-Kennzeichen.

In Syntax (2) werden, wenn *bedingung* erfüllt ist, alle Befehle bis zur nächsten ELSEIF-, ELSE- oder END IF-Zeile ausgeführt; einer ELSEIF-Zeile können Befehle folgen, die nur dann ausgeführt werden, wenn alle bisher getesteten Bedingungen nicht erfüllt wurden und die bei ELSEIF genannte erfüllt wird; einer ELSE-Zeile können Befehle folgen, die ausgeführt werden, wenn bisher alle Bedingungen unzutreffend waren. Wenn eine ELSE-Zeile benutzt wird, dürfen danach keine ELSEIF-Zeilen mehr folgen; es darf insgesamt nur eine ELSE-Zeile in einem Block enthalten sein. Die Befehle IF, ELSEIF, ELSE, END IF müssen jeweils der erste Befehl auf einer Zeile sein, und hinter einem THEN nach Syntax (2) darf in derselben Zeile kein Befehl mehr folgen.

Eine *bedingung* ist dann erfüllt, wenn ihr Wert nicht 0 ist. Es kann also auch eine einfache Variable als *bedingung* benutzt werden. Unwahre logische Ausdrücke, zum Beispiel `5 > 6`, haben den Wert 0, wahre den Wert –1.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch SELECT CASE (670).

$INCLUDE (Metabefehl)

Anwendung `REM $INCLUDE:'`*dateiname*`'`

Nutzen Lädt eine Include-Datei. Der Compiler arbeitet so, als wären die Zeilen aus der Datei *dateiname* an der Stelle, an der der $INCLUDE-Befehl steht, wirklich vorhanden.

Bemerkung • Wenn eine mit $INCLUDE angeforderte Datei nicht im aktuellen Verzeichnis steht, suchen sowohl VBDOS als auch der Compiler BC in dem Verzeichnis, das die DOS-Betriebssystemvariable INCLUDE angibt. Sie können diese (zum Beispiel in AUTOEXEC.BAT) mit dem SET-Befehl einstellen:

```
SET INCLUDE=C:\BC7\BI
```

Kompatibel PDS + VBWIN - Formen + Mathe -

INKEY$ (Funktion)

Anwendung *x$* = INKEY$

Nutzen Liest ein Zeichen aus dem Tastaturpuffer und gibt es zurück. Ist der Tastaturpuffer leer, so wird nicht gewartet, bis ein Zeichen gedrückt wurde, sondern es wird ein Leerstring zurückgegeben.

Die Information über die gedrückte Taste wird als ein oder zwei Zeichen langer String zurückgegeben. Ein String mit der Länge 1 entsteht, wenn es sich um gewöhnliche Tasten handelt, während ein String der Länge 2, der als erstes Zeichen immer CHR$(0) hat, den Code einer Sondertaste zurückgibt.

Gewöhnliche Tasten sind zum Beispiel alle Buchstaben, Ziffern, Satzzeichen, die Enter-, die Tab-, die Backspace- und die Leertaste. Auch die meisten Tastenkombinationen mit der CTRL-Taste zählen als gewöhnliche Tasten. Erweiterte Codes entstehen zum Beispiel bei Pfeil- und Funktionstasten, Kombinationen mit der ALT-Taste oder Shift-Tab. Die Tasten »Sys Req«, Num Lock, Scroll Lock, Caps Lock, Shift, ALT, CTRL, ALT GR und Pause können nicht mit INKEY$ abgefragt werden. Die Codes für alle Tasten und Kombinationen finden Sie im Anhang.

Bemerkung • Wenn einer Funktionstaste mit dem KEY-Befehl Text zugeordnet wurde, dann wird, wenn der Benutzer sie drückt, nicht der Code für die Funktionstaste, sondern der assoziierte Text Buchstabe für Buchstabe mit INKEY$ aus dem Tastaturpuffer geholt.

• Tasten, an die mit Hilfe von ON KEY GOSUB eine Event-Handling-Routine gekoppelt wurde, werden bei INKEY$ ignoriert.

• In einem mit /D kompilierten Programm führt das Drücken von CTRL Break zum Programmabbruch, während es in einem ohne /D kompilierten anstandslos als String mit dem Inhalt CHR$(0) + CHR$(3) zurückgegeben wird.

Kompatibel PDS + VBWIN - Formen - Mathe -

Siehe auch INPUT$ (614), INP (614)

INP (Funktion)

Anwendung *x%* = INP(*kanal*)

Nutzen Liest ein Byte von einem bestimmten Hardware-Kanal (I/O-Port). Als *kanal* sind die Zahlen 0 bis 65.535 erlaubt.

Bemerkung • Mit der Benutzung dieser Funktion begeben Sie sich ziemlich nahe an die Chip-Ebene des PCs heran, und deshalb gibt es auch eine Anzahl von Schwierigkeiten damit. Sie können zwar prinzipiell mit den meisten Chips des Rechners direkt »Kontakt aufnehmen«, so zum Beispiel mit dem DMA-Controller, dem Interrupt-Controller, dem Sound-Chip, dem Coprozessor, den Disketten- und Festplattencontrollern, den Grafikkarten, seriellen und parallelen Schnittstellen, aber dazu müssen Sie erstens die Port-Adressen der Chips und zweitens deren spezifische »Sprache« kennen.

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch OUT (642), WAIT (690).

INPUT$ (Funktion)

Anwendung *x$* = INPUT$(*anzahl* [, [#]*dateinummer*])

Nutzen Liest eines oder mehrere Zeichen aus dem Tastaturpuffer oder einer Datei ein. *anzahl* ist die Anzahl der Zeichen, die gelesen werden sollen, und *dateinummer* ist die Nummer der Datei, aus der die Zeichen gelesen werden sollen. Wenn Sie *dateinummer* weglassen, werden die Zeichen von der Tastatur geholt. Im Unterschied zu INKEY$ erkennt INPUT$ jedoch keine Sonderzeichen, so daß INPUT$(1) bei Druck auf eine Taste mit erweitertem Zeichencode erst einmal CHR$(0) zurückgibt und erst im zweiten Anlauf den eigentlichen Code liefert. Außerdem wartet INPUT$, wenn nicht genügend Zeichen vorhanden sind, so lange, bis es die gewünschte Anzahl von Zeichen liefern kann, während INKEY$ einfach einen Leerstring liefert, wenn nichts da ist.

Beim Lesen aus einer Datei hat INPUT$ die gleiche Wirkung wie das Lesen eines Strings der Länge *anzahl* mit GET, mit dem Unterschied, daß INPUT$ auch für Random-Access- und für sequentielle Dateien benutzbar ist. Bei Random-Access-Dateien (OPEN FOR RANDOM) darf *anzahl* jedoch nicht größer sein als die beim Öffnen mit LEN angegebene Satzlänge.

Kompatibel PDS + VBWIN - Formen - Mathe -

Siehe auch INKEY$ (613), INPUT (615), LINE INPUT (624).

INPUT (Befehl)

Anwendung (1) INPUT [;] ["Aufforderung" ;|,] *variable(n)*
(2) INPUT *#dateinummer*, *variable(n)*

Nutzen Liest einen oder mehrere Variablenwerte von der Tastatur (Syntax (1)) oder einer Datei (Syntax (2)) ein.

Wenn Sie bei Syntax (1) gleich hinter INPUT ein Semikolon eingeben, hat das die Wirkung, daß der Cursor nach der Eingabe nicht in eine neue Zeile gesetzt wird. Danach kann eine Eingabeaufforderung in Anführungszeichen aufgeführt werden, gefolgt von einem Semikolon oder einem Komma. Bei Verwendung eines Semikolons wird zusätzlich ein Fragezeichen ausgegeben, bei Verwendung des Kommas nicht. Schließlich folgt obligatorisch mindestens ein Variablenname. Der Benutzer muß durch Kommata getrennt ebensoviele Variableninhalte eingeben, wie *variable(n)* angegeben sind. Die vom Benutzer eingegebenen Werte müssen zum Typ der jeweils korrespondierenden Variable in *variable(n)* passen. Ist das nicht der Fall oder gibt der Benutzer eine falsche Zahl von Inhalten an, erscheint die Meldung *Redo from start*, und die Eingabe muß wiederholt werden.

Syntax (2) ist etwas einfacher, da hier eine Eingabeaufforderung entfällt. Hier werden einfach die Dateinummer einer mit OPEN FOR INPUT geöffneten Datei und dahinter eine Liste von Variablen angegeben. Zeilenende und Komma in der Datei werden als der Beginn eines neuen Inhaltes angesehen.

Kompatibel PDS + VBWIN - Formen - Mathe *

Die Mathematik-Library wird eingebunden, wenn Sie nicht mit NOFLTIN.OBJ linken.

Siehe auch LINE INPUT (624), INPUT$(615).

INSTR (Funktion)

Anwendung *x%* = INSTR([*anfang%*,] *string$*, *teilstring$*)

Nutzen Stellt fest, ob der *teilstring$* im *string$* vorkommt. Falls ja, ist der Funktionswert die Stelle in *string$*, an dem *teilstring$* beginnt, sonst ist der Funktionswert 0. Üblicherweise wird ab der ersten Position in *string$* gesucht; wenn Sie jedoch die INTEGER-Variable *anfang%* angeben, wird erst ab dieser Position in *string$* nach *teilstring$* gesucht.

Kompatibel PDS x VBWIN x Formen x Mathe o

INT (Funktion)

Anwendung *x* = INT (*y*)

Nutzen Übergibt die größte Ganzzahl, die kleiner oder gleich dem Argument *y* ist, als Funktionswert.

Bemerkung • Der Unterschied zwischen CINT, FIX und INT ist der: CINT rundet ab 0,5 auf, ist symmetrisch bezüglich 0 und darf nur für Werte zwischen –32.768 und 32.767 benutzt werden. FIX und INT können für beliebige Zahlen benutzt werden und runden beide nicht; der Unterschied zwischen beiden ist, daß FIX wirklich einfach die Nachkommastellen abschneidet, während INT die nächstkleinere ganze Zahl sucht. Das ist bei positiven Zahlen äquivalent; bei negativen Zahlen jedoch sind die Zahlen, die FIX zurückgibt, größer oder gleich dem Argument (der Betrag ist kleiner oder gleich dem Argument), während die INT-Zahlen kleiner oder gleich dem Argument sind.

Beispiel CINT(–0.6) und INT(–0.6) sind beide –1, FIX(–0.6) ist 0.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch CINT (576), INT (616).

INTERRUPT, INTERRUPTX (Prozeduren)

Anwendung INTERRUPT *nummer%, eingaberegister, ausgaberegister*
INTERRUPTX *nummer%, eingaberegister, ausgaberegister*

Nutzen Ruft einen Interrupt auf. *nummer%* ist die Nummer des Interrupts; *eingaberegister* und *ausgaberegister* sind Variablen vom Typ RegType bzw. RegTypeX für INTERRPUTX. Diese Typen sind in der Datei VBDOS.BI deklariert.

Um Interruptaufrufe mit diesen Prozeduren auszuführen, muß die Quick Library VBDOS.QLB geladen sein (Switch /L beim Aufruf); bei der Erzeugung von EXE-Dateien muß VBDOS.LIB hinzugelinkt werden. VBDOS erledigt das allerdings automatisch.

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch Kapitel 24.

IOCTL$ (Funktion)

Anwendung *x$* = IOCTL$([#]*dateinummer*)

Nutzen Liest einen Status-String von einem Gerätetreiber. Eine Datei auf diesem Gerätetreiber muß geöffnet sein, und die Dateinummer dieser Datei muß als *dateinummer* übergeben werden.

Bemerkung • IOCTL$ kann nur mit Gerätetreibern benutzt werden, die IOCTL-Strings verarbeiten (z.B. einige CD-ROM-Laufwerkstreiber); die Blocktreiber (A:, B:, C: etc.) und die Gerätetreiber LPT*x*, COM*x*, SCRN, CONS und PIPE können das nicht.

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch IOCTL (328).

IOCTL (Befehl)

Anwendung IOCTL [#]*dateinummer*, *befehl$*

Nutzen Sendet einen Befehl zu einem Gerätetreiber. Eine Datei auf diesem Gerätetreiber muß geöffnet sein, und die Dateinummer dieser Datei muß als *dateinummer* übergeben werden.

Bemerkung • Es gilt das bei IOCTL$ Gesagte.

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch IOCTL$ (616).

KEY (zur Belegung der Funktionstasten) (Befehl)

Anwendung
```
KEY nummer, text$
KEY ON
KEY OFF
KEY LIST
```

Nutzen

Mit dem Befehl KEY *nummer*, *text$* können Sie eine der Funktionstasten (*nummer* ist 1–10 für F1-F10 und 30 bzw. 31 für F11 und F12) mit einem bis zu 15 Zeichen umfassenden Text belegen. Dadurch wird das Drücken auf die entsprechende Taste äquivalent zur Eingabe der einzelnen Buchstaben, die Sie ihr zugeordnet haben. Mit KEY *nummer*, "" können Sie die Belegung wieder aufheben; die Taste gibt dann, wenn sie mit INKEY$ abgefragt wird, ihren Zwei-Zeichen-Tastencode zurück.

KEY LIST gibt eine Liste der Tastendefinitionen auf dem Bildschirm aus, KEY ON schaltet die Funktionstastenanzeige in der untersten Bildschirmzeile an, KEY OFF schaltet sie aus.

Kompatibel PDS + VBWIN - Formen * Mathe -

Siehe auch INKEY$ (613), INPUT$ (614), INPUT (615), LINE INPUT (624).

KEY (für Key-Trapping) (Befehle)

Anwendung
```
KEY nummer, text$
KEY(nummer) ON
KEY(nummer) OFF
KEY(nummer) STOP
```

Nutzen

Der Befehl KEY(*nummer*) ON schaltet das Key-Trapping (eine Form von Event-Trapping) für die Taste *nummer* ein. Es muß zuvor ein ON KEY(*nummer*) GOSUB-Befehl ausgeführt werden. Von da an wird immer dann, wenn der Benutzer die betreffende Taste drückt, in die mit ON KEY GOSUB spezifizierte Routine gesprungen. KEY OFF schaltet das Key-Trapping ab, während KEY STOP es nur unterbindet. Wenn nach einem KEY STOP-Befehl wieder ein KEY ON folgt, wird nachträglich noch auf einen Tastendruck reagiert, den der Benutzer eventuell in der Zwischenzeit getätigt hat, während bei KEY OFF die Taste wieder ihren ganz gewöhnlichen Status erhält. Eine Taste, deren Key-Trapping nur mit KEY STOP unterbunden ist, wird bei INKEY$ und INPUT$ ignoriert, während eine Taste, deren Key-Trapping mit KEY OFF abgeschaltet wurde, bei INKEY$ oder INPUT$ ihren üblichen Tastencode zurückgibt.

Folgende *nummern* sind möglich:

nummer	*Taste(n)*
1 bis 10	Die Funktionstasten F1-F10
30, 31	Die Funktionstasten F11 und F12
11	Pfeil aufwärts
12	Pfeil links
13	Pfeil rechts
14	Pfeil abwärts
15–25	benutzerdefinierte Tasten

Wenn Sie bei den ON-, OFF- oder STOP-Befehlen als *nummer* 0 angeben, wirkt der Befehl auf alle Tasten.

Die Nummern 15–25 können frei einer beliebigen Taste oder Tastenkombination zugeordnet werden. Dazu ordnet man mit dem KEY *nummer*, *text$* der Tastennummer einen zwei Zeichen langen String zu, der die Tastenkombination beschreibt, für die man ein Key-Trapping einrichten will. Als erstes muß er einen Shift-Code und als zweites den Scan-Code der Taste enthalten. Die Scan-Codes finden Sie im Anhang; die Shift-Codes sind:

Code	*Taste*
0	keine der genannten Tasten ist gedrückt oder an
1	Shift ist gedrückt
4	CTRL ist gedrückt
8	ALT ist gedrückt

32	Num Lock ist an
64	Caps Lock ist an
128	zusätzliche Taste bei Tastatur mit 101 Tasten ist gedrückt (eine solche Tastatur hat zum Beispiel zwei Enter-Tasten; wenn die zusätzliche am Nummernblock gemeint ist, benutzt man den Shift-Code 128)

Sie können Shift-Codes addieren, um Tastenkombinationen zu erzielen, zum Beispiel Code 12 = CTRL+ALT, Code 33 = Shift+Num Lock an usw.

Bemerkung • Während einer Tastatureingabe mit INPUT$, INPUT oder LINE INPUT sind Key-Trapping-Routinen nicht aktiv.

Kompatibel PDS + VBWIN - Formen - Mathe -

Siehe auch ON *event* GOSUB (634), EVENT ON/OFF (600).

KILL (Befehl)

Anwendung KILL *dateiname$*

Nutzen Löscht eine oder mehrere Dateien. *dateiname$* ist die Dateibezeichnung; sie kann, wie beim DEL-Befehl von DOS, Laufwerks- und Pfadangabe enthalten. In *dateiname$* sind die Wildcard-Zeichen * und ? erlaubt.

KILL kann nicht auf Dateien angewandt werden, die gerade geöffnet sind; es wird sonst ein *File already open*-Fehler verursacht.

Kompatibel PDS x VBWIN x Formen x Mathe o

LBOUND (Funktion)

Anwendung *x* = LBOUND(*arrayname* [, *dimension*])

Nutzen Gibt die untere Dimensionierungsgrenze eines beliebigen Arrays zurück. Bei mehrdimensionalen Arrays kann mit *dimension* noch angegeben werden, die untere Grenze welcher Dimension gewünscht ist; wird keine Dimension angegeben, ermittelt LBOUND die untere Grenze der ersten Dimension.

Die untere Grenze ist der kleinstmögliche Index. Bei Dimensionierungen mit einer TO-Angabe, wie DIM a%(1900 TO 1999), ist die untere Grenze die Zahl links von TO; bei einfachen Dimensionierungen wie DIM a%(255) wird die untere Grenze eines Arrays durch OPTION BASE festgelegt. Standardmäßig ist sie 0, wenn aber vor dem DIM- ein OPTION BASE 1-Befehl ausgeführt wurde, ist sie 1.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch DIM (590), UBOUND (686), OPTION BASE (642).

LCASE$ (Funktion)

Anwendung *x$* = LCASE$(*y$*)

Nutzen Wandelt alle Großbuchstaben in Kleinbuchstaben um. Der Funktionswert der Funktion ist ein String, der dieselbe Länge hat wie das Argument *y$*, in dem jedoch alle von VBDOS erkannten Großbuchstaben in Kleinbuchstaben umgewandelt wurden. Folgende Umwandlungen werden durchgeführt:

von ASCII	von Zeichen	nach ASCII	nach Zeichen	von ASCII	von Zeichen	nach ASCII	nach Zeichen
65–90	A-Z	97–122	a-z	146	Æ	145	æ
128	Ç	135	ç	153	Ö	148	ö
142	Ä	132	*	154	Ü	129	ü
143	Å	134	å	165	Ñ	164	ñ

Kompatibel PDS * VBWIN + Formen + Mathe -

Das BASIC PDS unterstützt nur die 26 Standard-Großbuchstaben.

Siehe auch UCASE$ (686).

LEFT$ (Funktion)

Anwendung *x$* = LEFT$(*y$, zeichen*)

Nutzen Gibt die ersten *zeichen* Zeichen seines Arguments *y$* als neuen String zurück. Wenn *zeichen* länger als *y$* selbst ist, wird der ganze *y$* zurückgegeben; es wird dann weder ein Fehler generiert, noch füllt LEFT$ den String mit Leerzeichen auf.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch RIGHT$ (659), MID$ (629).

LEN (Funktion)

Anwendung *x%* = LEN(*variable*|*string$*)

Nutzen Mit Strings benutzt, ergibt LEN die Länge des angegebenen Strings (bei Strings mit fester Länge immer deren bei der Vereinbarung definierte Länge). Mit einer beliebigen anderen Variable verwendet, gibt LEN die Anzahl an Bytes zurück, die diese Variable im Speicher belegt.

Bemerkung • LEN wird gerne benutzt, um festzustellen, wieviele Bytes eine Variable eines bestimmten selbstdefinierten Typs belegt. Beachten Sie dabei, daß Sie nicht die Länge eines Typs, sondern immer nur die Länge einer Variable des Typs bestimmen können.

Kompatib PDS x VBWIN # Formen x Mathe o

LET (Befehl)

Anwendung [LET] *variable* = *ausdruck*

Nutzen Ordnet einer Variable den Wert von *ausdruck* zu. Das Befehlswort LET wird in der Regel nicht mehr benutzt, da das Gleichheitszeichen ausreicht. Wenn Sie den Befehl benutzen, um einer Variable den Inhalt einer anderen zuzuweisen, werden numerische Variablen automatisch gerundet bzw. String-Variablen gekürzt, falls das nötig ist. Es darf jedoch bei einer solchen Zuweisung nie der zulässige Bereich für *variable* überschritten werden.

Sie können auch einer Variable selbstdefinierten Typs den Inhalt einer anderen zuordnen, aber nur, wenn beide Variablen denselben Typ haben.

Bemerkung • Die Zuweisung einer Variable selbstdefinierten Typs zu einer mit davon abweichendem selbstdefinierten Typ ist mit LET nicht möglich, selbst wenn die beiden selbstdefinierten Typen sich lediglich im Namen unterscheiden. Ziehen Sie für solche Aufgaben LSET heran.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch LSET (628).

LINE (Befehl)

Anwendung LINE [[STEP](*x1, y1*)] – [STEP](*x2, y2*) [,[*farbe*] [,[B|BF] [,*linientyp*]]]

Nutzen Zeichnet eine Linie oder ein Rechteck im Grafikmodus.

(*x1, y1*) sind die Koordinaten des Anfangspunkts der Linie. Setzen Sie ein STEP davor, werden die angegebenen Werte relativ zur aktuellen Position des Grafikcursors genommen. Lassen Sie *x1* und *y1* völlig weg, werden die aktuellen Koordinaten des Grafikcursors eingesetzt.

(*x2, y2*) sind die Koordinaten des Endpunkts der Linie, wobei keine Rolle spielt, ob sie größer oder kleiner als die des Anfangspunkts sind. Ein STEP vor diesen Koordinaten macht sie relativ zu denen des Anfangspunkts der Linie.

farbe ist das Farbattribut für die neue Linie (über die Zuordnung von Farben zu Farbattributen siehe SCREEN, COLOR und PALETTE). Wenn *farbe* weggelassen wird, benutzt LINE die aktuelle Zeichenfarbe.

[B|BF] Wenn Sie die Optionen B oder BF angeben, wird statt einer einfachen Linie ein Rechteck gezeichnet; bei BF wird es ausgefüllt, B alleine zeichnet nur den Rahmen.

linientyp ist eine 16-Bit-Maske, die den Linientyp beschreibt. Wenn Sie *linientyp* nicht angeben, wird eine durchgezogene Linie gezeichnet (das entspricht dem *linientyp* –1). *linientyp* wirkt auch auf einfache Rechtecke, nicht jedoch auf gefüllte.

Bemerkung • Mit *linientyp* können Sie verschiedene Arten von gepunkteten oder gestrichelten Linien erzeugen. *linientyp* ist eine INTEGER-Zahl mit 16 Bits. Jedes der Bits hat den Wert $2^{\text{bitnummer}-1}$, nur für das Bit Nr. 16 müssen Sie statt 32.768 –32.768 verwenden. Der Linientyp für eine Linie, bei der jedes zweite Pixel ausgelassen wird (also eine gepunktete Line) wäre dementsprechend 2^0 + 2^2 + 2^4 + ... + 2^14.

Beachten Sie, daß »Pixel auslassen« nicht bedeutet, daß ein Pixel, das an dieser Stelle schon auf dem Bildschirm ist, gelöscht wird. Das führt dazu, daß eine gepunktete Linie zum Beispiel nicht als solche erkennbar ist, wenn Sie über eine durchgezogene gezeichnet wird.

• Wenn Sie eine Linie zeichnen wollen, die über den aktuellen Viewport oder über den Bildschirmrand hinausgeht, wird nur der sichtbare Teil gezeichnet; es wird kein Fehler erzeugt.

Kompatibel PDS + VBWIN * Formen - Mathe -

Siehe auch CIRCLE (576), PSET (653), PRESET (650), DRAW (593), SCREEN (663), PALETTE (643), COLOR (579).

LINE INPUT (Befehl)

Anwendung (1) LINE INPUT [;] ["*Aufforderung*";] *variable$*
(2) LINE INPUT *#dateinummer, variable$*

Nutzen Liest einen String von der Tastatur oder eine Zeile aus einer Datei.

LINE INPUT unterscheidet sich von INPUT dadurch, daß mit LINE INPUT nur eine einzelne Stringvariable gelesen werden kann. Aus Dateien wird dabei eine ganze Zeile, ungeachtet der Kommata und/oder Anführungszeichen, und von der Tastatur die gesamte Benutzereingabe gelesen. Ungleich INPUT, schneidet LINE INPUT auch keine führenden Leerzeichen ab.

LINE INPUT zeigt bei Tastatureingaben nie ein Fragezeichen an.

Kompatibel PDS + VBWIN * Formen * Mathe -

(In VBWIN und während Formen angezeigt werden sind nur dateibezogene LINE INPUT-Anweisungen zulässig.)

Siehe auch INPUT (615).

LOC (Funktion)

Anwendung *x&* = LOC(*dateinummer*)

Nutzen Gibt die Position des Dateizeigers in einer geöffneten Datei zurück. Bei RANDOM-Dateien ist das die Nummer des zuletzt gelesenen oder geschriebenen Datensatzes, bei BINARY-Dateien die Nummer des zuletzt gelesenen oder geschriebenen Bytes und bei sequentiellen Dateien (OPEN FOR INPUT, OUTPUT, APPEND) die Nummer des 128-Byte-Blocks, in dem der Zeiger steht (also der Quotient aus Dateizeiger und 128, aufgerundet auf die nächste ganze Zahl).

Für Kommunikationsschnittstellen gibt LOC die Zahl der Zeichen an, die bereits über die Leitung gekommen, aber noch nicht von BASIC eingelesen sind, die sich also im Puffer befinden (die Puffergröße kann beim OPEN-Befehl mit angegeben werden).

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch EOF (597), LOF (626), OPEN (632).

LOCATE (Befehl)

Anwendung LOCATE [*zeile*][,[*spalte*][,[*cursor*][,*start*[,*stop*]]]]

Nutzen Setzt den Textcursor an eine beliebige Position, schaltet ihn ein oder aus, und stellt seine Größe ein.

zeile und *spalte* geben die neue Position für den Cursor an. Bei Weglassen (einer) der Angaben wird die jeweils alte Zeile bzw. Spalte beibehalten.

cursor gibt an, ob der Cursor angezeigt werden soll oder nicht (ein Wert ungleich 0 bedeutet anzeigen).

start und *stop* stellen die Cursorgröße ein. *start* ist die Linie, bei der der Cursor beginnt, *stop* diejenige, an der er aufhört. Wird *stop* weggelassen, hat der Cursor nur eine Zeile. Im Real Mode können Werte von 0–7 benutzt werden (0–13 für die Herculeskarte). Die angegebenen Werte werden vom BIOS der Grafikkarte an die jeweilige tatsächliche Auflösung angepaßt, so daß LOCATE ,,,7 den Cursor immer auf die unterste Zeile setzt (ausgenommen die Herculeskarte).

Kompatibel PDS + VBWIN - Formen - Mathe -

Verwenden Sie bei Formen die CurrentX- und CurrentY-Eigenschaft.

Siehe auch CLS (578).

LOCK, UNLOCK (Befehle)

Anwendung LOCK [#]*dateinummer* [, *anfang* [TO *ende*]]
...
UNLOCK [#]*dateinummer* [, *anfang* [TO *ende*]]

Nutzen Verhindert, daß irgendein anderer Prozeß im Netzwerk auf die Datei zugreift.

dateinummer ist die Nummer der geöffneten Datei, die vor fremdem Zugriff geschützt werden soll. Wird weiter nichts angegeben (oder handelt es sich um eine sequentielle Datei), schützt LOCK die gesamte Datei. Für RANDOM- und BINARY-Dateien kann auch ein bestimmter Teilbereich geschützt werden. Wenn Sie die Zahl *anfang* alleine angeben, wird nur dieser Datensatz (bei RANDOM-Dateien) bzw. dieses Byte (bei BINARY-Dateien) geschützt; setzen Sie noch TO *ende* hinzu, werden alle Datensätze bzw. Bytes von *anfang* bis *ende* geschützt.

Jeder LOCK-Befehl muß mit einem völlig gleichlautenden UNLOCK-Befehl aufgehoben werden, bevor die Datei geschlossen wird.

Bemerkung
- LOCK und UNLOCK funktionieren erst ab der DOS-Version 3.1, und auch dann nur, wenn zuvor das DOS-Programm SHARE.EXE zur Netzwerkunterstützung geladen wurde. Möglichkeiten, DOS-Version und SHARE-Zustand festzustellen, finden Sie in Kapitel 26.
- LOCK und UNLOCK können nur auf gewöhnliche Dateien, nicht auf ISAM-Dateien oder Geräte (wie LPT1:, COM1: etc) angewandt werden.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch OPEN (632).

LOF (Funktion)

Anwendung *x&* = LOF(*dateinummer*)

Nutzen Gibt die Größe einer geöffneten Datei in Byte zurück. Bei Kommunikationsschnittstellen müssen Sie LOC benutzen, um die Anzahl der im Puffer wartenden Zeichen zu erfahren; LOF gibt hier die Anzahl freier Zeichen im Ausgabepuffer zurück, das heißt, wenn LOF gleich Null ist, können keine Zeichen mehr auf die Kommunikationsschnittstelle ausgegeben werden, solange nicht die wartenden abgesandt sind.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch EOF (597), LOC (624).

LOG (Funktion)

Anwendung *x* = LOG (*zahl*)

Nutzen Errechnet den natürlichen Logarithmus einer Zahl. *zahl* muß größer als 0 sein.

Bemerkung • Logarithmen zu einer anderen Basis als *e* können errechnet werden, indem man den natürlichen Logarithmus einer Zahl durch den natürlichen Logarithmus der gewünschten Basis dividiert; so kann man den Logarithmus zu Basis 10 einer Zahl *x* mit der Formel Log10 = LOG(x) / LOG(10) berechnen.

Kompatibel PDS + VBWIN + Formen + Mathe +

Siehe auch EXP (601).

LPOS (Funktion)

Anwendung *x* = LPOS(*druckernummer*)

Nutzen Gibt die Anzahl Zeichen zurück, die seit dem letzten Zeilenanfang (oder CR-Zeichen CHR$(13)) an einen Drucker geschickt wurden. Mit *druckernummer* können Sie den Drucker auswählen: 0 oder 1 bedeuten LPT1:, 2 entspricht LPT2: und 3 heißt LPT3:. In der Praxis ist diese Funktion kaum brauchbar, da sie auch nicht druckbare Zeichen mitzählt.

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch WIDTH (691).

LPRINT (Befehl)

Anwendung LPRINT [USING *format$*;] [*ausdruck* [,|; *ausdruck*]...[,|;]]

Nutzen Gibt Daten auf der Druckerschnittstelle LPT1: aus. Alles weitere siehe PRINT.

Bemerkung • Benutzen Sie statt LPRINT besser OPEN "LPT1:" FOR OUTPUT AS #x ... PRINT #x, denn dann können Sie recht einfach zum Beispiel auf LPT2: umstellen. LPRINT und OPEN "LPT1:" sollten nicht beide im gleichen Programm vorkommen.

• Wenn Sie nicht mit WIDTH eine andere Zeilenbreite einstellen, nimmt LPRINT 80 an. Dann wird nach 80 in einer Zeile gesendeten Zeichen immer ein CR/LF (CHR$(13) und CHR$(10) an den Drukker gesendet, um eine neue Zeile anzufangen.

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch PRINT (651), LPOS (627), WIDTH (691).

LSET (Befehl)

Anwendung (1) LSET *string1$* = *string2$*
(2) LSET *variable1* = *variable2*

Nutzen In der ersten Syntax wird LSET benutzt, um den *string2$* in *string1$* linksbündig unterzubringen. Der Unterschied zwischen LSET string1$ = string2$ und string1$ = string2$ ist, daß bei LSET die Länge von *string1$* auf jeden Fall erhalten bleibt. *string2$* wird gekürzt oder rechts mit Leerzeichen aufgefüllt, um genau in *string1$* zu passen. Technisch heißt das, das bei LSET der Stringdeskriptor von *string1$* nicht geändert wird und der LSET-Befehl deshalb (unter Umständen signifikant) schneller ist als die Zuweisung mit =.

Mit der zweiten Syntax ist es möglich, zwei Variablen von verschiedenen selbstdefinierten Datentypen einander zuzuordnen. Wenn der Datentyp beider Variablen die gleiche Struktur hat, ist das kein Problem, unterscheidet sich jedoch die Struktur, kann das Resultat ein unerwünschtes sein, da einfach Byte für Byte kopiert wird. Im Gegensatz zur ersten Syntax wird hier jedoch nicht mit Leerzeichen aufgefüllt, sondern wenn Sie einen kürzeren Datentyp in einen längeren hineinkopieren, bleibt der Rest des längeren unberührt.

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch RSET (660)

LTRIM$ (Funktion)

Anwendung *x$* = LTRIM$(*y$*)

Nutzen LTRIM$ gibt als Funktionswert den String zurück, der ihr als Argument übergeben wurde, entfernt jedoch zuvor alle führenden Leerzeichen (CHR$(32)).

Bemerkung • Beachten Sie, daß Strings mit fester Länge vor ihrer ersten Verwendung nicht Leerzeichen, sondern CHR$(0)-Zeichen enthalten, die sich im Aussehen von Leerzeichen nicht unterscheiden, die LTRIM$ jedoch nicht entfernt.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch RTRIM$ (661), PRINT (651).

MID$ (Funktion und Befehl)

Anwendung (1) *x$* = MID$(*stringausdruck*, *anfang* [, *laenge*])
(2) MID$(*x$*, *anfang* [, *laenge*]) = *stringausdruck*

Nutzen In Syntax (1) gibt MID$ einen Teil von *stringausdruck* zurück. *anfang* ist die Position, ab der *stringausdruck* zurückgegeben werden soll, in *laenge* ist die Anzahl der Zeichen enthalten, die von der Position *anfang* aus übergeben werden. Ist *anfang* + *laenge* größer als die Länge von *stringausdruck* oder wird *laenge* ganz weggelassen, übergibt MID$ alle Zeichen aus *stringausdruck* von Position *anfang* bis zum Ende. Wenn *anfang* größer als die Länge von *stringausdruck* ist, wird ein Leerstring zurückgegeben.

Bei Syntax (2) funktioniert MID$ genau umgekehrt. Ein Teil eines Strings wird durch einen anderen ersetzt. Die Parameter sind dabei gleich wie in der Syntax 1, mit der Restriktion, daß *anfang* hier keinesfalls größer als die Länge von *x$* sein darf. Von der Position *anfang* an werden alle Zeichen in *x$* durch Zeichen aus *stringausdruck* ersetzt, so lange, bis entweder *laenge* Zeichen ersetzt wurden, bis das Ende von *x$* erreicht wurde oder *stringausdruck* keine Zeichen mehr bereithält. *x$* kann also durch eine MID$-Operation niemals länger oder kürzer werden. Die Syntax (2) darf nicht verwendet werden, um Objekteigenschaften zu verändern.

Sowohl in Syntax (1) als auch in Syntax (2) darf *anfang* nie kleiner als 1 werden.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch LEFT$ (621), RIGHT$ (659).

MINUTE (Funktion)

Anwendung *x%* = MINUTE(*zeitcode#*)

Nutzen Ermittelt die zum angegebenen *zeitcode#* gehörige Minute (0–59).

Kompatibel PDS * VBWIN + Formen + Mathe +

Siehe auch HOUR (611), SECOND (669).

MKDIR (Befehl)

Anwendung MKDIR *verzeichnisname$*

Nutzen Erstellt ein neues Verzeichnis auf dem Datenträger. *verzeichnisname$* ist der komplette Name dieses Verzeichnisses und darf nicht länger als 63 Zeichen sein.

Bemerkung • MKDIR ist in der Funktion identisch mit dem gleichnamigen DOS-Befehl.

Kompatibel PDS + VBWIN * Formen + Mathe -

Siehe auch CHDIR (575), CURDIR$ (583), RMDIR (660).

MKx$ (Funktionen)

Anwendung *x$* = MKI$(*y%*)
*x$* = MKL$(*y&*) (seit QB 4.0)
*x$* = MKS$(*y!*)
*x$* = MKD$(*y#*)
*x$* = MKC$(*y@*) (seit PDS 7.0)

Nutzen Diese Funktionen dienen dazu, Zahlen in Code-Strings umzuwandeln. Wenn BASIC Zahlen in Random-Access-Dateien schreibt (ob mit FIELD oder innerhalb eines TYPE-Records), werden sie so verschlüsselt. MKI$ erzeugt einen 2-Byte-String, MKL$ und MKS$ haben vier Bytes, MKD$ und MKC$ acht. MKD$ und MKS$ benutzen die IEEE-Codierung für Fließkommazahlen.

Bemerkung • Die zugehörigen CV*x*-Funktionen bewirken das Gegenteil.

• Der Compiler-Switch /MBF beeinflußt die Funktionen MKS$ und MKD$.

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch MK*x*MBF$ (630), CV*x* (584), FIELD (602).

MKxMBF$ (Funktionen)

Anwendung *x$* = MKSMBF$(*y!*)
*x$* = MKDMBF$(*y#*)

Nutzen Diese beiden Funktionen codieren wie ihre Nachfolger MKS$ und MKD$ SINGLE- bzw. DOUBLE-Zahlen in Zwei- oder Vier-Byte-Strings, allerdings nicht im IEEE-Verfahren, sondern im Microsoft-Binärformat, das MKS$ und MKD$ vor QuickBASIC 3.0 noch benutzten (siehe Bemerkung bei CV*x*MBF).

Kompatibel PDS + VBWIN - Formen + Mathe +

Siehe auch CV*x*MBF (584), MK*x*$ (630), FIELD (602).

MONTH (Funktion)

Anwendung *x%* = MONTH(*zeitcode#*)

Nutzen Ermittelt den zum angegebenen *zeitcode#* gehörigen Tag des Monats (1–31).

Kompatibel PDS * VBWIN + Formen + Mathe +

Siehe auch DAY (587), YEAR (694).

NAME (Befehl)

Anwendung (1) NAME *dateiname1$* AS *dateiname2$*
(2) NAME *directory1$* AS *directory2$*

Nutzen NAME ist ein – verglichen mit dem DOS-Befehl REN – ziemlich mächtiger Befehl zum Umbenennen und Verschieben von Dateien. In der Syntax (1) kann eine Datei umbenannt werden. *dateiname1$* muß der Name einer vorhandenen Datei sein, *dateiname2$* darf noch nicht vorhanden, muß aber gültig sein. *dateiname2$* muß auf demselben Laufwerk wie *dateiname1$* sein. Wenn *dateiname2$* eine andere Directory-Angabe als *dateiname1$* enthält, wird die Datei in das neue Directory »verschoben«. In Syntax (2) werden statt Dateinamen Directorynamen angegeben; dadurch kann ein ganzes Verzeichnis umbenannt werden. Hierbei darf allerdings nicht verschoben werden; NAME "\FRED\DATA" AS "\DOS\TEST" wäre zum Beispiel nicht möglich, weil das Directory DATA nicht nur einen neuen Namen bekommen, sondern auch in ein anderes Directory verschoben werden müßte. NAME "\FRED\DATA" AS "\FRED\TEST" hingegen wäre zulässig.

Kompatibel PDS x VBWIN x Formen x Mathe o

NOW (Funktion)

Anwendung *x#* = NOW

Nutzen Ermittelt den vollständigen Zeitcode des Augenblicks, in dem sie aufgerufen wird. NOW bedient sich dazu der Systemuhr. Statt x# = NOW könnte man auch schreiben: x# = DateValue(DATE$) + TimeValue(TIME$).

Bemerkung • Mit Hilfe von NOW und der FORMAT$ ist es ein Leichtes, das aktuelle Datum im deutschen Format auszugeben: Der Befehl dafür heißt PRINT FORMAT$(Now, "d.m.yy").

Kompatibel PDS * VBWIN + Formen + Mathe +

Siehe auch DATEVALUE (586), TIMEVALUE (684), FORMAT$ (604).

OCT (Funktion)

Anwendung *x$* = OCT$(*y*)

Nutzen OCT$ gibt ihr Argument in oktaler Schreibweise zurück. Obwohl eine Oktalzahl nur Ziffern enthält, wird ein String erzeugt.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch HEX$ (320).

ON ERROR (Befehl)

Anwendung ON [LOCAL] ERROR GOTO *zeilennummer/-label*
ON [LOCAL] ERROR RESUME NEXT
ON [LOCAL] ERROR GOTO 0

Nutzen Schaltet eine Fehlerbehandlungsroutine, einen Error-Handler, ein oder aus.

Das Wort LOCAL bedeutet in jedem der Befehle, daß ein lokaler Error-Handler gemeint ist. Es kann nur innerhalb einer Prozedur oder Funktion benutzt werden.

ON ERROR GOTO *zeilennummer/-label* setzt den globalen Error-Handler für das Modul, in dem der Befehl auftritt. Jeder vorhergehende ON ERROR GOTO-Befehl in diesem Modul verliert seine Bedeutung (nicht jedoch ON LOCAL ERROR). Von nun an wird bei jedem Fehler zur angegebenen Zeile gesprungen, die sich im Modulcode des Moduls befinden muß. Globale Error-Handler und der ON ERROR-Befehl ohne LOCAL können nur in Codemodulen, niemals in Formmodulen existieren, da Formmodule keinen Code auf Modulebene zulassen.

ON ERROR RESUME NEXT setzt ebenfalls jeden vorherigen ON ERROR GOTO-Befehl außer Kraft; von diesem Zeitpunkt an werden bei Auftreten eines Fehlers lediglich die ERR- und ERL-Variablen gesetzt, das Programm aber normal fortgeführt.

ON ERROR GOTO 0 schließlich setzt den gerade aktiven globalen Error-Handler außer Kraft. Da immer nur ein globaler Error-Handler aktiv sein kann, bedeutet das, daß in dem Modul nach ON ERROR GOTO 0 kein globaler Error-Handler mehr aktiv ist und jeder Fehler sofort zum Programmabbruch führt (mit Ausnahme der Fehler, die von lokalen Error-Handlern abgefangen werden).

Für lokale Error-Handler gibt es ebenfalls die drei oben genannten Befehle, jeweils mit dem Schlüsselwort LOCAL. Lokale Error-Handler sind nur in Prozeduren und Funktionen zulässig, und sie werden automatisch abgeschaltet, wenn die Prozedur verlassen wird.

Ein lokaler Error-Handler ist stärker als ein globaler, so daß eine Prozedur mit lokalem Error-Handler nur diesen und nicht den globalen benutzt. Durch lokale Error-Handler kann eine Prozedur also völlig unabhängig von ihrer Umgebung werden, weil sie nicht einmal mehr den globalen Error-Handler benutzen muß.

Zwar kann pro Prozedur nur ein lokaler Error-Handler aktiv sein, aber es ist möglich, daß alle Prozeduren einen eigenen Error-Handler haben und sich gegenseitig aufrufen, wodurch die Error-Handler ineinander geschachtelt werden. Der zuletzt aktivierte lokale Error-Handler hat immer Gültigkeit. Wenn in diesem Error-Handler selbst ein Fehler auftritt, wird der nächsthöhere Error-Handler aufgerufen usw., bis man beim globalen Error-Handler (falls existent) ankommt. Wenn in diesem selbst ein Fehler auftreten sollte, generiert BASIC eine Fehlermeldung, und das Programm bricht ab.

Ein lokaler Error-Handler kann nur von der Prozedur deaktiviert werden, die ihn eingerichtet hat, nicht aber von Prozeduren, die aus dieser heraus aufgerufen wurden.

Im Gegensatz dazu kann der globale Error-Handler von allen Prozeduren gleichberechtigt ein- oder ausgeschaltet oder verändert werden. Prozeduren und Funktionen, die in einem anderen Modul stehen als der globale Error-Handler, können ihn zwar deaktivieren, abr nicht einschalten, weil der ON ERROR GOTO-Befehl dann das Zeilenlabel nicht findet.

Siehe auch ERL, ERR (598), ERDEV (598), ERDEV$ (598).

ON event GOSUB (Befehl)

Anwendung ON *event* GOSUB *zeilennummer/-label*

Nutzen Etabliert eine Event-Trapping-Routine für ein bestimmtes Ereignis (*event*). Das Trapping für das entsprechende Ereignis muß mit *event* ON aktiviert werden, bevor die Event-Trapping-Routine wirklich benutzt wird. »Event Trapping« ist ein Programmierkonzept, das teilweise im Widerspruch zum neuen ereignisgesteuerten Programmierkonzept steht; außer ON KEY sind jedoch alle Varianten auch zusammen mit Formen einsetzbar.

Wenn Sie für *zeilennumer/-label* 0 einsetzen, wird die Event-Trapping-Routine deaktiviert. Folgende *events* sind möglich:

COM(*n*)	Ruft die Trapping-Routine auf, wenn am durch *n* spezifizierten Kommunikations-Port Zeichen anliegen.
KEY(*n*)	Ruft die Trapping-Routine auf, wenn die Taste *n* gedrückt wird. Zu möglichen Tastennummern siehe KEY.
PEN	Ruft die Trapping-Routine auf, wenn mit dem Lichtstift (Lightpen) auf eine Bildschirmposition gezeigt wird.
PLAY(*n*)	Ruft die Trapping-Routine auf, wenn die *n*te Note gespielt wurde und nun nur noch *n*–1 Noten zu spielen sind. ON PLAY GOSUB wirkt nur, wenn der PLAY-Befehl im Hintergrund arbeitet (siehe Befehl "MB" bei PLAY). *n* darf nicht größer als 32 sein.
STRIG(*n*)	Ruft die Trapping-Routine auf, wenn ein Feuerknopf an einem der beiden Joysticks gedrückt wurde. *n* ist 0 für das Drücken der ersten Taste am ersten Joystick, 2 für die erste Taste am zweiten, 4 für die zweite am ersten und 6 für die zweite Taste am zweiten Joystick.
TIMER(*n*)	Ruft die Trapping-Routine alle *n* Sekunden auf. *n* darf nicht größer als 86.400 und muß ganzzahlig sein.

UEVENT Ruft die Trapping-Routine auf, wenn ein UEVENT, ein selbstdefiniertes Ereignis auftritt. Von BASIC aus können Sie dieses Ereignis künstlich mit CALL SetUEvent eintreten lassen; eigentlich ist die Einrichtung jedoch dafür gedacht, mittels Assembler oder einer anderen Sprache ein BASIC-Ereignis auslösen zu können. SetUEvent hat keine Argumente und kann aus jeder anderen Sprache einfach aufgerufen werden.

Kompatibel PDS + VBWIN - Formen - Mathe -

Siehe auch COM (580), KEY (618), PEN (646), SIGNAL (673), STRIG (679), TIMER (683), UEVENT (687), EVENT ON/OFF (600).

ON...GOSUB, ON...GOTO (Befehle)

Anwendung ON *variable* GOTO *zeilennummern/-labels*
ON *variable* GOSUB *zeilennummern/-labels*

Nutzen Verzweigen in eine von mehreren angegebenen Zeilen abhängig vom Wert der angegebenen *variable*. *zeilennummern/-labels* ist jeweils eine Liste von bis zu 60 Zeilennummern und/oder -labels, die durch Komma getrennt werden.

Wenn der Wert von *variable* 0 oder größer als die Anzahl der angegebenen Zeilennummern bzw. -labels ist, wird überhaupt nicht verzweigt; bei einem positiven Wert wird in die entsprechende der angegebenen Zeilen verzweigt (bei 1 in die erste genannte, bei 2 in die zweite genannte usw.).

Negative Werte für *variable* führen zu einem *Funktionsaufruf unzulässig*.

Bemerkung • SELECT CASE eignet sich in den meisten Fällen besser, um unterschiedliche Verzweigungen auszulösen.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch GOSUB (610), GOTO (610), SELECT CASE (670).

OPEN (Befehl)

Anwendung (1) OPEN *dateiname$* [FOR *modus*] [ACCESS *zugriff*] [*fremdzugriff*] AS [#]*dateinummer* [LEN=*satzlaenge*]

(2) OPEN *schnittstelle$* [FOR *modus*] AS [#]*dateinummer* [LEN=*satzlaenge*]

Nutzen

Öffnet eine Datei oder Schnittstelle zur Ein- und/oder Ausgabe von Daten.

Die Syntax (1) wird für gewöhnliche Dateien und für KYBD:, SCRN:, LPT*x*: und CONS: benutzt, während die Syntax (2) ausschließlich verwendet wird, um Kommunikationsschnittstellen zu öffnen.

VBDOS unterstützt außerdem eine dritte Syntax, die von ihrer Funktion her eine Untermenge der ersten Syntax darstellt, um die Kompatibilität mit älteren Versionen von BASIC zu wahren. Diese veraltete Syntax wird hier nicht mehr beschrieben.

Syntax (1)

dateiname$ ist ein gültiger Dateiname, der Laufwerk und Pfad einschließen darf, oder der Name eines Gerätes wie CONS: oder LPT1:. *modus* ist einer der folgenden Modi:

RANDOM Die Datei wird für »wahlfreien Zugriff« geöffnet, zum Lesen und Schreiben beliebiger Datensätze. Bei RANDOM-Dateien spielt die Angabe LEN=*satzlaenge* eine große Rolle, da auf RANDOM-Dateien nur satzweise zugegriffen werden kann. Standard für *satzlaenge* ist hier 128 Bytes. RANDOM-geöffnete Datein können unter mehreren Dateinummern gleichzeitig geöffnet werden.

BINARY Die Datei wird im Binärmodus geöffnet, der das Lesen und Schreiben beliebiger Zeichen erlaubt. Im Gegensatz zu RANDOM ist die Satzlänge bei BINARY-Dateien redundant (sie wird ignoriert), da hier ohnehin byteweise vorgegangen wird. BINARY-geöffnete Dateien können unter mehreren Dateinummern gleichzeitig geöffnet werden.

INPUT Die Datei wird zum sequentiellen Lesen geöffnet. INPUT-geöffnete Dateien können unter mehreren Dateinummern gleichzeitig geöffnet werden.

OUTPUT Die Datei wird zum sequentiellen Schreiben geöffnet; falls sie schon existiert, wird sie überschrieben.

APPEND Die Datei wird zum sequentiellen Schreiben geöffnet; falls sie schon existiert, werden die neuen Daten an sie angehängt.

In den drei sequentiellen Modi ist *satzlaenge* die Länge des Dateipuffers. Je kleiner dieser, desto häufiger muß beim Lesen oder Schreiben auf die Festplatte zugegriffen werden. Standard ist 512 Bytes.

Der Zugriffsmodus, der mit ACCESS *zugriff* angegeben wird, ist nur für die beiden erstgenannten Dateiarten interessant, da nur hier zum Zeitpunkt des Öffnens der Datei unklar ist, welcher Art der Zugriff sein wird. *zugriff* kann entweder READ, WRITE oder READ WRITE sein und so den Zugriff auf die Datei einschränken. Das ist in Netzwerkumgebungen sinnvoll, in denen vielleicht ein anderer Prozeß dieselbe Datei geöffnet hat und das Schreiben in die Datei so lange verboten ist. Dann würde ein gewöhnliches OPEN FOR RANDOM einen Fehler (*Permission denied*) ergeben, weil die Datei nicht beschreibbar ist. Ein OPEN FOR RANDOM ACCESS READ hingegen würde nicht beanstandet.

Die ACCESS *zugriff*-Klausel kann nur mit DOS 3.0 und neueren Versionen benutzt werden; außerdem muß zuvor das Programm SHARE.EXE zur Netzwerkunterstützung geladen worden sein, da die Verwendung von ACCESS *zugriff* sonst zu einem *Feature unavailable*-Fehler führt.

Während *zugriff* also den eigenen Zugriff auf die Datei ankündigt, kann mit *fremdzugriff* eingestellt werden, welche Zugriffe auf die Datei von anderen Prozessen durchgeführt werden dürfen, solange sie geöffnet ist. Möglich sind hier SHARED, LOCK READ, LOCK WRITE und LOCK READ WRITE. SHARED bedeutet, daß jeder andere Prozeß nach Belieben in die Datei schreiben und aus ihr lesen darf. Die drei LOCK-Befehle verbieten anderen Prozessen das Lesen, das Schreiben, oder beides. Sie können nur dann wirksam werden, wenn nicht bereits ein anderer Prozeß Zugriff auf die Datei angemeldet hat, der ihm hiermit verboten würde.

Wenn aus »Netzwerk-Gründen« ein OPEN-Befehl fehlschlägt, wenn also ein anderer Prozeß in irgendeiner Weise Zugriff auf die Datei hat, der mit dem OPEN-Befehl in Konflikt kommt, wird ein *permission denied*-Fehler erzeugt.

dateinummer ist eine Nummer zwischen 1 und 255, unter der sich folgende Befehle auf die geöffnete Datei beziehen können. Zur maximalen Anzahl geöffneter Dateien siehe FREEFILE.

Syntax (2)

Diese Syntax wird benutzt, um eine Kommunikationsschnittstelle zur Ein- und Ausgabe zu öffnen. *schnittstelle$* enthält alle benötigten Informationen und hat die Syntax:

"COMn: baud,parität, databits, stopbits[,optionen]"

n ist die Nummer der Kommunikationsschnittstelle (1 oder 2); *baud* ist die Übertragungsgeschwindigkeit, die Baudrate, die angibt, wieviele Bits in einer Sekunde gesendet/empfangen werden. *baud* kann 75, 110, 150, 300, 600, 1200, 1800, 2400 oder 9600 sein. Höhere Geschwindigkeiten sind nicht möglich. *parität* kann entweder N (keine), O (ungerade), E (gerade), S (»space«) oder M (»mark«) sein; üblicherweise verwendet man N, und N ist auch die einzige Parität, die in Kombination mit 8 Datenbits zulässig ist. *databits* ist die Anzahl der Datenbits pro Byte (5, 6, 7 oder 8). Wenn Sie nicht 8 verwenden, können Sie bestimmte Zeichen nicht sen»en und empfangen. *stopbits* ist die Anzahl der Stopbits pro Byte (1, 1,5 oder 2). 1 ist die übliche Einstellung.

optionen wiederum ist eine Kette von bis zu neun durch Komma getrennten Befehlen, die das Verhalten der Schnittstelle beeinflussen. Folgende Befehle sind möglich:

BIN Der Standardmodus. Hier können alle Zeichen unverändert gesendet und empfangen werden. Zeilen werden, ungeachtet der WIDTH-Einstellung, niemals unterbrochen. Wenn BIN angegeben ist, wird LF ignoriert.

ASC ASCII-Modus. Das ASCII-Zeichen 26 (EOF) wird als Dateiende-Zeichen verstanden; das ASCII-Zeichen 9 (TAB) wird in entsprechend viele Leerzeichen (mindestens 1, maximal 8) umgewandelt. Beim CLOSE-Befehl wird als letztes Zeichen ein CHR$(26) an die Schnittstelle geschickt. Eine Zeile, die breiter als 80 Zeichen oder die mit WIDTH neu gesetzte Zeilenbreite ist, wird durch ein CR (Carriage Return, ASCII 13) unterbrochen.

LF (als Ergänzung zu ASC) Hinter jedem Carriage Return-Zeichen wird automatisch zusätzlich ein Linefeed-Zeichen (ASCII 10) gesendet, so daß die Ausgabe auf Druckern möglich ist.

RB *n* setzt den Empfangspuffer auf eine Größe von *n* Bytes. Der Empfangspuffer ist normalerweise 512 Bytes groß (für beide Schnittstellen zusammen), wenn nicht mit /C beim Kompilieren ein anderer Wert angegeben wurde. Je größer der Empfangspuffer, desto seltener müssen Sie die Daten an der Schnittstelle abrufen, weil dann mehr Daten zwischengespeichert werden können. Maximum für *n* ist 32.767.

TB *n* setzt den Sendepuffer auf eine Größe von *n* Bytes. Standard ist dieselbe Größe wie der Empfangspuffer. Bei langsamen Geschwindigkeiten (300 Baud oder weniger) brauchen Sie einen großen Sendepuffer, wenn Sie viele Daten auf einmal auf die Schnittstelle schreiben wollen, weil die Schnittstelle dann Daten zwischenspeichern muß. Maximum für *n* ist 32.767.

RS verhindert, daß die RTS-Leitung beim OPEN-Befehl unter Spannung gesetzt wird.

CD *n* verursacht einen Timeout-Fehler (ERDEV = 130), wenn die DCD-Leitung (Carrier Detect) der Schnittstelle *n* Millisekunden lang keine Spannung führt. Wenn CD oder der Parameter *n* weggelassen wird oder *n* 0 ist, wird die DCD-Leitung nicht beachtet. *n* liegt zwischen 0 und 65.535.

CS *n* verursacht einen Timeout-Fehler (ERDEV = 128), wenn die CTS-Leitung (Clear To Send) der Schnittstelle *n* Millisekunden lang keine Spannung führt. Wenn CS oder der Parameter *n* weggelassen wird, wird eine Sekunde gewartet; ist *n* 0, wird der Status der CTS-Leitung ignoriert. *n* liegt zwischen 0 und 65.535.

DS *n* verursacht einen Timeout-Fehler (ERDEV = 129), wenn die DSR-Leitung (Data Set Ready) der Schnittstelle *n* Millisekunden lang keine Spannung führt. Wenn DS oder der Parameter *n* weggelassen wird, wird eine Sekunde gewartet; ist *n* 0, wird der Status der DSR-Leitung ignoriert. *n* liegt zwischen 0 und 65.535.

OP *n* verursacht einen Timeout-Fehler, wenn nach *n* Millisekunden nicht alle Kommunikationsleitungen aktiv sind. *n* liegt zwischen 0 und 65.535. Bei 0 wird der Status der Leitungen ignoriert; wird OP weggelassen, ist der Standardwert das Zehnfache des Wertes von CD oder DS (des höheren von beiden). Wird OP angegeben, aber *n* weggelassen, wartet OP 10 Sekunden lang.

modus beim OPEN-Befehl der Syntax 2 kann entweder INPUT, OUTPUT oder RANDOM sein. RANDOM wird auch benutzt, wenn kein *modus* angegeben wird. Nur bei RANDOM ist die *satzlaenge* von Bedeutung; RANDOM-geöffnete Schnittstellen können wie mit RANDOM geöffnete Dateien behandelt werden, wobei hier natürlich Daten, die hineingeschrieben werden, sofort über die Schnittstelle geschickt werden und dann – im Gegensatz zu einer echten Random-Access-Datei – nicht mehr verfügbar sind.

dateinummer ist eine gültige Dateinummer (siehe erste Syntax).

Bemerkung

• Mit dem OPEN-Befehl der ersten Syntax können auch eine Anzahl von Geräten geöffnet werden:

Die Drucker LPT*x*: (nur OUTPUT). Eine Ausgabe in eine solche Datei entspricht der Ausgabe mit LPRINT. Wenn Sie an den Gerätenamen ein BIN anfügen (LPT*x*:BIN), wird nicht an jedem Zeilenende ein Carriage Return-Zeichen gesendet. Der Effekt ist derselbe, wenn Sie den Drucker ohne BIN öffnen und alle PRINT-Befehle auf den Drucker mit einem Semikolon beenden.

Die Tastatur KYBD: (nur INPUT). Sie können von der Tastatur Zeichen lesen. Im Gegensatz zu der Zeicheneingabe mit LINE INPUT, INPUT, INPUT$ oder INKEY$ kann die Eingabe für ein OPEN "KYBD:" FOR INPUT *nicht* durch das DOS-Umleitungszeichen < aus einer Datei gesteuert werden, sondern sie muß zwangsläufig von der Tastatur kommen.

Die Konsole CONS: und den Bildschirm SCRN: (bei beiden ist nur OUTPUT sinnvoll). Zum Vergleich zwischen gewöhnlichem PRINT-Befehl, PRINT in die Datei CONS: und PRINT in die Datei SCRN: die folgende Tabelle (alle drei Variationen geben Zeichen auf dem Bildschirm aus):

CONS: Zeichen werden von BASIC nicht kontrolliert, aber von DOS-Treibern (zum Beispiel ANSI.SYS) verarbeitet. So können mit ANSI.SYS Tasten umdefiniert werden. Die Ausgabe erfolgt in der Farbe, die auch DOS zum Zeitpunkt des BASIC-Programmaufrufs benutzte.

Die Ausgabe kann kann mit dem >-Zeichen von DOS in eine Datei umgeleitet werden.

SCRN: Zeichen werden von BASIC kontrolliert; ein PRINT #1, CHR$(12), wenn #1 die Datei SCRN: ist, würde beispielsweise den Bildschirm löschen. Die Zeichen werden nicht von einem DOS-Treiber wie ANSI.SYS verarbeitet. Sie erscheinen in der mit COLOR eingestellten Farbe; wenn COLOR nicht benutzt wurde, in Weiß auf Schwarz. Die Ausgabe kann nicht umgeleitet werden.

Wenn Sie mit TSCNIO (professionelle Ausgabe) linken, werden die Zeichen überhaupt nicht mehr kontrolliert, so daß selbst das Zeichen CHR$(13), das am Ende einer Zeile gesendet wird, als solches sichtbar ist.

PRINT Wie SCRN:, jedoch können die Zeichen umgeleitet werden, und Steuerzeichen sind bei Verwendung von TSCNIO*xx* überhaupt nicht sichtbar.

Der LOCATE-Befehl wirkt auf alle drei Arten der Zeichenausgabe, nur dann nicht, wenn in eine Datei umgeleitet wird. CONS: und SCRN: können nicht verwendet werden, während Formen angezeigt werden.

• OPEN dient auch zum Öffnen von ISAM-Datenbanken. Mehr dazu siehe im ISAM-Referenzteil.

Kompatibel PDS + VBWIN * Formen + Mathe -

Siehe auch LEN (622), WIDTH (691), LOC (624), LOF (626), CLOSE (578), DO...LOOP (592).

OPTION BASE (Befehl)

Anwendung OPTION BASE *n*

Nutzen Setzt die standardmäßig benutzte untere Grenze für Arrays auf *n*. *n* kann entweder 0 oder 1 sein. Üblicherweise, wenn Sie ein Array mit DIM *array*(*index*) dimensionieren, ist sein Bereich 0 bis *index*, es hat also *index*+1 Elemente. Nach einem OPTION BASE 1-Befehl wäre der Bereich des Arrays beim gleichen Befehl nur 1 bis *index*.

Bemerkung • Benutzen Sie lieber die TO-Klausel im DIM-Befehl (siehe dort); mit ihr kann man Indexgrenzen eines Arrays sehr viel differenzierter festlegen.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch DIM (590).

OPTION EXPLICIT (Befehl)

Anwendung OPTION EXPLICIT

Nutzen Sorgt dafür, daß alle Variablen mit DIM deklariert werden müssen, bevor sie benutzt werden können. Als Programmierer hat man dann etwas mehr Schreibarbeit, vermeidet aber lästige Fehler. Bisher konnte es vorkommen, daß man eine Variable Zahl% benutzte und irgendw im Porgramm versehentlich Zahl (ohne %) schrieb. BASIC verstand das dann als eine neue Variable, und oft dauerte es lange, einen solchen Fehler zu finden. Bei OPTION EXPLICIT wäre dieses unerwartete Auftreten von Zahl durch den Compiler bemänglet worden.

Kompatibel PDS - VBWIN + Formen + Mathe -

Siehe auch DIM (590).

OUT (Befehl)

Anwendung OUT *kanal, byte*

Nutzen Sendet auf einem Hardware-I/O-Kanal ein Byte. *kanal* ist die Kanalnummer (0 bis 65.535), *byte* das zu sendende Byte (0 bis 255).

Bemerkung
- Siehe Bemerkung zu INP.
- Unachtsame Anwendung von OUT kann im schlimmsten Falle bei bestimmten Konstellationen sogar Hardware-Defekte verursachen. Keine Experimente also mit diesem Befehl!

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch INP (614), WAIT (690).

PAINT (Befehl)

Anwendung PAINT [STEP] (*x, y*) [, [*fuellen*[, [*rahmenfarbe* [, *hintergrund$*]]]

Nutzen Füllt einen begrenzten Bereich auf dem Bildschirm von einem Punkt ausgehend mit einer Farbe oder einem Muster aus. (*x, y*) sind die Koordinaten des Punktes, von dem aus gefüllt werden soll. Wenn STEP angegeben wird, werden die Koordinaten relativ zur Position des Grafikcursors verstanden.

fuellen kann entweder ein String oder eine Zahl sein. Wenn Sie eine Zahl angeben, wird vom angegebenen Punkt aus in der Farbe *fuellen* gefüllt. Geben Sie einen String an, wird vom aktuellen Punkt aus mit dem Muster *fuellen* gefüllt. *fuellen* ist dann ein String, der bis zu 64 Zeichen umfassen kann und sowohl Füllfarbe als auch Füllmuster enthält. Die ausführliche Beschreibung der Konstruktion eines solchen Strings würde mich hier etwa acht Seiten kosten; deshalb sei es hier mit zwei Verweisen getan: Musterstrings für PAINT, die für fast alle Grafikzwecke ausreichen müßten, lassen sich mit den Routinen GetPattern$ und MakeChartPattern$ aus der Presentation Graphics-Toolbox (professionelle Ausgabe). Experimentieren Sie ruhig ein wenig mit diesem Stringparameter, es kann ja nichts schaden.

rahmenfarbe ist die Farbe, in der das Objekt gezeichnet wurde, das PAINT füllen soll. Wenn beim Füllen Linien der Farbe *rahmenfarbe* erreicht werden, hört PAINT dort mit dem Füllen auf. Wenn *rahmenfarbe* weggelassen wird, wird die gleiche Farbe wie *fuellen* angenommen.

hintergrund$ ist ein String, der ein Muster (wie *fuellen*) angibt, das beim Füllen nicht als Grenze erkannt, also übergangen werden soll.

Kompatibel PDS + VBWIN - Formen - Mathe -

Siehe auch SCREEN (663), GetPattern$ (Kap.27.3), MakeChartPattern$ (Kap. 27.3).

PALETTE (Befehl)

Anwendung (1) PALETTE [*farbattribut*, *farbe*]
(2) PALETTE USING *array* [*index*]

Nutzen Ordnet den Farbattributen bei EGA-, VGA- oder MCGA-Karten Farben zu. Farbangaben, die Sie bei BASIC-Befehlen machen, sind immer Nummern von Farbattributen. Die Anzahl der verfügbaren Farben ist meist größer als die der verfügbaren Farbattribute.

In Syntax (1) ist *farbattribut* die Nummer eines Farbattributs, und *farbe* ist die Nummer einer Farbe. Die Anzahl der verfügbaren Farben und Farbattribute hängt vom aktuellen Bildschirmmodus, von der Grafikkarte und vom verwendeten Monitor ab. Wenn Sie beide Parameter weglassen, wird die Standard-Farbpalette eingestellt.

Mit Syntax (2) können Sie mehrere Zuordnungen gleichzeitig vornehmen. *array* ist der Name eines Arrays, das mindestens so viele Elemente haben muß, wie es Farbattribute im aktuellen Bildschirmmodus gibt. Dem ersten Farbattribut wird dann die Farbe *array*(1) zugeordnet, dem zweiten die Farbe *array*(2) usw. Wenn Sie *index* angeben, beginnt die Zurodnung nicht mit dem ersten Array-Element, sondern mit dem Element Nr. *index*. Wenn im Array die Zahl –1 vorkommt, wird die dem betreffenden Farbattribut zugeordnete Farbe nicht geändert.

Bemerkung
- Der PALETTE-Befehl ist bei EGA-, VGA- und MCGA-Karten auch verwendbar, um andere als die 16 Standardfarben den Farbattributen zuzuordnen.
- Eine Übersicht über die Anzahl der verfügbaren Farbattribute und Farben bei verschiedenen Bildschirmen, Grafikkarten und Bildschirm-Modi finden Sie im Abschnitt über SCREEN.

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch SCREEN (663), COLOR (579).

PCOPY (Befehl)

Anwendung PCOPY *vonseite, nachseite*

Nutzen Je nach aktuellem Bildschirmmodus und vorhandener Grafikkarte passen in den Video-Speicher auf der Grafikkarte mehrere Bildschirmseiten. Sie können, wenn Sie mit einer Konstellation arbeiten, die mehrere Seiten unterstützt, den Inhalt eines ganzen Bildschirms mit PCOPY von der Bildschirmseite *vonseite* auf die Seite *nachseite* kopieren. Das kann zum Beispiel eine einfache Möglichkeit sein, einen ganzen Bildschirminhalt zu sichern, um ihn später sofort wieder abrufen zu können.

Bemerkung
- Eine ausführliche Beschreibung der einzelnen Bildschirmmodi mit der Anzahl der verfügbaren Bildschirmseiten finden Sie beim SCREEN-Befehl.

Kompatibel PDS + VBWIN - Formen - Mathe -

Siehe auch SCREEN (663).

PEEK (Funktion)

Anwendung *x%* = PEEK(*adresse*)

Nutzen Ermittelt den Inhalt einer Speicherstelle. *adresse* ist die (Offset-Adresse der Speicherstelle, die relativ zum aktuellen Segment (siehe DEF SEG) genommen wird. *adresse* muß zwischen 0 und 65.535 liegen. Wenn *adresse* im Bereich 32.768 bis –1 liegt, wird 65.536 hinzuaddiert. Zurückgegeben wird der Inhalt der gewünschten Speicherstelle als Zahl zwischen 0 und 255.

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch POKE (649), DEF SEG (589), BLOAD (571), BSAVE (571).

PEN (Funktion)

Anwendung *x* = PEN(*argument*)

Nutzen Gibt den Status des Lichtstiftes zurück. Die Zahl *argument* (0 bis 9) gibt an, welche Information gewünscht ist. Der Lichtstift gibt Koordinaten bezogen auf die gerade verwendete Grafikauflösung zurück, es sei denn, Sie lassen ihn durch eine Maus emulieren. PEN funktioniert nicht, solange der Maustreiber aktiv ist.

argument	*gibt zurück*
0	–1, wenn der Lichtstift seit dem letzten PEN-Aufruf betätigt wurde, 0, wenn nicht
1	die x-Koordinate, an der der Stift zum letzen Mal gedrückt wurde
2	die y-Koordinate, an der der Stift zum letzen Mal gedrückt wurde
3	–1, wenn der Lichtstift gerade gedrückt wird, 0, wenn nicht
4	die x-Koordinate, an der der Stift zum letzten Mal den Bildschirm verlassen hat
5	die y-Koordinate, an der der Stift zum letzten Mal den Bildschirm verlassen hat
6–9	wie 1, 2, 4, und 5, nur werden hier nicht Grafikkoordinaten, sondern Textzeilen bzw. -spalten zurückgegeben.

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch ON *event* GOSUB (610), PEN (646)

PEN (Befehle)

Anwendung
```
PEN ON
PEN OFF
PEN STOP
```

Nutzen Wie alle anderen *event*-Befehle ist dieser Befehl dazu geeignet, das Event-Trapping für den Lichtstift ein- und auszuschalten oder zu unterbrechen. PEN ON schaltet das Event-Trapping für den Lichtstift ein (bevor es wirksam wird, muß noch ein ON PEN GOSUB-Befehl ausgeführt werden); PEN OFF schaltet es ab, und PEN STOP ruft die Trapping-Routine bis zum nächsten PEN ON nicht mehr auf, dann (nach dem PEN ON) werden allerdings alle inzwischen aufgetretenen PEN-Events verarbeitet.

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch ON *event* GOSUB (634), EVENT ON/OFF (600), PEN (645).

PLAY (Funktion)

Anwendung *x%* = PLAY(*n*)

Nutzen Gibt die Anzahl der Noten zurück, die im Hintergrund noch gespielt werden. *n* ist ein Dummy-Argument und kann einen beliebigen Wert annehmen. Wenn die Musik im Vordergrund läuft, gibt PLAY(*n*) 0 zurück.

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch PLAY (647).

PLAY (für Event-Trapping) (Befehle)

Anwendung
```
PLAY ON
PLAY OFF
PLAY STOP
```

Nutzen Wie alle anderen *event*-Befehle ist dieser Befehl dazu geeignet, das Event-Trapping für den Musik-Hintergrundspeicher ein- und auszuschalten oder zu unterbrechen. PLAY ON schaltet das Event-Trapping für den Musik-Hintergrundspeicher ein (bevor es wirksam wird, muß noch ein ON PLAY GOSUB-Befehl ausgeführt werden); PLAY OFF schaltet es ab, und PLAY STOP ruft die Trapping-Routine bis zum nächsten PLAY ON nicht mehr auf, dann (nach dem PLAY ON) werden allerdings alle inzwischen aufgetretenen PLAY-Events verarbeitet.

Siehe auch ON *event* GOSUB (610), PLAY (Funktion) (646), PLAY (für Tonerzeugung) (647).

PLAY (für Tonerzeugung) (Befehl)

Anwendung PLAY *musik$*

Nutzen Spielt – wenn auch in bescheidener Klangqualität – Musik über den internen Lautsprecher ab. Ähnlich wie DRAW besitzt PLAY eine Art von eigener »Programmiersprache«, deren Befehle man ihm als String (*musik$*) übergibt. Folgende Befehle sind in *musik$* möglich:

Befehl	*Wirkung*
A,B,C,D,E,F,G	Spielt die angegebene Note. Der Buchstabe kann jeweils von einem + oder # (einen Halbton aufwärts) oder von einem – (einen Halbton abwärts) gefolgt werden. Danach kann eine Notenlänge (1 für ganze Note, 2 für halbe Note etc., bis 64) folgen. Außerdem kann auch ein Punkt (.) folgen, der die Notenlänge für diese Note um die Hälfte verlängert. Beachten Sie, daß im Amerikanischen unsere Note H B heißt, unser B wäre also ein H-.
P	Spielt eine Pause. Es gelten die gleichen Längenangaben wie für Noten.
N *n*	Spielt die Note Nr. *n* (*n* im Bereich 1 bis 84, oder 0 für Pause). Die Note wird mit der durch L festgelegten Länge gespielt.
L *n*	Legt die Länge für Noten fest, denen keine Längenangabe folgt. *n* ist die Notenlänge (1 für ganze Note bis 64 für Vierundsechzigstelnote).
O *n*	Wählt die Oktave *n* (*n* im Bereich von 0 bis 6).
<	wählt die nächsttiefere Oktave
>	wählt die nächsthöhere Oktave
MN	Setzt den normalen Spielmodus, in dem jede Note für 7/8 ihrer Länge angehalten wird und das verbleibende Achtel Pause ist.
ML	Setzt den Legato-Modus, in dem jede Note voll angehalten wird.
MS	Setzt den Staccato-Modus, in dem jede Note nur 3/4 ihrer Länge spielt und das restliche Viertel pausiert wird.
T *n*	Setzt das Spieltempo. *n* ist die Anzahl der Viertelnoten pro Minute und kann von 32 bis 255 reichen.

MF Setzt für PLAY- und SOUND-Befehle den Vordergrund-Modus, so daß die Ausführung eines PLAY- oder SOUND-Befehles erst dann beendet ist, wenn die Noten ausgeklungen sind. In diesem Modus können das PLAY-Event-Trapping und die Funktion PLAY nicht benutzt werden.

MB Setzt für PLAY- und SOUND-Befehle den Hintergrund-Modus, in dem bis zu 32 Noten in einem Puffer zwischengespeichert werden. Die Programmausführung läuft dann schon weiter, während noch Musik spielt. Nur in diesem Modus sind das PLAY-Event-Trapping und die Funktion PLAY anwendbar.

X*adr$* Führt einen weiteren PLAY-String aus, dessen Adresscode mit VARPTR$ ermittelt und in *adr$* eingetragen werden muß.

=*adr$* Kann anstelle einer Zahl (*n* in obigen Beispielen) gesetzt werden; *adr$* muß die mit VARPTR$ ermittelte Adresse einer Zahl im Speicher sein, die dann hier eingesetzt wird.

Bemerkung • Leerzeichen sind überhaupt nicht erforderlich, aber an jeder Position und in beliebiger Anzahl im PLAY-String erlaubt – mit einer Ausnahme: Zwischen X bzw. = und dem dazugehörigen VARPTR$ darf sich kein Leerzeichen befinden.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch PLAY (Funktion) (646), ON *event* GOSUB (634), SOUND (674).

PMAP (Funktion)

Anwendung *ausgabe* = PMAP(*eingabe*, *verfahren*)

Nutzen Wenn Sie im Grafikmodus mit dem WINDOW-Befehl arbeiten, also ein eigenes Koordinatensystem definieren, sind die Pixelkoordinaten nicht mehr identisch mit den logischen Koordinaten, die Sie bei allen folgenden Grafikbefehlen angeben müssen. In diesen Fällen können Sie PMAP benutzen, um die beiden Koordinatensysteme (das originale Pixelsystem und Ihr selbstdefiniertes) ineinander umrechnen zu lassen.

verfahren	*Funktion*
0	Eine logische x-Koordinate wird in eine Pixel-x-Koordinate umgerechnet.
1	Eine logische y-Koordinate wird in eine Pixel-y-Koordinate umgerechnet.

2	Eine Pixel-x-Koordinate wird in eine logische x-Koordinate umgerechnet.
3	Eine Pixel-y-Koordinate wird in eine logische y-Koordinate umgerechnet.

Pixel-Koordinaten sind relativ zum mit VIEW definierten Grafik-Viewport. Haben Sie VIEW nicht benutzt oder bei VIEW zusätzlich SCREEN angegeben, dann handelt es sich um absolute Pixelkoordinaten (Punkt 0,0 in der oberen linken Ecke).

Kompatibel PDS + VBWIN - Formen - Mathe +

Siehe auch VIEW (689), WINDOW (692).

POINT (Funktion)

Anwendung (1) *z%* = POINT(*x, y*)
(2) *z%* = POINT(*funktion*)

Nutzen In der ersten Syntax gibt POINT das Farbattribut des angegebenen Grafikpunktes zurück. *x* und *y* sind dabei, wie bei allen anderen Grafikbefehlen, logische Koordinaten, die von WINDOW beeinflußt werden. Wenn Sie WINDOW nicht benutzen, sind die logischen Koordinaten mit den Pixelkoordinaten identisch.

In Syntax (2) hängt der Funktionswert vom Argument *funktion* ab:

funktion	*Zurückgegebener Wert*
0	Pixel-x-Koordinate des Grafikcursors
1	Pixel-y-Koordinate des Grafikcursors
2	logische x-Koordinate des Grafikcursors
3	logische y-Koordinate des Grafikcursors

0

Kompatibel PDS + VBWIN - Formen - Mathe *

Siehe auch PMAP (648), VIEW (689), WINDOW (692), PSET (653).

POKE (Befehl)

Anwendung POKE *adresse, byte*

Nutzen Schreibt einen Wert in eine Speicherstelle. *adresse* ist die (Offset-)Adresse der Speicherstelle, die relativ zum aktuellen Segment (siehe DEF SEG) genommen wird. *adresse* muß zwischen 0 und 65.535 liegen. Wenn *adresse* im Bereich –32.768 bis –1 liegt, wird 65.536 hinzuaddiert. *byte* ist der Wert, der in die angegebene Speicherstelle geschrieben werden soll (zwischen 0 und 255).

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch PEEK (645), DEF SEG (589), BLOAD (571), BSAVE (571).

POS (Funktion)

Anwendung *x* = POS(*n*)

Nutzen POS gibt die Bildschirmspalte zurück, in der sich der Textcursor befindet. Der Funktionswert liegt also zwischen 1 und 80 bzw. zwischen 1 und 40 bei Bildschirm-Modi mit 40 Zeichen. *n* ist ein Dummy-Argument, das jeden beliebigen Wert haben kann.

Kompatibel PDS + VBWIN - Formen - Mathe -

Verwenden Sie die Eigenschaft *CurrentX*, um die aktuelle Spalte bei Formen festzustellen.

Siehe auch CSRLIN (583), LOCATE (625).

PRESET (Befehl)

Anwendung PRESET [STEP](*x, y*), *farbe*

Nutzen PRESET setzt einen Punkt im Grafikmodus. PRESET funktioniert exakt wie PSET, mit dem Unterschied, daß bei Weglassen von *farbe* hier die Hintergrundfarbe (üblicherweise Nr. 0) gewählt wird.

Kompatibel PDS + VBWIN - Formen - Mathe -

Siehe auch PSET (653), POINT (649).

PRINT (Befehl)

Anwendung PRINT [#*dateinummer*,] [USING *format$*;] [*ausdruck* [,|; *ausdruck*]... [,|;]]

Nutzen Gibt Daten auf dem Bildschirm oder in eine Datei aus. *dateinummer* ist eine gültige Dateinummer, die zuvor mit OPEN geöffnet wurde. Wenn Sie *#dateinummer* weglassen, erfolgt die Ausgabe auf dem Bildschirm.

format$ ist ein String, der angibt, wie die folgenden *ausdrücke* formatiert werden sollen. Wird USING *format$* weggelassen, werden die *ausdrücke* nicht formatiert. Strings werden dann so ausgegeben, wie sie sind; positive Zahlen werden von einem Leerzeichen, negative Zahlen von einem Minus-Zeichen eingeleitet, und hinter beiden wird noch ein Leerzeichen ausgegeben. Das Exponentialformat (beispielsweise 1E–7) wird für SINGLE-Zahlen benutzt, wenn sie nicht mit maximal sieben, für DOUBLE-Zahlen, wenn sie nicht mit maximal 15 Stellen exakt ausgedrückt werden können. Zahlen, deren Betrag kleiner 1 ist, werden nicht als 0.xxxx, sondern einfach als .xxxx ausgegeben.

Überzählige Dezimalstellen werden gerundet. Wenn die zu formatierende Zahl nicht in *format$* zu formatieren ist, weil sie zu groß ist, wird ein %-Zeichen gefolgt von der vollständigen Zahl (als wären genug Ziffern-Platzhalter vorhanden) ausgegeben. Die erlaubten Zahlformatierzeichen in *format$* sind:

Zeichen	*Bedeutung*
#	Ziffern-Platzhalter. Ziffern-Platzhalter nach dem Dezimalpunkt und der Ziffern-Platzhalter unmittelbar vor dem Dezimalpunkt werden mit Nullen gefüllt, wenn keine Ziffern vorhanden sind; alle anderen werden mit Leerzeichen gefüllt.
.	Dezimalpunkt. Legt fest, wieviele Ziffern vor und wieviele nach dem Dezimalpunkt stehen.
+	An der Stelle, an der im Format-String das Pluszeichen steht, wird das Vorzeichen der Zahl eingetragen (+ darf nur vor oder hinter den Ziffernplatzhaltern stehen).
-	Wie +, jedoch wird hier im Falle positiver Zahlen kein Vorzeichen ausgegeben.

**	Anstatt führende, von der Zahl nicht belegte Ziffern-Platzhalter mit Leerzeichen zu füllen, werden sie mit Sternen gefüllt. ** hat selbst die Funktion zweier Ziffern-Platzhalter.
$$	Vor der ersten Ziffer der Zahl wird ein Dollar-Zeichen ausgegeben. $$ hat selbst die Funktion eines weiteren Ziffern-Platzhalters.
**$	Wie ** und $$ zusammen; **$ hat selbst die Funktion zweier Ziffern-Platzhalter.
,	Links vom Dezimalpunkt sorgt das Komma dafür, daß alle drei Ziffern ein Komma (als Tausender-Trennzeichen) eingeschoben wird. Es zählt als weiterer Ziffern-Platzhalter.
^^^^	Die Zahl wird im Exponentialformat ausgegeben. Sie müssen ein fünftes ^-Zeichen anfügen, wenn der Exponent drei Ziffern benötigt (nur bei DOUBLE-Zahlen ist das möglich). Diese Zeichen müssen unmittelbar hinter dem letzten Ziffern-Platzhalter folgen. Der erste Ziffern-Platzhalter wird, wenn nicht durch – oder + anders spezifiziert, für das Vorzeichen verwendet.
_	(Unterstrich) Das folgende Zeichen wird ungeachtet seiner Bedeutung einfach als Zeichen ausgegeben.

USING kann auch Strings formatieren:

Zeichen	*Bedeutung*
!	Nur das erste Zeichen des Strings wird ausgegeben.
&	Der String wird unformatiert ausgegeben.
\\	Es werden so viele Zeichen des Strings ausgegeben, wie durch die beiden Backslashes und die zwischen ihnen eingeschlossenen Leerzeichen angegeben sind, also zwei plus der Anzahl der Leerzeichen zwischen den Backslashes.

Als *ausdruck* im PRINT-Befehl können Stringvariablen oder -konstanten, numerische Variablen oder Konstanten, Funktionen sowie Verknüpfungen aus all diesen angegeben werden. Sie sollten jedoch keine Funktionen im PRINT-Befehl verwenden, die ihrerseits den PRINT-Befehl benutzen, um in die gleiche Datei bzw. auch auf den Bildschirm zu schreiben, weil das unerwünschte Resultate haben kann.

Es können beliebig viele *ausdrücke* angegeben werden; sie werden durch Komma oder Semikolon getrennt. Bei der Verwendung von USING *format$* spielt es keine Rolle, welche Trennzeichen Sie benutzen; ansonsten sorgt ein Komma dafür, daß der nächste Ausdruck nach der nächsten durch 14 teilbaren Spalte ausgegeben wird, während ein Semikolon keinen zusätzlichen Abstand zum nächsten Ausdruck bedingt.

Wenn am Ende der *ausdruck*-Liste kein Komma oder Semikolon steht, wird der Cursor auf die erste Spalte der nächsten Zeile gesetzt.

Kompatibel PDS + VBWIN * Formen * Mathe -

In VBWIN und während der Formenanzeige ist für die Bildschirmausgabe die PRINT-Methode zu verwenden (siehe Objekt-Referenzteil); zur Dateiausgabe kann PRINT unbeschränkt verwendet werden.

Siehe auch LPRINT (627), OPEN (632), FORMAT$ (604), LEFT$ (621).

PSET (Befehl)

Anwendung PSET [STEP](*x, y*), *farbe*

Nutzen PSET setzt einen Punkt im Grafikmodus. *x* und *y* sind die Koordinaten des Punktes. Wird STEP angegeben, sind *x* und *y* keine absoluten Koordinaten, sondern relativ zur Position des Grafikcursors.

farbe ist die Farbe, in der der Punkt gesetzt werden soll. Wird *farbe* weggelassen, wählt PSET die aktuelle Vordergrundfarbe, und das ist auch der einzige Unterschied zu PRESET, das in diesem Falle die Hintergrundfarbe wählt.

Kompatibel PDS + VBWIN * Formen - Mathe -

VBWIN stellt eine gleichnamige Methode zur Verfügung.

Siehe auch PRESET (363), POINT (361), PMAP (361), SCREEN (378).

PUT (für Dateien) (Befehl)

Anwendung PUT [#]*dateinummer*[,[*satznummer*], [*satzvariable*]]

Nutzen Schreibt Daten in eine Datei, die FOR RANDOM oder FOR BINARY geöffnet wurde. Der Unterschied zwischen beiden Dateiarten beim PUT-Befehl ist, daß (a) bei RANDOM-Dateien *satznummer* die Nummer des Datensatzes ist (die Länge des Datensatzes wird beim Öffnen angegeben), während *satznummer* bei BINARY-Dateien die absolute Byte-Position innerhalb der Datei ist, ab der geschrieben werden soll, und daß (b) für BINARY-Dateien die Angabe einer *satzvariable* unerläßlich ist, während RANDOM-Dateien stattdessen auch mit dem FIELD-Befehl bearbeitet werden können.

Läßt man die *satznummer* weg, so wird anstelle dessen LOC(*dateinummer*) + 1 benutzt, die Stelle nach der zuletzt gelesenen oder beschriebenen Position in der Datei.

Bei RANDOM-Dateien wird der Inhalt der entsprechenden FIELD-Variablen in die Datei geschrieben oder, wenn eine *satzvariable* angegeben ist, diese Variable.

Siehe auch GET (für Dateien) (608), OPEN (632), LOC (624), FIELD (602).

PUT (für Grafik) (Befehl)

Anwendung PUT [STEP](*x,y*), *feldname*[(*index*)] [,*modus*]

Nutzen Schreibt einen rechteckigen Grafikbereich, der mit GET vom Grafikbildschirm in ein Array kopiert wurde, wieder auf den Bildschirm. Das Koordinatenpaar (*x,y*) beschreibt den Punkt, auf den die obere linke Ecke (bei WINDOW ohne SCREEN-Zusatz die untere linke Ecke) des Bereichs abgebildet wird. Gibt man STEP an, sind *x* und *y* relativ zum Grafikcursor.

feldname ist der Name des Datenfeldes, in dem der Grafikbereich steht. *index* steht für einen oder (je nach Dimension des angegebenen Feldes) mehrere Indizes, die, bei dem die Übertragung in den Bildschirmspeicher beginnen soll.

modus ist die Art und Weise, in der der Bereich auf den Bildschirm kopiert wird:

modus	*Wirkung*
AND	Punkte, die auf dem Bildschirm dieselbe Farbe haben wie im gespeicherten Bild, werden angezeigt; alle anderen nicht.
OR	Das gespeicherte Bild wird auf den Bildschirm kopiert, ohne das dort eventuell bereits vorhandene Bild vorher zu löschen. Es können sich, da die Farbwerte mit OR verknüpft werden, Farbverfälschungen ergeben.
PSET	Das gespeicherte Bild wird auf den Bildschirm kopiert; vorher wird der Bereich auf dem Bildschirm gelöscht, so daß danach allein das gespeicherte Bild in diesem Bereich sichtbar ist.
PRESET	Wie PSET, allerdings wird ein Negativ des gespeicherten Bildes kopiert.
XOR	Das gespeicherte Bild und das bereits vorhandene Bild werden mit einer exklusiven ODER-Operation verknüpft. Dabei ergeben sich zumeist Farbverfälschungen. Wenn dasselbe Bild ein zweites Mal an die gleiche Stelle geschrieben wird, wird durch die logische Beschaffenheit dieser Verknüpfung der ursprüngliche Zustand an dieser Stelle wiederhergestellt. Dadurch können mit XOR Objekte (wie etwa ein Cursor) über den Bildschirm bewegt werden, ohne den Bildschirminhalt zu verändern, indem man, bevor man das Objekt weiterbewegt, einen zweiten PUT-Befehl mit XOR ausführt.

Kompatibel PDS + VBWIN - Formen - Mathe -

Siehe auch GET (für Grafik) (609).

QBCOLOR (Funktion)

Anwendung *x%* = QBCOLOR[*Farbwert*]

Nutzen Wie man in QBCOLOR hineinruft, so schallt es heraus: Diese Dummy-Funktion gibt den ihr übergebenen *Farbwert* zurück. Wenn Sie diese Funktion in Ihren Programmen bei allen Farbzuweisungen verwenden – also z.B. `Form1.BackColor = QBCOLOR(10)` statt `Form1.BackColor = 10`, was in VBDOS völlig gleichwertig ist – können Ihre Programme leichter nach VBWIN übersetzt werden, weil VBWIN mit anderen Farbwerten arbeitet und die Funktion QBCOLOR in VBWIN dann den Windows-Farbwert zurückgibt, der der VBDOS-Farbe *Farbwert* am ehesten entspricht.

Kompatibel PDS - VBWIN + Formen + Mathe -

Siehe auch COLOR (579), RGB (659).

RANDOMIZE (Befehl)

Anwendung RANDOMIZE [*zahl*]

Nutzen Initialisiert den Zufallsgenerator. Wenn Sie RANDOMIZE nicht benutzen oder ein Konstante als *zahl* angeben, dann ergibt RND bei jedem Lauf Ihres Programms die gleichen Zufallswerte (und damit keine Zufallswerte) zurück. Sie müssen also, wenn Sie RND benutzen, sichergehen, daß bei jedem Programmstart RANDOMIZE mit einer anderen *zahl* als Argument aufgerufen wird. Wenn man *zahl* wegläßt, fragt das Programm nach einer Zahl.

Kompatibel PDS + VBWIN + Formen + Mathe +

Siehe auch RND (660), TIMER (683).

READ (Funktion)

Anwendung READ *variable* [, *variable*]...

Nutzen Liest die in DATA-Zeilen aufgeführten Konstanten in Variablen ein. In die erste angegebene Variable wird die DATA-Konstante eingelesen, auf die der DATA-Zeiger gerade zeigt. Dann wird der DATA-Zeiger auf die nächste DATA-Konstante gesetzt. Beim Start des Programms zeigt der DATA-Zeiger auf die erste DATA-Konstante im Modul. Jedes Modul hat einen eigenen DATA-Zeiger, und kein READ kann DATA-Konstanten aus einem anderen Modul als dem, in dem es sich selbst befindet, lesen. Mit dem Befehl RESTORE kann der DATA-Zeiger verschoben werden.

Bemerkung • READ und DATA werden häufig verwendet, um eine größere Anzahl von Konstanten (beispielsweise ein Array von Konstanten) zu definieren.

Siehe auch DATA (585), RESTORE (658).

REDIM (Befehl)

Anwendung
```
REDIM [PRESERVE] [SHARED] variable(array-bereich)
[AS typ]
      [, variable(array-bereich) [AS typ]]...
```

Nutzen REDIM redimensioniert dynamische Arrays. Wenn ein Array noch nicht dimensioniert ist, hat REDIM denselben Effekt wie DIM, mit dem Unterschied, daß ein Array, das mit REDIM dimensioniert wird, immer ein dynamisches Array ist. REDIM kann nicht auf statische Arrays angewendet werden.

Wenn ein Array bereits dimensioniert ist und Sie PRESERVE nicht benutzen, hat REDIM denselben Effekt wie ein ERASE- und ein DIM-Befehl, die nacheinander auf das Array angewendet werden. Das Array wird völlig gelöscht und dann wieder neu eingerichtet. Dabei dürfen mit REDIM nicht mehr oder weniger Dimensionen angegeben werden, als das Array zuvor hatte, und es darf auch nicht der Typ des Arrays verändert werden.

Wenn Sie das Schlüsselwort PRESERVE angeben, dürfen Sie nur die obere Grenze der höchsten Dimension des Arrays verändern (siehe Beispiel). Dann werden die Daten, die sich bereits im Array befinden, beibehalten. Dies ist eine nicht zu unterschätzende Methode zur dynamischen Speicherverwaltung (Listen können beliebig verkürzt und verlängert werden). Vergleiche dazu Kapitel 9.

Kompatibel PDS + VBWIN * Formen + Mathe -

VBWIN unterstüzt das PRESERVE-Schlüsselwort nicht.

Siehe auch DIM (299), ERASE (307).

REM (Befehl)

Anwendung
```
REM bemerkung
'bemerkung
```

Nutzen Leitet eine Bemerkung ein. *bemerkung* ist ein beliebiger Text, der vom Compiler nicht beachtet wird.

Kompatibel PDS + VBWIN + Formen + Mathe -

RESET (Befehl)

Anwendung RESET

Nutzen RESET schließt alle offenen Dateien. Die Wirkung ist dieselbe wie die von CLOSE ohne Parameter.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch CLOSE (578).

RESTORE (Befehl)

Anwendung RESTORE [*zeilennnummer/-label*]

Nutzen Setzt den DATA-Zeiger eines Moduls auf die nächste DATA-Konstante nach der Zeile, die durch *zeilennummer/-label* spezifiziert wird.

RESTORE ohne Argument setzt den DATA-Zeiger auf die erste DATA-Kostante im Modul.

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch DATA (585), READ (656).

RESUME (Befehl)

Anwendung RESUME *zeilennummer/-label*
RESUME
RESUME NEXT

Nutzen Setzt, nachdem ein Fehler aufgetreten und das Programm aufgrund eines ON ERROR GOTO-Befehls in die Error-Trapping-Routine verzweigt hat, die Programmausführung fort.

RESUME setzt das Programm bei dem Befehl fort, der den Fehler verursacht hat (führt ihn also nochmals aus). RESUME NEXT fährt hinter dem Befehl fort, der den Fehler verursachte, und RESUME *zeilennummer/-label* setzt die Programmausführung an der genannten Zeile fort. Die alte Schreibweise RESUME 0 entspricht RESUME ohne Argument.

RESUME darf nur in einem Error-Handler benutzt werden.

Bemerkung • Jede Form von RESUME außer RESUME *zeilennummer/-label* erfordert, daß das Programm mit /X kompiliert wird.

• RESUME und RESUME NEXT unterligen bei Mehrmodul-programmen einigen Beschränkungen: Siehe dazu Kapitel 12.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch ON ERROR (599), ERR, ERL (598).

RETURN (Befehl)

Anwendung RETURN [*zeilennummer/-label*]

Nutzen RETURN alleine kehrt hinter den zuletzt ausgeführten GOSUB-Befehl bzw. an die Stelle, von der aus der letzte Trapping-Aufruf erfolgte, zurück; RETURN mit Zeilenangabe verzweigt – genau wie GOTO – zur angegebenen Zeile. Beide Formen nehmen die letzte GOSUB-Markierung vom Stack, so daß für jedes GOSUB, das ausgeführt wird, auch nur ein RETURN ausgeführt werden darf.

In einer Event-Handling-Routine, die durch ON *event* GOSUB aktiviert und durch das Auftreten des betreffenden Ereignisses aufgerufen wurde, sorgt RETURN dafür, daß genau an die Stelle zurückgekehrt wird, an der das Ereignis auftrat, so daß das Programm problemlos weiterläuft.

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch GOSUB (610), SUB/FUNCTION (680), ON *event* GOSUB (634), ON...GOSUB (635).

RGB (Funktion)

Anwendung *farbwert* = RGB(*rot*, *grün*, *blau*)

Nutzen RGB gibt einen VBDOS-Farbwert (0 bis 15) zurück, der der WINDOWS-Farbe, die aus den Farbanteilen *rot*, *grün* und *blau* (jeweils 0 bis 255) gebildet wird, am ehesten entspricht. Diese Funktion wird u.U. benötigt, um Programme von VBWIN nach VBDOS zu übersetzen; in VBWIN liegt der zurückgegebene Farbwert dieser Funktion nicht im Bereich von 0 bis 15.

Kompatibel PDS - VBWIN + Formen + Mathe -

Siehe auch COLOR (579), QBCOLOR (655).

RIGHT$ (Funktion)

Anwendung *x$* = RIGHT$(*y$, zeichen*)

Nutzen Gibt die letzten *zeichen* Zeichen seines Arguments *y$* als neuen String zurück. Wenn *zeichen* länger als *y$* selbst ist, wird der ganze *y$* zurückgegeben.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch LEFT$ (621), MID$ (Befehl) (629).

RMDIR (Befehl)

Anwendung RMDIR *verzeichnisname$*

Nutzen Löscht ein leeres Verzeichnis vom Datenträger. *verzeichnisname$* ist der komplette Name dieses Verzeichnisses und darf nicht länger als 63 Zeichen sein. Es kann nur ein Verzeichnis gelöscht werden, das keine Dateien und keine Unterverzeichnisse mehr enthält.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch CHDIR (575), CURDIR$ (583), MKDIR (629).

RND (Funktion)

Anwendung *x!* = RND [(*argument*)]

Nutzen RND gibt eine Zufallszahl zwischen 0 und 1 zurück ($0 \leq x! < 1$). Mit *argument* = 0 wird immer die zuletzt erzeugte Zufallszahl erneut zurückgegeben, während ein beliebiges *argument* größer 0 die nächste Zufallszahl in der mit RANDOMIZE initialisierten Serie zurückgibt. Solche »Zufallsserien« haben eine unbegrenzte Zahl von Elementen. Ein gleiches RANDOMIZE führt auch zu der gleichen Zufallsserie. Ein *argument* < 0 gibt immer eine Zahl zwischen 0 und 1 zurück, die sich direkt aus dem *argument* berechnet, also unabhängig von RANDOMIZE ist. Ihre Erzeugung ist beliebig wiederholbar.

Kompatibel PDS + VBWIN + Formen + Mathe +

Siehe auch RANDOMIZE (656).

RSET (Befehl)

Anwendung RSET *string1$ = string2$*

Nutzen RSET schreibt den *string2$* rechtsbündig in *string1$* und füllt *string1$* am linken Rand mit Leerzeichen auf, falls noch Platz ist. Wenn *string2$* länger als *string1$* ist, wird er abgeschnitten.

Bemerkung • REST kann nicht, wie LSET, benutzt werden, um zwei verschiedene Variablen selbstdefinierten Typs einander zuzuordnen.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch LSET (628).

RTRIM$ (Funktion)

Anwendung *x$* = RTRIM$(*y$*)

Nutzen RTRIM$ gibt als Funktionswert den String zurück, der ihr als Argument übergeben wurde, entfernt jedoch zuvor alle Leerzeichen (CHR$(32)) vom rechten Rand des Strings.

Bemerkung • Beachten Sie, daß Strings mit fester Länge vor ihrer ersten Verwendung nicht Leerzeichen, sondern CHR$(0)-Zeichen enthalten, die sich im Aussehen von Leerzeichen nicht unterscheiden, die RTRIM$ jedoch nicht entfernt.

• ISAM (professionelle Ausgabe) führt beim Speichern von Strings automatisch ein RTRIM$ aus, so daß nie Leerzeichen am rechten Rand gespeichert werden.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch LTRIM$ (628)

RUN (Befehl)

Anwendung (1) RUN [*zeilennummer*]
(2) RUN *programmname$*

Nutzen Beide Versionen von RUN löschen alle Variablen und Vereinbarungen, schließen sämtliche offenen Dateien, entfernen einen eventuellen Grafik-Viewport und setzen den DATA-Zeiger wieder auf die erste DATA-Konstante.

In der ersten Syntax RUN *zeilennummer* startet daraufhin das Programm von der angegebenen Zeilennummer aus neu. *zeilennummer* muß eine Zeile im Modulcode bezeichnen. Wird die Zeilennummer weggelassen, startet RUN von der ersten Zeile des Modulcodes aus neu.

In der zweiten Syntax RUN *programmname$* wird das angegebene Programm geladen und ausgeführt. Dabei dürfen auch Nicht-BASIC-Programme aufgerufen werden, solange es COM- oder EXE-Dateien sind. Im Gegensatz zum SHELL-Befehl wird das Programm, das den RUN-Befehl enthält, nicht mehr weiter ausgeführt, wenn das aufgerufene Programm beendet ist. Dem aufgerufenen Programm steht mehr Speicher zur Verfügung als beim SHELL-Befehl.

Bemerkung • Der Unterschied zum CHAIN-Befehl besteht darin, daß CHAIN die Übergabe von Daten ermöglicht, daß Dateien geöffnet bleiben, und daß das Runtime-Modul (falls vorhanden) beim Aufruf eines anderen BASIC-Programms nicht neu geladen wird. Mit RUN kann eine beliebige ausführbare Datei gestartet werden; CHAIN stürzt bei Nicht-BASIC-Programmen meistens ab.

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch CHAIN (574), SHELL (673)

SADD (Funktion)

Anwendung *x%* = SADD(*variable$*)

Nutzen Gibt die (Offset-)Adresse einer Stringvariablen zurück. *variable$* ist die Stringvariable, deren Adresse ermittelt werden soll.

Um die vollständige Adresse eines Strings zu erfahren, muß zusätzlich die Funktion SSEG benutzt werden, die die Segmentadresse des Strings zurückgibt. Erst diese beiden Zahlen zusammen ermöglichen das Auffinden eines Strings im Speicher.

SADD wird nur für Strings variabler Länge benutzt. Verwenden Sie VARPTR für Strings mit fester Länge und andere Variablen..

Bemerkung • Die Adresse eines Strings kann von BASIC laufend verändert werden. Ermitteln Sie die Adresse deshalb stets unmittelbar, bevor Sie sie verwenden.

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch SSEG (675), SSEGADD (676), DEF SEG (589), PEEK (645), POKE (649).

SCREEN (Funktion)

Anwendung *x%* = SCREEN(*zeile*, *spalte* [, *farbwert*])

Nutzen Gibt den ASCII-Wert oder die Farbe des Zeichens, das an einer bestimmten Bildschirmposition steht, zurück. *zeile* und *spalte* geben die Position an; wenn *farbwert* 0 ist, dann wird der ASCII-Wert des Zeichens zurückgegeben; ist *farbwert* ungleich 0, wird stattdessen der Farbcode (ein Attribut, das Vorder- und Hintergrundfarbe in einem Byte kombiniert). Im Grafikmodus gibt die SCREEN-Funktion keine Farbcodes zurück.

Kompatibel PDS + VBWIN - Formen - Mathe -

SCREEN (Befehl)

Anwendung SCREEN [*modus*] [, [*farbe*] [, [*aseite*] [, *vseite*]]]

Nutzen Schaltet in einen bestimmten Bildschirmmodus (*modus*). Die erlaubten Modi sind untenstehend ausführlich beschrieben; welche Modi auf einem definierten System eingesetzt werden können, hängt von der Grafikkarte und dem verwendeten Bildschirm ab. Bei VGA- und EGA-Karten ist es zum Teil auch relevant, ob die Karte mit 64, 128 oder 256 KB (bei EGA) bzw. 256 oder 512 KB (bei VGA) ausgestattet ist. Die meisten EGA-Karten besitzen heute 256 KB; bei VGA-Karten ist die Konfiguration 256 KB leider noch recht häufig anzutreffen.

aseite und *vseite* geben die aktuelle und die virtuelle Bildschirmseite an. Auf die aktuelle Bildschirmseite wird mit PRINT, CLS, den Grafikbefehlen etc. geschrieben, während die virtuelle Bildschirmseite sichtbar ist. Die Standardeinstellung ist 0 für beide Werte. Unterstützt eine Grafikkarte in einem bestimmten Modus jedoch mehrere Bildschirmseiten, so kann eine Seite angezeigt werden, während auf eine andere geschrieben wird, oder man kann eine Hilfsseite o.*. immer im Hintergrund, unsichtbar, auf der zweiten Bildschirmseite »aufbewahren«, um sie dann blitzartig durch einfaches Umschalten der vseite anzuzeigen. *aseite* und *vseite* können sich zwischen 0 und 7 bewegen; wieviele Bildschirmseiten ein Modus wirklich unterstützt, ist in der Aufstellung angegeben.

Die unterstützen Grafikkarten sind: *MDPA* (IBM Monochrome Display and Printer Adapter), *Hercules* (Standard-Monochrom-Grafikkarte), *CGA* (Color Graphics Adapter), *EGA* (Enhanced Graphics Adapter), *VGA* (Video Graphics Array) und *MCGA* (Multicolor Graphics Array). Außerdem wird mit dem Grafikmodus 4 eine exotische Olivetti- und AT&T-Grafikkarte unterstützt.

Modus 0 – keine Grafik

Dieser Modus läßt sich auch mit einer EGA-Karte betreiben, an die nur ein CGA-Bildschirm angeschlossen ist.

MDPA 80x25 Zeichen, eine Bildschirmseite, 16 Farbattribute (0 und 8 = schwarz, 1–7 = weiß, 9–15 = leuchtend weiß).

CGA 80x25 Zeichen (vier Bildschirmseiten) oder 40x25 Zeichen (acht Bildschirmseiten), 16 Farbattribute (Standardfarben 0–15).

Hercules 80x25 Zeichen, eine Bildschirmseite, 16 Farbattribute (0 und 8 = schwarz, 1 und 9 = weiß und unterstrichen, 2–7 und 10–15 = weiß).

Olivetti wie Hercules.

EGA *mit CGA-Bildschirm*: wie CGA, jedoch auch im 80x25-Modus acht Bildschirmseiten, wenn die Karte mit mehr als 64 KB Bildschirmspeicher ausgestattet ist (das ist üblich).

EGA *mit Monochrom-Bildschirm*: 80x25 Zeichen (vier oder oder Bildschirmseiten) oder 80x43 Zeichen (zwei oder vier Bildschirmseiten); jeweils 16 Farbattribute wie MDPA (die kleinere Zahl bei der Angabe der Bildschirmseiten gilt für 64 KB Bildschirmspeicher).

EGA *mit EGA- oder Multisync-Bildschirm*: 40x25, 40x43, 80x25 und 80x43 Zeichen; jeweils 16 Farbattribute, denen mit PALETTE 64 verschiedene Farben zugeordnet werden können; acht Bildschirmseiten bei 40x25, vier oder acht bei 40x43 und 80x25, zwei oder vier bei 80x43 (die kleinere Zahl gilt bei 64 KB Bildschirmspeicher).

MCGA 40x25 oder 80x25 Zeichen, acht Bildschirmseiten, 16 Farbattribute (Standardfarben 0–15). Es gibt 64 Farben, aber die Zuordnung zu den Attributen kann mit PALETTE nicht geändert werden.

VGA 40x25, 40x43, 40x50, 80x25, 80x43 oder 80x50 Zeichen; 8 Bildschirmseiten in den Modi 40x25, 40x43 und 80x25, sonst vier Bildschirmseiten; bis auf den 80x50-Modus 16 Farbattribute, denen mit PALETTE 64 Farben zugeordnet werden können.

Modus 1 – Grafik 320x200 Punkte

CGA 40x25 Zeichen; eine Bildschirmseite; 16 Hintergrundfarben und zwei Gruppen von je vier Vordergrundfarben (nur eine Gruppe kann benutzt werden, siehe COLOR).

EGA 40x25 Zeichen; eine Bildschirmseite; 4 Farbattribute, denen mit PALETTE 16 Farben zugeordnet werden können.

MCGA wie EGA, jedoch kann PALETTE nicht eingesetzt werden.

VGA wie EGA.

Modus 2 – Grafik 640x200 Punkte

CGA 80x25 Zeichen; eine Bildschirmseite; 2 Farbattribute (0 = schwarz, 1 = weiß).

EGA wie CGA, jedoch können den 2 Farbattributen 16 Farben mit PALETTE zugeordnet werden.

MCGA wie CGA.

VGA wie EGA.

Modus 3 – Grafik 720x348 Punkte

Dieser Modus kann nur dann benutzt werden, wenn das speicherresidente Programm MSHERC.COM zuvor geladen wurde. Auch fertig kompilierte EXE-Programme erfordern leider das vorherige Laden dieses Programms, das Sie jedoch an Benutzer Ihrer Programme weitergeben dürfen.

Die untersten zwei Pixelzeilen der Bildschirmzeile 25 werden abgeschnitten, so daß Buchstaben mit Unterlänge (wie zum Beispiel g) dort schlecht lesbar sind.

Hercules 80x25 Zeichen; effektiv nur eine Bildschirmseite; zwei Farbattribute (0 = schwarz, 1 = weiß).

Modus 4 – Grafik 640x400 Punkte

Dieser Modus wird von den Olivetti-Rechnern M24, M240, M28, M280, M380, M380/C, M380/T und der AT&T–6300-Rechnerserie unterstützt.

Olivetti 80x25 Zeichen; eine Bildschirmseite; 2 Farbattribute: Hintergrundfarbe immer schwarz, als Vordergrundfarbe kann mit COLOR eine von 16 Farben gewählt werden.

Modus 7 – Grafik 320x200 Punkte

Dieser Modus läßt sich auch mit einer EGA-Karte betreiben, an die nur ein CGA-Bildschirm angeschlossen ist.

EGA 40x25 Zeichen; zwei, vier oder acht Bildschirmseiten (bei 64, 128 und 256 KB Bildschirmspeicher); 16 Farbattribute, denen 16 Farben mit PALETTE zugeordnet werden können.

VGA wie EGA, jedoch immer acht Bildschirmseiten.

Modus 8 – Grafik 640x200 Punkte

Dieser Modus läßt sich auch mit einer EGA-Karte betreiben, an die nur ein CGA-Bildschirm angeschlossen ist.

EGA 80x25 Zeichen; eine, zwei oder vier Bildschirmseiten (bei 64, 128 und 256 KB Bildschirmspeicher); 16 Farbattribute, denen 16 Farben mit PALETTE zugeordnet werden können.

VGA wie EGA, jedoch vier Bildschirmseiten bei 256 KB und acht Bildschirmseiten bei 512 KB Bildschirmspeicher.

Modus 9 – Grafik 640x350 Punkte

EGA 80x25 oder 80x43 Zeichen; eine (bei 64 oder 128 KB) oder zwei (bei 256 KB) Bildschirmseiten; 16 Farbattributen können 64 Farben mit PALETTE zugeordnet werden (bei 64 KB stehen nur 4 Farbattribute, aber dennoch 64 Farben zur Verfügung).

VGA 80x25 oder 80x43 Zeichen; zwei (bei 256 KB) oder 4 (bei 512 KB) Bildschirmseiten; 16 Farbattribute, denen 64 Farben mit PALETTE zugeordnet werden können.

Modus 10 – Grafik 640x350 Punkte

Dieser Modus wird nur mit monochromen Monitoren betrieben.

EGA 80x25 oder 80x43 Zeichen; eine, zwei oder vier Bildschirmseiten (bei 64, 128 und 256 KB Bildschirmspeicher); vier Farbattribute, denen mit PALETTE folgende 9 »Farben« zugeordnet werden können: 0 = schwarz, 1 = blinkend chwarz/weiß, 2 = blinkend schwarz/leuchtend weiß, 3 = blin•kend schwarz/weiß; 4 = weiß, 5 = blinkend weiß/leuchtend weiß, 6 = blinkend leuchtend weiß/schwarz, 7 = blinkend leuchtend weiß/weiß, 8 = leuchtend weiß; die Standardzuordnung ist (Farbattribut/Farbe): 0/0, 1/4, 2/1, 3/8.

VGA wie EGA, jedoch vier oder acht Bildschirmseiten bei 256 oder 512 KB Bildschirmspeicher.

Modus 11 – Grafik 640x480 Punkte

MCGA 80x30 oder 80x60 Zeichen; eine Bildschirmseite; zwei Farbattribute, denen mit PALETTE 262.144 Farben zugeordnet werden können.

VGA wie MCGA.

Modus 12 – Grafik 640x480 Punkte

VGA 80x30 oder 80x60 Zeichen; eine Bildschirmseite; 16 Farbattribute, denen mit PALETTE 262.144 Farben zugeordnet werden können.

Modus 13 – Grafik 320x200 Punkte

MCGA 40x25 Zeichen; eine Bildschirmseite; 256 Farbattribute, denen mit PALETTE 262.144 Farben zugeordnet werden können.

VGA wie MCGA.

Die »Standardfarben 0–15« sind:

Farbattribut	*EGA-Farbe*	*Farbe*	*monochrom*
0	0	schwarz	schwarz
1	1	blau	weiß unterstrichen
2	2	grün	weiß
3	3	cyan	weiß
4	4	rot	weiß
5	5	violett	weiß
6	20	braun	weiß
7	7	weiß	weiß
8	56	grau	schwarz
9	57	hellblau	leuchtend weiß unterstr.
10	58	hellgrün	leuchtend weiß
11	59	hellcyan	leuchtend weiß
12	60	hellrot	leuchtend weiß
13	61	hellviol.	leuchtend weiß
14	62	gelb	leuchtend weiß
15	63	hellweiß	leuchtend weiß

Bei EGA-Karten sind den Farbattributen, wenn keine Änderungen mit PALETTE gemacht werden, die unter »EGA-Farbe« genannten Farben zugeordnet. Bei VGA und MCGA existieren ebenfalls solche Standard-Farben, die zwar auf dem Bildschirm identisch mit den genannten aussehen, denen aber andere, unter Umständen von Karte zu Karte verschiedene Werte zugrundeliegen. Diese Standardeinstellung kann jederzeit durch PALETTE ohne Parameter wiederhergestellt werden. Bei CGA-Karten existieren nur diese 16 Farben, und die Farb- und Farbattributnummern sind identisch.

Bemerkung

- Bei VGA- und MCGA-Karten liegen die gültigen Farbcodes dort, wo 262.144 Farben möglich sind, nicht kontinuierlich von 0 ab gezählt, sondern im Bereich von 0 bis 4.144.959. Ein gültiger Farbwert errechnet sich aus der Formel `farbwert = 65.536 * blauwert + 256 * grünwert + rotwert`, wobei für *blauwert*, *grünwert* und *rotwert* nur die Zahlen 0 bis 63 erlaubt sind.

- Bei der EGA-Karte berechnen sich die 64 Farben ebenfalls aus einem *blauwert*, einem *grünwert* und einem *rotwert*, jedoch sind hier für diese drei Zahlen nur die Werte 0–3 erlaubt. Die Formel lautet: `farbwert = (blauwert AND 1)+2*((grünwert AND 1)+2*(((rotwert AND 1)+(blauwert AND 2)) + 2 * ((grünwert AND 2)+2*(rotwert AND 2))))`. Anders ausgedrückt: Bit 1 und 4 des Farbwertes repräsentieren den *blauwert*, Bit 2 und 5 den *grünwert* und Bit 3 und 6 den *rotwert*.

- Wenn an einer VGA-Karte ein Schwarz-Weiß-VGA- oder -Multisync-Monitor betrieben wird, errechnet sich die benutzte Graustufe (0 bis 63 sind möglich) aus der Formel `graustufe = 0,11*blauwert+0,59*grünwert+0,3*rotwert`. Die VGA-Farbe Nr. 1.969.233 zum Beispiel läßt sich nach der in der ersten Bemerkung erwähnten Formel zerlegen in *blauwert* = 30, *grünwert* = 12 und *rotwert* = 81. Die Graustufe dieser Farbe auf einem Schwarz-Weiß-Monitor wäre also nach der Graustufenformel 34. Das gilt allerdings nur für die gewöhnlichen Kathodenstrahl-Bildschirme; die meisten Laptop-LCD- oder Plasma-Schirme können keine 64 Graustufen darstellen und benutzen deshalb eigene Formeln.

Beispiel Durch trickreiche Ausnutzung kann ein Programm mit dem SCREEN-Befehl recht detailliert feststellen, um welches Grafiksystem es sich handelt. Die Funktion GETGRAF (auf der beiliegenden Diskette) ermittelt, welche Grafikkarte angeschlossen ist; bei EGA- und VGA-Karten wird auch festgestellt, wieviel Bildschirmspeicher die Karte hat, und ob ein Monochrom- oder Farbbildschirm angeschlossen ist.

Kompatibel PDS + VBWIN - Formen - Mathe -

Siehe auch PALETTE (643), COLOR (579).

SECOND (Funktion)

Anwendung *x%* = SECOND(*zeitcode#*)

Nutzen Ermittelt die zum angegebenen *zeitcode#* gehörige Sekunde (0–59).

Kompatibel PDS * VBWIN + Formen + Mathe +

Siehe auch HOUR (611), MINUTE (629).

SEEK (Funktion)

Anwendung *x&* = SEEK(*dateinummer*)

Nutzen Gibt die aktuelle Position in einer geöffneten Datei zurück.

Im Unterschied zu LOC gibt SEEK für RANDOM-Dateien die Nummer des Datensatzes, der als nächstes gelesen oder geschrieben würde, zurück (also ist der Funktionswert von SEEK um eins größer als der von LOC). Für BINARY-Dateien gilt dasselbe mit Bytes. Bei sequentiellen Dateien, also OPEN FOR INPUT, OUTPUT oder RANDOM, wird bei SEEK ebenfalls die Nummer des Bytes, das als nächstes gelesen oder geschrieben werden wird, zurück.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch LOC (624), LOF (626), SEEK (Befehl) (669).

SEEK (Befehl)

Anwendung SEEK [#]*dateinummer*, *position&*

Nutzen Setzt die aktuelle Position in einer beliebigen Datei. Auch in sequentiellen Dateien kann die aktuelle Position auf diese Weise »verbogen« werden.

dateinummer ist die Nummer der Datei, bei der die Position gesetzt werden soll, und *position&* ist die neue aktuelle Position. Bei RANDOM-Dateien ist *position&* eine Satznummer, bei allen anderen Dateiarten (ISAM-Dateien werden nicht unterstützt) ist *position&* eine Byte-Position (relativ zum ersten Byte der Datei, das die Nummer 1 hat). *position&* muß zwischen 1 und 2.147.483.647 (der Obergrenze für LONG-Variablen) liegen.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch SEEK (Funktion) (669), GET (für Dateien) (608), PUT (für Dateien) (654).

SELECT CASE (Befehl)

Anwendung

```
SELECT CASE prüfausdruck
CASE ausdrücke
   [befehle]
[CASE ausdrücke
   [befehle]]...
[CASE ELSE
   [befehle]]
END SELECT
```

Nutzen SELECT CASE kann aufwendige IF-Konstruktionen ersetzen. Wenn der Wert von *prüfausdruck* in einer der CASE *ausdrücke*-Zeilen vorkommt, werden alle Befehle bis zum nächsten CASE oder bis zum END SELECT ausgeführt, danach geht es hinter END SELECT weiter. Es dürfen beliebig viele CASE *ausdrücke*-Zeilen benutzt werden, und Befehle können auch noch direkt in der CASE-Zeile (mit Doppelpunkt abtrennen) stehen, was manchmal übersichtlicher ist.

Nach CASE ELSE darf keine weitere CASE-Zeile folgen. Wenn BASIC die CASE ELSE-Zeile erreicht und bisher keine von den ausdrücken in allen CASE-Zeilen zutraf, werden die Befehle zwischen CASE ELSE und END SELECT ausgeführt.

Wenn *prüfausdruck* in mehreren CASE *ausdrücke*-Zeilen enthalten ist, wird nur die Befehlsgruppe ausgeführt, die der ersten zutreffenden CASE *ausdrücke*-Zeile folgt.

ausdrücke hat die Form

```
vergleichsausdruck [, vergleichsausdruck]...
```

Das heißt, daß beliebig viele Ausdrücke der Form *vergleichsausdruck* durch Kommata getrennt aneinandergereiht werden können. *vergleichsausdruck* wiederum kann folgende Formen annehmen (links die Form für *vergleichsausdruck*, rechts die IF-Zeile, der das entspräche):

Form für vergleichsausdruck	*entspräche der IF-Zeile*
ausdruck	IF *prüfausdruck = ausdruck*
ausdruck1 TO *ausdruck2*	IF *ausdruck1* <= *prüfausdruck* AND *prüfausdruck* <= *ausdruck2*
IS *operator ausdruck*	IF *prüfausdruck operator ausdruck*

In der letztgenannten Form mit IS sind als *operator* zulässig: =, <, <=, >, >= und <>.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch IF...THEN...ELSE (611).

SETMEM (Funktion)

Anwendung *x%* = SETMEM(*farspeicheraenderung*)

Nutzen Verändert die Menge an Far-Speicher, die das Programm benutzt. Üblicherweise beansprucht ein BASIC-Programm für sich den ganzen Far-Speicher (»Far Heap«, den Speicher, der im 640K-Bereich noch übrigbleibt, wenn man das abzieht, was das Programm und sein Haupt-Datensegment DGROUP benötigen), egal, wieviel davon wirklich benutzt wird. Das ist in Ordnung, solange nicht irgendeine Routine versucht, von DOS Speicher für eigene Zwecke zugeordnet zu bekommen. Wenn Ihr Programm nun zum Beispiel eine in C programmierte Routine aufruft, die mittels der C-Funktion halloc versucht, Speicher für ihre eigenen Zwecke zu reservieren, wird das fehlschlagen, weil das BASIC-Programm »auf dem Speicher sitzt«. Sie können mit SETMEM die Menge des von BASIC belegten Far-Speichers reduzieren. *farspeicheraenderung* gibt an, wieviele Bytes das Programm vom Far-Speicher für andere Prozeduren freigeben soll (negative Zahl) oder wieviel Far-Speicher es wieder zusätzlich belegen darf (positive Zahl).

Als Funktionswert gibt SETMEM die nach der Änderung nun belegte Menge Far-Speichers in Bytes zurück.

Bemerkung • PRINT SETMEM(0) ergibt die Menge an Far-Speicher, die augenblicklich belegt ist.

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch FRE (604), STACK (676), CLEAR (577).

SGN (Funktion)

Anwendung *x* = SGN(*y*)

Nutzen Entsprechend der Signum-Funktion in der Mathematik gibt SGN –1 zurück, wenn das Argument *y* negativ, 0, wenn *y* 0, und 1, wenn *y* positiv ist.

Kompatibel PDS + VBWIN + Formen + Mathe -

SHARED (Befehl)

Anwendung SHARED *variable*[()] [AS *typ*] [, *variable*[()] [AS *typ*]]...

Nutzen SHARED kann nur im Prozedurcode auftreten und sorgt dafür, daß die betreffende Prozedur die angegebenen Variablen aus dem Modulcode benutzen kann, obwohl sie weder mit COMMON SHARED oder DIM SHARED als globale Variable definiert noch der Prozedur als Parameter übergeben wurden.

Durch Verwendung von SHARED können Sie ein Mittelding zwischen globalen und lokalen Variablen schaffen. Die Variablen sind nicht, wie bei DIM SHARED, allen Prozeduren verfügbar, sondern nur denen, in den sie mit SHARED expliziert angefordert werden.

Runde Klammern hinter dem Variablennamen verwendet man, wenn es sich um ein ganzes Array handelt.

Bemerkung • Die AS *typ*-Klausel wird bei DIM näher erläutert.

• Variablen in einem SHARED-Befehl müssen auch im Hauptprogramm definiert sein, wenn Sie OPTION EXPLICIT verwenden, sonst nicht.

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch DIM (590), COMMON (581), STATIC (677).

SHELL (Befehl)

Anwendung SHELL [*befehl$*]

Nutzen Führt ein anderes Programm oder einen Betriebssystem-Befehl aus. Die Ausführung des BASIC-Programms wird solange unterbrochen; nach Beendigung des aufgerufenen Programms wird das BASIC-Programm weiter ausgeführt.

SHELL lädt eine Kopie des Befehlsinterpreters (COMMAND.COM) in den Speicher und übergibt diesem mit der /C-Option den *befehl$*. Dadurch wird erreicht, daß jeder Befehl oder Programmaufruf, den man unter DOS bzw. OS/2 direkt eingeben könnte, auch mit SHELL aufgerufen werden kann. SHELL ohne Parameter lädt einfach nur einen Befehlsprozessor, so daß beliebig viele Programme aufgerufen und Befehle ausgeführt werden können; durch die Eingabe von EXIT wird der Prozessor verlassen und das aufrufende Programm fortgesetzt.

Bemerkung • Wenn SHELL den Befehlsprozessor nicht finden kann, wird ein *File not found*-Fehler erzeugt.

• Die Funktion SHELL, die im BASIC PDS unter OS/2 und in VBWIN unter WINDOWS verfügbar ist, gibt es in VBDOS nicht.

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch RUN (661), CHAIN (574).

SIN (Funktion)

Anwendung *x* = SIN(*y*)

Nutzen Gibt den Sinus des Arguments *y* zurück. Das Ergebnis wird mit doppelter Genauigkeit berechnet, wenn es sich bei *y* um eine Zahl vom Typ LONG, DOUBLE oder CURRENCY handelt.

y wird als Winkel im Bogenmaß erwartet. Wenn ein Winkel in Grad vorliegt, muß dieser zunächst durch 180/π (57,2957795130824) geteilt werden.

Kompatibel PDS + VBWIN + Formen + Mathe +

Siehe auch COS (583), TAN (682), ATN (570).

SLEEP (Befehl)

Anwendung SLEEP *sekunden*

Nutzen Hält das Programm für *sekunden* Sekunden an. Die Pause wird sofort beendet, wenn eine Taste gedrückt wird oder ein Ereignis eintritt, für das eine Event-Trapping-Routine aktiv ist. *sekunden* muß eine Ganzzahl sein.

Kompatibel PDS + VBWIN - Formen - Mathe -

Verwenden Sie ein Zeitmesser-Steuerelement oder die Systemvariable TIMER, um zu pausieren, während Formen angezeigt werden.

Siehe auch ON *event* GOSUB (634).

SOUND (Befehl)

Anwendung SOUND *frequenz, dauer*

Nutzen Aktiviert den Tongenerator. Es wird ein Ton von der Frequenz *frequenz* (37 bis 32.767 Hz) und der Länge *dauer* generiert, wobei *dauer* ein Wert zwischen 0 und 65.535 ist und die Anzahl der 1/18,2-Sekunden-Intervalle angibt.

Wenn mit dem MB-Kommando des PLAY-Befehls die Musikbefehle auf Hintergrund geschaltet wurden, kann die Programmausführung sofort weitergehen, wenn nicht, wartet der SOUND-Befehl solange, bis der angegebene Ton gespielt wurde.

Bemerkung • Die ungefähren Frequenzwerte der Musiknoten sind: C = 261,25; D = 293,65; E = 329,63; F = 349,23; G = 392,99; A = 440; H = 493,88. Durch Halbieren erreichen Sie eine tiefere, durch Verdoppeln eine höhere Oktave.

Kompatibel PDS + VBWIN - Formen + Mathe +

Siehe auch PLAY (647), BEEP (571).

SPACE$ (Funktion)

Anwendung *x$* = SPACE$(*anzahl*)

Nutzen Erzeugt einen String mit *anzahl* Leerzeichen. *anzahl* darf zwischen 1 und 32.767 liegen.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch SPC (675), STRING$ (679).

SPC (Funktion)

Anwendung SPC(*anzahl*)

Nutzen Diese Funktion nimmt eine Sonderrolle ein, denn sie hat keinen Funktionswert und kann außerdem nur innerhalb von PRINT- und LPRINT-Befehlen benutzt werden. Sie gibt *anzahl* MOD *zeilenbreite* Leerzeichen auf dem Bildschirm bzw. Drucker aus und ist ein Relikt aus grauer BASIC-Vorzeit.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch SPACE$ (394), TAB (406).

SQR (Funktion)

Anwendung *x* = SQR(*y*)

Nutzen Errechnet die Quadratwurzel von *y*. Die Berechnung erfolgt in doppelter Genauigkeit, wenn *y* vom Typ DOUBLE, LONG oder CURRENCY ist. *y* muß größer oder gleich 0 sein.

Bemerkung • Andere Wurzeln als die Quadratwurzel zieht man, indem man den Radikanden mit dem Kehrwert des Exponenten potenziert (für die dritte Wurzel aus x also x^(1/3)).

Siehe auch LOG (626).

SSEG (Funktion)

Anwendung *x%* = SSEG(*variable$*)

Nutzen Gibt die Segmentadresse einer Stringvariable als INTEGER zurück. *variable$* ist die Stringvariable, deren Adresse ermittelt werden soll. Um die vollständige Adresse eines Strings zu erfahren, muß zusätzlich die Funktion SADD benutzt werden, die die Offsetadresse des Strings innerhalb seines Segments zurückgibt. Erst diese beiden Zahlen zusammen ermöglichen das Auffinden eines Strings im Speicher.

SSEG wird nur für Strings variabler Länge benutzt. Verwenden Sie VARSEG für Strings mit fester Länge und andere Variablen.

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch SADD (662), SSEGADD (676), DEF SEG (589), PEEK (645), POKE (649).

SSEGADD (Funktion)

Anwendung *x&* = SSEGADD(*variable$*)

Nutzen Gibt die Segment- und die Offsetadresse einer Stringvariable zugleich als LONG-Variable zurück. *variable$* ist die Stringvariable, deren Adresse ermittelt werden soll.

Bemerkung • Die Zahl, die SSEGADD zurückgibt, kann als »Far Pointer« in der Kommunikation mit anderen Sprachen eingesetzt werden

Siehe auch SADD (662), SSEG (675), VARPTR (688), VARSEG (689).

STACK (Funktion und Befehl)

Anwendung *x* = STACK
STACK stackgroesse

Nutzen Die Funktion STACK gibt die Menge an Speicher zurück, die maximal als Stackspeicher zugeordnet werden kann. Der Befehl STACK ordnet die angegebene *stackgroesse* als Stackspeicher zu.

Bemerkung • Stackspeicher wird benötigt, um Funktionen und Prozeduren aufzurufen, für GOSUB-Befehle usw.

• Wie stark Ihr Programm den Stack benutzt, können Sie mit der FRE-Funktion ermitteln.

• Die Standardgröße für den Stack ist 3 KB; die minimale Größe sind 325 Bytes.

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch CLEAR (577).

$STATIC (Metabefehl)

Anwendung REM $STATIC

Nutzen Alle DIM-Befehle, die auf diesen Metabefehl folgen, erzeugen statische Arrays (Arrays, deren Speicherplatz schon beim Start des Programms zugeordnet wird). Ausgenommen hiervon sind Arrays, die mit Variablen als Index dimensioniert werden und Arrays in nicht-statischen (also automatischen) Subroutinen.

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch DIM (590), $DYNAMIC (595).

STATIC (Befehl)

Anwendung STATIC *variable*[()] [AS *typ*] [, *variable*[()] [AS *typ*]]...

Nutzen STATIC macht bestimmte Variablen lokal zu einer Prozedur. STATIC kann nur im Prozedurcode und in DEF FN-Funktionsdefinitionen benutzt werden. Variablen, die mit STATIC vereinbart sind, sind statische Variablen, das heißt, ihr Wert wird zwischen den Prozeduraufrufen beibehalten. Außerdem setzt der STATIC-Befehl eventuell gültige globale Variablen oder Konstanten außer Kraft (sie werden durch lokale Variablen »überschattet«).

Handelt es sich um Arrays, wird einfach ein Paar runder Klammern angegeben. Es muß dann allerdings noch ein DIM- oder REDIM-Befehl für das Array folgen.

Bemerkung • Eine Beschreibung der AS-Klauseln finden Sie bei DIM. STATIC wird außerdem im Kapitel 9 behandelt.

Kompatibel PDS + VBWIN * Formen + Mathe -

Siehe auch DIM (590), COMMON (581), SHARED (672).

STICK (Funktion)

Anwendung *x* = STICK(*argument*)

Nutzen Erfragt den Status eines Steuerknüppels (Joystick). Der zurückgegebene Funktionswert richtet sich nach dem *argument*:

argument	*Funktionswert*
0	X-Koordinate für ersten Joystick
1	Y-Koordinate für ersten Joystick
2	X-Koordinate für zweiten Joystick
3	Y-Koordinate für zweiten Joystick

Der Aufruf der Funktion mit dem Argument 0 gibt nicht nur die X-Koordinate des ersten Joysticks zurück, sondern ermittelt auch die Werte für die anderen Argumente, die dabei in einen Zwischenspeicher geschrieben werden. Es hat also keinen Sinn, STICK(1) aufzurufen, wenn nicht zuvor ein STICK(0)-Aufruf erfolgte, weil der Zwischenspeicher sonst keine aktuellen Werte enthält.

Die zurückgegebenen Werte haben den Bereich 1 bis 200.

Bemerkung • Den Status der Feuerknöpfe an den Joysticks erfragt man mit der Funktion STRIG.

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch STRIG (Funktion) (678), ON *event* GOSUB (634).

STOP (Befehl)

Anwendung STOP [*exitcode*]

Nutzen STOP hat keinen Nutzen. In der VBDOS-Umgebung wirkt es wie ein Breakpoint, in einem kompilierten Programm wie END oder SYSTEM, mit dem Unterschied, daß eine Meldung angezeigt wird.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch END (595), SYSTEM (681).

STR$ (Funktion)

Anwendung *x$* = STR$(*y*)

Nutzen Wandelt die Zahl *y* in einen String um und gibt diesen String als Funktionswert zurück. Negative Zahlen beginnen mit einem Minus-Zeichen, positive mit einem Leerzeichen. Dezimalstellen werden gemäß amerikanischer Schreibweise immer mit einem Punkt abgetrennt.

Bemerkung • Die Funktion FORMAT$ ist in vielen Fällen bequemer einzusetzen als STR$.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch FORMAT$ (604), VAL (687).

STRIG (Funktion)

Anwendung *x* = STRIG(*argument*)

Nutzen Erfragt den Status der Feuerknöpfe der Joysticks. Welcher Status zurückgegeben wird, hängt vom *argument* ab (der Funktionswert ist immer –1 für ja und 0 für nein):

argument	*Funktion*
0	Wurde der untere Feuerknopf des ersten Joysticks seit dem letzten STRIG(0) gedrückt?
1	Wird der untere Feuerknopf des ersten Joysticks gerade gedrückt?
4	Wurde der obere Feuerknopf des ersten Joysticks seit dem letzten STRIG(0) gedrückt?

5	Wird der obere Feuerknopf des ersten Joysticks gerade gedrückt?
2,3,6,7	Wie 0, 1, 4 und 5, jedoch für zweiten Joystick

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch STICK (677).

STRIG (Befehl)

Anwendung STRIG(*nummer*) ON
STRIG(*nummer*) OFF
STRIG(*nummer*) STOP

Nutzen Schaltet das Event-Trapping für die Feuerknöpfe der Joysticks ein (bevor es wirksam wird, muß noch ein ON STRIG GOSUB-Befehl ausgeführt werden), schaltet es aus oder unterbricht es. Die gültigen Werte für *nummer* sind 0, 2, 4, und 6 (siehe ON *event* GOSUB).

Siehe auch ON *event* GOSUB (634), STRIG (Funktion) (678).

STRING$ (Funktion)

Anwendung (1) *x$* = STRING$(*anzahl, zeichen*)
(2) *x$* = STRING$(*anzahl, zeichen$*)

Nutzen STRING$ bildet einen String aus beliebig vielen gleichen Zeichen. *anzahl* gibt an, wieviele Zeichen der Ergebnis-String enthalten soll (1 bis 32.767; bei 0 gibt es einen *Funktionsaufruf unzulässig*). *zeichen* ist ein numerischer Wert von 0 bis 255 und gibt den ASCII-Code des Zeichens an, mit dem der String gefüllt werden soll. Wird stattdessen ein String angegeben, so wird der Ergebnisstring mit dem ersten Zeichen des angegebenen *zeichen$* gefüllt; die Wirkung von STRING$(*anzahl, zeichen$*) ist also identisch mit der von STRING$(*anzahl*, ASC(*zeichen$*)).

Beispiel STRING$ ist praktisch zum Zeichen von Linien usw.:

```
PRINT "À"; STRING$(78, "Ä"); "Ù"
```

Bedenken Sie aber, daß dabei erst ein temporärer String erzeugt werden muß, was in zeitkritschen Anwendungen ein Nachteil sein kann.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch SPACE$ (674).

SUB/FUNCTION (Befehl)

Anwendung

(1) [STATIC] SUB *prozedurname* [*parameter*]
(2) [STATIC] FUNCTION *funktionsname* [*parameter*] [AS *typ*]
...
(1) END SUB
(2) END FUNCTION

wobei *parameter* die Form

[BYVAL] *variable*[()] [AS *typ*] [, *parameter*]

hat.

Nutzen

Definiert eine Prozedur oder Funktion. Aller Programmcode zwischen dem SUB- und END SUB-Befehl (bzw. dem FUNCTION- und END FUNCTION-Befehl) ist Prozedurcode.

prozedurname ist ein beliebiger gültiger Name für die Prozedur bzw. Funktion. Bei FUNCTION darf dieser Name einen Typenbezeichner enthalten, um den Typ des Rückgabewertes anzuzeigen; in VBDOS ist es jedoch erstmals möglich, diesen Rückgabetyp auch mit einem ordentlichen *AS typ* am Ende der Zeile anzugeben.

parameter ist die optionale Liste von Variablen, die übergeben werden sollen. Wenn Sie hier Angaben machen, dann müssen in jedem Aufruf der Funktion oder Prozedur Variablen des angegebenen Typs in der gleichen Reihenfolge angegeben werden. Die Namen der Variablen spielen hingegen keine Rolle. Innerhalb der Prozedur werden sie unter dem Namen benutzt, der im SUB- oder FUNCTION-Befehl steht, außerhalb aber können sie ganz andere Namen haben.

Wenn ein ganzes Array als Parameter übergeben werden soll, müssen hinter den Variablennamen eine offene und eine geschlossene runde Klammer zur Kennzeichnung gesetzt werden. Die Typen der Parameter können entweder durch Typenbezeichner oder durch das Schlüsselwort AS gefolgt von einem der Typen INTEGER, LONG, CURRENCY, SINGLE, DOUBLE, STRING oder dem Namen eines selbstdefinierten Typen gekennzeichnet werden. In VBDOS sind zusätzlich die Objekttypen FORM und CONTROL erlaubt (siehe S. Kap. 25). Strings mit fester Länge können nicht in der Liste auftauchen, allerdings können solche Strings auch an Prozeduren übergeben werden, die einen String variabler Länge erwarten.

Das Schlüsselwort BYVAL vor einem Parameter bewirkt, daß dieser Parameter nur als Werteparameter übergeben wird. Beim Aufruf wird der Prozedur nur der Wert der Variablen übergeben, quasi eine Kopie der Variable. Die Prozedur kann dann zwar an diesem Wert manipulieren, aber nach dem Verlassen der Prozedur hat die Variable wieder den ursprünglichen Wert. Im Normalfall wird eine Variable als Variablenparameter übergeben, so daß Änderungen, die die Prozedur an ihr macht, auch im aufrufenden Programm gültig bleiben. Bei der Übergabe von Objekteigenschaften an Prozeduren ist ausschließlich die BYVAL-Übergabe möglich; will man eine Objekteigenschaft an eine Prozedur übergeben, die eine Variable »by reference« erwartet, muß sie in Klammern eingeschlossen werden (vgl. CALL).

Wenn Sie das Schlüsselwort STATIC bei der Prozedur- oder Funktionsdefinition angeben, dann definieren Sie eine statische Subroutine. Das bedeutet, daß alle Variablen statisch sind, so, als wären sie alle einzeln mit dem STATIC-Befehl vereinbart worden. Im Gegensatz zum STATIC-Befehl sind globale Konstanten und Variablen jedoch dann weiterhin gültig.

Mit dem Befehl EXIT SUB bzw. EXIT FUNCTION kann die Prozedur/Funktion jederzeit verlassen werden.

Bei Funktionen wird der Funktionswert mittels einer gewöhnlichen Zuweisung (Gleichheitszeichen) irgendwo in der Funktion zugeordnet.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch CALL (572), DECLARE (586).

SWAP (Befehl)

Anwendung SWAP *variable1, variable2*

Nutzen Vertauscht den Inhalt beider angegebenen Variablen. Sie müssen exakt den gleichen Datentyp haben. Lediglich bei Strings ist BASIC kulant: sie müssen nicht dieselbe Länge haben, um mit SWAP getauscht werden zu können.

Kompatibel PDS + VBWIN - Formen + Mathe -

SYSTEM (Befehl)

Anwendung SYSTEM [*errorlevel%*]

Nutzen Beendet ein Programm. Die Wirkung von SYSTEM ist völlig identisch mit der von END.

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch STOP (678), END (595).

TAB (Funktion)

Anwendung TAB(*n*)

Nutzen Ähnlich wie auch SPC nimmt TAB eine Sonderrolle unter den Funktionen an, weil TAB nur bei den Ausgabebefehlen PRINT und LPRINT (und der PRINT-Methode) benutzt werden kann und, anstatt einen Funktionswert zurückzugeben, den Cursor in eine bestimmte Spalte setzt. Diese Spalte wird mit *n* angegeben. Ist *n* kleiner als die Anzahl der bisher gesendeten Zeichen auf der aktuellen Zeile (die mit POS(0) oder LPOS ermittelt werden kann), so wird eine neue Zeile begonnen, um die gewünschte Spalte erreichen zu können.

TAB zu verwenden wird unmöglich, sobald die Anzahl der gesendeten (bzw. ausgegebenen) Zeichen nicht mehr direkt auf die Cursorposition schließen läßt, wenn Sie also zum Beispiel Zeichen drucken, die nicht sichtbar sind.

Kompatibel PDS + VBWIN * Formen + Mathe -

Siehe auch SPC (675), SPACE$ (674).

TAN (Funktion)

Anwendung *x* = TAN(*y*)

Nutzen Gibt den Tangens des Arguments *y* zurück. Das Ergebnis wird mit doppelter Genauigkeit berechnet, wenn es sich bei *y* um eine Zahl vom Typ LONG, DOUBLE oder CURRENCY handelt.

y wird als Winkel im Bogenmaß erwartet. Wenn ein Winkel in Grad vorliegt, muß dieser zunächst durch 180/π (57,2957795130824) geteilt werden.

Kompatibel PDS + VBWIN + Formen + Mathe +

Siehe auch SIN (673), COS (583), ATN (570).

TIME$ (Systemvariable)

Anwendung
```
x$ = TIME$
TIME$ = x$
```

Nutzen TIME$ enthält stets die Uhrzeit der Systemuhr im 24-Stunden-Format *hh*:*mm*:*ss*. Man kann in diese Variable eine neue Systemzeit eintragen; dabei kann auf Sekunden und Minuten verzichtet werden.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch DATE$ (585), NOW (631), TIMER (683).

TIMER (Funktion)

Anwendung
```
x = TIMER
```

Nutzen TIMER gibt die Anzahl der Sekunden zurück, die seit 0 Uhr verstrichen sind. Die Zahl liegt also zwischen 0 und 86.400; auch Sekundenbruchteile werden zurückgegeben.

Bemerkung • TIMER ist praktisch, um den Zufallsgenerator mit RANDOMIZE zu initialisieren.

Kompatibel PDS + VBWIN + Formen + Mathe +

Siehe auch TIMER$ (683).

TIMER (Befehle)

Anwendung
```
TIMER ON
TIMER OFF
TIMER STOP
```

Nutzen Schaltet das Event-Trapping für die Systemuhr ein (bevor es wirksam wird, muß noch ein ON TIMER(*n*) GOSUB-Befehl ausgeführt werden), schaltet es aus oder unterbricht es. Weitere Erklärungen finden Sie bei ON *event* GOSUB.

Siehe auch ON *event* GOSUB (610), TIMER (Funktion).

TIMESERIAL (Funktion)

Anwendung *x#* = TIMESERIAL(*stunde%, minute%, sekunde%*)

Nutzen Gibt den Zeitcode der angegebenen Uhrzeit zurück. Für alle drei Werte können Sie nahezu beliebige Zahlen angeben. Wenn Sie normale (allgemeingültige) Zeitangaben machen, wird sich der Funktionswert zwischen 0 (für 0 Uhr, 0 Minuten, 0 Sekunden) und 0,99999 (für 23 Uhr, 59 Minuten und 59 Sekunden) bewegen.

Solange Ihre Werte nicht über die 1-Tag-Grenze hinausgehen, ist der Wert, den Sie durch die Funktion erhalten, ein reiner Uhrzeit-Zeitcode, also eine Zahl zwischen 0 und 0,99999, ohne Vorkomma-Anteil. Überschreiten Sie diese Grenze, indem Sie zum Beispiel –10 Stunden angeben, was einem Zeitpunkt am Vortag entspricht, müssen Sie sinnvollerweise noch einen reinen Datums-Zeitcode addieren, damit diese Überschreitung korrekt behandelt wird.

Kompatibel PDS * VBWIN + Formen + Mathe +

Siehe auch DATESERIAL (586), TIMEVALUE (684).

TIMEVALUE (Funktion)

Anwendung *x#* = TIMEVALUE(*text$*)

Nutzen Diese Funktion arbeitet wie TimeSerial, mit dem Unterschied, daß ihr nicht drei INTEGER-Zahlen für Stunde, Minute und Sekunde übergeben werden, sondern eine beliebig formatierte Zeitangabe. Außerdem dürfen hier die Werte nicht, wie das bei TimeSerial möglich war, den üblichen Bereich überschreiten (0–23 für die Stunde, 0–59 für Minute und Sekunde). TIMEVALUE erkennt auch 12-Stunden-Zeitangaben mit AM oder PM korrekt. In *text$* enthaltene Datumsangaben werden ignoriert, wenn sie gültig sind, und führen zu einem *Funktionsaufruf unzulässig*, wenn sie ungültig (oder nicht in amerikanischer MM/DD-Notation) sind.

Kompatibel PDS * VBWIN + Formen + Mathe +

Siehe auch DATEVALUE (586), TIMESERIAL (684).

TRON, TROFF

Anwendung
```
TRON
TROFF
```

Nutzen Diese Befehle sind ein Relikt aus grauer BASIC-Vorzeit. Innerhalb von VBDOS hatt die Ausführung eines TRON-Befehls dieselbe Wirkung wie die Aktivierung des Menüpunktes »Ablauf verfolgen« aus dem Menü »Testen«. TROFF deaktiviert den Verfolgungsmodus. In kompilierten Programmen führt ein TRON zur Ausgabe aller Zeilennummern (nicht -labels), die passiert werden.

Kompatibel PDS + VBWIN - Formen + Mathe -

TYPE (Befehl)

Anwendung
```
TYPE typname
   element[(index)] AS typ
   ...
END TYPE
```

Nutzen Vereinbart einen selbstdefinierten Typ. Eine solche TYPE-Vereinbarung sollte immer möglichst weit am Programmanfang stehen.

typname ist der Name des neuen Typs. Die Erfahrung hat gezeigt, daß es praktisch ist, den Namen immer auf ...typ enden zu lassen.

element ist der Name eines Elements innerhalb dieses Typs (wie ein Variablenname frei wählbar). Wenn Sie hinter *element* in Klammern einen *index* (entweder als einfache Zahl oder in der Form zahl1 TO zahl2) angeben, definieren Sie ein Array innerhalb des Typs. Hinter der AS-Klausel steht der Typ dieses Elements; das kann entweder INTEGER, LONG, CURRENCY, SINGLE, DOUBLE, STRING * *x* (String mit fester Länge) oder der Name eines anderen, zuvor definierten Typs sein. Strings mit variabler Länge sind nicht erlaubt.

Bedenken Sie, wenn Sie den Typ zum Zugriff auf ISAM-Dateien (professionelle Ausgabe) benutzen wollen, daß Sie dann keine SINGLE-Elemente benutzen dürfen und daß ein einzelner Elementname ebenso wie der Typ-Name nicht länger als 30 Zeichen sein darf.

Generell kann ein Typ insgesamt nicht mehr als 65.535 Bytes umfassen.

Bemerkung • Um Variablen von selbstdefiniertem Typ zu vereinbaren, ist in der Hauptsache der DIM-Befehl zu verwenden.

Kompatibel PDS + VBWIN * Formen + Mathe -

Siehe auch DIM (590).

UBOUND (Funktion)

Anwendung *x* = UBOUND(*arrayname* [, *dimension*])

Nutzen Gibt die obere Dimensionierungsgrenze eines beliebigen Arrays zurück. Bei mehrdimensionalen Arrays kann mit *dimension* noch angegeben werden, die obere Grenze welcher Dimension gewünscht ist; wird keine Dimension angegeben, ermittelt UBOUND die obere Grenze der ersten Dimension. (Die obere Grenze ist der größtmögliche Index.)

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch DIM (590), LBOUND (620).

UCASE$ (Funktion)

Anwendung *x$* = UCASE$(*y$*)

Nutzen Wandelt alle Kleinbuchstaben in Großbuchstaben um. Der Funktionswert der Funktion ist ein String, der dieselbe Länge hat wie das Argument *y$*, in dem jedoch alle von VBDOS erkannten Kleinbuchstaben in Großbuchstaben umgewandelt werden. Folgende Umwandlungen werden durchgeführt:

von ASCII	Zeichen	nach ASCII	Zeichen	von ASCII	Zeichen	nach ASCII	Zeichen
97–122	a-z	65–90	A-Z	141	ì	73	I
129	ü	154	Ü	145	æ	146	Æ
130	é	69	E	147	ô	79	O
131	â	65	A	148	ö	153	Ö
132	*	142	Ä	149	ò	79	O
133	à	65	A	150	û	85	U
134	å	143	Å	151	ù	85	U
135	ç	128	Ç	152	ÿ	89	Y
136	ê	69	E	160	á	65	A
137	ë	69	E	161	í	73	I
138	è	69	E	162	ó	79	O
139	ï	73	I	163	ú	85	U
140	î	73	I	164	ñ	165	Ñ

Bemerkung • Beachten Sie, daß Text$=LCASE$(UCASE$(Text$)) nicht immer wahr sein muß, da LCASE$ weniger Zeichen umwandelt als UCASE$.

Kompatibel PDS * VBWIN + Formen + Mathe -

Das PDS unterstützt nur die 26 Standard-Buchstaben.

Siehe auch LCASE$ (621).

UEVENT (Befehle)

Anwendung UEVENT ON
UEVENT OFF
UEVENT STOP

Nutzen Aktiviert, deaktiviert oder unterbricht das Event-Trapping für den »User-defined Event«. UEVENT ON hat keinen Sinn, wenn nicht zuvor ein ON UEVENT GOSUB ausgeführt wurde. Der »User-defined Event« tritt auf, wenn irgendein Programm oder eine in anderer Sprache geschriebene Prozedur die Prozedur SetUEvent aufruft. Ist UEVENT ON aktiv, wird dann die Event-Trapping-Routine, die mit ON UEVENT GOSUB angegeben wurde, aufgerufen. UEVENT OFF beendet diese Bearbeitung des UEVENT-Ereignisses; UEVENT STOP unterbricht sie, aber beim nächsten UEVENT ON wird auf alle UEVENTs reagiert, die während der UEVENT STOP-Zeit auftraten.

Bemerkung • Mehr über SetUEvent finden Sie unter ON *event* GOSUB.

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch ON *event* GOSUB (634).

VAL (Funktion)

Anwendung *x* = VAL(*y$*)

Nutzen Ermittelt den Wert von *y$* und gibt ihn als Funktionswert zurück. Tab-Zeichen (ASCII 9), Linefeed-Zeichen (ASCII 10) und Leerzeichen werden völlig ignoriert; der erste Dezimalpunkt im String wird als Dezimalpunkt interpretiert, ein vor der ersten Ziffer befindliches Minus-Zeichen wird ebenfalls erkannt. Jedes andere Zeichen sowie jeder weitere Dezimalpunkt und auch Minus-Zeichen nach Ziffern führen zum Abbruch der Auswertung.

Kompatibel PDS + VBWIN + Formen + Mathe +

Die VAL-Funktion benötigt die Mathematik-Bibliotheken, wenn nicht mit NOFLTIN.OBJ (professionelle Ausgabe) gelinkt wird.

Siehe auch STR$ (678).

VARPTR (Funktion)

Anwendung *x* = VARPTR(*variable*)

Nutzen Gibt die (Offset-)Adresse einer beliebigen Variable zurück. Lediglich für Strings ist die Verwendung von VARPTR eingeschränkt; hier wird die Adresse des Stringdeskriptors zurückgegeben. Der Stringdeskriptor enthält wiederum die Adresse des Strings selbst sowie seine Länge, aber viel praktischer ist es, die Stringadresse mit SSEG und SADD abzufragen, weil man sich dann nicht um den Stringdeskriptor kümmern muß.

Die Adresse von Variablen ist nötig, um sie beispielsweise an Routinen in anderen Sprachen zu übergeben, sie mit PEEK und POKE direkt zu manipulieren oder sie mit BSAVE und BLOAD zu laden und zu speichern.

Bemerkung • Die komplette Adresse einer Variable besteht aus der Segment- und der Offsetadresse. Die Segmentadresse erhält man mit VARSEG. Nur Variablen im Segment DGROUP können auch ohne Segmentadresse angesprochen werden, aber man geht auf Nummer sicher, wenn man grundsätzlich die Segmentadresse mitbenutzt.

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch VARSEG (689), SSEG (675), SADD (662), BSAVE (571), BLOAD (571), PEEK (645), POKE (649).

VARPTR$ (Funktion)

Anwendung *x$* = VARPTR$(*variable*)

Nutzen VARPTR$ gibt einen als drei Byte langer String codierten Zeiger auf die *variable* zurück. VARPTR$ sollte ausschließlich zur Verwendung in Verbindung mit PLAY und DRAW dienen (siehe dort).

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch PLAY (647), DRAW (593).

VARSEG (Funktion)

Anwendung `x = VARSEG(variable)`

Nutzen Gibt die Segmentadresse einer beliebigen Variable zurück. Für Strings gelten dieselben Einschränkungen wie bei VARPTR. Bevor man mit einem Befehl wie PEEK, POKE, BSAVE, BLOAD o.*. auf die Variable zugreift, muß man mit DEF SEG = *segment-adresse* die Segmentadresse der Variable einstellen, da bei den Befehlen selbst immer nur die Offsetadresse angegeben wird.

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch VARPTR (688), BSAVE (571), BLOAD (571), PEEK (645), POKE (649).

VIEW (Befehl)

Anwendung `VIEW [[SCREEN] (x1, y1)-(x2, y2) [,[farbe] [,rahmen]]]`

Nutzen VIEW kann nur im Grafikmodus angewendet werden und definiert einen Grafik-Viewport. Alle zukünftigen Grafikbefehle wirken nur noch in dem durch VIEW definierten Bildschirmausschnitt; der Bereich um den Ausschnitt herum wird von Grafikbefehlen nicht verändert.

Wenn SCREEN nicht angegeben wird, dann werden alle Koordinaten in zukünftigen Grafikbefehlen als relativ zur oberen linken Ecke des Viewports interpretiert, so daß der Punkt (0,0) also stets im Viewport liegt. Mit dem SCREEN-Schlüsselwort jedoch wird vereinbart, daß Koordinaten weiterhin als relativ zur oberen linken Ecke des *Bildschirms* zu verstehen sind, also läge dann der Punkt (0,0) nur im Viewport, wenn dieser in der oberen linken Bildschirmecke begänne.

farbe ist das Farbattribut, mit dem der Viewport gefüllt werden soll. Wenn Sie *farbe* weglassen, wird der Viewport nicht gelöscht oder gefüllt.

rahmen ist ebenfalls ein Farbattribut, diesmal jedoch für den Rahmen, der um den neuen Viewport gezeichnet werden soll. Das Weglassen von *rahmen* verhindert das Zeichnen eines Rahmens.

Wenn Sie alle Angaben weglassen und nur den Befehl VIEW benutzen, wird der ganze Bildschirm zum Viewport.

Kompatibel PDS + VBWIN - Formen - Mathe -

VIEW PRINT (Befehl)

Anwendung VIEW PRINT [*vonzeile* TO *biszeile*]

Nutzen Bestimmt den Text-Viewport. Alle Textausgabe-Befehle richten sich auf den Text-Viewport; im Unterschied zum Grafik-Viewport werden bei der Einrichtung eines Text-Viewports jedoch immer noch die alten Koordinaten beibehalten, und außerdem kann der Text-Viewport nur zeilen- und nicht spaltenweise festgelegt werden.

Läßt man *vonzeile* und *biszeile* weg, so wird der ganze Bildschirm zum Text-Viewport. Das ist der Normalzustand. Gibt man hingegen *vonzeile* und *biszeile* an, so ist die Textausgabe von da an nur noch im angegebenen Zeilenbereich zulässig; ein LOCATE außerhalb dieses Bereichs führt zum Fehler *Funktionsaufruf unzulässig*, und wenn durch PRINT der Bereich überschritten würde, wird der Inhalt des Bereichs nach oben verschoben (»gescrollt«), so, wie es auch üblich ist, wenn ein PRINT-Befehl in der letzten Bildschirmzeile ausgeführt wird.

Kompatibel PDS + VBWIN - Formen - Mathe -

Siehe auch WINDOW (692).

WAIT (Befehl)

Anwendung WAIT *kanal*, *and-bedingung* [, *xor-bedingung*]

Nutzen Wartet so lange, bis an einem bestimmten Hardware-Kanal eine bestimmte Byte-Kombination auftritt. *and-bedingung* und *xor-bedingung* werden mit jedem eingehenden Byte verknüpft (beide Bedingungen sind Werte zwischen 0 und 255; wenn die *xor-bedingung* weggelassen wird, ist sie 0). Wenn dabei ein Wert übrigbleibt, der nicht 0 ist, wird das Programm fortgesetzt, sonst wird weiter gewartet.

Der Befehl WAIT *k, a, x* läßt sich umschreiben mit DO: *z* = INP(*a*): LOOP UNTIL ((*z* XOR *x*) AND *a*) > 0.

Kompatibel PDS + VBWIN - Formen + Mathe -

Siehe auch INP (614), OUT (642).

WEEKDAY (Funktion)

Anwendung *x%* = WEEKDAY(*zeitcode#*)

Nutzen Gibt den Wochentag des Datums zurück, das im angegebenen *zeitcode#* enthalten ist. Der Funktionswert reicht von 1 (Sonntag) bis 7 (Samstag).

Kompatibel PDS * VBWIN + Formen + Mathe +

Siehe auch DAY (587).

WHILE...WEND (Befehl)

Anwendung WHILE *bedingung*

...

WEND

Nutzen WHILE...WEND ist ein Strukturbefehl, der völlig identisch ist mit DO WHILE *bedingung*...LOOP. WHILE existierte jedoch schon vor DO...LOOP, das vielseitiger ist, und wurde aus Kompatibilitätsgründen beibehalten. Die Befehle zwischen WHILE und WEND werden solange ausgeführt, wie *bedingung* nicht 0 ist.

Kompatibel PDS + VBWIN + Formen + Mathe -

Siehe auch DO...LOOP (592).

WIDTH (Befehl)

Anwendung (1) WIDTH [*spalten*] [, *zeilen*]

(2) WIDTH {LPRINT|*#dateinummer*|*geraet$*}, *zeilenbreite*

Nutzen Setzt die Breite (und Höhe) des Bildschirms oder einer Datei bzw. eines Geräts.

Am Bildschirm (Syntax 1) macht sich die geänderte Breite direkt bemerkbar, weil der Bildschirmmodus umgeschaltet wird, was ein CLS verursacht. *spalten* kann hier entweder 40 oder 80 sein; *zeilen* darf 25, 43, 50 oder 60 sein, je nach Grafikkarte und Bildschirmmodus, der mit SCREEN eingestellt wird. Entweder *zeilen* oder *spalten* kann weggelassen werden, dann wird der aktuelle Wert beibehalten.

Auflösung	*ist möglich mit...*
80x25	allen Grafikkarten im SCREEN 0, außerdem in den SCREEN-Modi 2, 3, 4, 8, 9 und 10.
40x25	CGA, EGA, MCGA, VGA im SCREEN 0, außerdem in den SCREEN-Modi 1, 7 und 13.
80x30	SCREEN-Modi 11 und 12
80x43	EGA und VGA im SCREEN 0, außerdem in den SCREEN-Modi 9 und 10.
40x43	EGA und VGA im SCREEN 0
80x50	VGA im SCREEN 0.
40x50	VGA im SCREEN 0.
80x60	SCREEN-Modi 11 und 12.

Die Syntax 2 wird benutzt, um die Zeilenbreite für LPRINT-Befehle oder bestimmte Geräte (wie zum Beispiel COM1:) zu setzen. WIDTH LPRINT und WIDTH *#dateinummer*, wobei *dateinummer* eine Dateinummer sein muß, unter der ein Gerät geöffnet ist, wirken sofort. Ein WIDTH *geraet$*, zum Beispiel WIDTH "LPT1:", wirkt erst dann, wenn das genannte Gerät das nächste Mal mit OPEN geöffnet wird.

Wenn für eine Datei oder ein Gerät eine Zeilenbreite gesetzt ist, sorgt BASIC dafür, daß keine Zeile länger als diese Zeilenbreite wird. Notfalls fügt es einfach die Steuersequenz für »neue Zeile« (CHR$(13)+CHR$(10)) ein, um eine Zeile in zwei Teile zu zerschneiden. WIDTH #dateinummer hat auf gewöhnliche Dateien keine Wirkung.

Kompatibel PDS + VBWIN * Formen * Mathe -

In VBWIN ist nur die Dateivariante WIDTH # zulässig. In VBDOS darf WIDTH, während Formen angezeigt werden, nicht auf den Bildschirm angewandt werden; wohl aber zuvor. Die 40-Spalten-Modi sind jedoch mit Formen nicht einsetzbar.

Siehe auch LPOS (339).

WINDOW (Befehl)

Anwendung WINDOW [[SCREEN] (*x1*, *y1*)-(*x2*, *y2*)]

Nutzen Mit dem WINDOW-Befehl läßt sich das übliche Pixel-Koordinatensystem im Grafikmodus durch ein logisches Koordinatensystem ersetzen.

Üblicherweise ist die obere linke Ecke des Bildschirms der Punkt (0,0), und von da an breiten sich in x-Richtung 320 bis 720 und in y-Richtung 200 bis 480 »Pixel« (Bildschirmpunkte) aus. Mit WINDOW ist es möglich, ein logisches Koordinatensystem (zum Beispiel –1 bis 1 in beiden Richtungen) zu installieren, das das starre Pixel-System ersetzt. Die neuen Koordinaten können dann bei jeder künftigen Grafikanweisung benutzt werden; sie werden von BASIC in Pixel umgerechnet und auf dem Bildschirm dargestellt.

Wenn Sie das Schlüsselwort SCREEN mit angeben, dann wird die gewohnte Richtung der y-Achse (kleinster Wert oben, größter unten) beibehalten. Lassen Sie SCREEN weg, ist von nun an der kleinste y-Wert unten und der größte oben.

x1 und *y1* sind die neuen logischen Koordinaten für die linke obere (mit SCREEN) oder linke untere (ohne SCREEN) Ecke. *x2* und *y2* sind die logischen Koordinaten der gegenüberliegenden Ecke.

WINDOW allein, ohne Parameter, stellt das originäre Pixel-Koordinatensystem wieder her.

Bemerkung

- Geschickt eingesetzt, kann der WINDOW-Befehl viel Arbeit ersparen. Wenn Sie zum Beispiel ein Programm, das für die CGA-Auflösung 640x200 Punkte programmiert wurde, nun auf einer EGA-Karte mit 640x350 Punkten laufen lassen wollen, brauchen Sie im Prinzip nur einen Befehl wie `WINDOW SCREEN (0,0)-(640,200)` auf der EGA-Karte auszuführen, und schon läuft das Programm anstandslos.
- Wenn ein VIEW-Befehl aktiv ist, dann bezieht sich der WINDOW-Befehl nur auf den damit definierten Grafik-Viewport.

Kompatibel PDS + VBWIN - Formen - Mathe +

Siehe auch SCREEN (663), VIEW (689).

WRITE (Befehl)

Anwendung `WRITE [#`*`dateinummer`*`,] [`*`liste`*`]`

Nutzen Schreibt eine Anzahl von Werten auf den Bildschirm oder in eine Datei. *liste* enthält die auszugebenden Werte, durch Kommata getrennt. Alle Werte werden, ebenfalls durch Komma getrennt, auf dem Bildschirm bzw. in die Datei ausgegeben. Strings werden dabei automatisch in Anführungszeichen eingeschlossen.

Bemerkung • WRITE stammt noch aus der Zeit, in der es keine andere Möglichkeit zur Datenspeicherung gab, als die Daten in eine sequentielle Datei zu schreiben und mit INPUT wieder zu lesen.

Kompatibel PDS + VBWIN * Formen * Mathe -

In VBWIN und während der Anzeige von Formen ist nur die Datei-Version von WRITE zulässig.

Siehe auch PRINT (651), INPUT (615).

YEAR (Funktion)

Anwendung *x%* = YEAR(*zeitcode#*)

Nutzen Ermittelt das zum angegebenen *zeitcode#* gehörige Jahr (1753–2078).

Kompatibel PDS * VBWIN + Formen + Mathe +

Siehe auch DAY (587), MONTH (631).

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Frederik Ramm

Programmieren mit Visual Basic für DOS

Software-Entwicklung mit der Standard- und der Professional-Edition

Was das Buch bietet . . .

- eine professionelle Anleitung zum effizienten Software-Design mit einem visuellen Entwicklungssystem und eine fundierte Unterweisung in die Konzepte der ereignisorientierten Programmierung.

Worum es geht . . .

- Ereignisorientierte Programmierung
- Praktische Algorithmik
- ISAM-Datenbanken
- DOS- und BIOS-Interrupts
- Nutzung von EMS mit Visual Basic

Und außerdem . . .

- Trickreiche Programmodule (z. B. effiziente Sortieralgorithmen und ein phonetischer Suchalgorithmus)
- „Programmpatches" zur Nutzbarmachung von Features der Professional Edition in der Standardversion von Visual Basic

Was der Leser benötigt . . .

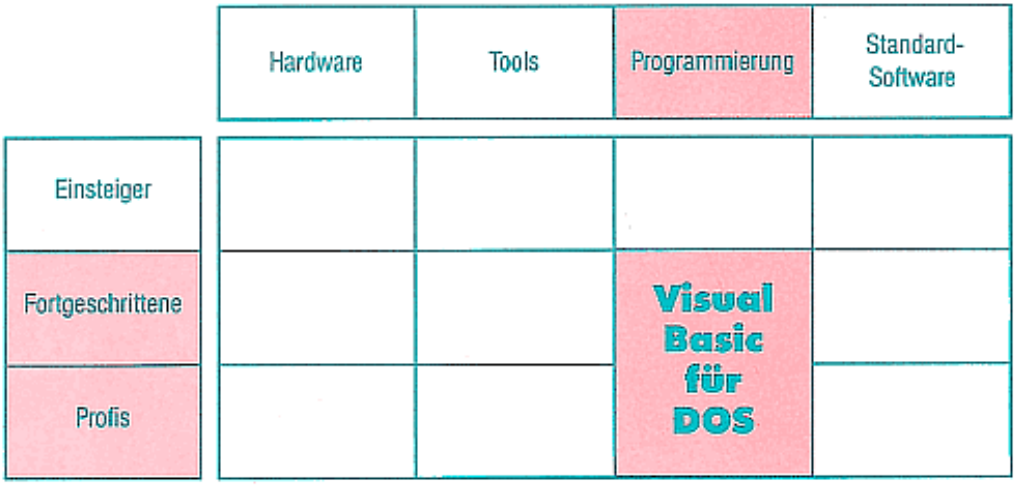
- HARDWARE: IBM PC und Kompatible
- SOFTWARE: Visual Basic für DOS (Standard- oder Professional Edition)

Besondere Kennzeichen . . .

- Das Buch orientiert sich in allen Punkten stark an den Erfordernissen der modernen Softwareentwicklung. Es ist gespickt mit zahlreichen ausgeklügelten Programmiertricks aus dem Erfahrungsschatz eines „Praktikers".

Der Autor . . .

- Frederik Ramm ist Student des Faches Wirtschaftsingenieurwesen. Ferner ist er in der Softwareentwicklung und als Fachbuchautor tätig.



9 783528 053383 07900

ISBN 3-528-05338-0 DM 079.00

Buchregal-Hinweis:
Programmierung