



FREDERIK RAMM

**Microsoft®**

# **BASIC PDS**

**Version 7.1**

**inkl. Diskette**

Programmieren  
mit dem  
Microsoft Professional  
Development System





FREDERIK RAMM

# ***Microsoft®*** **BASIC PDS 7.1**

**Programmieren mit dem Microsoft  
Professional Development System**

**2., überarbeitete Auflage**



Die Deutsche Bibliothek – CIP Einheitsaufnahme für die 2. Auflage

**Ramm, Frederik:**

Microsoft BASIC PDS 7.1: Programmieren mit dem Microsoft  
Professional Development System / Frederik Ramm. –  
2., überarb.Aufl. – Braunschweig; Wiesbaden: Vieweg, 1992  
ISBN 3-528-15152-8

1. Auflage 1991

2., überarbeitete Auflage 1992

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

Das Werk einschließlich aller seiner Teile ist urheberrechtlich geschützt.  
Sämtliche Verwertungsrechte liegen beim Autor Frederik Ramm –  
<http://www.remote.org/frederik/> . Eine Veröffentlichung des Buches in  
elektronischer Form, auch im Internet, und in gedruckter Form ist nur mit  
ausdrücklicher Genehmigung des Autors gestattet

# 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 unverantwortlich, durch ganze Absätze hindurch die berüchtigten Doppelsubjekt-Querstrichkonstruktionen zu führen, und auch die moderne Schreibweise mit großem I im Wort stört den Lesefluß zu sehr. So benutze ich die althergebrachte Schreibweise und hoffe auf Ihr Verständnis.*

BASIC ("Beginner's all-purpose symbolic instruction code", also etwa "Allzweck-Programmiersprache für Anfänger") ist eine Programmiersprache mit einer langen Geschichte. Ihr Siegeszug auf PCs begann damit, daß diese serienmäßig neben dem Betriebssystem auch mit einem BASIC-Interpreter ausgestattet wurden. Die Bedienung dieser Interpreter (BASICA, GWBASIC) war unkomfortabel, und die Programme liefen äußerst langsam (das typische Interpreter-Handicap\*).

Bald gab es erste BASIC-Compiler. Vielleicht haben Sie schon einmal auf einer alten Diskette eine Datei namens "BASRUN.EXE" entdeckt, die das System zum Absturz brachte, wenn man sie als Programm starten wollte. Das ist die Signatur des ersten IBM BASIC-Compilers 1.0 - sein "Runtime-Modul", das von den kompilierten Programmen gebraucht wurde.

Aber die Konkurrenz schlief nicht: Turbo Pascal kam auf den Markt und in Mode, und BASIC-Programmierer, die vor jeden Befehl mühsam eine Zeilennummer setzen mußten, deren Programme ein Fremder kaum lesen konnte, wurden gern belächelt.

Da erblickte QuickBASIC 1 das Licht der Welt - ein gewöhnlicher Compiler, der allerdings einige für die damalige Zeit interessante Möglichkeiten offenbarte. Man konnte Zeilennummern weglassen und Prozeduren schreiben - fast wie die "richtigen" Programmierer. Zwar war bei Pascal und Konsorten immer noch vieles besser, aber nun durfte man hoffen.

---

\* Ein Interpreter übersetzt immer nur den Befehl, den er gerade ausführen muß, ein Compiler dagegen übersetzt das ganze Programm einmalig. Dadurch sind Interpreter, besonders bei Befehlswiederholungen, sehr viel langsamer als Compiler.

Es folgte QuickBASIC 2, das neben dem Compiler eine Programmierumgebung besaß, die in puncto Übersichtlichkeit manches vereinfachte. QuickBASIC 3 mit seinen leicht verbesserten Struktur- und Fehlersuchemöglichkeiten war nur ein kurzes Gastspiel vergönnt, denn die Software-Ingenieure bei Microsoft setzten bereits auf ein neues Konzept, das zwar ähnlich aussah wie die Vorgänger, aber innerlich völlig davon abwich: QuickBASIC 4. Eine Ver"quick"ung von Interpreter und Compiler vereint hier die Vorteile beider Systeme; die Fehlersuche wurde wesentlich vereinfacht, und der Programmierer erhielt zahlreiche Mittel zur strukturierten Programmgestaltung. Der Nachfolger, QuickBASIC 4.5, hatte nochmals einige zusätzliche Befehle, vor allem aber ein stark überarbeitetes Hilfesystem zu bieten, so daß man nun für die meiste Programmierarbeit endlich das Handbuch im Schrank lassen konnte.

Parallel zu QuickBASIC 4.5 gab es dessen auf professionelle Programmierer zugeschnittenes Pendant BASIC 6.0, das als Entwicklungsumgebung noch QuickBASIC 4 enthielt und im Sprachumfang mit der Version 4.5 identisch war; lediglich einige Dinge wie OS/2- und wählbare Coprozessor-Unterstützung sowie andere Feinheiten machten den Unterschied aus.

Hier geht es jetzt um den jüngsten Sproß dieser alten Familie, das "Microsoft BASIC 7.1 Professional Development System", wie der vollständige Name lautet, kurz PDS. Das System erfüllt professionelle Ansprüche und ist schlicht der beste BASIC-Compiler, den ein PC-Benutzer kaufen kann. Seine ausgefeilte Benutzeroberfläche und der große Sprachumfang heben es auch in diesen Punkten über viele Compiler anderer Sprachen hinweg.

Dieses Buch ist für professionelle BASIC-Programmierer gedacht und solche, die es werden wollen. Wer schon mit QuickBASIC 4.0 oder 4.5 (oder Microsoft BASIC 6.0) gearbeitet hat, findet hier in den drei Kapiteln der Sektion III eine detaillierte Beschreibung aller neuen Möglichkeiten, die das PDS-System bietet, sowie eine Vielzahl von Tips und Ratschlägen, wie diese am besten auszunutzen sind.

Auch dem reinen QuickBASIC-Anwender wird dieses Buch, obwohl nicht auf ihn zugeschnitten, ein wertvoller Ratgeber zur Verbesserung seiner Programmier-technik sein.

Compiler und Entwicklungsumgebung werden ausführlich behandelt. Die Sektion IV widmet sich in neun Kapiteln ausschließlich Techniken, die helfen, professionellen Anforderungen gerecht zu werden: Sie enthält einerseits eine Anzahl von Standard-Algorithmen für häufig anzutreffende Aufgaben, andererseits praktische Tips zur Datenverwaltung, die speziell auf die Fähigkeiten von BASIC 7.1 PDS eingehen.

Wenn Sie also der Grundlagen des strukturierten Programmierens in Quick-BASIC mächtig sind (denn die werde ich nicht zum 1001. Mal erklären), geben

Ihnen der neue Compiler und dieses Buch alle Mittel an die Hand, um Software zu entwickeln, die der am Markt erhältlichen in nichts nachsteht.

Ein paar Worte noch zur Terminologie: Ich habe darauf verzichtet, die angestammten englischen Ausdrücke und Namen ins Deutsche zu übersetzen. Das Programm LINK.EXE heißt hier Linker und nicht Binder oder Verknüpfer, BC.EXE ist der Compiler und nicht der Übersetzer, ein Runtime-Modul ist ein Runtime-Modul und keine Laufzeitbibliotheksdatei usw.. Wenn Sie die Begriffe kennen, werden Sie das zu schätzen wissen, wenn nicht, brauchen Sie trotzdem kein Lexikon, denn zumindest bei der Einführung eines Begriffs wird seine Bedeutung geklärt. Wenn Sie die englischen Ausdrücke kennen, fällt es Ihnen auch wesentlich leichter, Fehlermeldungen des Programms und andere Dinge zu verstehen, nach deren Bedeutung Sie sonst mühsam forschen müßten. Eine deutsche Version des BASIC PDS ist nicht geplant, und ich halte es für keinen Frevel, wenn die Arbeitskraft bei Microsoft für sinnvollere Dinge eingesetzt wird.

Abschließend möchte ich noch Franz Fabian, Karsten Fourmont sowie Rolf und Dirk Thierbach meinen Dank für die Durchsicht einer Vorversion des Buches und die konstruktive Kritik daran aussprechen.

Ich wünsche Ihnen viel Erfolg bei der Arbeit mit BASIC PDS und diesem Buch.

*Frederik Ramm*

PS: Noch ein wichtiger Hinweis zum Abtippen der Listings! Wie Sie wahrscheinlich wissen, muß ein BASIC-Befehl oder ein Funktionsaufruf stets komplett in einer Zeile stehen. Dafür kann eine Zeile auch bis zu 255 Zeichen enthalten. Die Breite dieses Buches ist jedoch begrenzt. In den Programmbeispielen ergab sich die Notwendigkeit, eine logische Zeile (so, wie Sie sie eingeben müssen) auf zwei gedruckte Zeilen zu verteilen. An diesen Stellen finden Sie am Ende des ersten Teils der Zeile das Symbol  $\neg$ .

Wenn Sie also beim Abtippen eines Programmbeispiels auf das Zeichen  $\neg$  stoßen, wissen Sie: Der Inhalt der folgenden Zeile gehört noch zur aktuellen Zeile. Mir gefiel diese Lösung besser, als das Buch im Querformat zu drucken.





# Inhalt

## Sektion I: Gestatten, BASIC PDS...

<b>1</b>	<b>Was gibt es Neues?</b>	3
1.1	BASIC PDS 7.1 gegenüber BASIC PDS 7.0	3
1.2	BASIC PDS 7.0 gegenüber QuickBASIC 4 und BASIC 6.0	4
<b>2</b>	<b>Die Installation des BASIC PDS</b>	5
2.1	Voraussetzungen	5
2.2	Start der Installation	5
2.3	Die Setup-Menüs	6
	Directories	6
	Libraries und Runtime-Module	7
	ISAM-Unterstützung	9
	Zu installierende Dateien	10
2.4	Endlich: Die Installation beginnt	10
2.5	PWB-Zusätze für C-Programmierer	11
<b>3</b>	<b>Grundlagen</b>	13
3.1	Variablen und Datentypen	13
3.2	Operatoren	13
	Mathematische Operatoren	13
	Vergleichsoperatoren	14
	Logische und bitweise Operatoren	14
3.3	Arrays und Records	16
	Arrays	16
	Records	16
	Kombinationen	17
3.4	Prozeduren, Funktionen, Parameter, globale Variablen	18
3.5	Dateien	21
	Sequentielle Dateien	21
	Random Access-Dateien	22
	Binäre Dateien	22
3.6	Grafik und Text - Der Bildschirm	23
	Textmodus	23
	Grafikmodus	23

## Sektion II: Drei Möglichkeiten zur Programmerstellung

<b>4</b>	<b>Die Entwicklungsumgebung QBX</b>	27
4.1	Der Aufruf von QBX	27
	Beispiele zum QBX-Programmaufruf	29
4.2	Editor und Tastenbelegung	29
4.3	Dialogboxen	32
4.4	Programme, Module, Dateien - die Terminologie	34
4.5	Menüs	35
	Das Menü "File"	35
	Das Menü "Edit"	37
	Das Menü "View"	37
	Das Menü "Search"	38
	Das Menü "Utility"	38
	Das Menü "Options"	39
	Das Menü "Help"	39
4.6	Programme starten und "debuggen"	40
	Terminologie	40
	Das Menü "Run"	40
	Das Menü "Debug"	41
	Fehlersuche	42
	Eingrenzen des Fehlers	43
	Verfolgen von Variablenwerten	43
	Prozeduraufrufe	43
	Fehlerbehandlungsroutinen	44
	Verfolgen des Programmablaufs	44
4.7	Quick Libraries	45
4.8	EXE-Programme und Libraries	46
	Libraries	46
	EXE-Programme	47
<b>5</b>	<b>Die Entwicklungsumgebung PWB</b>	49
5.1	Der Aufruf der PWB	51
	TOOLS.INI	52
5.2	Der PWB-Editor	52
	Neue Tasten	53
	Marken	53

	Text markieren .....	53
	Pseudo-Dateien .....	54
5.3	Ein Projekt .....	54
	Programmliste .....	54
	Debug und Release .....	55
	Build .....	56
5.4	Der Source Browser .....	56
	Goto Definition .....	57
	Goto Reference .....	57
	View Relationship .....	57
	Call Tree .....	58
	Reference List .....	58
	Outline .....	58
5.5	Ausblick .....	59
<b>6</b>	<b>Separates Kompilieren mit BC und LINK .....</b>	<b>61</b>
6.1	Die verschiedenen Dateiarnten .....	61
6.2	Beispiele für das separate Kompilieren .....	62
6.3	Der Compiler BC.EXE .....	64
	Beispiele zu BC-Aufrufen .....	68
6.4	LINK bringt's zum Laufen .....	68
	LINK-Switches .....	71
	LINK-Steuerungsdatei .....	73

## Sektion III: Neue Sprachelemente - neue Möglichkeiten

<b>7</b>	<b>Das ISAM-Datenbanksystem .....</b>	<b>77</b>
7.1	Was ist ISAM? Wozu ISAM? .....	77
7.2	Das ISAM-Konzept .....	78
	ISAM-Funktionsweise .....	78
	ISAM-Begriffe .....	78
7.3	ISAM-Datenfiles auf der Platte .....	79
7.4	Die Schnittstelle zwischen ISAM und BASIC .....	79
	Die aktuelle Position .....	80
	Erstellen und Benutzen eines neuen Index .....	81

	Suchen nach einem bestimmten Datensatz .....	82
	Wie ISAM Strings sortiert .....	82
	Zugriff auf die Daten .....	83
	Transaktionen .....	83
	Weitere ISAM-Befehle und -Funktionen .....	84
	Programmbeispiel .....	85
7.5	ISAM und QBX .....	93
7.6	ISAM in kompilierten Programmen .....	95
7.7	ISAM - Programmierdetails und Vorsichtsmaßnahmen .....	95
	Anzahl gleichzeitig geöffneter	
	ISAM-Dateien und -Datenbanken .....	95
	Probleme mit den speicherresidenten ISAM-Routinen .....	96
	Datentyp SINGLE .....	96
	Interne Datenbanken .....	96
7.8	ISAM-Utilities .....	97
	ISAMCVT .....	97
	Btrieve-Konvertierung .....	98
	dBase-Konvertierung .....	98
	MS/ISAM-Konvertierung .....	99
	ISAMIO .....	99
	Import .....	99
	Export .....	101
	ISAMREPR .....	102
	ISAMPACK .....	102
<b>8</b>	<b>Add-On-Libraries .....</b>	<b>103</b>
8.1	Die Format-Add-On-Library .....	104
	Inhalt der Library .....	104
	Einbinden der Routinen .....	104
	Beispiele für numerische Formatierungen .....	105
	Beispiele für Zeitcode-Formatierungen .....	105
8.2	Die Date-Add-On-Library .....	106
	Inhalt der Library .....	106
	Was sind Zeitcodes? .....	106
	Einbinden der Routinen .....	107
8.3	Die Finance-Add-On-Library .....	108
	Inhalt der Library .....	108
	Einbinden der Routinen .....	108

<b>9</b>	<b>Toolboxen</b>	109
9.1	Die Matrizenmathematik-Toolbox	110
	Inhalt der Toolbox	110
	Einbinden der Routinen	110
	Allgemeine Hinweise	111
	Anwendungsbeispiele	111
9.2	Die Font-Toolbox	115
	Inhalt der Toolbox	115
	Einbinden der Routinen	115
	Verändern der Toolbox	116
	Allgemeine Hinweise	117
	Anwendung	117
	Die Schriftart-Dateien	118
	Die interne Schriftart	119
	Anwendungsbeispiel	119
	Fehlermeldungen der Font-Toolbox	121
	Schrifttabelle	121
	Programmfehler in der Font-Toolbox	123
9.3	Die Presentation Graphics-Toolbox	124
	Inhalt der Toolbox	124
	Einbinden der Routinen	125
	Verändern der Toolbox	126
	Allgemeine Hinweise	126
	Anwendungsbeispiele	127
	Die ChartEnvironment-Variable und ihre Subtypen	128
	RegionType für Bereichsbeschreibungen	129
	TitleType für Titel	129
	AxisType für Achsenbeschreibungen	130
	LegendType für die Legende	131
	ChartEnvironment - wo die Fäden zusammenlaufen	132
	Die Analyze-Routinen	135
	Die Farb- und Musterpalette	135
9.4	Die General-Toolbox	136
	Inhalt der Toolbox	136
	Einbinden der Routinen	136
9.5	Die User Interface-Toolbox	137
	Inhalt der Toolbox	137
	Einbinden der Routinen	137

Verändern der Toolbox .....	138
Allgemeine Hinweise .....	139
Mausroutinen .....	139
Menüroutinen .....	139
Fenster-Routinen .....	140

## Sektion IV: Aus der Praxis - Probleme und Lösungen

<b>10 Praktische Algorithmik .....</b>	<b>145</b>
10.1 Sortieren .....	145
Quicksort .....	145
Insertsort .....	147
Bucketsort .....	148
Sortieren durch Mischen ("Mergesort") .....	149
10.2 Suchen .....	153
Binäres Suchen .....	153
Binäres Suchen in ASCII-Dateien .....	155
10.3 Veränderungen an den Sortier- und Suchalgorithmen .....	156
Datentypen .....	156
Sortierfolge .....	157
10.4 Rekursive Programmierung .....	158
Backtracking .....	159
Rekursive und nicht-rekursive Algorithmen .....	163
<b>11 Dateiverwaltung .....</b>	<b>165</b>
11.1 Parameterdatei .....	165
Wo sollte die Parameterdatei abgelegt werden? .....	166
Datei als Teil des Programms .....	167
Speicherungsart für Parameter .....	167
11.2 Druckertreiber .....	169
Eine oder mehrere Dateien? .....	169
Verzeichnis vorhandener Drucker .....	169
11.3 Formularbeschreibungen .....	170
11.4 Formularbeschriftungen .....	171
Wie speichert man eine Formularbeschreibung? .....	171
11.5 Text-Datenbank .....	173

<b>12</b>	<b>Variablenverwaltung im Speicher .....</b>	<b>175</b>
12.1	Boolesche Variablen .....	175
12.2	Strings mit fester und variabler Länge .....	177
12.3	Far Strings .....	178
	Wozu Far Strings? .....	178
	Near und Far Strings im Vergleich .....	179
	Far String-Speicher für Großverbraucher .....	179
	Vor- und Nachteile beider String-Varianten .....	181
	Direktzugriff auf Far Strings .....	181
	Kompatibilität zu alten Toolboxes und Quick Libraries ..	181
12.4	Statische und dynamische Felder .....	182
	Arrays aus Strings mit variabler Länge .....	183
	Numerische Arrays .....	183
	Huge Arrays .....	183
12.5	Statische und automatische Variablen .....	184
	Selbständige Subroutinen .....	185
<b>13</b>	<b>Umfangreiche Programmsysteme .....</b>	<b>187</b>
13.1	Programmsysteme mit CHAIN .....	187
	Runtime-Module .....	188
13.2	Overlays - der Schlüssel zum Megabyte-Programm .....	189
	Overlays und EMS .....	189
	Programmieren mit Overlays .....	190
13.3	Die verschiedenen Konzepte im Vergleich .....	191
<b>14</b>	<b>Die Mathematik-Bibliotheken und der Coprozessor .....</b>	<b>195</b>
14.1	Vor- und Nachteile der Emulator-Library .....	195
14.2	Vor- und Nachteile der Alternate Math-Library .....	196
14.3	Fazit .....	196
<b>15</b>	<b>Fehlerbehandlung .....</b>	<b>197</b>
15.1	Die einfachste Lösung .....	197
15.2	Fortgeschrittene Methoden .....	198
	Lokale Fehlerbehandlung .....	200
	Fehlerbehandlung bei mehreren Modulen .....	203
	Resumée .....	203

<b>16</b>	<b>DOS-Interrupts und ihre Nutzung</b>	205
16.1	Wozu Interrupts?	205
16.2	Datenübergabe bei Interrupt-Aufrufen	206
16.3	Wie man einen Interrupt aufruft	207
	Zum Beispiel...	207
	Schwierigkeiten mit der Integer-Rechnung	208
16.4	Weitere Beispiele zur Interrupt-Nutzung	209
	Inhaltsverzeichnis	209
	Konfiguration des Systems	215
	Freier Platz auf einem Datenträger	216
<b>17</b>	<b>Extended &amp; Expanded Memory</b>	217
17.1	EMS und XMS innerhalb QBX	218
17.2	EMS und XMS bei kompilierten Programmen (EXE-Files)	219
17.3	Zusammenfassung	219
<b>18</b>	<b>OS/2-Programmierung</b>	221
18.1	Einschränkungen	222
18.2	OS/2-Betriebssystemfunktionen	223
18.3	Runtime-Module	223

## Sektion V: Der letzte Schliff

<b>19</b>	<b>Optimierung</b>	227
19.1	Die Ausführungsgeschwindigkeit eines Programms	227
	80286-spezifischer Code	227
	Alternate Math Library	227
	Verschiedene Datentypen	227
	GOTO und GOSUB - alt, aber rüstig	227
	Datenübergabe bei Prozeduraufrufen	228
	Statische Subroutinen	228
	Zeilennummern- und -labels	228
	Arrays statt Funktionen	228
	Leerstrings	228
	Auswahl der Datentypen	229



	Schleifen .....	229
	IF...THEN-Abfragen .....	229
	String-Verknüpfungen .....	230
	Interrupts .....	230
	Funktions- und Prozeduraufrufe .....	230
19.2	Programmgröße und RAM-Speicherplatz .....	231
	Konstanten .....	231
	Statische Subroutinen .....	231
	Dateibuffer bei sequentiellen Dateien .....	231
	Variablen nach dem SCREEN-Befehl .....	231
	Fließkomma-Arithmetik .....	232
	Event-Trapping .....	232
	Error-Trapping .....	232
	Verzicht auf Coprozessor-Emulation .....	233
	Statische und dynamische Felder .....	233
	Compiler- und Linker-Schalter .....	233
19.3	Verzicht-Files .....	234

## Sektion VI: Zusatzprogramme zum Compiler

<b>20</b>	<b>BUILDRTM und Runtime-Module .....</b>	<b>239</b>
20.1	Wozu Runtime-Module? .....	239
20.2	Standard-Runtime-Module .....	241
20.3	Erstellen eines Runtime-Moduls mit BUILDRTM .....	242
20.4	Runtime-Module und Verzicht-Files .....	244
20.5	Runtime-Module und ISAM .....	245
20.6	Add-On-Libraries und Toolboxen in Runtime-Modulen ..	245
20.7	Inkompatible Runtime-Module .....	246
<b>21</b>	<b>LIB und Libraries .....</b>	<b>247</b>
21.1	LIB-Aufruf .....	247
	Beispiele für LIB-Aufrufe .....	249
	Steuerungsdateien .....	249

21.2	Linken mit Libraries .....	250
21.3	Granularität und Inhalt einer Library .....	251
<b>22</b>	<b>Automatisierte Programmerstellung mit NMAKE .....</b>	<b>253</b>
22.1	Erste NMAKE-Anwendungen .....	253
22.2	Der Aufruf von NMAKE .....	255
22.3	Die NMAKE-Funktionsweise .....	256
22.4	NMAKE-Programmierdetails .....	258
	Ableitungsregeln .....	258
	Sonderzeichen .....	259
	Wildcards .....	259
	Makros .....	260
	Die wichtigsten Spezialmakros .....	260
	Makrodefinitionen und Ableitungsregeln in TOOLS.INI ..	261
	Unechte Abhängigkeiten .....	261
	Die NMAKE-Ablaufsteuerung .....	262
	Steuerungsdateien .....	263
22.5	Der NMAKE-Zwilling NMK.COM .....	264
<b>23</b>	<b>MKKEY - Veränderung der Tastaturdefinition .....</b>	<b>267</b>

## Sektion VII: Referenzteil

Standard-BASIC-Befehle und Funktionen .....	273
ISAM .....	421
Date-Library .....	435
Format-Library .....	440
Finance-Library .....	443
Matrizenmathematik-Toolbox .....	453
Font-Toolbox .....	458
Presentation Graphics-Toolbox .....	470
User Interface-Toolbox .....	482
Menü-Routinen .....	482
Window-Routinen .....	492
Maus-Routinen .....	509
General-Routinen .....	512

## Sektion VIII: Anhänge

<b>A</b>	<b>Alle Limits auf einen Blick .....</b>	<b>521</b>
<b>B</b>	<b>Alle Switches auf einen Blick .....</b>	<b>524</b>
	QBX .....	524
	BC .....	525
	PWB .....	526
	LINK .....	526
	LIB .....	527
	BUILDRTM .....	528
	NMAKE / NMK .....	529
<b>C</b>	<b>Alle Fehlermeldungen auf einen Blick .....</b>	<b>530</b>
C.1	BASIC-Fehlermeldungen (nach Nummern) .....	530
C.2	BASIC-Fehlermeldungen (alphabetisch) .....	531
	Unerklärliche Systemfehler .....	557
	Unerklärliche Strukturfehler .....	557
C.3	LINK-Fehler (Auswahl) .....	558
C.4	LIB-Fehler (Auswahl) .....	561
C.5	Fehlercodes der Toolboxen .....	562
	Presentation Graphics .....	562
	Fonts .....	563
<b>D</b>	<b>Tastatur- und Zeichencodes .....</b>	<b>564</b>
D.1	Scancodes .....	564
D.2	ASCII-Codes der verschiedenen Tastenkombinationen ...	566
	Äquivalente Tastenbezeichnungen .....	568
D.3	Standard-ASCII-Codes .....	569
D.4	ISAM-Sortiertabellen .....	573

<b>E</b>	<b>Ausgewählte Interrupts .....</b>	<b>576</b>
<b>F</b>	<b>Mausfunktionen .....</b>	<b>585</b>
<b>G</b>	<b>Tabellen dieses Buches .....</b>	<b>590</b>
<b>H</b>	<b>Liste der Programmbeispiele .....</b>	<b>592</b>
<b>I</b>	<b>Befehlsreferenz – funktionsorientiert gegliedert .....</b>	<b>593</b>
	<b>Stichwortverzeichnis .....</b>	<b>609</b>

---

<b>Sektion I</b>	<b>Gestatten, BASIC PDS...</b>
----------------------	------------------------------------

---

- **Was gibt es Neues?**
  - **Die Installation des  
BASIC PDS**
  - **Grundlagen**
-



# 1 Was gibt es Neues?

Wer bisher schon mit QuickBASIC oder einem anderen BASIC-Compiler gearbeitet hat, der möchte natürlich wissen, was sich alles verändert hat. Ich handele zunächst die wenigen Veränderungen ab, die es von BASIC PDS 7.0 zu BASIC PDS 7.1 gab, und beschäftige mich danach mit all dem, was von den QuickBASIC-Versionen 4.0 und 4.5 (beziehungsweise dem damit fast gleichwertigen Compiler BASIC 6.0) auf BASIC PDS 7.0 neu hinzugekommen ist.

Im Standard-Referenzteil ist bei jedem Befehl vermerkt, seit welcher QuickBASIC-Version er in der heute vorliegenden Form verfügbar ist.

## 1.1 BASIC PDS 7.1 gegenüber BASIC PDS 7.0

In der Sprache hat sich zwischen diesen beiden Versionen kaum etwas geändert. Ein paar kleine Fehler des PDS 7.0 wurden behoben (siehe beispielsweise RUN); Werteparameter können jetzt komfortabler als vorher übergeben werden (siehe SUB/FUNCTION und CALL), und es besteht die Möglichkeit, auch Arrays aus Strings von fester Länge als Parameter zu übergeben. Der REDIM-Befehl hat jetzt die Fähigkeit, Daten beizubehalten, die bereits in einem Array stehen.

Die Benutzeroberfläche QBX hat sich nur in kleinsten Details geändert. Der Editor "M" wird nicht mehr mitgeliefert, stattdessen aber die neueste Version 3.0 von CodeView und die Microsoft Programmer's WorkBench (PWB), eine multilinguale Entwicklungsumgebung (siehe dazu Kapitel 5).

Nähert man sich dem System 7.1 von der technischen Seite, stellt man fest, daß an den Systemfunktionen noch gefeilt wurde, so daß viele Libraries kleiner geworden sind; durch zusätzliche Optimierungen werden die EXE-Programme gerade bei Schleifen und anderen Kontrollstrukturen noch kleiner und schneller (siehe dazu aber die Bemerkung "Compiler-Optimierung" weiter unten). Microsoft BASIC PDS 7.1 ist voll C 6.0-kompatibel. Das bedeutet, daß der produzierte Code sich mit den Erzeugnissen von Microsoft C 6.0 bestens versteht. Sie können so Subroutinen für BASIC in C und umgekehrt schreiben. "Problemlos" ist natürlich relativ zu verstehen; wenn Sie zum Beispiel mit Strings arbeiten, ist einiger Aufwand erforderlich, um diese in C zu manipulieren.

## 1.2 BASIC PDS 7.0 gegenüber QuickBASIC 4 und BASIC 6.0

Es folgt eine ausführliche Darstellung der neuen Features, an die sich ein ehemaliger QuickBASIC-Programmierer mit Hilfe dieses Buches herantasten kann:

### **Far Strings**

Stark erweiterter Stringspeicherplatz. Die 64K-Grenze ist gefallen. Siehe Kapitel 12.3.

### **Overlays**

EXE-Programme bis zu 16 MB. Programmteile werden je nach Bedarf automatisch nachgeladen. Siehe Kapitel 13.2.

### **ISAM**

Eingebaute Datenbanksprache zur Verwaltung großer Datenbestände. Siehe Kapitel 7.

### **Datentyp Currency**

Neuer Datentyp aus 8 Bytes mit 3 Nachkommastellen. Keine Rundungsfehler mehr. Siehe Anhang A.

### **Add-On-Libraries und Toolboxes**

Fertige Routinen für die Bereiche Präsentationsgrafik, Matrizenmathematik, Benutzeroberfläche, Datum/Uhrzeit und Finanzen.

### **Spracherweiterungen**

Wichtigste neue Befehle sind DIR\$ und ON LOCAL ERROR (siehe Referenzteil). Selbstdefinierte Datentypen dürfen jetzt auch Felder enthalten.

### **Compiler-Optimierung**

Die Compiler-Routinen sind in vielen Punkten verfeinert worden. Die Coprozessor emulation und die Alternate-Math-Library wurden verbessert; der Compiler produziert kleinere und schnellere EXE-Files (als vorher, nicht etwa als andere Compiler).

### **Entwicklungsumgebung**

QBX - Quick BASIC Extended - hat ein ausgezeichnetes Hilfesystem und unterstützt EMS (siehe Kapitel 17.1). Es gibt eine Undo-Funktion, die bis zu 20 zuletzt getätigte Änderungen rückgängig machen kann. Die Tastenbelegung des Editors kann programmiert werden (siehe Kapitel 23).



# 2 Die Installation des BASIC PDS

Entgegen allen bisherigen BASIC-Compilern wird das BASIC PDS nicht einfach auf die Festplatte kopiert, sondern besitzt ein umfangreiches SETUP-Programm. Dieses Programm kopiert und entkomprimiert nicht nur alle benötigten Dateien, sondern erstellt auch individuelle Libraries und Runtime-Module nach den Vorgaben des Benutzers. Folgende Punkte sollten bereits vor der Installation von Ihnen geklärt worden sein:

- In welchem Umfang wollen Sie ISAM einsetzen?
- Welche Grafikadapter sollen von Ihren Programmen unterstützt werden?
- Auf welchen Zielsystemen sollen Ihre Programme eingesetzt werden (Coprozessor)?
- Wollen Sie Programme für OS/2 erstellen?
- Können Sie schon jetzt festlegen, ob Sie mit Runtime-Modulen arbeiten wollen?

Diese und weitere Details, die die Installation beeinflussen, werde ich nun im einzelnen behandeln.

## 2.1 Voraussetzungen

Das volle System benötigt etwa 15 MB an Plattenkapazität. Wenn Sie auf OS/2 verzichten, reduziert sich der Platzbedarf auf etwa 10 MB. Der Verzicht auf die PWB spart ein weiteres Megabyte ein. Eine arbeitsfähige Konfiguration (mit QBX, Hilfesystem und ISAM) kann man schon ab 5 MB erreichen.

Sie brauchen mindestens das Betriebssystem DOS 3.0 oder OS/2 1.1, um das BASIC PDS benutzen zu können; kompilierte Programme funktionieren jedoch auch schon ab DOS 2.1.

Ihr Rechner sollte mit 640 KB RAM ausgerüstet sein; EMS-Speicher ist für das separate Kompilieren unnötig, für QBX sinnvoll und für PWB fast unerlässlich.

## 2.2 Start der Installation

Es gibt zwei Möglichkeiten zur Installation. Bei der Standard-Installation erledigt das SETUP-Programm sowohl das Kopieren und Dekomprimieren als auch die

Erstellung der Libraries und Runtime-Module durch die Programme LINK, LIB und BUILDRTM.

Bei der Batch-Installation wird SETUP mit dem Switch /batch aufgerufen. Es kopiert und dekomprimiert dann nur, ohne die LINK-, LIB- und BUILDRTM-Aufrufe selbst auszuführen. Diese Aufrufe werden vielmehr in eine Batch-Prozedur namens BUILD.BAT auf der Festplatte geschrieben, die Sie nach Abschluß des Installationsvorgangs "von Hand" aufrufen müssen. Der Vorteil dieses Batch-Modus liegt darin, daß LINK, LIB und BUILDRTM mehr Speicher zur Verfügung haben und das ganze Verfahren dadurch nicht gar so lange dauert.

Apropos Dauer: Auf langsamen Rechnern müssen Sie mit mindestens einer Stunde Installationszeit rechnen; nur bei sehr schnellen Systemen (abhängig natürlich von Prozessor- und Festplattenleistung) kommen Sie unter einer halben Stunde weg.

Wenn Sie einfach nur so schnell wie möglich eine Arbeitsgrundlage erhalten wollen, übergehen Sie einfach alle Menüs (SETUP hat sinnvolle Voreinstellungen), und wählen Sie sofort "I". Lesen Sie dann weiter bei 2.4.

## 2.3 Die Setup-Menüs

### Directories

Das Menü "Specify Paths and Directories" ermöglicht die Einstellung der Pfade für die verschiedenen Gruppen von Dateien, die installiert werden. Wenn Sie hier Directories angeben, die nicht existieren, werden diese neu angelegt.

Die Angaben, die Sie in diesem Menü machen, sind nur für das Kopieren und anschließende Erstellen der Libraries und Runtime-Module und für die QBX-Einstellung relevant. Die Pfadeinstellung von QBX können Sie allerdings nachträglich jederzeit ändern.

Geben Sie am besten für "bound executables" und "real mode executables" das gleiche Directory an, wenn Sie unter DOS arbeiten, und nehmen Sie für "bound" und "protected mode executables" das gleiche Directory, wenn Sie OS/2 betreiben.

Vier der Directories, die Sie hier angeben, sollten Sie auch in Ihre AUTOEXEC.BAT-Prozedur aufnehmen: Der Pfad für "bound executables"\*, also zum Beispiel LINK, BUILDRTM, LIB, BC usw., gehört in den PATH (etwa

---

\* "bound executables" sind Programme, die sowohl unter DOS als auch unter OS/2 eingesetzt werden können.

PATH C:\BC7\BIN), damit Sie die Programme von überall her aufrufen können. Der Pfad für "BASIC Source and include files" sollte mit SET INCLUDE=pfadname "publik" gemacht werden, damit QBX und BC wissen, wo sie Include-Files zu suchen haben. Der Pfad für Libraries kann dem LINK-Programm mit SET LIB=pfadname mitgeteilt werden, so daß es im angegebenen Pfad seine Libraries und OBJ-Dateien sucht, falls sie im aktuellen Directory nicht gefunden werden. Mit SET HELPPFILES=pfadname können Sie außerdem noch den Pfad für die Hilfe-Dateien bekanntgeben, so daß Programme wie LINK oder QH ihre Hilfstexte finden und anzeigen können.

SETUP erzeugt beim Installieren eine Prozedur namens NEW-VARS.BAT (oder NEW-VARS.CMD für OS/2), die all diese SET-Befehle mit den richtigen Argumenten enthält. Übernehmen Sie am besten die Angaben aus NEW-VARS.BAT in die Prozedur AUTOEXEC.BAT (beziehungsweise aus NEW-VARS.CMD in die Prozedur CONFIG.SYS bei OS/2).

## Libraries und Runtime-Module

Dieses Menü besteht aus drei Auswahlbildschirmen. Der erste davon bezieht sich gleichermaßen auf Runtime-Module und Libraries, die Bildschirme 2 und 3 wirken nur auf Runtime-Module.

### Der erste Bildschirm

Im ersten Bildschirm können Sie in vier Bereichen je eine oder zwei Optionen anwählen.

Wählen Sie "Stand-alone EXE", wenn Sie später (auch) Programme erzeugen möchten, die ohne jedes Runtime-Modul funktionieren. Wählen Sie "EXE requiring BRT module", wenn Sie (auch) Programme erstellen möchten, die mit Runtime-Modulen arbeiten.

Wählen Sie "80X87 or Emulator Math", wenn Sie Programme erstellen möchten, die die Emulator-Library benutzen (d.h. den Coprozessor unterstützen können, siehe Kapitel 14), und "Alternate Math", wenn Sie Programme mit der Alternate Math-Library erstellen wollen.

Außerdem können Sie hier noch wählen, ob Sie Programme mit Near und/oder Far Strings und Programme für DOS und/oder OS/2 erstellen möchten.

Jedes Kreuz, das Sie im ersten Bildschirm setzen, steht für eine *Möglichkeit*. Wenn Sie in einem der Felder beide Möglichkeiten ankreuzen, wählen Sie erst später beim Kompilieren, welcher Art das zu erstellende Programm sein wird (zum Beispiel, ob es mit Far Strings arbeiten soll oder nicht). Kreuzen Sie in einem Feld nur eine Möglichkeit an, dann können Sie später beim Kompilieren nicht mehr wählen, sondern müssen immer die eine Auswahl benutzen, die Sie

bei der Installation getroffen haben. Wenn Sie also bei der Installation "Near Strings" und "Far Strings" anwählen, können Sie später durch Verwendung des Compiler-Switches /Fs entscheiden, ob Far Strings benutzt werden sollen oder nicht. Wählen Sie aber bei der Installation nur "Near Strings" oder nur "Far Strings", bleibt Ihnen später nicht mehr die Wahl.

Mit der Auswahl in diesem Bildschirm bestimmen Sie, welche Libraries und Runtime-Module erstellt werden. Wenn Sie alle acht Kreuze setzen, entstehen:

- acht BCL71xyz.LIB-Libraries (je bis zu 250 KB lang),
- acht BRT71xyz.LIB-Libraries (je etwa 50 KB lang) und
- acht BRT71xyz.EXE beziehungsweise BRT71xyz.DLL-Runtime-Module (je bis zu 170 KB lang).

Die BCL-Libraries entstehen, wenn Sie "Stand-alone EXE" ankreuzen. Die BRT-Libraries und Runtime-Module entstehen, wenn Sie "EXE requiring BRT module" wählen.

Die drei Platzhalter xyz stehen für die anderen Optionen:

- x ist A, wenn Sie "Alternate Math", E, wenn Sie "80X87 or Emulator Math" wählen.
- y ist N, wenn Sie "Near Strings", F, wenn Sie "Far Strings" wählen,
- z ist R für "Real Mode" oder P für "Protected Mode".

## **Der zweite und dritte Bildschirm**

Im zweiten Bildschirm, der sich ausschließlich auf Runtime-Module bezieht und für Stand-alone-Programme keine Bedeutung hat, legen Sie fest, welche Grafikmöglichkeiten die Runtime-Module zur Verfügung stellen sollen. Je mehr Optionen Sie auswählen, desto länger werden die Runtime-Module.

Für den dritten Bildschirm gilt dasselbe. Allerdings beziehen sich Ihre Eingaben hier nicht auf Grafikmodi, sondern auf andere allgemeine Funktionen des Runtime-Moduls.

Den Optionen in beiden Bildschirmen entsprechen bestimmte Verzicht-Files (siehe Kapitel 19.3), die in die Runtime-Module eingebunden werden oder nicht, je nachdem, wie Sie die Kreuze setzen. Im zweiten Bildschirm handelt es sich um die Verzicht-Files TSCNIOyz.OBJ, NOGRAPH.OBJ, NOCGA.OBJ, NOHERC.OBJ, NOOGA.OBJ, NOEGA.OBJ und NOVGA.OBJ. Im dritten Bildschirm gibt es folgende Entsprechungen:

<b>Option</b>	<b>Bedeutung</b>
COMn Device OPEN and I/O	Kein Kreuz: NOCOM.OBJ wird in das Runtime-Modul eingebunden.
LPTn Device OPEN and I/O	Kein Kreuz: NOLPT.OBJ wird eingebunden.
Overlays in DOS 2.1	Ein Kreuz: OVLDOS21.OBJ wird eingebunden.
EMS Support for Overlays	Kein Kreuz: NOEMS.OBJ wird eingebunden.
Transcendental Math	Kein Kreuz: NOTRNEMz.LIB wird eingebunden.
INPUT Floating-Point Values	Kein Kreuz: NOFLTIN.OBJ wird eingebunden.
Math Coprocessor Required	Ein Kreuz: 87.LIB wird eingebunden.
EVENT Trapping	Kein Kreuz: NOEVENT.OBJ wird eingebunden.
Detailed Error Messages	Kein Kreuz: SMALLERR.OBJ wird eingebunden.
Full-Power INPUT Editor	Kein Kreuz: NOEDIT.OBJ wird eingebunden.

## ISAM-Unterstützung

Das Menü "Specify ISAM Support" besteht aus zwei Bildschirmen. Im ersten wählen Sie, wie den fertigen Programmen die ISAM-Routinen verfügbar gemacht werden. Die Entscheidung muß schon bei der Installation getroffen werden, weil sie Einfluß auf die Erstellung der Standard-Libraries und Runtime-Module hat. Die folgende kleine Übersicht zeigt, welche Wirkung die vier möglichen Auswahlen haben.

### ISAM Routines in TSR

Egal, ob Ihr Programm mit Runtime-Modul oder ohne arbeitet, muß jedesmal vor dem Programmaufruf eines der beiden Programme PROISAM.EXE oder PROISAMD.EXE geladen werden. Ihre Programme und Runtime-Module bleiben dadurch recht klein, jedoch entsteht der Zwang zum Laden der speicherresidenten Programme.

### ISAM Routines in LIB, Support Database Creation and Access

Programme, die mit Runtime-Modulen arbeiten, benutzen die ISAM-Routinen aus den Runtime-Modulen; in Stand-Alone-Programme werden die ISAM-Routinen direkt eingebunden. Runtime-Module und Stand-Alone-Programme werden dadurch länger, aber es müssen keine speicherresidenten Programme geladen werden.

### ISAM Routines in LIB, Support Database Access Only

Wie oben, allerdings wird hier nur ein Teil der ISAM-Routinen eingebunden (entspricht PROISAM.EXE), so daß es nicht möglich ist, neue Indizes oder Datenbanken zu erstellen oder alte zu löschen.

### No ISAM support

ISAM wird überhaupt nicht unterstützt.

Im zweiten Bildschirm zum Thema ISAM geben Sie an, welche nationale Sortiertabelle Sie wünschen. Die Unterschiede entnehmen Sie bitte dem Anhang C.4.

## Zu installierende Dateien

Schließlich können Sie auswählen, welche Programme und Dateigruppen wirklich installiert werden sollen. SETUP zeigt dabei an, wieviel Platz auf der Festplatte benötigt wird und wieviel vorhanden ist. Falls ein Teil von BASIC PDS schon installiert ist (und Sie SETUP benutzen, um einen Teil zusätzlich zu installieren, den Sie bei der ersten Installation vergaßen), wird das in die Berechnung mit einbezogen.

Die einzelnen Menüpunkte bedeuten:

### **BASIC Tools and Utilities**

Der Compiler BC.EXE, LINK.EXE, LIB.EXE und andere unverzichtbare Programme.

### **BASIC Libraries, Run-Time Modules, and Add-On LIBs**

Die benötigten Libraries und Runtime-Module.

### **BASIC Sample Programs, Toolboxes, and Source Code**

Beispielprogramme und Toolboxen sowie die Quelltexte der Toolboxen.

### **QuickBASIC Extended Environment and Quick Libraries**

QBX und die dazugehörigen Quick Libraries (\*.QLB) der Add-On-Libraries und Toolboxen.

### **Mixed-Language Tools for DOS or OS/2 Real Mode**

Die Programmer's Workbench und CodeView für DOS.

### **Mixed-Language Tools for OS/2 Protected Mode**

CodeView und einige Utility-Programme für OS/2.

### **Help Files for Chosen Tools**

Hilfesystem für QBX, PWB und Utilities.

## 2.4 Endlich: Die Installation beginnt

Wenn Sie alle notwendigen Angaben gemacht haben, können Sie mit der eigentlichen Installation anfangen. Wählen Sie "Install with current options". Es erscheint dann noch die Frage, ob die Komponentendateien, also die Dateien, aus denen die Libraries usw. zusammengesetzt sind, nach dem Zusammenstellen gelöscht werden sollen. Sie können die Dateien löschen lassen, wenn Sie es

vorerst nicht beabsichtigen, eigene Runtime-Module zu erstellen, denn das ist die einzige Operation, zu der sie benötigt werden. Wenn sie dann einmal gebraucht werden, kann man sie immer noch kopieren.

Nun beginnt die Installation, und Sie werden sich von Zeit zu Zeit, den Aufforderungen von SETUP Folge leistend, als Discjockey betätigen müssen. Falls Dateien, die SETUP installieren will, schon vorhanden sind, müssen Sie bestimmen, ob die Dateien überschrieben werden sollen oder nicht.

Wenn Sie beim Aufruf von SETUP /batch als Switch angegeben haben, müssen Sie zum Schluß noch die Prozedur BUILD.BAT aufrufen. Sie wird von SETUP in einem der Directories angelegt, das Sie angegeben haben. Haben Sie hingegen den Batch-Modus nicht benutzt, dann erzeugt SETUP die Libraries und Runtime-Module sofort. Sofern dabei ein Fehler auftritt, fragt SETUP nach, ob es fortfahren soll.

## **2.5 PWB-Zusätze für C-Programmierer**

Zwei Zusatzdateien für die Programmer's WorkBench, nämlich PWBC.PXT und PWBC.MXT, sind zwar auf den Disketten vorhanden, werden aber nicht automatisch installiert. Benutzen Sie die Befehle

```
UNPACK A:PWBC.PXT C:\BC7\BIN\PWBC.PXT  
UNPACK A:PWBC.MXT C:\BC7\BIN\PWBC.MXT
```

um die beiden Dateien ebenfalls zu installieren. Setzen Sie gegebenenfalls statt C:\BC7\BIN einen anderen Directory-Namen ein.





# 3 Grundlagen

Die folgenden Abschnitte stellen kurz die elementaren Grundlagen für das Programmieren in BASIC dar. Auch als professioneller Anwender sollten Sie sie kurz überfliegen.

## 3.1 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-Befehl oder einen DEFxxx-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 diesem 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*. Siehe dazu den Abschnitt 3.3 in diesem Kapitel.

## 3.2 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 bis 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 beziehungsweise 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 folgende Wahrheitstabellen.

<b>x AND y</b>	<b>x = T</b>	<b>x = F</b>	<b>x OR y</b>	<b>x = T</b>	<b>x = F</b>
y = T	T	F	y = T	T	T
y = F	F	F	y = F	T	F
<b>x EQV y</b>	<b>x = T</b>	<b>x = F</b>	<b>x XOR y</b>	<b>x = T</b>	<b>x = F</b>
y = T	T	F	y = T	F	T
y = F	F	T	y = F	T	F
<b>x IMP y</b>	<b>x = T</b>	<b>x = F</b>	<b>NOT x</b>	<b>x = T</b>	<b>x = F</b>
y = T	T	T		F	T
y = 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:

```
x=5 ist FALSE; x>10 ist TRUE; y<7 ist FALSE
FALSE OR TRUE ist TRUE; NOT FALSE ist TRUE
TRUE AND TRUE ist TRUE
```

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) macht eine Ausnahme, es hat den Wert -32.768 statt 32.768.

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.

## 3.3 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. Diese 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 über 1,8 Millionen) Elemente einen eigenen Namen haben kann, bekommt es nur einen 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 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 Geburtstagskinds 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! Zum Thema "Schachtelung von Records" können Sie auch einmal einen Blick auf die Beschreibung des selbst-definierten Typs ChartEnvironment im Kapitel 9.3 über die Presentation Graphics-Toolbox werfen.

## 3.4 Prozeduren, Funktionen, Parameter, globale Variablen

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

Ein QuickBASIC-Programmierer, der keine Prozeduren und Funktionen benutzt, ist den Compiler nicht wert. (Das klingt hart, aber QuickBASIC zu benutzen, ohne Prozeduren und Funktionen zu verwenden, ist wie einen Lamborghini zu kaufen und damit nur 30 zu fahren.)

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 31
...
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 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$
    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.

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%.

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

- 1 Man übergibt sie als Parameter (wie Dateiname\$ im ersten Beispiel).
- 2 Man erwähnt sie in einem SHARED-Befehl in der betreffenden Prozedur. Dann ist sie nur dieser Prozedur zugänglich.
- 3 Man vereinbart sie im Hauptprogramm mit DIM SHARED. Dann ist sie allen Prozeduren und Funktionen dieses Programms zugänglich.
- 4 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.

Mehr über verschiedene Module ist in den Kapiteln 4.4 (QBX) und 6 (separates Kompilieren) zu finden.



## 3.5 Dateien

Ein paar Worte zum Thema "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 erst seit PDS 7.0 verfügbaren ISAM-Dateien, die in vielem eine Ausnahme bilden und in diesem Buch noch ausführlich genug 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. Dieser 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 einem ASCII-Zeichen 10 steht (der Code 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, wie sie hier zum Beispiel in Kapitel 10.2 (Binäres Suchen für sequentielle Dateien) oder im Referenzteil unter SEEK dargestellt werden, aber dennoch gilt: Sequentiell ist eben nicht "random access", nicht Direktzugriff.

# Random Access-Dateien

RANDOM-Dateien sind Dateien, die Datenblocks (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 Prinzip 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 oder einen bestimmten Text irgendwo einzutragen.

## 3.6 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 Farbkombinationen 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, daß 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, 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 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 ausgestüftelte Programme beweisen. Obwohl es schwieriger ist, mit den Blockgrafikzeichen Grafik 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 in Anhang D.3.

### 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 Pixel-spalten 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).

Im Grafikmodus lassen sich also nicht so schnell wie im Textmodus Bildbereiche abspeichern, verschieben oder verändern. Dafür besteht allerdings eine viel umfassendere Kontrolle über den Bildschirminhalt.

Spezielle Grafikbefehle erleichtern die Erzeugung von Grafiken sowie das Kopieren und Verschieben von Ausschnitten. Mit der Font-Toolbox 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 - aber das ist mit großem Aufwand und viel Testarbeit verbunden.

---

## **Sektion II      Drei Möglichkeiten zur Programmerstellung**

---

- **Die Entwicklungs-  
umgebung QBX**
  - **Die Entwicklungs-  
umgebung PWB**
  - **Separates Kompilieren  
mit BC und LINK**
-



# 4 Die Entwicklungsumgebung QBX

Der Editor der Entwicklungsumgebung QBX ist optimal auf die Bedürfnisse des BASIC-Programmierers abgestimmt. Zwar liefert Microsoft im PDS 7.1 die Workbench PWB mit, aber trotzdem wird QBX in den meisten Fällen am nützlichsten sein.

Sowohl PWB als auch dem Editor "M", der mit dem PDS 7.0 mitgeliefert wurde, ist QBX überlegen, weil es eingegebene Programmzeilen sofort vorkompiliert und deshalb fehlende Klammern etc. umgehend bemerkt, hie und da ein fehlendes Semikolon einfügt, alle BASIC-Befehle groß schreibt und auch für eine einheitliche Schreibung der Variablen sorgt. Außerdem sind die Debug-Möglichkeiten von QBX in den meisten Fällen hilfreicher als die der PWB.

## 4.1 Aufruf von QBX

Die QBX-Befehlszeile sieht folgendermaßen aus:

```
QBX [switches] [/RUN] programname [/CMD command$]
```

*programname* ist der Name des Programms, das Sie bearbeiten wollen. Das Programm mit dem angegebenen Namen wird geladen, als ob Sie es mit dem "Open"-Befehl aus dem "File"-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 12.4). Alle EXE-Programme, die aus QBX heraus erstellt werden, werden ebenfalls mit /AH kompiliert, wenn dieser Switch angegeben wird.
/B	Die QBX-Bildschirmanzeige erfolgt nur schwarz/weiß.

(Fortsetzung nächste Seite)

Switch	Wirkung
/C: <i>b</i>	Setzt den Standard-Kommunikations-Buffer auf <i>b</i> Bytes (für das Empfangen von Daten über OPEN COM).
/CMD: <i>command\$</i>	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 "Modify COMMAND\$" im "Run"-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.
/E: <i>ems</i>	Limitiert die für QBX verfügbare EMS-Größe auf <i>ems</i> KB. Wenn dieser Switch nicht angegeben wird, kann QBX sämtlichen EMS-Speicher belegen.
/Es	Teilt Expanded Memory zwischen QBX und Routinen anderer Sprachen auf (nur für gemischtsprachliches Programmieren). /Es ist nicht mit /Ea kompatibel.
/G	(nur für CGA-Karten): Beschleunigt den QBX-Bildschirmaufbau, kann bei manchen CGA-Karten aber zu unruhigem Bild führen.
/H	QBX benutzt, wenn es die Hardware erlaubt, eine kleinere Schrift, so daß mehr als 25 Zeilen auf den Schirm passen.
/K: <i>tastaturdatei</i>	QBX lädt die angegebene, mit MKKEY erstellte (oder mitgelieferte) Tastaturdefinitionsdatei. Sie bleibt so lange aktiv, bis QBX wieder einmal mit einem anderen /K-Parameter aufgerufen wird.
/L [ <i>quicklibrary</i> ]	Lädt die angegebene Quick Library. Es kann nicht mehr als eine Quick Library geladen werden; wenn Sie den Namen <i>quicklibrary</i> weglassen, lädt QBX die Library QBX.QLB, die die Routinen für Interruptaufrufe und absolute Funktionsaufrufe enthält.
/MBF	Ersetzt die Funktionen MKS\$, MKD\$, CVS und CVD durch ihre Pendanten mit dem Anhängsel MBF, um Kompatibilität mit Uralt-Datenbanken zu gewährleisten. Wenn Sie beim Aufruf von QBX /MBF angeben, werden auch alle aus QBX heraus erzeugten EXE-Files mit /MBF kompiliert.
/NOF	Die Menüs "Utility", "Help" und "Options" sowie das gesamte Hilfe-System werden außer Kraft gesetzt; dadurch stehen etwa 19 KB zusätzlich für Programme zur Verfügung. Dieser Switch sollte angewendet werden, wenn kein EMS vorhanden ist und deshalb der Speicher bei größeren Programmen knapp wird.

(Fortsetzung nächste Seite)



Switch	Wirkung
/NOHI	QBX zeigt Texte und Zeichen, die sonst in heller Farbe hervorgehoben werden, anders an (je nach Einstellung beispielsweise invers). Ein Switch für Monitore, die keine High-Intensity-Farben darstellen können.
/RUN	Das genannte Programm wird sofort gestartet. Wenn es einen SYSTEM-Befehl enthält, wird QBX dadurch sofort verlassen.

## Beispiele zum QBX-Programmaufruf

QBX PROG1

Startet QBX und lädt PROG1.BAS.

QBX PROG1 /L

Startet QBX, lädt PROG1.BAS und die Quick Library QBX.QLB.

QBX /RUN PROG1 /L FONTBEFR /CMD C:\\*.\*

Startet QBX, lädt die Quick Library FONTBEFR.QLB und das Programm PROG1.BAS, startet das Programm sofort und schreibt in COMMAND\$ den Text "C:\\*.\*" hinein.

QBX /NOF/NOHI/B

Diese Zeile könnte auf einem Laptop mit LCD-Bildschirm eingegeben worden sein: Hier wird QBX gestartet und erhält die Anweisung, alles in schwarz/weiß auszugeben und keine hervorgehobenen (hellen) Farbattribute zu benutzen; außerdem werden zum Einsparen von Speicherplatz die Hilfe-Funktion und die Menüs "Utility" und "Options" deaktiviert.

## 4.2 Editor und Tastenbelegung

Die Tastenkombinationen zum Aufruf verschiedener Funktionen lassen sich beim QBX-Editor selbst definieren (siehe "MKKEY und die Tastendefinitionen", Kapitel 23); ich behandle hier aber nur die Standard-Tastenkombinationen, deren jede aus irgendeinem anderen Programm wahrscheinlich schon bekannt ist und die deshalb nicht viel Umgewöhnung erfordern.

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.

Pfeiltasten	Cursorbewegung
Insert (Einfüg)	Zwischen Einfüge- (kleiner Cursor) und Überschreibmodus (großer Cursor) umschalten
CTRL links	Wort links
CTRL rechts	Wort rechts
PgUp (Bild auf)	Seite aufwärts
PgDn (Bild ab)	Seite abwärts (Eine Seite hat so viele Zeilen, wie im Fenster dargestellt werden können.)
CTRL PgUp	Bildschirm links
CTRL PgDn	Bildschirm rechts
CTRL Home	zum ersten Zeichen im Fenster
CTRL End	zum letzten Zeichen im Fenster

### Weitere Tasten zum Bearbeiten:

Del (Entf)	Markierten Text löschen
Shift-Del	Markierten Text löschen, aber zuvor ins Clipboard kopieren (Das Clipboard ist ein Zwischenspeicher, den man zum Verschieben von Text benutzen kann.)
CTRL Y	Aktuelle Zeile löschen, aber zuvor ins Clipboard kopieren
CTRL Insert	Text aus dem Clipboard an der Cursorposition einfügen
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).

Die letzten 20 Veränderungen an geladenen Dateien behält QBX "im Kopf", ganz gleich, ob es sich um das Einfügen eines Buchstabens oder das Löschen einer ganzen Prozedur handelt.

ALT Backspace (Rückschritt)	Die letzte Änderung rückgängig machen ("Undo")
CTRL Backspace	Die letzte rückgängig gemachte Änderung doch wieder einfügen ("Redo", funktioniert nur unmittelbar nach "Undo")

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.

Für jede Unteroutine Ihres Programms (also jedes SUB und jede FUNCTION) wird ein eigenes Fenster angelegt. In dem Augenblick, in dem Sie "SUB" (gefolgt von einem Namen für die Routine) eingeben, eröffnet QBX sofort ein neues Fenster und schreibt als letzte Zeile gleich "END SUB" hinein.

Beim Start von QBX sehen Sie zwei Fenster: Der Programmbereich umfaßt fast den ganzen Bildschirm und ist mit dem Namen des geladenen Programms (oder "Untitled") überschrieben. Ein zweites, winzig kleines Fenster am unteren Bildschirmrand trägt den Titel "Immediate". In diesem Fenster wird jeder Befehl, den man eingibt, sofort ausgeführt. Die Überschrift des aktuellen Fensters (also des Fensters, in dem sich der Cursor gerade befindet) wird farblich hervorgehoben.

F6	Ins nächste Fenster springen
Shift-F6	Ins vorherige Fenster springen (bei nur zwei Fenstern identisch mit F6)
CTRL F10	Das aktuelle Fenster auf Bildschirmgröße "aufblasen"; alle anderen Fenster verschwinden, bis nochmals CTRL F10 gedrückt wird.
ALT Plus	Das aktuelle Fenster um eine Zeile vergrößern
ALT Minus	Das aktuelle Fenster um eine Zeile verkleinern

Maximal können vier Fenster auf dem Bildschirm sein: Das Hilfsfenster, das bei Druck auf F1 geöffnet wird und mit ESC wieder geschlossen werden kann, zwei Programmfenster und das Immediate-Fenster. Der Aufruf eines zweiten Programmfensters wird später beschrieben. Wie schon QuickBASIC 4.0 und vor allem 4.5 besitzt QBX eine überaus praktische Hilfefunktion.

F1	(im Programmtext) Öffnet das Hilfsfenster zum BASIC-Befehl oder zum Variablen- oder Prozedurnamen, auf dem der Cursor steht
Shift-F1	Öffnet das Haupthilfsfenster, von dem aus man (über "Index") sämtliche Hilfstexte abrufen kann
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
CTRL F1	(im Hilfsfenster) Zeigt, nachdem man ALT F1 benutzt hatte, wieder die danach gewählte Hilfsseite an
ESC	schließt das Hilfsfenster

## 4.3 Dialogboxen

Die Taste F2 öffnet eine Dialog-Box auf dem Bildschirm, die alle geladenen Dateien mit ihren Namen anzeigt und zusätzlich, unter dem Dateinamen, zu dem sie gehören, alphabetisch sortiert, sämtliche FUNCTION- oder SUB-Prozeduren. 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.

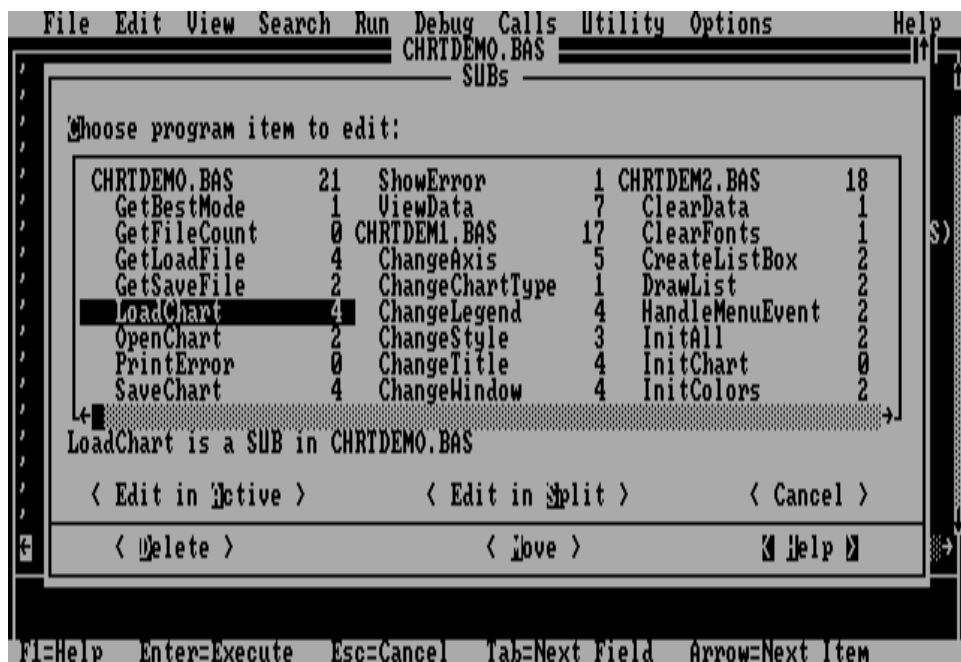


Abbildung 4-1: Die "View Subs"-Dialogbox

Hier nun zunächst ein kleiner Exkurs in das Dialogbox-Prinzip von QBX.

Eine Dialogbox enthält zumeist mehrere "Buttons", Knöpfe also, die in <>-Zeichen eingeschlossen sind. Außerdem gibt es häufig noch Eingabe- und Auswahlfelder. In unserem Beispiel, der View Subs-Dialogbox, haben wir es mit einem Auswahlfeld und 6 Buttons zu tun.

Mit der Tab-Taste kann man zwischen den einzelnen Feldern in der Dialogbox hin- und herspringen.

Einer der Buttons ist meist mit inversen oder anders hervorgehobenen Klammern versehen. Bei Betätigen der ENTER-Taste wird dieser Button "gedrückt".

Mit der Tab-Taste können Sie den Cursor zwischen den einzelnen dargestellten Buttons und Feldern hin- und herbewegen, mit der Leertaste oder den Pfeiltasten lassen sich in Auswahlfeldern Optionen selektieren, und in Eingabefelder schreibt man ganz gewöhnlich mit der alphanumerischen Tastatur.

Die Dialogbox wird erst geschlossen, wenn Sie die Enter-Taste betätigen oder einen der Buttons mit einer ALT-Kombination aktivieren. Dann werden - es sei denn, sie wählen "Cancel" oder "Help" - die zuletzt eingestellten Optionen und Eingaben akzeptiert. Fast alle Felder und Buttons haben einen andersfarbigen Buchstaben in ihrem Namen. Man kann dann durch Betätigen von ALT und diesem Buchstaben sofort in das betreffende Feld springen beziehungsweise den Button "drücken".

Natürlich können Sie eine Dialogbox auch vollständig mit der Maus bedienen - aber dazu bedarf es wohl keiner weiteren Erklärung.

Sie werden solchen Dialogboxen ständig in QBX begegnen - ob Sie nun speichern, laden, drucken, Farben verstellen oder ein Programm kompilieren wollen -, und Sie werden auch die Werkzeuge an die Hand bekommen, Dialogboxen selbst in Ihren Programmen zu verwenden (siehe "Die User Interface-Toolbox", Kapitel 9.5).

Zurück zur "View Subs"-Dialogbox: Hier können Sie auswählen, welche der Subroutinen Sie als nächstes bearbeiten wollen. Wählen Sie sie einfach mit dem Leuchtbalken an, und drücken Sie ENTER.

Außerdem können Sie von hier aus auch Subroutinen löschen und ein zweites Programmfenster öffnen beziehungsweise entscheiden, ob Sie die ausgewählte Prozedur im ersten oder zweiten Programmfenster edieren möchten.

F2	(im Programm) Öffnet die "View Subs"-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
Shift-F2	(im Programm) Schaltet in die nächste Subroutine (alphabetische Reihenfolge) weiter
CTRL F2	(im Programm) Schaltet in die Subroutine vor der gerade bearbeiteten (alphabetische Reihenfolge)
Pfeiltasten	(in der View Subs-Dialogbox) den Leuchtbalken bewegen
ENTER oder ALT A	(in der View Subs-Dialogbox) Wählt die Subroutine/das Programm, auf der/dem der Leuchtbalken steht, zur Bearbeitung in dem Programmfenster, aus dem heraus die View Subs-Dialogbox aufgerufen wurde

*(Fortsetzung nächste Seite)*

ALT S	Wählt die Subroutine/das Modul, auf der/dem der Leuchtbalken steht, zur Bearbeitung im anderen Programmfenster; wenn erst ein Programmfenster existiert, wird ein zweites geöffnet
ALT D	(in der View Subs-Dialogbox) Löscht nach einer Sicherheitsabfrage das Modul beziehungsweise die Subroutine, auf dem/der der Leuchtbalken steht
ALT M	(in der View Subs-Dialogbox) Ermöglicht das Verschieben der Subroutine, auf der der Leuchtbalken steht, in ein anderes geladenes Modul

## 4.4 Programme, Module, Dateien - die Terminologie

An dieser Stelle wird es Zeit, ein wenig auf die Terminologie einzugehen, die in QBX und demzufolge auch in diesem Buch benutzt wird.

### Dateien

Sie können jede beliebige Datei in QBX laden; beim Laden müssen Sie jedoch angeben, ob es sich dabei um ein Modul, eine Include-Datei oder ein Dokument handelt.

### Module

Ein Modul ist eine .BAS-Datei, die Modulcode und/oder Prozedurcode enthalten kann. Eines der geladenen Module muß immer das Hauptmodul sein, und wenn Ihr Programm nur aus einer einzigen BAS-Datei besteht, ist das Hauptmodul auch gleichzeitig das Programm (wenn ich später von "einem Modul" spreche und Ihr Programm nur eins umfaßt, können Sie sich stattdessen auch "Ihr Programm" denken). Das "aktuelle Modul" ist das Modul, in dem sich gerade der Cursor befindet. Die bis zu zwei Programmfenster tragen als Überschrift den Modulnamen und, wenn der Cursor sich nicht im Modulcode befindet, dahinter einen Doppelpunkt und den Namen der Subroutine, in der der Cursor steht.

### Modulcode

Modulcode nenne ich alle BASIC-Zeilen, die nicht Teil eines SUBs oder einer FUNCTION sind. Bei mehreren geladenen Modulen kann immer nur der Modulcode eines der Module ausgeführt werden; dieses Modul ist das Hauptmodul.

### Prozedurcode

Prozedurcode heißen alle BASIC-Zeilen, die zu einem SUB oder einer FUNCTION gehören.

### Include-Dateien

Include-Dateien sollten die Extension .BI haben. Eine Include-Datei wird in ein Modul eingeschlossen, indem Sie im Modul den Befehl REM \$INCLUDE ver-

wenden. Include-Dateien, die Sie explizit in QuickBASIC laden, werden wie Module behandelt, mit dem Unterschied, daß veränderte Include-Dateien immer gespeichert werden, bevor Sie das Programm starten, damit ein REM \$INCLUDE-Befehl, der sich auf diese Include-Datei bezieht, die korrekte (geänderte) Version lädt.

### **Dokumente**

Dokumente sind Dateien mit beliebigem Namen, die als ASCII-Text abgespeichert sein müssen. Sie können beliebig viele Dokumente in QBX laden (wenn der Speicherplatz ausreicht); sie beeinflussen den Programmablauf nicht. Auch Module können - wenn sie im Textformat gespeichert sind - als Dokument geladen, aber dann nicht ausgeführt werden. Außerdem würden die SUBs und FUNCTIONs eines als Dokument geladenen Moduls nicht in der View Subs-Dialogbox angezeigt werden.

### **Programme**

Ein Programm besteht aus einem Hauptmodul und beliebig vielen weiteren Modulen. Das Hauptmodul ist ein gewöhnliches Modul, mit dem Unterschied, daß der Modulcode des Hauptmoduls als einziger beim Programmstart ausgeführt wird. Der Modulcode aller anderen Module wird ignoriert. Das Hauptmodul ist in der "View Subs"-Liste immer als erstes aufgeführt, alle anderen Module werden alphabetisch sortiert.

## **4.5 Menüs**

### **Das Menü "File"**

#### **New Program**

Löscht alle Module aus dem Speicher. Fragt vorher nach, wenn Module noch nicht gespeichert sind.

#### **Open Program...**

Lädt ein neues Hauptmodul. Falls für das neue Hauptmodul eine .MAK-Datei existiert, werden alle dort erwähnten Module mitgeladen. Alle eventuell im Speicher befindlichen Module, Dokumente und Include-Dateien werden aus dem Speicher gelöscht.

#### **Merge...**

Fügt ein beliebiges Modul an die Cursorposition; dieses Modul muß als Text-Datei (und nicht im QBX-Format) gespeichert sein. Prozedurcode des Moduls erhält - wie üblich - eigene Fenster für jede Prozedur; der Modulcode wird - falls das Modul solchen enthält - direkt an der Cursorposition eingesetzt.

**Save**

Speichert das aktuelle Modul ("aktuell" ist immer das Modul beziehungsweise die Prozedur, in dem/der der Cursor steht)

**Save As...**

Speichert das aktuelle Modul, fragt aber vorher nach Dateityp (QBX- oder Textformat) und dem Dateinamen.

**Save All**

Speichert alle geladenen Module und erzeugt, wenn mehrere Module geladen sind, eine .MAK-Datei unter dem Namen des Hauptmoduls, in der alle geladenen Module erwähnt werden.

**Create File...**

Erzeugt ein neues, leeres Modul (oder eine Include-Datei oder ein Dokument) im Speicher; wenn eine gleichnamige Datei auf der Platte existiert, wird diese zunächst nicht berührt, beim Speichern später aber überschrieben.

**Load File...**

Lädt ein Modul, eine Include-Datei oder ein Dokument hinzu (es können keine Module geladen werden, die Subroutinen enthalten, die mit gleichem Namen schon in einem geladenen Modul existieren).

**Unload File...**

Löscht ein einzelnes Modul, eine Include-Datei oder ein Dokument aus dem Speicher; eine Sicherheitsabfrage findet statt, wenn die Datei verändert, aber noch nicht gespeichert wurde.

**Print...**

Druckt alle geladenen Dateien, das aktuelle Modul oder den markierten Text.

**DOS Shell**

Ruft den DOS-Interpreter auf, so daß Befehle eingegeben und andere Programme gestartet werden können. Diesen Programmen steht dann allerdings signifikant weniger Speicherplatz zur Verfügung. Mit EXIT kann von DOS wieder in QBX zurückgekehrt werden.

**Exit**

Verläßt QBX. Wenn veränderte, aber noch nicht gespeicherte Dateien geladen sind, findet eine Sicherheitsabfrage statt.

## Das Menü "Edit"

**Undo**

Macht die letzte Änderung rückgängig (wie ALT Backspace, s.o.)



**Redo**

Nimmt das letzte "Undo" zurück (wie CTRL Backspace, s.o.)

**Cut**

Löscht markierten Text, kopiert ihn zuvor ins Clipboard (wie Shift-Del, s.o.)

**Copy**

Kopiert markierten Text ins Clipboard (wie CTRL Insert, s.o.)

**Paste**

Fügt Text aus dem Clipboard an der Cursorposition ein (wie Shift-Insert, s.o.)

**Clear**

Löscht markierten Text (wie Del, s.o.)

**New SUB...**

Öffnet ein Fenster für ein neues SUB (gleichwertig mit der Eingabe von "SUB *name*")

**New FUNCTION...**

Öffnet ein Fenster für eine neue FUNCTION (gleichwertig mit der Eingabe von "FUNCTION *name*")

## Das Menü "View"

**SUBs...**

Öffnet die "View Subs"-Dialogbox (wie F2, s.o.)

**Next SUB**

Springt in das nächste SUB (wie Shift-F2, s.o.)

**Split**

Öffnet ein zweites Programmfenster, wenn es erst eines gibt beziehungsweise schließt dasjenige von beiden Programmfenstern, in dem der Cursor gerade nicht steht, wenn es bereits zwei gibt.

**Next Statement**

Wenn das Programm durch einen Breakpoint, einen Fehler oder den Benutzer abgebrochen wurde, wird der Cursor auf den Befehl gesetzt, der als nächstes ausgeführt worden wäre.

## **Output Screen**

Zeigt den Bildschirmaufbau des Programms an, der durch QBX nicht überschrieben wird (zurück zu QBX mit beliebiger Taste). Die Taste F4 hat dieselbe Funktion.

## **Included File**

Wenn der Cursor auf einer REM \$INCLUDE-Zeile steht, wird das Include-File zur Bearbeitung in ein eigenes Fenster geladen.

## **Included Lines**

Alle Zeilen einer mit REM \$INCLUDE eingebundenen Datei werden in einer anderen Farbe angezeigt. Solange "Included Lines" eingeschaltet ist, können jedoch keine Programmänderungen vorgenommen werden.

# **Das Menü "Search"**

## **Find...**

Sucht einen bestimmten Text im aktuellen Fenster, im aktuellen Modul oder in allen geladenen Dateien.

## **Selected Text**

Sucht den gerade markierten Text.

## **Repeat Last Find**

Wiederholt die letzte Suchaktion (identisch mit Taste F3).

## **Change...**

Ersetzt einen beliebigen Text durch einen anderen.

## **Label...**

Sucht ein Zeilenlabel. Ein Zeilenlabel muß am Anfang einer Zeile stehen und mit einem Doppelpunkt enden. Der Doppelpunkt wird bei "Search Label" nicht mit eingegeben.

# **Das Menü "Utility"**

## **Run DOS Command...**

Führt einen einzelnen DOS-Befehl aus (ähnlich wie "DOS Shell" im "File"-Menü, mit dem Unterschied, daß dort mehrere Befehle ausgeführt werden können).

## **Customize Menu...**

Ermöglicht es, eigene Menü-Einträge in das Utility-Menü aufzunehmen, die dann per Tastendruck aufgerufen werden können.

# Das Menü "Options"

## **Display...**

Erlaubt die Einstellung von Farben, Verschiebepalken und Tab-Länge. Es ist sinnvoll, die Tab-Länge von 8 auf 3 herunterzusetzen, um praktischer einrücken zu können.

## **Set Paths...**

Erlaubt die Einstellung der Directories für verschiedene Dateiarnten.

## **Right Mouse...**

Hier kann eingestellt werden, ob das Drücken der rechten Maustaste als Hilfe-Anforderung verstanden werden soll oder als Befehl, das Programm bis zur Cursorposition auszuführen.

## **Syntax Checking**

Wenn "Syntax Checking" eingeschaltet ist, prüft QBX jede eingegebene Zeile und zeigt sofort eine Fehlermeldung an, wenn die Zeile nicht korrekt ist.

# Das Menü "Help"

## **Index**

Zeigt das alphabetische Stichwortverzeichnis der Hilfe-Funktion an.

## **Contents**

Zeigt eine Übersicht über den Inhalt des Hilfe-Systems.

## **Topic: xxxx**

Wenn der Cursor gerade auf einem Befehlswort steht, zeigt diese Hilfsfunktion die zugehörige Informationsseite an (äquivalent zu F1).

## **Using Help**

Zeigt eine kurze Anleitung zur Benutzung der Hilfe-Funktion an (wie Shift-F1).

Die Menüs "Run" und "Debug" werde ich behandeln, nachdem einige Worte über das Starten von Programmen und die Fehlersuche gesagt sind.

## 4.6 Programme starten und "debuggen"

### Terminologie

Auch für dieses Kapitel müssen erst einige Begriffe geklärt werden.

#### Aktueller Befehl

Der aktuelle Befehl ist der Befehl, bei dem die Programmausführung unterbrochen wurde und bei dem sie fortgesetzt wird, wenn man "Continue" anwählt. Wenn ein Programm gar nicht unterbrochen, sondern korrekt beendet wurde - etwa durch einen END-Befehl im Programm - oder wenn es noch gar nicht gestartet ist, dann ist kein Befehl im Programm "aktuell".

#### Breakpoint

Ein Breakpoint ist eine Zeile, bei der der Compiler die Programmausführung unterbrechen soll. Die Wirkung eines Breakpoints ist die gleiche wie die eines STOP-Befehls im Programm. Ein Breakpoint ist eine *positionsabhängige Abbruchbedingung*.

#### Watchpoint

Ein Watchpoint ist ein Ausdruck (siehe Kapitel 3.2), der von QBX nach jedem Befehl ausgewertet wird. Sobald der Wert des Ausdrucks nicht 0 ist, wird das Programm unterbrochen. Ein Watchpoint ist eine *variablenabhängige Abbruchbedingung*. Jeder Watchpoint etabliert automatisch eine "Watch" für den betreffenden Ausdruck.

#### Watch

Hier handelt es sich um eine Zeile, die zwischen Programmfenster und der Menüzeile angezeigt wird und den Wert einer bestimmten Variablen oder eines Ausdrucks enthält.

Die Verwendung von Watchpoints und Watches, aber auch der Einsatz der Trace- und History-Funktionen (siehe weiter unten) verlangsamen den Programmablauf erheblich.

### Das Menü "Run"

#### Start

Startet das Programm neu (wie Shift-F5).

#### Restart

Löscht, wie "Start", alle Variablen und macht den ersten Befehl im Modulcode des Hauptmoduls zum aktuellen Befehl. "Restart" beginnt im Gegensatz zu "Start" nicht mit der Programmausführung.

### **Continue**

Führt das Programm vom aktuellen Befehl an weiter aus.

### **Modify COMMAND\$...**

Ermöglicht es, für die Funktion COMMAND\$ beliebigen Text einzutragen.

### **Make EXE File...**

Produziert aus allen geladenen Modulen ein lauffähiges EXE-Programm (Details siehe weiter unten).

### **Make Library...**

Produziert aus allen geladenen Modulen eine Library (Details siehe weiter unten).

### **Set Main Module...**

Ermöglicht es, ein beliebiges geladenes Modul zum Hauptmodul zu machen. Nur der Modulcode des Hauptmoduls wird bei Abarbeitung des Programms ausgeführt.

## **Das Menü "Debug"**

### **Add Watch...**

Eine neue Variable in den Watch-Bereich aufnehmen

### **Instant Watch...**

Den Wert einer Variable, auf der der Cursor steht, anzeigen (auch über Shift-F9 zugänglich)

### **Watchpoint...**

Einen neuen Watchpoint setzen (wird im Watch-Bereich angezeigt)

### **Delete Watch...**

Eine Variable oder einen Watchpoint aus dem Watch-Bereich löschen

### **Delete All Watch**

Alle Variablen und Watchpoints aus dem Watch-Bereich löschen

### **Trace On**

Schaltet den Trace-Modus ein oder aus. Im Trace-Modus ist eine Markierung neben "Trace On" sichtbar. Das Programm läuft extrem langsam. Die gerade bearbeitete Programmzeile wird hervorgehoben, so daß man den Ablauf beobachten kann. Außerdem schließt "Trace On" auch "History On" mit ein (siehe dort).

## **History On**

Schaltet den History-Modus ein oder aus. Im History-Modus ist eine Markierung neben "History On" sichtbar. Wenn das Programm unterbrochen wird, können mit Shift-F8 und Shift-F10 die letzten 20 ausgeführten Befehle des Programms vorwärts und rückwärts durchlaufen werden. "Trace On" und "Break on Errors" aktivieren automatisch den History-Modus, ohne daß eine Markierung hier sichtbar ist.

## **Toggle Breakpoint**

Macht die Zeile, auf der der Cursor steht, zu einer Breakpoint-Zeile oder wieder zu einer normalen Zeile, wenn sie bereits eine Breakpoint-Zeile ist. Vor der Ausführung einer Breakpoint-Zeile wird das Programm angehalten.

## **Clear All Breakpoints**

Macht alle Breakpoint-Zeilen wieder zu gewöhnlichen Zeilen.

## **Break on Errors**

Wenn "Break on Errors" eingeschaltet ist, wird die Programmausführung angehalten, sobald ein Fehler auftritt, selbst dann, wenn eine Fehlerbehandlungsroutine im Programm sich um diesen Fehler kümmert.

## **Set Next Statement**

Macht nach einer Programmunterbrechung den Befehl, auf dem der Cursor steht, zum aktuellen Befehl. Bei "Continue" wird das Programm dann hier fortgesetzt.

# **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 QBX nicht mehr dauernd zwischen Compiler und Editor hin- und herspringen, eine Zeile ändern, kompilieren, 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 QBX einfach mit Shift-F5. QBX 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 without IF"- oder ähnliche Fehler, die das Fehlen von Struktur-anweisungen bemängeln, erfordern meist eine etwas längere Suche, da QBX nicht anzeigen kann, wo sich der Fehler genau befindet. Das folgende Programm erzeugt zum Beispiel einen "LOOP without DO"-Fehler, anstatt daß QBX merkt,

daß ein END IF fehlt. Das müssen Sie bei der Korrektur der Fehler, die QBX meldet, im Kopf behalten.

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

Beschäftigen wir uns aber nun mit den wirklich schwierigen Fehlern. Das sind logische Fehler, die QBX gar nicht als solche erkennt. Fehler, die QBX 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.

## 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 Breakpoints unterbrechen und Variablenwerte mit Shift-F9 oder PRINT abfragen. Wenn Sie immer einen Breakpoint 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 einen Watchpoint, oder versuchen Sie herauszufinden, wo dieser Variable ein Wert zugewiesen wird. Mit der Find-Funktion aus dem Search-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 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. Aber auch mit Breakpoints läßt sich das bewerkstelligen.

## Prozeduraufrufe

Wenn Sie den Verdacht haben, daß eine bestimmte Prozedur aufgerufen oder nicht aufgerufen wird, obwohl das Gegenteil der Fall sein sollte, bauen Sie ein-

fach einen Breakpoint in die erste Zeile der Prozedur ein. Wenn die Prozedur dann aufgerufen wird, können Sie im Call-Menü ablesen, von wo aus das geschah, und auch, von wo aus die Prozedur oder Funktion aufgerufen wurde, die die betreffende Funktion aufrief usw. Wenn Sie im Call-Fenster eine Prozedur anwählen, wird diese im Programmfenster angezeigt, und der Cursor steht an der Stelle, von der aus der Aufruf erfolgte, der im Call-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 "Break on Errors" aktivieren, damit QBX 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 History-Funktion einsetzen. Noch flexibler geht es allerdings im "Handbetrieb": Setzen Sie einen Breakpoint an die Stelle, von der an es interessant wird, und starten Sie das Programm. Wenn es am Breakpoint anhält, haben Sie neben der Taste F5, die die Programmausführung normal fortsetzt, auch noch folgende Möglichkeiten:



<b>Taste</b>	<b>Bedeutung</b>
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, überspringt man, indem man den Cursor hinter die Passage setzt und F7 benutzt.

## 4.7 Quick Libraries

Ein paar Worte noch zu Quick Libraries, die nun schon mehrfach erwähnt, aber nie genau erklärt wurden:

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 QBX zugeschnitten sind.

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

Allerdings können Routinen, die in einer Quick Library sind, nicht mehr in Fehlersuche-Aktionen innerhalb QBX einbezogen werden. Wenn in einer Quick Library ein Fehler auftritt, kann QBX lediglich anzeigen, in welcher Routine das war. Sie können in der Quick Library keine Breakpoints setzen etc., weil der Source-Code nicht mehr verfügbar ist.

Im Gegensatz zu gewöhnlichen Libraries, die immer wieder verändert und aktualisiert werden können, muß eine Quick Library für jede Änderung völlig neu

erstellt werden. Es gibt auch keine Möglichkeit, eine Routine aus einer Quick Library zu entnehmen.

Mehrere Quick Libraries können auch nicht einfach zu einer einzigen Quick Library verbunden werden, wie das mit gewöhnlichen Libraries möglich ist. Mit einer Quick Library kann man also nichts anderes machen als sie unter QBX zu benutzen. Nicht zu Unrecht schreibt Microsoft: Wenn Sie Hersteller von Toolboxes o.ä. für BASIC sind, können Sie ohne weiteres Quick Libraries ihrer Toolboxes frei verteilen. Dann kann zwar jedermann mit QBX Ihre Routinen ausprobieren, aber niemand kann ein lauffähiges Programm erstellen - denn dafür reicht die Quick Library nicht, sondern man braucht eine gewöhnliche Objectcode-Library. Es gibt keine Möglichkeit, diese aus einer Quick Library zurückzugewinnen.

Man kann jedoch eine gewöhnliche Library in eine Quick Library umwandeln, und zwar mit dem Befehl

```
LINK /Q libname, quicklibname,,QBXQLB;
```

Für *libname* setzen Sie den Namen der Library ein, für *quicklibname* den Namen der zu erstellenden Quick Library.

Bedenken Sie: Alle Routinen, die in einer Quick Library unter QBX Dienst tun sollen, müssen mit der Emulator-Library (/FPi), für den Real Mode (/LR) und mit Far Strings (/Fs) kompiliert werden.

## 4.8 EXE-Programme und Libraries

Sie können aus QBX heraus EXE-Programme und Libraries erstellen, und zwar mit den Befehlen "Make EXE" und "Make Library" aus dem "Run"-Menü. Beide Befehle erfordern, daß das Programm keine Fehler enthält.

### Libraries

Die Erstellung von Libraries mit QBX ist fast gleichwertig mit der separaten Methode, bei der BC, LINK und LIB benutzt werden müssen, um die gleichen Resultate zu erzielen. Das Problem dabei ist, daß nur Libraries für Far Strings, DOS und die Emulator-Library erzeugt werden können. Für Quick Libraries ist das zwar ohnehin die einzige Möglichkeit, aber bei den normalen Objectcode-Libraries sind auch alle anderen Kombinationen möglich. Außerdem ist es nicht möglich, Routinen einzubinden, die in anderen Sprachen geschrieben sind und als OBJ-Dateien vorliegen. Deshalb müssen zum Beispiel die Quick Libraries der Font-, der Presentation Graphics- und der User Interface-Toolbox immer noch auf separatem Wege produziert werden.

Wenn Sie die "Make Library"-Funktion benutzen, erstellt QBX sowohl eine Quick Library (.QLB) als auch eine gewöhnliche Objectcode-Library (.LIB). Die Quick Library wird für QBX 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 - bis auf notwendige COMMON-, DIM-, DECLARE- und DEFxxx-Befehle und eventuell Fehlerbehandlungs- oder Event-Trapping-Routinen - gelöscht werden, da er die Libraries nur unnötig verlängern würde. Wählen Sie dann "Make Library" aus dem "Run"-Menü. 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 QBX, dem *integrierten Kompilieren*, besteht gegenüber dem separaten Kompilieren (siehe Kapitel 6) eine Anzahl von Einschränkungen:

- Wenn Sie mit einer Quick Library arbeiten, nimmt QBX 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 DOS, mit Far Strings und mit der Emulator-Library erstellen können. Bei der separaten Compilierung und bei QBX ohne Quick Library haben Sie dagegen die Auswahl zwischen Near und Far Strings, zwischen DOS und OS/2 und zwischen Emulator- und Alternate Math-Library.
- Sie können mit der "Make EXE"-Funktion keine EXE-Programme erstellen, die mit eigenen Runtime-Modulen (siehe Kapitel 20) arbeiten.
- Für Programme, die ganz ohne Runtime-Modul funktionieren, kann man meistens beim Linken diverse Verzicht-Files angeben (siehe Kapitel 19.3), die das Programm unter Umständen erheblich verkürzen. Das ist mit der "Make EXE"-Funktion nicht möglich.
- LINK kann durch die Switches /F/PACKC und /NON schnellere beziehungsweise kürzere Programme herstellen (siehe die Kapitel 6.4 und 19.1). QBX unterstützt diese Switches nicht, man kann sie höchstens in die Betriebssystemvariable LINK eintragen.
- Es lassen sich mit der "Make EXE"-Funktion keine Programme erstellen, die mit Overlays arbeiten.

Wenn Sie sich von alldem nicht schrecken lassen: Laden Sie alle Module, die Bestandteil des EXE-Programms werden sollen. Nur der Modulcode im als Hauptmodul deklarierten Modul 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 "Make EXE", und stellen Sie die gewünschten Optionen ein (siehe auch Kapitel 6.3). Switches, die Sie hier nicht einstellen können und die QBX auch nicht selbst setzt (wie zum Beispiel /S), können Sie in das "Additional Options"-Feld (erst ab PDS 7.1) eintragen.

QBX erzeugt dann das EXE-Programm unter dem gewünschten Namen.

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

# 5 Die Entwicklungsumgebung PWB

Noch in der Version 7.0 wurde der Microsoft-Editor "M" mit dem BASIC PDS ausgeliefert. Es handelte sich dabei um einen leistungsfähigen und programmierbaren Editor, aus dem heraus man auch den Compiler aufrufen und sich Fehler im Programm anzeigen lassen konnte. Mit der kontinuierlichen Verbesserung von QuickBASIC hatte der Editor aber inzwischen seine Daseinsberechtigung verloren, und bei Microsoft wurde etwas neues ausgeheckt.

Das neue High-End-Produkt trägt den Namen PWB, "Programmer's WorkBench". Die PWB ist trotz allem nicht mehr als ein komfortabler Editor, wie der folgende Vergleich mit QBX zeigen soll.

QBX	PWB
Kompiliert eingegebene Programmzeilen sofort. Findet auf diese Weise auch sofort Tippfehler, schreibt BASIC-Befehle in Großbuchstaben, fügt Leerzeichen ein.	Fungiert als reiner Editor, keines der links genannten Features ist verfügbar.
Öffnet für jedes SUB beziehungsweise jede FUNCTION ein neues Fenster.	Kann zwar beliebig viele Dateien in beliebig vielen Fenstern bearbeiten, bietet jedoch diese Automatik nicht.
Kann ein Programm im Interpreter-Modus ablaufen lassen, Breakpoints setzen und den Programmablauf verfolgen.	Bietet diese Möglichkeit nicht; stattdessen muß das Programm auf besondere Weise kompiliert werden, um dann mit CodeView im Debug-Modus bearbeitet werden zu können.
Programm kann nach einer kleinen Änderung sofort weiter ausgeführt werden.	Damit eine Änderung wirksam wird, muß das ganze Programm neu kompiliert und gelinkt werden.

Ein weiterer Punkt ist zu beachten: Wenn Sie keinen 386er-Rechner mit genügend Speicher und schneller Festplatte haben, wird Ihnen das Arbeiten mit der PWB keinen Spaß machen. Bedingt durch ihr Konzept muß die PWB ständig Dateien laden, speichern, durchsuchen, sie muß andere Programme aufrufen und deren Ausgabe wieder einlesen usw. Auf einem AT, selbst wenn es ein Modell der neuesten Generation ist, kann die PWB den Kaffee- oder Teekonsum des Programmierers gefährlich erhöhen.

Die PWB ist eine sogenannte "Shell", also ein Programm, das andere Programme aufruft. Wenn Sie mit der PWB ein einfaches kleines BASIC-Programm schreiben und dieses kompilieren wollen, ruft die PWB, die selbst ja Ihr Programm überhaupt nicht "verstehen" (wie QBX das tun würde), erst einmal den BASIC-Compiler BC auf. Wenn Ihr Rechner über kein EMS verfügt, wird die PWB sich erst einmal selbst aus dem Speicher entfernen, was den unangenehmen Nebeneffekt hat, daß sie sich später wieder von der Platte laden muß, und das dauert lange.

Der Compiler tut dann seine gewohnte Arbeit, und wenn er fertig ist, wertet die PWB das aus, was er ausgegeben hat. Befinden sich darunter Fehlermeldungen, zeigt die PWB diese an. Ist das Programm einwandfrei, wird nun mit LINK das EXE-Programm erstellt und kann dann schließlich gestartet und/oder auf Fehler untersucht werden.

Obwohl es bisher nicht so klingt, hat die PWB auch einige Vorteile gegenüber QBX. Diese sind:

**OS/2-Tauglichkeit:** Die PWB kann ohne Einschränkungen auch unter OS/2 eingesetzt werden. QBX läuft nur unter DOS.

**Source Browser:** Mit dieser Funktion können Sie auch bei Programmsystemen, die aus mehreren Modulen bestehen, schnell erkunden, welche Variablen und / oder Prozeduren in welchem Modul definiert sind, wo sie benutzt werden usw.

**Gemischtsprachliches Programmieren:** Wie bereits gesagt, kümmert sich die PWB nicht selbst darum, ob Sie BASIC-Code oder einen Brief an Ihre Mutter eingeben (bei letzterem käme QBX aus dem Protestieren nicht mehr heraus). Die Verarbeitung Ihrer Eingaben übernehmen bei der PWB die Compiler selbst. Das bedeutet, daß Sie mit der PWB auch in mehreren Sprachen gleichzeitig programmieren können, vorausgesetzt, die PWB unterstützt die entsprechenden Compiler und die Compiler kennen ihrerseits die PWB. Die PWB arbeitet mit allen PDS-Systemen von Microsoft zusammen - neben anderen zum Beispiel BASIC, C oder Assembler, und weitere werden folgen.

Meine persönliche Empfehlung: Wenn Sie nur in BASIC programmieren, lassen Sie die Finger von der PWB. Sie macht alles nur komplizierter. Begnügen Sie sich mit QBX, schreiben Sie sich Prozeduren, die das Compilieren erleichtern, oder benutzen Sie NMAKE. Wenn Sie hauptsächlich in BASIC programmieren und nur ab und zu eine kleine Routine in Assembler oder C erstellen, um diese dann mit Libraries oder Quick Libraries in Ihre Programme einzubinden, sind Sie auch mit QBX bestens bedient. Der Einsatz der PWB lohnt sich wirklich nur dann, wenn Sie so viel in anderen Sprachen programmieren, daß Sie die Vorteile von QBX gar nicht mehr voll ausnutzen können. Wenn Sie ständig in mehreren Sprachen arbeiten, so daß der QBX-Horizont für Sie nicht weit genug ist - dann erst sollten Sie sich mit der PWB beschäftigen. Oder natürlich dann, wenn Sie

partout nicht nur Programme schreiben wollen, die unter OS/2 laufen, sondern die Programme auch unter OS/2 entwickeln wollen.

Ich werde mich hier darauf beschränken, das Konzept der PWB verständlich zu machen und einige Begriffe und Verfahrensweisen zu erklären.

## 5.1 Der Aufruf der PWB

Zum Aufruf der PWB benutzen Sie die Syntax:

PWB [*switches*] [*programmname*] [*programmname*]...

Wenn Sie mindestens einen Programmnamen angeben, wird dieses Programm in den Editor geladen. Die PWB verlangt, daß die Extension mit angegeben wird, weil sie ja nicht wissen kann, ob Sie ein BASIC- oder C-Programm edieren wollen. Alle weiteren genannten Dateien kommen in die "Warteschlange", so daß sie mit "Load Next" aus dem "File"-Menü geladen werden können.

Geben Sie keinen Dateinamen an, lädt die PWB automatisch den letzten Stand, also alle Dateien, die geladen waren, als Sie zuletzt die PWB verließen.

Die folgenden *switches* sind möglich.

Switch	Bedeutung
/D[A][S][T]	/DA verhindert, daß die PWB-Zusätze (siehe unten) automatisch geladen werden. /DS verhindert, daß die Datei CURRENT.STS eingelesen wird und so der letzte Stand Ihrer Arbeit wiederhergestellt wird. /DT verhindert, daß die PWB versucht, Befehle für sich aus der Datei TOOLS.INI zu entnehmen. /D alleine wirkt wie /DA, /DS und /DT zugleich.
/E <i>text</i>	Führt die in <i>text</i> genannten PWB-Befehle sofort aus.
/M <i>cursor</i>	Positioniert den Cursor an der durch <i>cursor</i> spezifizierten Stelle in einer geladenen Datei. <i>cursor</i> hat das Format <i>zeile.spalte [dateiname]</i> , also zum Beispiel 3.28 FORMAT.BAS für Zeile 3, Spalte 28 in FORMAT.BAS.
/PF <i>datei</i>	Lädt die NMAKE-kompatible Programmliste <i>datei</i> .
/PL	Lädt die letzte in der PWB benutzte Programmliste.
/PP <i>datei</i>	Lädt die PWB-kompatible Programmliste <i>datei</i> .
/R	Verhindert, daß irgendeine geöffnete Datei in verändertem Zustand auf die Platte geschrieben werden kann.
/T	(nur unmittelbar vor einem Dateinamen erlaubt) Legt fest, daß die genannte Datei eine temporäre Datei ist, die von der PWB nicht in der "Warteschlange" gehalten werden soll.

Um sich an verschiedene Umgebungen besser anpassen zu können, arbeitet die PWB mit einem modularen Konzept. Das bedeutet, daß die Grundfunktionen (Menüs usw.) durch bestimmte Zusätze erweitert werden, je nachdem, welche

Anforderungen man hat. Je mehr Erweiterungen benutzt werden, desto länger dauert es jedesmal, bis die PWB geladen ist.

Im Lieferumfang des BASIC PDS sind Zusätze für BASIC (PWBBASIC), C (PWBC), Source Browser (PWBROWSE), Hilfe-System (PWBHELP) und die Unterstützung der Hilfsprogramme LINK, NMAKE und CodeView (PWBUTIL) enthalten. Die Dateien haben die Extension .MXT für den Real Mode und .PXT für den Protected Mode.

## TOOLS.INI

Beim Aufruf der PWB werden alle vorhandenen Zusätze automatisch geladen. Sie können dies jedoch durch den Switch /DA verhindern. Wenn Sie trotzdem selektiv einige der Zusätze laden wollen, geht das mit einem Eintrag in der Datei TOOLS.INI. TOOLS.INI sollte in einem Directory stehen, das in der Betriebssystemvariable INIT erwähnt ist (zum Beispiel mit SET INIT=C:\BC7). PWB durchsucht diese Datei - wenn Sie nicht den Switch /DT benutzen - bei jedem Aufruf nach Befehlen. Dabei werden alle Befehle, die einer Zeile namens

[pwb]

folgen, als PWB-Befehle aufgefaßt, solange, bis ein anderer Text in eckigen Klammern auftritt. Der Befehl, um Zusätze zu laden, heißt LOAD und wird gefolgt von einem Doppelpunkt und den durch Semikolon getrennten Namen (eventuell mit Pfad) der betreffenden Zusätze für Real und Protected Mode.

Zum Beispiel:

```
LOAD: PWBBASIC.MXT; PWBBASIC.PXT
```

Sie können außerdem in TOOLS.INI auch noch Fallunterscheidungen bezüglich geladener Dateien vornehmen, in dem Sie die Markierung [pwb] erweitern:

[pwb]	Befehle, die immer ausgeführt werden
[pwb - .BAS .BI]	Befehle, die ausgeführt werden, wenn ein .BAS oder .BI-Programm von PWB geladen wird
[pwb - .C]	Befehle, die ausgeführt werden, wenn ein .C-Programm von PWB geladen wird

## 5.2 Der PWB-Editor

In vielem ist der PWB-Editor dem QBX-Editor ähnlich. Das betrifft das "File"-Menü ebenso wie die Möglichkeiten zum Bewegen des Cursors innerhalb eines Fensters (Pfeiltasten) oder von Fenster zu Fenster (mit F6), die Clipboard-Funktionen wie auch die Undo- und Redo-Befehle.



## Neue Tasten

Die wichtigsten neuen oder unterschiedlichen Tasten sind:

F4	Letzten gesuchten Text rückwärts suchen.
F8 und F9	PWB verlassen, ohne zu speichern.
ALT F4	PWB verlassen und speichern.
F2	Die zuletzt bearbeitete Datei laden.
Shift-F7	Alle bisher an der Datei durchgeführten Änderungen werden unwirksam.
CTRL F8	Die Größe eines Fensters verändern. Nachdem CTRL F8 gedrückt wurde, kann das Fenster entweder mit Pfeil auf/Pfeil ab oder mit Pfeil links/Pfeil rechts in der Größe verändert werden, bis man ENTER drückt.
Shift-F3	Nächsten gefundenen Fehler anzeigen.
Shift-F4	Vorherigen gefundenen Fehler anzeigen.
CTRL F4	Fenster, in dem der Cursor steht, schließen.

## Marken

In QBX lassen sich mit speziellen Tastenkombinationen vier Markierungen setzen, zu denen man jederzeit wieder springen kann. In der PWB können Sie beliebig viele Stellen in einer Datei mit dem "Define Mark"-Befehl aus dem "Search"-Menü markieren. Dabei müssen Sie einen Namen für die Stelle vergeben; später kann jederzeit durch Aufruf von "Goto Mark" aus demselben Menü eine Liste aller vereinbarten Markierungen abgerufen und zu einer dieser Markierungen gesprungen werden.

Wenn Sie den "Set Mark File"-Befehl aus dem "Search"-Menü aufrufen und den Namen einer Markierungsdatei angeben, werden alle Markierungen, die Sie setzen, in der angegebenen Datei gespeichert, und alle Markierungen, die in der Datei bereits vermerkt sind, sind wieder zugänglich.

## Text markieren

Wie auch in QBX können in der PWB durch Drücken von SHIFT und Bewegen des Cursors Texte markiert werden. Dabei ist jedoch zu beachten, daß die PWB drei verschiedene Markierungsmodi kennt, die im "Edit"-Menü gewählt werden: "Box"-, "Stream"- und "Line"-Modus sind verfügbar. Der erstgenannte Modus markiert immer den rechteckigen Bereich von der Stelle, an der mit der Markierung begonnen wurde, bis zu der Stelle, an der der Cursor gerade steht. Der Stream-Modus entspricht der Art und Weise, wie auch in Microsoft Word Texte markiert werden: Von der Position, an der mit der Markierung begonnen wurde, immer nach rechts, bei Zeilenende weiter auf die nächste Zeile, bis zu der Stelle, an der der Cursor steht. Der letzte Markierungsmodus schließlich, Line, markiert

die Zeile, in der mit der Markierung begonnen wurde, die Zeile, in der der Cursor jetzt steht, sowie alle dazwischenliegenden.

Über diese verschiedenen Markierungsmodi hinaus gibt es mittels der Funktionen "Set anchor" und "Select to anchor" aus dem "Edit"-Menü die Möglichkeit, große Textmengen leicht zu markieren, indem man dort, wo die Markierung anfangen soll, den "Set anchor"-Befehl ausführt und später dann mit "Select to anchor" alles von der Cursorposition bis zu diesem "Anker" markiert.

## Pseudo-Dateien

Für umfangreichere Einstellungen oder Informationen (zum Beispiel Ergebnisse eines Compiler-Laufs) öffnet die PWB stets eigene Fenster, die sogenannte Pseudo-Dateien enthalten. Dabei handelt es sich um Dateien, die nicht wirklich auf der Platte existieren, sondern deren Existenz von der PWB nur "vorgegaukelt" wird. Man ändert oder betrachtet sie wie gewöhnliche Dateien, und wenn die Information nicht mehr benötigt wird, schließt man das Fenster mit CTRL F4.

## 5.3 Ein Projekt

Sie können die PWB auch benutzen, um nur an einer einzigen Datei zu arbeiten und diese dann zu linken und zu kompilieren. Ich fange allerdings gleich etwas anspruchsvoller an und betrachte ein Programmsystem, das aus mehreren Dateien besteht.

### Programmliste

Um ein Projekt in der PWB zu bearbeiten, müssen Sie zunächst eine Programmliste erstellen. Eine Programmliste enthält die Namen aller Programme, die zu dem Projekt gehören, also zu einem EXE-Programm zusammengelinkt werden.

Wählen Sie "Set Program List" aus dem "Make"-Menü, und geben Sie einen Namen für Ihr Projekt (und damit auch für die Programmliste) an.

Da beim ersten Mal die zugehörige .MAK-Datei noch nicht existiert, wird die PWB fragen, ob sie neu erstellt werden soll. Antworten Sie mit "Yes", und Sie erhalten eine Dialogbox, mit der Sie hintereinander beliebig viele Dateien auswählen können, die dann in der Programmliste angezeigt werden. Wählen Sie eine Datei, die bereits in der Liste steht, ein zweites Mal, wird sie wieder aus der Liste gestrichen.

Wenn Sie mit dem Erstellen der Liste fertig sind, wählen Sie "Save List". Daraufhin prüft die PWB zunächst, ob sie mit den Dateien, die Sie angegeben

haben, etwas anfangen kann. Wenn Sie .BAS-Dateien angeben und die BASIC-Zusätze nicht geladen sind, wird zum Beispiel eine Fehlermeldung angezeigt.

Nehmen wir an, Sie haben die zwei .BAS-Programme HAUPT.BAS und TOOLS1.BAS sowie ein in C programmiertes Modul namens TOOLS2.C. Dann müßten Sie diese drei Dateien angeben und danach die Programmliste speichern lassen. Wichtig ist, daß die BASIC- und die C-Zusätze geladen sind (siehe Kapitel 2.5). Include-Dateien werden übrigens in der Programmliste nicht mit angegeben, da die jeweiligen Compiler sie beim Kompilieren selbständig laden.

Die Programmlisten, die die PWB als .MAK-Dateien speichert, sind nicht kompatibel mit denen, die QuickBASIC benutzt, aber kompatibel zu denen von NMAKE. Das heißt, Sie können ein PWB-.MAK-File auch außerhalb der PWB mit NMAKE ablaufen lassen. Umgekehrt ist diese Kompatibilität jedoch eingeschränkt. Wenn Sie ein von Hand erstelltes .MAK-File, das ursprünglich für den Gebrauch mit NMAKE vorgesehen war, in der PWB als Programmliste verwenden wollen, müssen Sie bei der Auswahl der Programmliste "Use as Non-PWB-Makefile" anwählen. Dann haben die Compiler- und Linker-Switches, die Sie im "Options"-Menü setzen, keinen Einfluß mehr. Der Source Browser kann dann nur benutzt werden, wenn Sie sich selbst darum kümmern, daß durch NMAKE die entsprechenden Browser-Datenbanken mit Hilfe des Programms PWBRMAKE erstellt werden.

## Debug und Release

Alle Compiler und der Linker haben verschiedene Switches, die dazu dienen, den erzeugten Code zu komprimieren oder Debugger-Informationen einzufügen usw.

Damit Sie es leicht haben, zwischen einer ordentlichen, kompaktierten Programmversion und einer zum Debugging mit CodeView geeigneten umzuschalten, merkt sich die PWB stets zwei Gruppen von Compiler- und Linker-Switches für ein Projekt: Die "Debug Options" und die "Release Options". "Debug Options" sind die Switches, die bei den verschiedenen Compilern und dem Linker angewandt werden sollen, wenn es gilt, eine Codeview-kompatible Programmversion zu erzeugen, ein EXE-File also, das eine ganze Anzahl an prinzipiell überflüssigen Daten enthält. Die "Release Options" hingegen sind die Switches, die die PWB benutzen soll, wenn Sie ein auslieferungsfertiges, kompaktes Programm ohne Debug-Optionen erzeugen wollen.

Im "Options"-Menü gibt es verschiedene Menüpunkte für die Programme, die von der PWB aus aufgerufen werden. Besonders wichtig sind dabei die "LINK Options", die "BASIC Compiler Options" und die "C Compiler Options". Bei allen dreien müssen Sie einige grundsätzliche Parameter einstellen und können darüberhinaus Switches angeben, die nur für den Debug- oder nur für den Release-Modus benutzt werden sollen.

Bei "Build Options" stellen Sie ein, ob die PWB gerade im Debug- oder im Release-Modus arbeiten soll. Außerdem können Sie dort mittels "Save Current Build Options" alle eingestellten Switches und Parameter für die Compiler und den Linker unter einem beliebigen Stichwort, zum Beispiel "EXE für DOS mit Runtime und Emulator-Library", in der Datei TOOLS.INI speichern. Dann können Sie diese Switch-Kombination jederzeit mit "Set Initial Build Options" aus dem "Options"-Menü wieder für ein neues Projekt benutzen.

Welche Standard-Switches ("Initial Build Options") von vornherein zur Auswahl stehen, hängt davon ab, welche Sprache Sie zur "Main Language" (ebenfalls in der "Build"-Dialogbox) erklären. Wählen Sie BASIC.

## Build

Wenn Sie alle Optionen passend eingestellt haben, können Sie auch schon beginnen und das Projekt kompilieren und linken lassen.

Wählen Sie dazu "Build" aus dem "Make"-Menü. Die PWB fängt nun an, mit den Compilern und dem Linker das EXE-File zu erzeugen. In unserem Beispiel würde der C-Compiler mit den C-Compiler-Switches für TOOLS2.C gestartet, der BASIC-Compiler hätte sich mit den für ihn eingestellten Switches um HAUPT.BAS und TOOLS1.BAS zu kümmern, und LINK dürfte schließlich alles zusammenlinken. "Build" erzeugt - im Gegensatz zu "Rebuild All" - nur die Dateien neu, die sich verändert haben.

Wenn während des Build-Vorgangs Fehler auftraten, werden diese in einem Fenster angezeigt, und Sie können außerdem, wenn Sie Programme geladen haben, die am Build-Prozeß beteiligt waren, mit den Tasten Shift-F3 und Shift-F4 die Fehler im Programmtext sofort lokalisieren.

Erst, wenn das Programm fehlerlos erstellt werden konnte, können Sie wählen, ob Sie CodeView aufrufen, das Programm starten oder die Browser-Informationen betrachten wollen.

## 5.4 Der Source Browser

Der Source Browser kann nur benutzt werden, wenn der Zusatz PWBROWSE.MXT (oder .PXT) geladen wurde und wenn Sie im "Options"-Menü die "Browse Options" aktiviert haben.

Beim Kompilieren werden dann Informationsdateien für den Source Browser generiert, so daß Sie schnell und bequem feststellen können, wo in Ihrem Programm ein bestimmtes Symbol (das heißt, eine Variable oder eine Prozedur) vorkommt. Beim Source Browser werden übrigens Prozeduren und Funktionen als "Functions", selbstdefinierte Typen als "Types" und Konstanten als "Macros"

bezeichnet, eine durch die Flexibilität bedingte Terminologie (der Browser ist ja nicht BASIC-spezifisch), an die Sie sich gewöhnen müssen.

## Goto Definition

Mittels des Befehls "Goto Definition" aus dem "Browse"-Menü können Sie eine Liste aller Symbole in Ihrem Projekt abrufen, sehen, welches Symbol wo definiert ist, und den Cursor auf Wunsch dorthin setzen lassen.

## Goto Reference

"Goto Reference" zeigt eine Liste aller Symbole in Ihrem Projekt und welches Symbol von wo aufgerufen wird. Auf Wunsch wird der Cursor dorthin gesetzt.

## View Relationship

Mit "View Relationship" können Sie verschiedene Informationen über die Beziehung zwischen zwei Symbolen in Ihrem Programm abrufen. Beispiele für mögliche Anfragen sind "Welche Routinen werden von der Prozedur x aufgerufen", "Welche Variablen werden von der Funktion y benutzt" oder "Wo findet die Konstante z Anwendung". Wählen Sie "View Relationship" aus dem "Browse"-Menü. Wählen Sie dann zunächst in der "Operation"-Liste am rechten Bildschirmrand "Program Symbols", um eine Liste aller Symbole angezeigt zu bekommen. Dann können Sie eines der Symbole auswählen und in der "Operation"-Liste einen der folgenden Punkte aktivieren:

Operation	Bedeutung
Contains	Welche Symboldefinitionen enthält die angewählte Datei?
Calls	Welche Prozeduren und Funktionen ruft die angewählte Funktion / Prozedur auf?
Called By	Von welchen Prozeduren / Funktionen wird die angewählte Funktion / Prozedur aufgerufen?
Uses	Welche Symbole benutzt die angewählte Funktion / Prozedur?
Used By	Von welchen Prozeduren / Funktionen wird die angewählte Variable / Konstante benutzt?
Used In	In welchen Prozeduren / Funktionen wird die angegebene Variable / Konstante benutzt (ohne Prozeduren/Funktionen, in denen sie definiert wird)?
Defined in	Wo ist die angegebene Variable / Konstante / Prozedur / Funktion definiert?

Sie können außerdem mit der "Show only"-Box rechts unten einschränken, was Sie als Antwort auf Ihre Frage sehen möchten.

## Call Tree

Ein "Call Tree" kann für eine bestimmte Funktion oder für ein ganzes Programm angelegt werden. Er ist eine Liste aller Funktionen und Prozeduren, die aus der genannten Prozedur/Funktion beziehungsweise aus allen Prozeduren und Funktionen in dem genannten Programm aufgerufen werden. Prozedurnamen werden mit einem Fragezeichen versehen, wenn es sich um externe Prozeduren handelt, deren Definition nicht verfügbar ist. Sie erhalten eine in eckigen Klammern angegebene Nummer, wenn sie mehrmals aufgerufen werden (die Nummer gibt dann die Anzahl der Aufrufe an). Drei Punkte folgen der Erwähnung einer Prozedur oder Funktion im "Call Tree", wenn sie an anderer Stelle genauer ausgeführt wird. Wenn zum Beispiel zwei Prozeduren die Prozedur "DisplayInfo" aufrufen, wird DisplayInfo im Call Tree nur einmal vollständig mit allen seinen Subroutinen angezeigt, das zweite Mal mit drei Punkten.

## Reference List

Die "Reference List" ist im Prinzip dasselbe wie der "Call Tree", nur daß hier umgekehrt vorgegangen wird: Bei jeder Prozedur wird angezeigt, von welchen Prozeduren sie aufgerufen wird, anstatt anzuzeigen, welche Prozeduren sie aufruft.

## Outline

Eine "Program Outline" ist einfach eine Liste aller Symbole im Programm mit der Angabe, ob es sich dabei um eine Funktion/Prozedur, eine Variable oder ein anderes Programmelement handelt, und ob es sich um eine lokale oder globale Definition handelt (in BASIC sind Funktionen und Prozeduren immer globale Definitionen).

## 5.5 Ausblick

Ich will mich hier nicht weiter mit der PWB beschäftigen. Wenn die PWB jedoch von der Programmer's WorkBench zu Ihrer Personal WorkBench avancieren soll und Sie sie tatsächlich als Umgebung für Ihre künftigen Programmierarbeiten nutzen wollen, sollten Sie noch dieses wissen:

Die PWB besteht aus einer großen Anzahl von Funktionen. Vielen Funktionen ist eine Taste oder eine Menüauswahl zugeordnet, so zum Beispiel "Undo" = ALT Backspace. Aber eine ganze Anzahl von Funktionen ist überhaupt nicht direkt ansprechbar und kann von Ihnen beliebig entweder einer Tastenkombination zugeordnet oder aus einem Makro heraus aufgerufen werden. Die Funktionen bilden so etwas wie eine Programmiersprache, in der Sie eigene Makros programmieren können. Mit Hilfe dieser Makros können Sie für häufig benötigte Anwendungen leicht automatische Abläufe programmieren, die Ihnen die Arbeit mit der PWB erleichtern. Allerdings erfordert das zunächst die Einarbeitung in die PWB-Funktionen und (vor allem) die Kenntnis der Funktionsnamen. Aus meiner Sicht lohnt sich diese Investition von Arbeit nicht - aber entscheiden Sie selbst.





# 6 Separates Kompilieren mit BC und LINK

In vielen Fällen ist es notwendig oder zumindest praktischer, seine Programme nicht über die Make EXE-Funktion aus dem QBX-Menü heraus, sondern direkt mit dem eigentlichen Compiler BC.EXE zu erzeugen, den QBX 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 QBX allein. Bestimmte Compiler-Optionen sind über QBX gar nicht zugänglich, und auch Hilfsprogramme wie BUILDRTM (siehe "BUILDRTM und Runtime-Module", Kapitel 20) und NMAKE ("Automatisierte Programmerstellung mit NMAKE", Kapitel 22) werden von QBX nicht unterstützt. 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 zum *integrierten Kompilieren*, das von QBX automatisch auf Befehl ausgeführt wird.

Die PWB bietet ein Mittelding zwischen beiden Verfahrensweisen, hat jedoch auch nicht die Flexibilität, die dem separaten Kompilieren innewohnt.

## 6.1 Die verschiedenen Dateiarnten

Zur Arbeit mit BC und LINK ist zunächst das Verständnis der fünf Dateiarnten erforderlich, mit denen man dabei zu tun bekommt:

**.BAS** ist die Extension der BASIC-Source-Dateien, die mit QBX oder mit einem beliebigen Texteditor erzeugt werden. Für jedes lauffähige EXE-Programm ist mindestens ein BAS-Modul (das Hauptmodul) erforderlich, das Modulcode besitzt, also Befehle, die nicht zu einer Prozedur gehören. Dieses Programm kann darüberhinaus natürlich auch Prozeduren enthalten. Ich nenne dieses BAS-Programm in den folgenden Beispielen HAUPT.BAS. Außerdem können beliebig viele weitere BAS-Module verwendet werden, die nur Prozeduren enthalten. Eventueller Modulcode von weiteren BAS-Programmen wird nicht berücksichtigt, es sei denn, es handelt sich um Definitionsbefehle wie COMMON, DIM usw.

**.OBJ** heißen die Object-Files, die Objektdateien, die der Kompiler BC aus BAS-Programmen macht. Eine solche Datei ist kein lauffähiges Programm, sondern nur ein Zwischenprodukt auf dem Wege dahin. Eine Objektdatei ist sprachunab

hängig, 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 lauffähiges Programm zusammenzubinden, aber wenn man gewisse Regeln beachtet, kann man durchaus "multilinguale" Programme erstellen. Das OBJ-File ist die einfache Übersetzung des Programmes in Maschinensprache. Es enthält viele Funktionsaufrufe 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 einem oder mehreren Object-Files zusammengesetzt und ist ausschließlich zur Nutzung innerhalb von QBX zu gebrauchen. In QBX können nie mehrere Quick Libraries gleichzeitig benutzt werden.

**.LIB** ist die Bezeichnung für eine gewöhnliche Library, die mittels des LIB-Programmes aus einem oder mehreren Object-Files erzeugt wird. Eine Library ist nichts weiter als eine Sammlung von OBJ-Dateien. Sie kann nicht in QBX 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 (es sei denn, man arbeitet mit einem Runtime-Modul - siehe dazu Kapitel 20). Auch Runtime-Module unter DOS haben die Extension EXE.

## 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.)

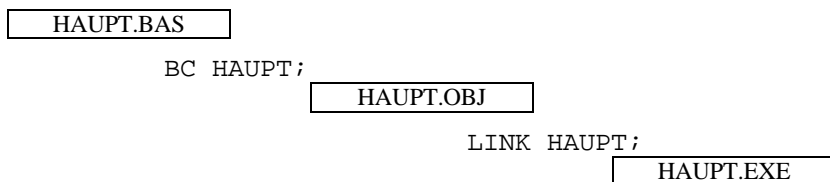
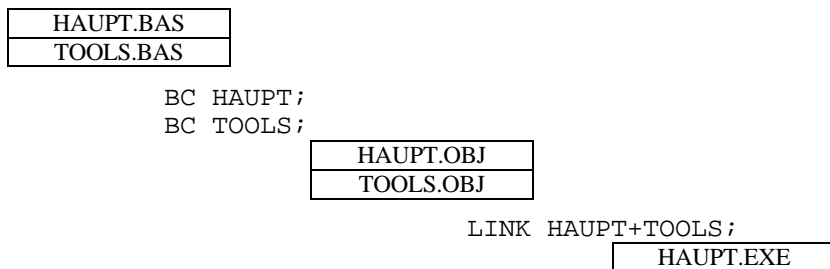


Abbildung 6-1: Simples separates Kompilieren

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 QBX einsetzen, es sei denn, man möchte beim Kompilieren oder Linken bestimmte Switches benutzen, die von QBX aus nicht aktiviert werden können, oder Verzicht-Files einbinden.

Betrachten wir einen etwas komplizierteren Fall:



*Abbildung 6-2: Separates Kompilieren mit zwei Dateien*

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 ein, wenn ein Programm so groß wird, daß man einige seiner Subroutinen in eine zweite Datei (hier TOOLS.BAS) auslagern muß, weil es sich sonst nicht mehr kompilieren läßt, oder weil man Wert auf ordentliches Modularisieren und Strukturieren legt.

Hat man mehrere solcher Tools-Dateien, kann man auch mit einer Library arbeiten, wie es Abbildung 6-3 zeigt.

Auf diese Weise vereinigt man alle TOOLS-Routinen aus den drei BAS-Programmen 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 in "LIB und Libraries", Kapitel 21. Das letztgenannte Beispiel werde ich außerdem noch zur Erläuterung der Runtime-Module in Kapitel 20 und auch bei der Behandlung des Hilfsprogramms NMAKE (Kapitel 22) heranziehen.

HAUPT.BAS
TOOLS1.BAS
TOOLS2.BAS
TOOLS3.BAS

```
BC HAUPT;
BC TOOLS1;
BC TOOLS2;
BC TOOLS3;
```

HAUPT.OBJ
TOOLS1.OBJ
TOOLS2.OBJ
TOOLS3.OBJ

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

TOOLS.LIB
-----------

```
LINK HAUPT,,,TOOLS;*
```

HAUPT.EXE
-----------

Abbildung 6-3: Benutzung von LIB

## 6.3 Der Compiler BC.EXE

BC hat die folgende Aufrufsyntax:

```
BC [modulname [, [objectname] [, 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, nimmt der Compiler .BAS an. Geben Sie als Modulnamen USER an, wird die Eingabe direkt von der Tastatur entgegengenommen.

*objectname* 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.

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

---

\* 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.

Das optionale Semikolon verhindert ein Nachfragen des Compilers: Bei fehlendem Namen für Object- oder List-File 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 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 QBX heraus kompiliert. "Nicht automatisch" bedeutet, daß Sie diesen Switch bei QBX das Feld "Additional Options" eintragen müssen, wenn Sie ihn verwenden möchten.

Switch	Bedeutung
/A	Wenn ein <i>listname</i> zusammen mit diesem Switch angegeben wird, erzeugt der Compiler zusätzlich ein Listing des Assembler-Codes, den er produziert. In QBX: Nicht automatisch.
/Ah	"Huge Arrays" - Ermöglicht dynamischen Arrays, größer als 64 K zu sein. Für die Herstellung von Quick Libraries, die mit /Ea in QBX verwendet werden sollen, muß entweder /Ah oder /D beim Kompilieren verwendet werden. In QBX: Wird zwangsläufig gesetzt, wenn QBX mit /Ah aufgerufen wird, sonst nicht automatisch.
/C:b	Setzt den Standard-Kommunikations-Buffer auf <i>b</i> 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 QBX: Wird automatisch eingestellt.
/D	Produziert "debugging"-Code: Mit /D kompilierte Module können jederzeit mit CTRL 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; dafür ist /Zi zuständig). In QBX: Wählbar.
/E	Muß angegeben werden, wenn das Modul die Befehle ON ERROR und RESUME mit Zeilennummer enthält. /E ist eine Untermenge von /X, wird also durch /X erübrigt. In QBX: Wird automatisch eingestellt, wenn erforderlich.
/Es	Teilt Expanded Memory zwischen BASIC und Routinen anderer Sprachen auf (nur für gemischtsprachliches Programmieren). In QBX: Wird automatisch gesetzt, wenn QBX mit /Es aufgerufen wurde, sonst nicht automatisch.

(Fortsetzung nächste Seite)

Switch	Bedeutung
/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 "Vor- und Nachteile der Alternate Math-Library", Kapitel 14.2.) In QBX: Wählbar.
/FBr	Erzeugt Browser-Informationen, die von der PWB ausgewertet werden, nur für globale Definitionen. In QBX: Nicht automatisch.
/FBx	Erzeugt vollständige Browser-Informationen für die PWB. In QBX: Nicht automatisch.
/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 QBX: Wählbar.
/Fs	Modul benutzt <i>Far Strings</i> . Siehe Kapitel 12.3. In QBX: Wählbar.
/G2	Produziert Code, der nur ab 80286-Prozessoren aufwärts funktioniert (das führt zu höherer Effizienz für diese Prozessoren). In QBX: Wählbar.
/Ib:x	Spezifiziert die Anzahl von ISAM-Puffern; 9 sind für vollen Funktionsumfang (6 für den reduzierten PROISAM-Betrieb, siehe Kapitel 7.5 und 7.6) mindestens erforderlich, 512 maximal erlaubt. Das Minimum wird benutzt, wenn der Switch nicht angegeben wird. In QBX: Nicht automatisch.
/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 1,2MB EMS. Ein Wert von -1 verbietet ISAM den EMS-Zugriff. In QBX: 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 QBX: Nicht automatisch.
/LP	Erzeugt ein OS/2-Protected Mode-Object File (das entstehende Programm ist für OS/2); bestimmte Befehle (hauptsächlich Grafik und Sound) werden unbenutzbar. In QBX: Wählbar.
/LR	(Standard) Erzeugt ein Real Mode-Object File für DOS beziehungsweise den OS/2 Real Mode. /LP und /LR schließen sich gegenseitig aus. In QBX: Wählbar.
/MBF	Ersetzt die Funktionen MKS\$, MKD\$, CVS und CVD durch ihre Pendanten mit dem Anhängsel MBF, um Kompatibilität mit Uralt-Datenbanken zu gewährleisten. In QBX: Wird zwangsläufig gesetzt, wenn QBX 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 lauffähig sind. 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 20. In QBX: Wählbar.

(Fortsetzung nächste Seite)

Switch	Bedeutung
/Ot	Optimiert die Geschwindigkeit für SUB-, FUNCTION- und DEF FN-Aufrufe. Wirksam nur, wenn /D und /Fs nicht benutzt werden und wenn eine Prozedur keine FN-Aufrufe, GOSUB-, RETURN- und ON ERROR-Befehle enthält. Lesen Sie dazu den Abschnitt "Funktions- und Prozeduraufrufe" in Kapitel 19.1, "Die Ausführungsgeschwindigkeit eines Programms". In QBX: 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 QBX: Wählbar.
/S	Schreibt Strings direkt in das Object-File und nicht in die Symbol-Tabelle; spart Platz bei Modulen mit vielen String-Konstanten. In QBX: Wählbar.
/T	Unterdrückt die Warnungen, die der Compiler üblicherweise bei weniger schweren Fehlern ausgibt (zum Beispiel "Array not dimensioned" o.ä.) In QBX: Immer gesetzt.
/V	Ermöglicht das Event Trapping; prüft nach jedem Befehl, ob eine der Trap-Bedingungen (Timer, Strig, Play, Pen, Uevent, Signal, Key) auftritt. Siehe bei ON <i>event</i> GOSUB im Referenzteil. In QBX: Automatisch gesetzt, wenn ein ON <i>event</i> GOSUB-Befehl im Porgramm vorkommt.
/W	Wie /V, prüft aber nur an jeder Zeilennummer beziehungsweise an jedem Zeilenlabel (!). /W ist eine Untermenge von /V und wird deshalb bei Einsatz von /V unnötig. In QBX: Nicht möglich, da QBX immer /V setzt, wenn das Programm Event Trapping enthält.
/X	Erlaubt ON ERROR und RESUME NEXT (erweitertes /E). In QBX: Automatisch gesetzt, wenn erforderlich.
/Z	Listet die Fehler während des Kompilierens so, daß der "M"-Editor und die PWB sie richtig verstehen können. In QBX: Nicht automatisch.
/Zd	Erzeugt ein Object-File, das (zur Bearbeitung mit dem Microsoft-Programm SYMDEB) die Zeilennummern des Source-Files enthält. In QBX: Nicht automatisch.
/Zi	Fügt Informationen für den Microsoft CodeView-Debugger in das Object-File ein (Achtung: später nicht mit /E linken). In QBX: Nicht automatisch.

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

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 Kapitel 21) oder ein eigenes Runtime-Modul produzieren (siehe Kapitel 20).

## Beispiele zu BC-Aufrufen

```
BC PROG1 ;
```

PROG1.BAS wird zu PROG1.OBJ kompiliert. Kein Switch wurde angegeben, also wird standardmäßig mit Near Strings und der Emulator-Library (als hätten Sie /FPi angegeben) gearbeitet; wenn Sie in DOS arbeiten, wird ein Object-File für DOS erstellt (wie /LR), sonst eines für den Protected Mode (wie /LP). 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, wenn Sie aus DOS heraus kompilieren, BRT71ENR.EXE: E für Emulator, N für Near Strings, R für Real Mode).

```
BC PROG1 /S/OT ;
```

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

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

Kompiliert PROG1.BAS in PROGRAMM.OBJ, erzeugt dabei ein Programm-listing 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 bringt's zum Laufen

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 QBX 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 dem BASIC PDS 7.1 mitgeliefert 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 linkern.



Während des LINK-Vorgangs werden unter anderem die BASIC-Hilfsroutinen zu Ihrem Programm hinzugefügt; deshalb benötigen Sie zum Linken immer auch eine der Libraries BRT71xyz.LIB oder BCL71xyz.LIB, es sei denn, Sie arbeiten mit einem eigenen Runtime-Modul (siehe Kapitel 20).

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

Die Syntax des LINK-Programms ist folgende:

```
LINK objectname [+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 sozusagen 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 zunächst 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 Modulcode enthalten. Eine Ausnahme davon ist nur die Erstellung von Quick Libraries, bei der es nicht sinnvoll ist, überhaupt Modulcode zu haben, weil eine Quick Library niemals ausgeführt wird (siehe Switch /Q).

Jedes angegebene OBJ-File wird vollständig eingebunden, egal ob es 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

---

\* Falls doch, finden Sie im Kapitel 21 die Beschreibung, wie Sie eine Liste aller in einer Library enthaltenen Routinen ausgeben lassen können.

Extension .LIB. Sie werden ebenfalls 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 (BCL71xyz.LIB oder BRT71xyz.LIB); diesen Standard-Namen, der abhängig von den Switches /FPi oder /FPa, /Fs, /LR oder /LP ist, die Sie beim Kompilieren angeben, notiert aber schon der Compiler in der OBJ-Datei, so daß LINK automatisch die richtige Library verwendet, ohne daß Sie sie explizit angeben müssen.

Sie werden das Library-Feld zum Beispiel benutzen, wenn Sie ein Programm erstellen, das eine der Toolboxes oder Add-On-Libraries benötigt. In der Beschreibung der einzelnen Toolboxes steht genau, welche Libraries Sie zu erwähnen haben, wenn Sie die jeweilige Toolbox verwenden.

Nochmal im Klartext: Sie müssen für gewöhnliche Programme in diesem Feld überhaupt nichts angeben; erst bei der Verwendung von Add-On-Libraries und Toolboxes (oder Interruptaufrufen) wird das nötig.

Das fünfte Feld auf der Befehlszeile, *definition*, gibt den Namen einer speziellen Definitionsdatei an, die für OS/2-Applikationen einige betriebssystemspezifische Details regelt. Die Definitionsdatei zu beschreiben, würde den Rahmen dieses Kapitels sprengen. Konsultieren Sie ein LINK-Handbuch, wenn Sie Näheres über die Definitionsdatei wissen wollen. Sie können auch ohne die Definitionsdatei problemlos Programme für OS/2 erstellen.

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 Object-Dateien 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 kompiliert wurde).
/E	"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. Benutzen Sie diese Switches nicht, wenn Sie mit Overlays arbeiten. 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" - Ruft das Programm QH.EXE auf und aktiviert das Hilfesystem für LINK. Wenn Sie nur eine Liste der Switches sehen möchten, benutzen Sie stattdessen den Switch /?.
/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.

*(Fortsetzung nächste Seite)*

Switch	Bedeutung
/M	"Map" - Schließt in die Listendatei, falls sie überhaupt erzeugt wird, eine sortierte Liste aller globalen Symbole ein. Standard für das Feld <i>listname</i> ist dann <i>exename</i> mit angehängtem .MAP (und nicht länger NUL).
/NOD	"No default library search" - Wie oben 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 "... use /NOE" aus), und zwar dann, wenn dieselbe Routine in mehreren angegebenen Libraries vorhanden ist. Wenn Sie zum Beispiel sowohl die User Interface- als auch die Presentation Graphics-Toolbox im gleichen Programm benutzen (beide Libraries werden beim Linken angegeben), erhalten Sie die Meldung, daß INTERRUPT und INTERRUPTX mehrfach definiert sind. Benutzen Sie dann /NOE.
/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 <i>nr</i> 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).
/PAU	Legt eine Pause ein und wartet auf einen Tastendruck, bevor das EXE-File erzeugt wird (für eventuellen Diskettenwechsel).
/PM:typ	"Presentation Manager" - Gibt (nur für OS/2-Programmierung) den Typ der Anwendung an; für BASIC-Programme gibt es nur zwei Möglichkeiten: NOVIO (Standard) bedeutet, daß die Anwendung nicht Presentation Manager-kompatibel ist; VIO heißt, daß sie es ist. Sie können VIO verwenden, wenn Ihr Programm weder Grafik noch Event-Handling (ON <i>event</i> GOSUB) benutzt. Ansonsten ist die Einstellung NOVIO erforderlich.

(Fortsetzung nächste Seite)

Switch	Bedeutung
/Q	"Quick Library" - Mit dem Switch /Q erzeugt LIB kein EXE-Programm, sondern eine Quick Library. Für die Herstellung von Quick Libraries ist es entgegen dem für EXE-Programme Gesagten <i>nicht</i> erforderlich (und sogar unsinnig), daß das erste der genannten OBJ-Files Modulcode enthält. Sie müssen, wenn Sie eine Quick Library erstellen wollen, im Library-Feld die Library QBXQLB.LIB mit angeben. /Q sollte nicht zusammen mit anderen Switches verwendet werden, die auf das zu erzeugende EXE-File einwirken (wie /LI, /F/PACKC oder /E).
/SE:x	"Segments" - Setzt die maximal erlaubte Anzahl von Segmenten für das Programm. Der Standard ist 128, und wenn Sie beim Linken die Meldung <i>Too many segments</i> erhalten, sollten Sie /SE mit einem x größer 128 angeben, bis es klappt. Für kleine Programme können Sie die Segmentzahl auch herabsetzen. Dies hat jedoch keinerlei Auswirkung auf das entstehende Programm, sondern höchstens auf die Geschwindigkeit, mit der LINK arbeitet. Das Maximum für x ist 3072.

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 19.

Eine Übersicht über Fehlermeldungen, von denen LINK eine ganze Reihe parat hat, finden Sie in Anhang C.3.

## 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 Object-Files, 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. Für OS/2 sollten Sie eine andere Extension benutzen, denn CMD ist dort für Batch-Prozeduren vorgesehen.

Für das zweite und dritte Feld, die in der LINK-Zeile ja leere Felder sind, weil zwischen den Kommata nichts steht, werden in der Steuerungsdatei leere Zeilen eingesetzt. 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.

Weitere Informationen über den LINK-Prozeß finden Sie in den Kapiteln über Optimierung (19), Runtime-Module (20), LIB (21) und Overlays (13.2).

---

## **Sektion III      Neue Sprachelemente - neue Möglichkeiten**

---

- **Das ISAM-Datenbanksystem**
  - **Add-On-Libraries**
  - **Toolboxen**
-





# 7 Das ISAM-Datenbanksystem

In den ersten Abschnitten dieses Kapitels wird kurz auf das ISAM-Prinzip eingegangen und dann die Anwendung erläutert. Die einzelnen ISAM-Befehle und Funktionen werden dabei nur im Kontext ihrer Aufgaben und Möglichkeiten erwähnt. Eine detaillierte Beschreibung ist in der ISAM-Befehlsreferenz am Ende des Buches enthalten. Gegen Ende des Kapitels werden die technischen ISAM-Interna und -Utilities diskutiert, soweit sie für den BASIC PDS-Programmierer von Nutzen sind.

Auf ISAM wird außerdem in den Anhängen A und C kurz eingegangen, in denen es um Limits beziehungsweise Fehlermeldungen geht.

## 7.1 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".

BASIC 7.1 PDS enthält viele neue Befehle, die es ermöglichen, Daten mit der ISAM-Methode 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.

## 7.2 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 Eingabe-Reihenfolge. Die durch den Index determinierte Reihenfolge kann sich innerhalb von Sekundenbruchteilen völlig verändern, indem einfach auf einen anderen Index umgeschaltet wird.

Die eigentlichen (unsortierten) Daten bilden zusammen mit der Typenbeschreibung der Daten eine Tabelle ("Table"), 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 bei 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.

### ISAM-Begriffe

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* (die *Datentabelle*) 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*.

Um Verwirrungen zu vermeiden: In den englischen Handbüchern heißt das, was ich ISAM-Datenfile nenne, *database*, und meine Datenbanken sind dort nur *tables*. Ich halte meine Begriffe jedoch für zutreffender und habe deshalb diese Umdefinition vorgenommen.

## 7.3 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, also eine Datei mit einer Datenbank, die nur einen Datensatz und überhaupt keinen Index enthält, belegt bereits 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, eine Datenbank 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.

## 7.4 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 zunächst 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 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 ISAM-File 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 innerhalb des angegebenen Files genannt werden, die benutzt wird. 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 ("current record") 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, Doubletten 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 annimmt, dann dürfen keine zwei Datensätze aus der Datenbank im angegebenen Feld denselben Inhalt haben. Ist "universell" hingegen Null, so kommt es darauf nicht an.

Um einen Ortsindex für die Adressendatei anzulegen, könnte man also schreiben:

```
CREATEINDEX 1, "Ortsindex", 0, "PLZOrt"
```

"Universell" muß Null 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", 1, Nachname, Vorname
```

Hier wird primär nach dem Familiennamen sortiert, bei gleichen Familiennamen aber noch der Vorname herangezogen. In diesem Beispiel könnte "universell" schon eher auf 1 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 einem professionellen Programm jedoch darauf verzichten). Solche kombinierten Indizes sind dadurch beschränkt, daß die Summe der Längen der angegebenen Felder 255 nicht übersteigen darf.

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 ersten Datensatz nach der neu festgelegten Sortierung.

# 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, "5300 Bonn 1"
```

würde also - ausgehend vom aktuellen Index - den ersten Datensatz suchen, dessen Ortsangabe exakt "5300 Bonn 1" 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"
```

## 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.4); 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 ist auch sehr kulant, was die "Gleichheit" anbetrifft: "Aarßbürgen" ist zum Beispiel identisch mit "AARSSBÜRGEN".

Diese Gleichheit kann unter Umständen zu Schwierigkeiten mit dem "universell"-Parameter des CREATEINDEX-Befehls (siehe oben) führen.

TEXTCOMP vergleicht generell nur die ersten 255 Zeichen eines Strings, aber im ISAM-Referenzteil finden Sie eine Routine, die Sie auch diese Hürde überbrücken 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 Interesse.

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 Transaktions-Mitschrift 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 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 (keine Fenstertechnik, keine Soundeffekte, keine Farbeinstellung...).

```
' ISAMDEMO.BAS - Adreßverwaltung zur Demonstration der
' ISAM-Fähigkeiten

DECLARE SUB AnzeigeDatensatz (Modus AS INTEGER)
DECLARE SUB DatenbankOeffnen ()
DECLARE SUB Datenliste ()
DECLARE SUB DatensatzEingabe (Vorgabe AS ANY, NeuerSatz AS ANY)
DECLARE SUB HauptMenue ()
DECLARE SUB SucheDatensatz (Titel AS STRING)

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
Leer.Strasse = "": Leer.PLZ = ""      ' die "leere Adresse" lauter
Leer.Ort = "": Leer.Telefon = ""      ' CHR$(0)- Zeichen, und
                                     ' RTRIM würde nicht
                                     ' funktionieren. Hierdurch
                                     ' wird die "leere Adresse"
                                     ' mit CHR$(32), also Leer-
                                     ' zeichen gefüllt.

DatenbankOeffnen                      ' Eingangsmeldung anzeigen &
                                     ' Datenbank als Datei #1
                                     ' öffnen
```

(Fortsetzung nächste Seite)

*(Fortsetzung)*

```
HauptMenue                                ' Hauptmenü

PRINT "Sollen die Änderungen, die"
PRINT "Sie gemacht haben, gespei-"
PRINT "chert 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 ge-
                                           ' macht werden müssen: ROLLBACK ALL
                                           ' stellt den Status zum Zeitpunkt
                                           ' des BEGINTRANS-Befehls wieder her
COMMITTRANS: CLOSE                        ' COMMITTRANS wäre eigentlich gar
                                           ' nicht notwendig, da CLOSE es au-
                                           ' tomatisch ausführt. Nach COMMIT-
                                           ' TRANS kann mit ROLLBACK nichts
                                           ' mehr zurückgenommen werden.

SYSTEM                                    ' Programmende
```

```
' Diese Prozedur zeigt den Datensatz, der in der globalen Variable
' 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
```

*(Fortsetzung nächste Seite)*

```
' Diese Prozedur öffnet die ISAM-Datenbank (gemäß den Konstanten im Modul-
' code). Außerdem werden die beiden benötigten Indexlisten erstellt.
'
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 ist ja egal.
    CREATEINDEX #1, "NameVorname", 1, "Name", "Vorname"
    CREATEINDEX #1, "PLZOrt", 0, "PLZ", "Ort"
ELSE
    'Programm abbrechen.
    IF ERR = 73 THEN
        PRINT "ISAM ist nicht geladen."
    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
' Postleitzahlgebietes (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"
```

(Fortsetzung nächste Seite)

(Fortsetzung)

```
PRINT a$
PRINT

SELECT CASE a$
CASE "A"
    SETINDEX 1, "NameVorname"
    MOVEFIRST 1
    DO UNTIL EOF(1)
        RETRIEVE 1, Zugriff
        AnzeigeDatensatz Kurz
        MOVENEXT 1
    LOOP
CASE "P"
    PRINT "Von PLZ (ganze PLZ oder";
    PRINT " Anfangsziffern; 0 = ";
    PRINT " von Anfang): ";
    LINE INPUT VonPLZ
    PRINT "Bis PLZ (ganze PLZ oder";
    PRINT " Anfangsziffern; 0 = ";
    PRINT "bis Ende):    ";
    LINE INPUT BisPLZ
    SETINDEX 1, "PLZOrt"
    IF VonPLZ = "0" THEN
        MOVEFIRST 1
    ELSE
        SEEKGE 1, VonPLZ
    END IF
    DO UNTIL EOF(1)
        RETRIEVE 1, Zugriff
        IF BisPLZ <> "0" THEN
            IF TEXTCOMP(LEFT$(Zugriff.PLZ, LEN(BisPLZ)), BisPLZ) = 1 THEN EXIT DO
        END IF
        AnzeigeDatensatz Kurz
        MOVENEXT 1
    LOOP
END SELECT
END SUB
```

' LISTE NACH NAMEN SORTIERT:  
' Indexliste mit Sortierung nach  
' Namen anwählen  
' Zeiger auf ersten Datensatz nach  
' neuer Sortierung setzen (eigent-  
' lich hier überflüssig, weil von  
' SETINDEX automatisch ausgeführt)  
' Solange, bis der Zeiger hinter  
' letzten Datensatz zeigt:  
' Datensatz, auf den der Zeiger  
' zeigt, in "Zugriff" lesen,  
' ...anzeigen...  
' und Zeiger weiterstellen.

' LISTE NACH PLZ sortiert:  
'  
' Index für PLZ-Sortierung setzen  
' Wenn "Von Anfang" gewählt, dann  
' Zeiger auf erstes Element setzen,  
' sonst auf das erste Element, das  
' >= der Anfangs-PLZ ist  
'  
' Solange, bis der Zeiger hinter den  
' letzten Datensatz zeigt:  
' Datensatz, auf den der Zeiger  
' zeigt in "Zugriff" einlesen  
' wenn nicht "bis Ende" gewählt wurde:  
' prüfen, ob die PLZ des gelesenen  
' Datensatzes schon außerhalb des  
' Bereiches liegt; wenn ja, Schleife  
' verlassen.  
' Datensatz anzeigen  
' Zeiger weiterstellen

(Fortsetzung nächste Seite)

*(Fortsetzung)*

```
' Diese Prozedur lässt den Benutzer - unkomfortabel zwar, aber ausreichend -  
' einen Datensatz eingeben. In der Variable Vorgabe steht, was eingesetzt  
' wird, wenn der Benutzer einfach ENTER drückt. Die Eingaben werden in die  
' Variable NeuerSatz geschrieben. Wenn NeuerSatz schon Daten enthält, wer-  
' den 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  
    END IF  
    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  
    END IF  
    LINE INPUT "Vorname:        "; temp$  
    IF temp$ = "" THEN temp$ = Vorgabe.Vorname  
    NeuerSatz.Vorname = temp$  
  LOOP  
  
  DO UNTIL LEN(RTRIM$(NeuerSatz.Strasse))  
    IF LEN(RTRIM$(Vorgabe.Strasse)) THEN  
      PRINT "(Vorgabe)          "; Vorgabe.Strasse  
    END IF  
    LINE INPUT "Strasse:         "; temp$  
    IF temp$ = "" THEN temp$ = Vorgabe.Strasse  
    NeuerSatz.Strasse = temp$  
  LOOP  
  
  DO UNTIL LEN(RTRIM$(NeuerSatz.PLZ))  
    IF LEN(RTRIM$(Vorgabe.PLZ)) THEN  
      PRINT "(Vorgabe)          "; Vorgabe.PLZ  
    END IF  
    LINE INPUT "PLZ:           "; temp$  
    IF temp$ = "" THEN temp$ = Vorgabe.PLZ  
    NeuerSatz.PLZ = temp$  
  LOOP  
  
  DO UNTIL LEN(RTRIM$(NeuerSatz.Ort))  
    IF LEN(RTRIM$(Vorgabe.Ort)) THEN  
      PRINT "(Vorgabe)          "; Vorgabe.Ort  
    END IF
```

*(Fortsetzung nächste Seite)*

(Fortsetzung)

```
LINE INPUT "Ort:           "; temp$
IF temp$ = "" THEN temp$ = Vorgabe.Ort
NeuerSatz.Ort = temp$
LOOP

DO UNTIL LEN(RTRIM$(NeuerSatz.Telefon))
IF LEN(RTRIM$(Vorgabe.Telefon)) THEN
PRINT "(Vorgabe)           "; Vorgabe.Telefon
END IF
LINE INPUT "Telefon:       "; temp$
IF temp$ = "" THEN temp$ = Vorgabe.Telefon
NeuerSatz.Telefon = temp$
LOOP

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
```

(Fortsetzung nächste Seite)

```
CASE 2
  SucheDatensatz "Welchen Datensatz löschen?"
  IF NOT EOF(1) THEN
    ' Wenn der Zeiger nicht hinter dem
    ' letzten Datensatz steht,
    x% = SAVEPOINT
    ' SAVEPOINT-Markierung setzen
    DELETE #1
    ' Datensatz, auf den der 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
    Zugriff = Leer
    ' für Neueingabe:
    DatensatzEingabe Vorgabe, Zugriff
    x% = SAVEPOINT
    ' SAVEPOINT-Markierung setzen
    UPDATE 1, Zugriff
    ' Datensatz, auf den der 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
    ' Den Zustand der Datenbank zum
    ' Zeitpunkt der letzten SAVEPOINT-
    ' Markierung wiederherstellen
CASE 9
  EXIT DO
    ' Menü verlassen
CASE ELSE
  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 "den 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.

(Fortsetzung)

```

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$)
            CASE 1
                ' SUCHEN NACH NAMEN
                SETINDEX 1, "NameVorname" ' Namensindex aktivieren
                PRINT "Nachname (oder Anfangsbuchstaben)";
                PRINT "staben dessen): ";
                LINE INPUT NachName
                SEEKGE 1, NachName ' Zeiger auf ersten Datensatz
                ' setzen, dessen Nachname >= dem
                ' eingegebenen Namen ist
            DO UNTIL EOF(1) ' Solange bis der Datenzeiger hinter
                ' dem letzten Datensatz steht:
                RETRIEVE 1, Zugriff ' Datensatz, auf den der Datenzeiger
                ' zeigt, in Zugriff einlesen
                IF TEXTCOMP(LEFT$(Zugriff.Name, LEN(NachName)), NachName) THEN
                    ' Wenn der eingelesene Name schon
                    ' zu weit hinten im Alphabet steht
                    ' (wenn die ersten Buchstaben nicht
                    ' mehr mit den eingegebenen 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 und
                ' weitersuchen.
        LOOP
        IF a$ = "J" THEN
            EXIT SUB ' Bei JA: Prozedur verlassen
        END IF
    END SUB

```

(Fortsetzung nächste Seite)



```
ELSE
    MOVELAST 1: MOVENEXT 1                ' Bei ABBRUCH oder wenn keine DS
                                           ' mehr gefunden wurden, Zeiger hin-
                                           ' ter letzten DS stellen.

    IF NOT a$ = "A" THEN
        PRINT "** Keine weiteren Datensätze gefunden."
    END IF

END IF

CASE 2
    SETINDEX 1, "PLZOrt"                  ' SUCHEN NACH PLZ
    PRINT "Postleitzahl (oder erste ";    ' (gleiches Prinzip wie oben)
    PRINT "Ziffern davon): "
    LINE INPUT Postleitzahl
    SEEKGE 1, Postleitzahl
    DO UNTIL EOF(1)
        RETRIEVE 1, Zugriff
        IF TEXTCOMP(LEFT$(Zugriff.PLZ, LEN(Postleitzahl)), Postleitzahl) THEN
            EXIT DO
        END IF
        PRINT RTRIM$(Zugriff.Vorname); " "; RTRIM$(Zugriff.Name); ", "
        PRINT "      "; RTRIM$(Zugriff.PLZ); " "; RTRIM$(Zugriff.Ort);
        PRINT TAB(60); "OK (j/n/a)? ";
    DO
        DO: a$ = UCASE$(INKEY$): LOOP UNTIL LEN(a$)
        LOOP UNTIL INSTR("JNA", a$)
        PRINT a$
        IF a$ = "J" OR a$ = "A" THEN EXIT DO
        MOVENEXT 1
    LOOP
    IF a$ = "J" THEN
        EXIT SUB
    ELSE
        MOVELAST 1: MOVENEXT 1
        IF NOT a$ = "A" THEN
            PRINT "** Keine weiteren Datensätze gefunden."
        END IF
    END IF
END SELECT

END SUB
```

Listing 7-1: ISAMDEMO.BAS

## 7.5 ISAM und QBX

Damit ISAM von Programmen aus benutzt werden kann, die in der Entwicklungsumgebung QBX laufen, muß vor dem Aufruf von QBX ein speicherresistentes Programm geladen werden, das die ISAM-Routinen zur Verfügung stellt.

Zwei Programme werden hierzu mitgeliefert: PROSIAM.EXE und PROISAMD.EXE. Das kürzere PROSIAM.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 PROSIAM.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 PROSIAM.EXE lautet so (PROISAMD entsprechend):

```
PROSIAM [/Ib:puffer] [/Ie:emsrest] [/Ii:indexanzahl]  
PROSIAM /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 6.3) 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 PROSIAM 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:  $puffer = 1 + \text{maximale Anzahl gleichzeitig geöffneter Datenbanken} + \text{Anzahl der Indizes, die das Programm benutzt (Null-Index nicht eingerechnet)}$ . 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 PROSIAM beziehungsweise 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* kann benutzt werden, um ISAM vorzuschreiben, daß eine bestimmte Menge von EMS frei bleiben soll. *emsrest* wird in KB angegeben und legt damit fest, wieviel EMS ISAM nicht belegen darf. Üblicherweise, wenn Sie den /Ie-Switch weglassen, darf ISAM das EMS unbegrenzt benutzen (es benötigt aller-

dings 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.

## 7.6 ISAM in kompilierten Programmen

Wenn Sie EXE-Programme erstellen, die ISAM-Funktionen benutzen, müssen die ISAM-Routinen auf irgendeine Weise diesen Programmen verfügbar gemacht werden. Wie, das wird schon beim Setup, also bei der Installation des Compilers auf der Festplatte, festgelegt. Sie müssen dort bestimmen, ob die Routinen in die Runtime-Module und in die Compiler-Libraries eingebunden werden sollen oder nicht. Im ersten Fall werden die Programme beziehungsweise die Runtime-Module signifikant länger; im zweiten Fall wird es erforderlich, daß vor dem Start Ihrer Programme PROISAM.EXE oder PROISAMD.EXE wie im vorigen Abschnitt beschrieben geladen werden müssen.

Beim Kompilieren eines Programms, das die ISAM-Routinen selbst enthalten oder aus den Runtime-Modulen entnehmen wird, das also PROISAM.EXE nicht benötigt, können Sie für den Compiler BC.EXE dieselben Switches angeben, die im vorigen Abschnitt als Switches zu PROISAM.EXE behandelt wurden.

Weitere Informationen entnehmen Sie bitte dem Abschnitt 3, "ISAM-Unterstützung", des Kapitels 2, "Die Installation des BASIC PDS".

## 7.7 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 bitte folgendes: Sie können maximal vier ISAM-Dateien gleichzeitig geöffnet haben, und die Anzahl der gleichzeitig geöffneten Datenbanken ist  $1 + 3 * (5 - n)$ , 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.

## 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.

## 7.8 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.

Die vier Programme werden erst bei der Installation erstellt. Wenn Sie im SETUP-Programm entscheiden, daß Ihre Programme die ISAM-Routinen aus dem speicherresidenten Programm benutzen sollen, gilt das auch für ISAMIO, ISAMCVT und ISAMPACK. PROISAMD.EXE muß dann zuvor geladen werden.

### 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  
        [satzbeschreibungdatei]
```

*modus* ist entweder M (für MS/ISAM), D (für dBase) oder B (für Btrieve). Beim Konvertieren von Btrieve-Dateien muß hinter *isamdateiname* noch eine *satzbeschreibungdatei* 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.

## Btrieve-Konvertierung

Für die Konvertierung von Btrieve-Dateien muß das speicherresidente Btrieve-Programm geladen sein.

*satzbeschreibungdatei* 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 Felds 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 *satzbeschreibungdatei* 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.

## dBase-Konvertierung

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**

Textfeld mit Länge *n* oder Memo

Logisch

Zahl ohne Dezimalstellen

Zahl mit Dezimalstellen oder Datum

## **BASIC-Datentyp**

STRING \* *n*

INTEGER

INTEGER oder LONG (je nach Bereich)

DOUBLE (Datum wird zu Zeitcode, siehe Kapitel 8.2)

Für Indizes gilt das, was auch schon zu Btrieve-Dateien gesagt wurde.

## **MS/ISAM-Konvertierung**

MS/ISAM ist, wie gesagt, eine alte und wenig verbreitete Form des ISAM-Systems, sozusagen ein entfernter Vorläufer des heutigen ISAM. Für die Konvertierung 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 auf der Festplatte, 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 beziehungsweise aus der die Datensätze importiert werden sollen.

## **Import**

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, gröÙe, 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.

```
TYPE ZugriffType
    KundenNummer AS LONG
    BestellMenge AS INTEGER
    ArtikelNummer AS DOUBLE
    Preis AS CURRENCY
    Kommentar AS STRING * 50
    KundenInfo(0 TO 9) AS INTEGER
END TYPE
```

Zum vorher abgebildeten ISAM-Datentyp "ZugriffType" gehört die folgende Satzbeschreibungsdatei:

```
LONG,,KundenNummer
INTEGER,,BestellMenge
REAL,,ArtikelNummer
CURRENCY,,Preis
VS,50,Kommentar
BINARY,,KundenInfo
```

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,Allertz
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 4x4 LONG-Zahlen müßte also als 64-Byte-String (eine LONG-Zahl braucht 4 Bytes Speicherplatz) in der Datei stehen. Ein solcher String ließe sich zwar mit der Funktion MKL\$ erzeugen, doch ist es nicht unwahrscheinlich, daß er dann Anführungszeichen oder Dateende-Kennzeichen als Bestandteil der Codierung enthielte, die ISAMIO ziemlich durcheinanderbringen würden.

## Export

Der Export von Daten aus einer ISAM-Datei in ein ASCII-File ist wesentlich problemloser. Der Befehl lautet einfach:

```
ISAMIO /E asciifile isamfile datenbank
```

wobei es erlaubt ist, noch eine Satzbeschreibungsdatei anzugeben. In diesem Fall wird die angegebene Satzbeschreibungsdatei aufgrund der exportierten Daten neu erzeugt.

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 "Variabletext" exportiert, das heißt, daß ein Array von 10 INTEGER-Zahlen als 20-Byte-Codestring exportiert wird und nicht als 10 Zahlen.

# ISAMREPR

Das Programm ISAMREPR ist in der Lage, eine durch Diskettenfehler, "Abstürze" oder 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 *Database needs repair* 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 Datensatz 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*.

## 8 Add-On-Libraries

Die drei Add-On-Libraries (eigentlich nur zwei, denn die Format- und Date-Routinen stehen in denselben Dateien) sind, im Gegensatz zu den Toolboxen, nicht in BASIC programmiert. Ihre Source-Codes werden nicht mitgeliefert, und deshalb können Sie auch an den Add-On-Libraries keine Änderungen vornehmen.

Diese Libraries haben einen wesentlich höheren Rang als die Toolboxen, denn sie sind ausgiebig getestet. Die Wahrscheinlichkeit, daß eine der Add-On-Library-Funktionen nicht das tut, was sie soll, ist sehr gering, während sich in die Toolboxen eine ganze Anzahl von offensichtlichen Fehlern und Nachlässigkeiten eingeschlichen haben.

Wenn Sie Routinen aus einer Add-On-Library benutzen, müssen Sie einerseits entsprechende DECLARE-Anweisungen in Ihr Programm schreiben oder eine geeignete Include-Datei laden und andererseits beim Linken und beim Aufruf von QBX den Namen der zu verwendenden Library-Datei angeben. Sie können diesen Namen den Tabellen in den folgenden Sektionen entnehmen.

BASIC 7.1 PDS stellt drei Add-On-Libraries zur Verfügung, die ich im folgenden beschreibe:

- Die Format-Add-On-Library
- Die Date-Add-On-Library
- Die Finance-Add-On-Library

# 8.1 Die Format-Add-On-Library

## Inhalt der Library

Diese Library enthält einige schnelle Routinen zur Formatierung von numerischen Daten in Strings, also etwas komfortablere Varianten der STR\$-Funktion. Die Format-Routine für DOUBLE-Zahlen kann außerdem benutzt werden, um Zeitcodes (siehe "Was sind Zeitcodes?" im Abschnitt 8.2) auf einfache Weise als Datum und/oder Uhrzeit darzustellen.

Für jeden numerischen Datentyp ist eine Format-Routine in der Library enthalten. Die sechste und letzte Routine heißt SetFormatCC und dient zur landesspezifischen Einstellung der Dezimaltrennzeichen (siehe Format-Referenzteil).

## Einbinden der Routinen

Die Format-Routinen sind unveränderbar in den mitgelieferten Libraries enthalten. Zu verwenden ist die DTFMT-Library (DTFMT für Date & Format), die in maximal vier LIB- und einer QLB-Version daherkommt; die letzten beiden Buchstaben des Librarynamens geben Mathematik-Library und Prozessormodus an:

### Wenn Sie so kompilieren...

```
BC PROGRAMM /FPi /LR
BC PROGRAMM /FPa /LR
BC PROGRAMM /FPi /LP
BC PROGRAMM /FPa /LP
```

### ...benötigen Sie diese Library:

```
DTFMTER.LIB
DTFM TAR.LIB
DTFMTEP.LIB
DTFM TAP.LIB
```

### für QBX:

```
QBX PROGRAMM /L DTFMTER          DTFMTER.QLB
```

Wie Sie sehen, sind die benötigten Libraries dieselben, die auch für die Date-Add-On-Library benutzt werden. In der Tat sind beide Sammlungen eigentlich eine einzige Add-On-Library; nur die Include-Dateien sind nicht identisch. Zur Benutzung der Format-Routinen müssen Sie die Include-Datei FORMAT.BI mit REM \$INCLUDE: 'FORMAT.BI' in Ihr Programm einbinden.

## Beispiele für numerische Formatierungen

Die Beispiele gehen davon aus, daß mit SetFormatCC 49 die deutsche "Zeichensetzung" gewählt wurde.

```
PRINT FormatS$(3.798, "###,##")
      3,8
PRINT FormatS$(3.798, "000,00")
      003,80

PRINT FormatS$(3798, "0,000 E-##")
      3,798 E3
PRINT FormatS$(3798, "0,000 e-00")
      3,798 e03

PRINT FormatS$(3798, "0,000 E+00")
      3,798 E+03

PRINT FormatC$(35.46, "\D\M ###,##")
      DM 35,46

PRINT FormatS$(.12, "##,##")
      ,12
PRINT FormatS$(.12, "#0,##")
      0,12

PRINT FormatS$(10, "000,00 \H;000,00 \S;\±\0")
      010,00 H
PRINT FormatS$(-10, "000,00 \H;000,00 \S;\±\0")
      010,00 S
PRINT FormatS$(0, "000,00 \H;000,00 \S;\±\0")
      ±0
```

## Beispiele für Zeitcode-Formatierungen

```
PRINT FormatD$(33259.445#, "d\.m\.yyyy hh\.mm \U\h\r")
      21.1.1991 10.40 Uhr
PRINT FormatD$(33259.445#, "dd\.mm\.yy hh\.mm \U\h\r")
      21.01.91 10.40 Uhr
PRINT FormatD$(33259.445#, "dd\.mm\.yy hh:mm\.ss \U\h\r")
      21.01.91 10:40.48 Uhr
PRINT FormatD$(33259.445#, "dddd\, mmmm d\, yyyy")
      Monday, January 21, 1991
PRINT FormatD$(33259.445#, "hh:mm am/pm")
      10:40 am
```

## 8.2 Die Date-Add-On-Library

### Inhalt der Library

Die Datums-Library enthält Routinen für die Arbeit mit Zeitcodes (*date/time serial numbers*). Zeitcodes sind Zahlen doppelter Genauigkeit (DOUBLE), die einen Zeitpunkt repräsentieren. Die Routinen aus der Library ermöglichen es, Zeit- und Datumsangaben in Zeitcodes umzurechnen und umgekehrt.

### Was sind Zeitcodes?

Zeitcodes 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 nun 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 ganzzahligen 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#
```

Es empfiehlt sich, für Zeitcodes konsequent den Typenbezeichner # zu verwenden, da es sich ja um Zahlen doppelter Genauigkeit handelt.

Ebenso kann durch Addition aus reinem Datums- und reinem Uhrzeit-Zeitcode wieder ein kompletter Zeitcode gewonnen werden.

# Einbinden der Routinen

Die Date-Routinen sind als fertige Libraries auf den Disketten. Da es sich um eine Add-On-Library handelt, werden keine Source-Codes mitgeliefert.

## **Wenn Sie so kompilieren...**

BC PROGRAMM /FPi /LR

BC PROGRAMM /FPa /LR

BC PROGRAMM /FPi /LP

BC PROGRAMM /FPa /LP

## **...benötigen Sie diese Library:**

DTFMTER.LIB

DTFMTER.LIB

DTFMTER.LIB

DTFMTER.LIB

für QBX:

QBX PROGRAMM /L DTFMTER

DTFMTER.QLB

Die Include-Datei DATIM.BI enthält die benötigten DECLARE-Zeilen. Die genannten Library-Namen sind identisch mit denen der Format-Add-On-Library, da sich beide Libraries in denselben Dateien befinden.

## 8.3 Die Finance-Add-On-Library

### Inhalt der Library

Die Library enthält insgesamt 13 Funktionen hauptsächlich aus der Zinsrechnung und verwandten Gebieten. Dazu zählen Abschreibung, zukünftiger Wert, interner Zinsfuß, Gegenwartswert zukünftiger Zahlungen, Zinssatz und andere. Auch wenn Sie mit den Fachbegriffen nichts anzufangen wissen, sollten die Beispiele zu den meisten Funktionen im Referenzteil Ihnen nützliche Hinweise geben.

Die genaue Beschreibung finden Sie - wie üblich - im Referenzteil.

### Einbinden der Routinen

Die Finance-Routinen sind nicht in BASIC programmiert. Die Source-Codes werden nicht mitgeliefert, sondern nur die fertigen Libraries. Die folgende Tabelle zeigt, welche Library Sie einbinden müssen, abhängig davon, mit welcher Coprozessor-Option sie kompilieren und ob das Programm unter DOS oder OS/2 laufen soll.

#### **Wenn Sie so kompilieren...**

BC PROGRAMM /FPi /LR  
BC PROGRAMM /FPa /LR  
BC PROGRAMM /FPi /LP  
BC PROGRAMM /FPa /LP

#### **...benötigen Sie diese Library:**

FINANCER.LIB  
FINANCAR.LIB  
FINANCEP.LIB  
FINANCAP.LIB

#### **für QBX:**

QBX PROGRAMM /L FINANCER      FINANCER.QLB

Die Anwendung der Finance-Routinen erfordert außerdem die Einbindung der Include-Datei FINANC.BI, die die DECLARE-Anweisungen für die verwendeten Funktionen enthält.



# 9 Toolboxes

Die Toolboxes will Microsoft nicht als fertiges System verstanden wissen, sondern vielmehr als Basis für eigene Weiterentwicklungen. Nichtsdestotrotz sind die Toolboxes durchdacht genug, um fast ohne Änderungen (einige Fehler sind hier mit Korrekturanleitung abgedruckt) direkt in eigene Programme eingebunden werden zu können.

Der erfreuliche Vorteil, den diese Toolboxes gegenüber den Add-On-Libraries haben, ist der, daß hier der Source-Code mitgeliefert wird, daß Sie also an allen Routinen nach Herzenlust herumbasteln und sie an Ihre eigenen Bedürfnisse anpassen können.

In welchen Library-Dateien die Toolbox-Routinen jeweils enthalten sind, welche BAS- und Include-Files dazugehören und wie Sie die Libraries neu erstellen, wenn Sie Änderungen gemacht haben, beschreiben die folgenden Abschnitte im Detail.

Die Funktionsweise der Toolboxes wird in diesem Buch nur so weit erklärt, daß Sie sie in vollem Umfang *benutzen* können. Wenn Sie Änderungen machen möchten, müssen Sie sich unter Umständen etwas tiefer in die Materie einarbeiten, was jedoch aufgrund der gut dokumentierten Quelltexte keine Schwierigkeit darstellen sollte.

Folgende Toolboxes sind mitgeliefert:

- Die Matrizenmathematik-Toolbox
- Die Font-Toolbox
- Die Presentation Graphics-Toolbox
- Die General-Toolbox
- Die User Interface-Toolbox

# 9.1 Die Matrizenmathematik-Toolbox

## Inhalt der Toolbox

Berechnungen mit Matrizen sind elementarer Bestandteil vieler mathematischer Operationen. Das Lösen von linearen Gleichungssystemen ist nur ein Beispiel dafür. Die Matrizenmathematik-Toolbox enthält Routinen zur Addition, Subtraktion und Multiplikation zweier Matrizen, zum Ermitteln der Determinante, zum Invertieren einer quadratischen Matrix und schließlich auch zum Lösen linearer Gleichungssysteme.

## Einbinden der Routinen

Die Matrizenmathematik-Toolbox ist im Source-Code-File MATB.BAS gespeichert. Um sie zu benutzen, müssen Sie in Ihrem Programm die Include-Datei MATB.BI angeben und die adäquate Library benutzen. Die Libraries heißen alle MATB gefolgt von drei Buchstaben, die die Mathematik-Library, den Prozessormodus und die Stringspeichermethode abkürzen:

### Wenn Sie so kompilieren...

```
BC PROGRAMM /FPi /LR
BC PROGRAMM /FPi /Fs /LR
BC PROGRAMM /FPa /LR
BC PROGRAMM /FPi /Fs /LR
BC PROGRAMM /FPi /LP
BC PROGRAMM /FPi /Fs /LP
BC PROGRAMM /FPa /LP
BC PROGRAMM /FPi /Fs /LP
```

### ...benötigen Sie diese Library:

```
MATBENR.LIB
MATBEFR.LIB
MATBANR.LIB
MATBAFR.LIB
MATBENP.LIB
MATBEFP.LIB
MATBANP.LIB
MATBAFP.LIB
```

für QBX:

```
QBX PROGRAMM /L MATBEFR            MATBEFR.QLB
```

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 MATB.BAS zusätzliche, undokumentierte Routinen enthält, die intern Verwendung finden und die Sie eventuell mitkopieren müssen.

Wenn Sie Änderungen an den Routinen machen, können Sie die neuen Libraries gleich aus QBX heraus erstellen.

# Allgemeine Hinweise

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: 'MATB.BI'

DIM Variable AS INTEGER, Eingabe AS STRING
DIM Zahl AS SINGLE, XPos AS INTEGER, YPos AS INTEGER
DIM ReturnCode AS INTEGER

DO
  CLS
  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
NEXT
```

*(Fortsetzung nächste Seite)*

*(Fortsetzung)*

```
PRINT
PRINT "Bitte geben Sie nacheinander die Koeffizienten ein."
PRINT "Geben Sie QUIT ein, um abzuberechnen; 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$(13) simuliert Drücken der
' ENTER-Taste)
KEY 1, CHR$(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); ""
```

*(Fortsetzung nächste Seite)*

(Fortsetzung)

```
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"; ReturnCode  
    PRINT "Lösung nicht möglich"  
END SELECT  
  
END
```

### Listing 9-1: 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. Auf der folgenden Seite die Routine, die als Ergänzung zur Matrizen-Toolbox benutzt werden kann. 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.

```

' 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 Matrix3 wird das Ergebnis der Division von Matrix1 durch
' Matrix2 geschrieben.)
FUNCTION MatDivS (Matrix1() AS SINGLE, Matrix2() AS SINGLE, Matrix3() AS
SINGLE)

    DIM FehlerCode AS INTEGER

    FehlerCode = MatInvS(Matrix2())
    IF FehlerCode = 0 THEN
        FehlerCode = MatMultS(Matrix1(), Matrix2(), Matrix3())
    END IF
    MatDivS% = FehlerCode

END FUNCTION

```

***MATDIV.BAS***

## 9.2 Die Font-Toolbox

### Inhalt der 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). Zweitens kann man seine Größe nicht verändern. Außerdem benutzen professionelle grafische 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.

### Einbinden der Routinen

Die Font-Routinen sind Teil einer Toolbox, also größtenteils in BASIC programmiert (Datei FONTB.BAS). Einige Unterfunktionen wurden allerdings auch in Assembler programmiert und befinden sich - schon assembliert - in der Datei FONTASM.OBJ (der Assembler-Quelltext wird als FONTASM.ASM mitgeliefert).

Beim Installieren wird eine Quick Library namens FONTBEFR.QLB erzeugt, die die beiden oben genannten Dateien enthält. Diese muß geladen werden, um die Font-Routinen in QBX benutzen zu können. Da alle Teile von FONTBEFR.QLB auch in der Presentation-Graphics-Library CHRTBEFR.QLB enthalten sind, muß die Font-Toolbox nicht extra geladen werden, wenn man mit den Chart-Routinen arbeitet.

Außerdem entstehen bei der Installation Libraries der Form FONTB<sub>xyz</sub>.LIB (*x* ist A oder E für Alternate- oder Emulator-Library; *y* ist N oder F für Near oder Far Strings, *z* ist R für Real Mode/DOS beziehungsweise P für Protected Mode/OS/2), je nachdem, ob Sie bei der Installation zum Beispiel Far String-Unterstützung gefordert haben oder nicht. Diese Libraries enthalten jeweils dieselben Routinen wie die Quick Library in der entsprechend kompilierten Version.

Wenn Sie die Presentation Graphics-Toolbox verwenden, müssen Sie trotzdem beim Linken die Font-Libraries mit angeben. Entgegen der Presentation Graphics-

---

\* Proportionale Schriftarten sind solche, bei denen die Buchstaben verschieden breit sind, ein w zum Beispiel wesentlich mehr Platz belegt als ein i.

Quick Library enthalten die gewöhnlichen Presentation Graphics-Libraries nicht auch die Font-Routinen.

### **Wenn Sie so kompilieren...**

```
BC PROGRAMM /FPI /LR
BC PROGRAMM /FPI /FS /LR
BC PROGRAMM /FPA /LR
BC PROGRAMM /FPI /FS /LR
BC PROGRAMM /FPI /LP
BC PROGRAMM /FPI /FS /LP
BC PROGRAMM /FPA /LP
BC PROGRAMM /FPI /FS /LP
```

### **...benötigen Sie diese Library:**

```
FONTBENR.LIB
FONTBEFR.LIB
FONTBANR.LIB
FONTBAFR.LIB
FONTBENP.LIB
FONTBEFP.LIB
FONTBANP.LIB
FONTBAFP.LIB
```

### **für QBX:**

```
QBX PROGRAMM /L CHRTBEFR          FONTBEFR.QLB
```

Jedes Programm, in dem die Font-Routinen benutzt werden sollen, muß die Include-Datei FONTB.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 Verfügung: HELVA, HELVB, HELVE, TMSRA, TMSRB, TMSRE, COURA, COURB und COURE. Die Bedeutung der Buchstabenzusätze wird im Abschnitt "Die Schriftart-Dateien" weiter unten in diesem Kapitel erläutert.

Beachten Sie, daß die Font-Toolbox sehr viel String-Speicherplatz verbraucht (je nachdem, wieviele Schriftarten gleichzeitig geladen sind). Es ist deshalb ratsam, wenn das Programm selbst auch noch eine Anzahl von String-Daten verwaltet, mit Far Strings zu kompilieren, wenn man extensiven Gebrauch von den Font-Routinen machen will.

## **Verändern der Toolbox**

Wenn Sie Änderungen an den Font-Routinen vornehmen, müssen Sie alle Font-Libraries und die Quick Library der Presentation Graphics-Toolbox neu erstellen. Die Erstellung der Grafik-Quick Library ist im Abschnitt "Verändern der Toolbox" im Kapitel 9.3 über die Presentation Graphics-Toolbox geschildert.

Die Font-Quick Library wird so produziert:

```
BC FONTB /X/Ot/LR/FPI/FS;
LINK /Q FONTB+FONTASM, FONTBEFR.QLB, , QBXQLB;
```



Dabei entsteht die neue Quick Library FONTBEFR.QLB mit den geänderten Routinen.

Die FONTxyz.LIB-Libraries ändern Sie so:

```
BC FONTB /X/Ot switches;  
LIB FONTxyz.LIB -+FONTB;
```

Dabei wählen Sie *switches* und *xyz* gemäß der obenstehenden Tabelle; Sie müssen die Prozedur bis zu achtmal durchführen, je nachdem, welche Libraries Sie benötigen. Sie setzt übrigens voraus, daß die Libraries schon existieren. Sonst müßte der zweite Befehl heißen:

```
LIB FONTBxyz.LIB +FONTASM+FONTB;
```

## Allgemeine Hinweise

Wenn man mit den mitgelieferten Schriftarten (\*.FON) arbeiten will, müssen die benötigten Dateien vorhanden sein, wenn das Programm läuft. 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.

## 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 - selbstverständlich - beliebig 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 in toto registriert werden, also entweder alle Fonts aus der Datei oder gar keinen.

---

\* Im Gegensatz zu einem Vektorfont werden die Buchstaben eines Bitmapfonts einfach als kleine Bilder gespeichert, die eine feste Anzahl von Pixeln benötigen und deshalb nicht vergrößert oder verkleinert werden können.

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 Programmes: 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

Unter den Font-Dateien von Microsoft Windows befinden sich auch noch eine C- und eine D-Gruppe, um die Seitenverhältnisse noch besser angleichen zu können. Diese beiden Gruppen befinden sich jedoch nicht im PDS-Lieferumfang und werden auch nicht bei der automatischen Auswahl des besten Seitenverhältnisses (siehe Eintrag zu "LoadFont" im Referenzteil der Font-Toolbox) berücksichtigt.

Unter Umständen kann es auch wünschenswert sein, absichtlich einen "falschen" Font zu benutzen, um Text mit ungewöhnlichen Seitenverhältnissen oder in Zwischengrößen auszugeben.

# Die interne Schriftart

Damit im schlimmsten Fall - wenn aus irgendeinem Grunde keine Schriftart-dateien 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 selbst nicht in der Assembler-Datei FONTASM.OBJ, sondern in CHRTASM.OBJ, die zur Presentation Graphics-Toolbox gehört. Wenn Sie mit der Presentation Graphics-Toolbox arbeiten, verfügen Sie automatisch über die interne Schriftart. Tun Sie das nicht, so müssen Sie in Ihr Programm die Zeile

```
DECLARE SUB DefaultFont (SEG Segment%, SEG Offset%)
```

einfügen und die Datei CHRTASM.OBJ in die Quick Library einbauen (schreiben Sie in die LINK-Zeile bei "Verändern der Toolbox" weiter oben im Kapitel einfach hinter FONTASM noch +CHRTASM dazu). Außerdem müssen Sie CHRTASM.OBJ entweder bei jedem separaten Kompilervorgang einzeln angeben (zum Beispiel LINK PROGRAMM+CHRTASM, , , FONTBEFR;) oder in die Font-Libraries einbauen (zum Beispiel LIB FONTBEFR +CHRTASM;).

Die interne Schriftart muß mit einer speziellen Prozedur registriert und wie alle anderen geladen werden, bevor sie benutzt werden kann. Details über die interne Schriftart finden Sie im Font-Referenzteil unter RegisterMemFont und SetGCharSet.

## Anwendungsbeispiel

Die einfachsten Font-Funktionen zeigt 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 Schrift-höhe 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: 'FONTB.BI'

DIM Beschreibung AS FontInfo, Text AS STRING
DIM 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 automatisch gelöscht)
    x% = LoadFont("n" + STR$(a%))

    ' SelectFont ist nicht erforderlich, denn es ist ja
    ' nur ein Font geladen

    ' Text ausgeben
    Text="Datei "+FontDatei+", Schriftart Nr."+STR$(a%)
    x% = OutGText(0, PixelZeile, Text)

    ' Beschreibung zu aktuellem (=geladenem) Font holen
    ' und Zeilenzähler erhöhen
    GetFontInfo Beschreibung
    PixelZeile = PixelZeile+5+Beschreibung.PixHeight

  NEXT
LOOP

```

*Listing 9-2: FONTDEMO.BAS*

Die Ausgabe dieses Programms sieht etwa so aus:

Datei TMSRA.FON, Schriftart Nr. 1

**Datei TMSRA.FON, Schriftart Nr. 2**

Datei TMSRA.FON, Schriftart Nr. 3

**Datei TMSRA.FON, Schriftart Nr. 4**

Datei TMSRA.FON, Schriftart Nr. 5

**Datei TMSRA.FON, Schriftart Nr. 6**

Datei HELVB.FON, Schriftart Nr. 1

Datei HELVB.FON, Schriftart Nr. 2

Datei HELVB.FON, Schriftart Nr. 3

Datei HELVB.FON, Schriftart Nr. 4

**Datei HELVB.FON, Schriftart Nr. 5**

**Datei HELVB.FON, Schriftart Nr. 6**

Datei COURC.FON, Schriftart Nr. 1

Datei COURC.FON, Schriftart Nr. 2

Datei COURC.FON, Schriftart Nr. 3

Datei TMSRE.FON, Schriftart Nr. 1

Datei TMSRE.FON, Schriftart Nr. 2

Datei TMSRE.FON, Schriftart Nr. 3

**Datei TMSRE.FON, Schriftart Nr. 4**

Datei TMSRE.FON, Schriftart Nr. 5

**Datei TMSRE.FON, Schriftart Nr. 6**

## 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.5.

## Schrifttabelle

Es folgt eine Liste aller Schriftarten und -größen. Die erste Spalte gibt an, in welcher Datei sich die Schrift befindet und damit auch, welche Schriftart es ist. INT ist die interne Schriftart.

Die zweite Spalte enthält die *Größe in Punkt*, die benutzt werden kann, um die verschiedenen Schriftgrößen auf den verschiedenen Bildschirmen etwa vergleichen zu können (also: COURA 12 Punkt auf einer Bildschirmauflösung, die für die A-Typen geeignet ist, ist etwa gleich groß wie COURE 12 Punkt auf einer für die E-Typen geeigneten Bildschirmauflösung).

<b>Datei</b>	<b>Größe (Punkt)</b>	<b>Höhe</b>	<b>Rand</b>	<b>mittl. Breite</b>	<b>maxim. Breite</b>
INT	8	8	0	8	8
COURA	8	6	0	8	8
	10	8	1	9	9
	12	10	2	12	12
COURB	8	10	2	8	8
	10	12	2	9	9
	12	15	3	12	12
COURE	8	13	2	8	8
	10	16	3	9	9
	12	20	4	12	12
HELVA	8	6	0	5	14
	10	8	1	7	19
	12	10	2	8	19
	14	12	3	9	17
	18	15	3	12	24
	24	19	3	16	32
HELVB	8	10	2	5	11
	10	12	2	7	14
	12	15	3	8	16
	14	18	4	9	19
	18	22	4	12	24
	24	28	4	16	32
HELVE	8	13	2	5	11
	10	16	3	7	14
	12	20	4	8	16
	14	24	6	9	19
	18	29	5	12	24
	24	37	5	16	32
TMSRA	10	8	1	7	14
	12	6	0	6	11
	12	10	2	8	16
	14	11	2	9	18
	18	14	2	12	25
	24	18	2	16	33
TMSRB	8	10	2	5	11
	10	12	2	6	14
	12	15	3	8	18
	14	16	2	9	20
	18	20	2	12	25
	24	26	2	16	33
TMSRE	8	13	2	5	11
	10	16	3	6	14
	12	19	3	8	18
	14	21	3	9	17
	18	27	3	12	23
	24	35	3	16	33

Wie Sie vielleicht wissen, ist "Punkt" eine typographische Maßeinheit, in der die Schrifthöhe ausgedrückt werden kann. "Punkt" läßt sich linear in Millimeter umrechnen. Es ist nicht ganz zutreffend, diese Schriftgrößen auf Schriften für den Bildschirm zu übertragen, denn schließlich gibt es größere und kleinere Bildschirme, verschiedene Grafikkarten usw., so daß man nie genau weiß, wie groß ein Buchstabe nun wirklich wird. Trotzdem hat Microsoft bei Windows die Maßeinheit "Punkt" eingeführt, um eine wenigstens ungefähre Vergleichbarkeit von Schriftgrößen auf verschiedenen Systemen zu gewährleisten. Zehn bis zwölf Punkt, das sollten Sie sich merken, ist in etwa die normale Schriftgröße. Größere Zahlen bedeuten höhere Buchstaben.

Als *Höhe* ist die maximale Höhe eines Zeichens der betreffenden Schriftart in Pixel angegeben. Wenn man von dieser Zahl den Wert in der Spalte *Rand* abzieht, erhält man die Anzahl von Pixeln, die ein Buchstabe belegen würde, der sowohl Ober- als auch Unterlänge hat. *Rand* gibt an, wieviele Pixel in der Schriftart über einem Großbuchstaben noch leer sind. Wenn Sie in einer Schriftart mit einem *Rand* von 4 den Befehl `x% = OutGText (100, 100, "Hallo")` abschicken, wird das H tatsächlich erst in der Pixelzeile 104 anfangen (Die Anzahl der Pixelzeilen und -spalten, die der Bildschirm darstellt, ist abhängig vom gewählten Grafikmodus und reicht von 320x200 bis zu 640x480).

Die beiden letzten Spalten geben die mittlere und die maximale *Breite* der Buchstaben aus dieser Schriftart an. Sind beide Werte identisch, handelt es sich um eine nichtproportionale Schrift (Courier und Intern). Bei proportionalen Schriften ist die mittlere Breite viel kleiner als die maximale. Die mittlere Breite kann nützlich sein, um zu überschlagen, wieviele Pixel ein Text in der Breite benötigt. Zwar kann man die Gesamtbreite auch genau berechnen lassen (mit der Funktion `GetGTextLen`), aber erstens muß der Text dazu bekannt sein, und außerdem dauert das wesentlich länger als eine Überschlagsrechnung.

Hinweis: Für die Bildschirm-Modi mit einer Auflösung von 320 x 200 Pixel werden die Punktgrößen der Schriften mit etwa 3/2 multipliziert, damit sie vergleichbar bleiben.

## 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 alle Font-Libraries neu, die Sie benutzen (siehe "Verändern der Toolbox" weiter vorne in diesem Kapitel.).

Wenn die Funktion `flGetNextSpec` in der Datei `FONTB.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
```

## 9.3 Die Presentation Graphics-Toolbox

### Inhalt der Toolbox

Die Grafik-Toolbox dient dazu, typische Geschäftsgrafiken einfach und schnell zu erstellen. Ihre Produkte ähneln denen des Tabellenkalkulationsprogramms Excel. 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 Grafikfunktionen "das besondere 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 Anforederungen hat, desto weniger Arbeit kann die Toolbox übernehmen.

Wenn man mit der Standard-Grafik zufrieden ist, kann man mit vier, fünf Zeilen eine wirklich vorzeigbare Grafik auf den Bildschirm zaubern, und ein bißchen Benutzeroberfläche reicht aus, um ein fast schon marktfähiges Grafikpaket (nämlich die CHRTDEMO-Programme von Microsoft, die im PDS-Lieferumfang enthalten sind) zu erstellen.

Die Toolbox besteht aus einigen Routinen, die Grafik darstellen, einigen Hilfs-routinen, die die vorhergehenden Berechnungen erleichtern und schließlich noch



ein paar Prozeduren, die zusätzliche Texte an die Achsen schreiben oder bei der Berechnung von Muster-Paletten helfen.

## Einbinden der Routinen

Die Presentation Graphics-Routinen sind in einer Toolbox enthalten. Das bedeutet, daß die meisten Routinen in BASIC programmiert sind. Die Quellcodes werden in der Datei CHRTB.BAS mitgeliefert. Außerdem gehören die Daten der internen Schriftart dazu, die sich in der Datei CHRTASM.OBJ befinden (der Assembler-Quelltext heißt CHRTASM.ASM).

Die Presentation Graphics-Toolbox läuft nicht ohne die Font-Toolbox (FONTB.BAS und FONTASM.OBJ).

Beim Installieren wird eine Quick Library namens CHRTBEFR.QLB erzeugt, die alle vier oben genannten Dateien enthält und die Benutzung der Presentation Graphics-Toolbox innerhalb QBX ermöglicht.

Außerdem entstehen bei der Installation Libraries der Form CHRTB<sub>xyz</sub>.LIB (*x* ist A oder E für Alternate- oder Emulator-Library; *y* ist N oder F für Near oder Far Strings, *z* ist R für Real Mode/DOS beziehungsweise P für Protected Mode/OS/2), je nachdem, ob Sie bei der Installation zum Beispiel Far String-Unterstützung gefordert haben oder nicht. Diese Libraries enthalten jeweils nur CHRTB.BAS und CHRTASM.OBJ, so daß man, wenn man separat kompilieren will, nicht nur die CHRTB<sub>xyz</sub>-Library, sondern auch noch die dazugehörige FONTB<sub>xyz</sub>-Library angeben muß.

### Wenn Sie so kompilieren...

```
BC PROGRAMM /FPi /LR
BC PROGRAMM /FPi /Fs /LR
BC PROGRAMM /FPa /LR
BC PROGRAMM /FPi /Fs /LR
BC PROGRAMM /FPi /LP
BC PROGRAMM /FPi /Fs /LP
BC PROGRAMM /FPa /LP
BC PROGRAMM /FPi /Fs /LP
```

### ...benötigen Sie diese Libraries:

```
CHRTBENR.LIB, FONTBENR.LIB
CHRTBEFR.LIB, FONTBEFR.LIB
CHRTBANR.LIB, FONTBANR.LIB
CHRTBAFR.LIB, FONTBAFR.LIB
CHRTBENP.LIB, FONTBENP.LIB
CHRTBEFP.LIB, FONTBEFP.LIB
CHRTBANP.LIB, FONTBANP.LIB
CHRTBAFP.LIB, FONTBAFP.LIB
```

für QBX:

```
QBX PROGRAMM /L CHRTBEFR      CHRTBEFR.QLB
```

Jedes Programm, das die Presentation Graphics-Toolbox benutzen will, muß die Dateien CHRTB.BI und FONTB.BI als Include-Dateien laden.

# Verändern der Toolbox

Wenn Sie Änderungen an den BASIC-Chart-Routinen vornehmen, müssen Sie sämtliche Chart-Libraries neu erstellen. Für die Quick Library geht das so:

```
BC CHRTB /X/Ot/LR/FPi/Fs;  
BC FONTB /X/Ot/LR/FPi/Fs;  
LINK /Q CHRTB+CHRTASM+FONTB+FONTASM,CHRTBEFR.QLB,,QBXQLB;
```

Dann entsteht eine neue Quick Library namens CHRTBEFR.QLB mit den geänderten Routinen.

Die CHRTB<sub>xyz</sub>.LIB-Libraries ändern Sie so:

```
BC CHRTB /X/Ot switches;  
LIB CHRTBxyz.LIB -+CHRTB;
```

Dabei wählen Sie *switches* und *xyz* gemäß der obenstehenden Tabelle; Sie müssen die Prozedur bis zu achtmal durchführen, je nachdem, welche Libraries Sie benötigen. Sie setzt übrigens voraus, daß die Libraries schon existieren. Sonst müßte der zweite Befehl heißen:

```
LIB CHRTBxyz.LIB +CHRTASM+CHRTB;
```

## Allgemeine Hinweise

Die Grafik-Toolbox kennt vier Arten von Grafiken: Balken- (horizontal und vertikal), Linien-, Kuchen- und Punktgrafiken.

Balken-, Linien- und Kurvengrafiken 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 (der später noch eingehend behandelt wird) 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: 'FONTB.BI'
REM $INCLUDE: 'CHRTB.BI'
DIM Wert(1 TO 5) AS SINGLE, Label(1 TO 5) AS STRING
DIM GrafikArt AS ChartEnvironment

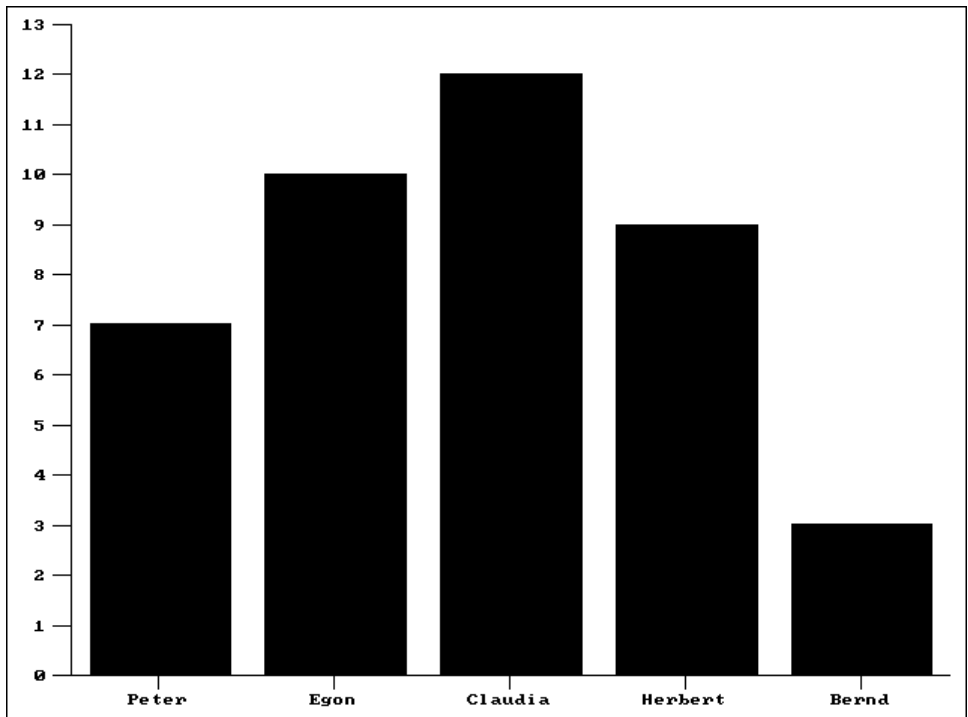
Wert(1) = 7: Label(1) = "Peter"
Wert(2) = 10.5: Label(2) = "Egon"
Wert(3) = 12: Label(3) = "Claudia"
Wert(4) = 8.9: Label(4) = "Herbert"
Wert(5) = 3: Label(5) = "Bernd"

ChartScreen 11
DefaultChart GrafikArt, cColumn, cPlain
Chart GrafikArt, Label(), Wert(), 5

DO WHILE LEN(INKEY$) = 0: LOOP
```

*Listing 9-3: GRDEMO1.BAS*

...reicht schon aus, um diese Grafik zu erzeugen:



Die im Programm benutzten Konstanten cColumn und cPlain sind, ebenso wie eine ganze Reihe weiterer Konstanten, in CHRTB.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 Grafikart in die Beschreibungsvariable GrafikArt zu schreiben. Dieser Routine werden der gewünschte Grafiktyp (im Beispiel cColumn für vertikale Balken) und eine von zwei Variationen 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 CHARTB.BI definiert und beginnen alle mit einem C. Auch die Konstanten der Font-Library 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, mit der man beliebig viele Datenreihen gleichzeitig zeichnen kann, zwei für Punktgrafik und eine für Kuchengrafik, die alle nach dem hier gezeigten Prinzip funktionieren und im Referenzteil genau beschrieben werden.

## 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 CHARTB.BI definiert.

Es reicht, wenn Sie die folgenden Seiten zunächst kurz überfliegen, um einen Eindruck zu bekommen, wieviele und welche Manipulationen an der Standardgrafik möglich sind. Wie das funktioniert, sehen Sie dann in einem Beispiel.

# RegionType für Bereichsbeschreibungen

RegionType wird benutzt, um einen bestimmten Bildschirmbereich zu beschreiben, und zwar genau für drei Bereiche: Den Bereich, der der Grafik insgesamt zur Verfügung steht, den Bereich, der mit den Daten aufgefüllt werden soll, und den Bereich, den die Legende benutzen soll.

```
TYPE RegionType
  X1 AS INTEGER
  X2 AS INTEGER
  Y1 AS INTEGER
  Y2 AS INTEGER
  Background AS INTEGER
  Border AS INTEGER
  BorderStyle AS INTEGER
  BorderColor AS INTEGER
END TYPE
```

*(x1, y1)* ist die linke obere, *(x2, y2)* die rechte untere Ecke des Bereichs. Beim Grafikbereich werden diese Werte absolut von der oberen linken Bildschirmcke aus gezählt, bei den beiden anderen relativ von der oberen linken Ecke des Grafikbereichs.

*background* ist die Farbe, mit der der Hintergrund des betreffenden Bereiches gefüllt werden soll (0  $\frac{3}{4}$  background  $\frac{3}{4}$  cPalLen). Der Standardwert ist 0.

*border* ist entweder cYes oder cNo und legt fest, ob ein Rahmen um den Bereich gezeichnet wird oder nicht. *borderstyle* und *bordercolor* (beide zwischen 0 und cPalLen, beide standardmäßig 1) geben Art und Farbe des Rahmens an.

## TitleType für Titel

TitleType beschreibt einen Titel in der Grafik. Insgesamt enthält eine Variable vom Typ ChartEnvironment sechs solche Titel: Einen Haupt- und einen Untertitel der Grafik, einen Titel für jede Achse und noch eine Achsenbeschriftung an jeder Achse (zum Beispiel "in DM").

```
TYPE TitleType
  Title AS STRING * 70
  TitleFont AS INTEGER
  TitleColor AS INTEGER
  Justify AS INTEGER
END TYPE
```

*title* ist der eigentliche Text, *titlefont* die Schriftart, in der der Titel angezeigt werden soll (Standard und Minimum ist 1, Maximum ist die Anzahl der geladenen Schriftarten); *titlecolor* ist die Farbe, in der der Text geschrieben werden soll (Standard ist 1, Minimum 0 und Maximum cPalLen), und *justify* kann entweder cLeft, cCenter oder cRight sein, je nachdem, wie der betreffende Text in dem ihm zur Verfügung stehenden Bereich ausgerichtet sein soll.

# AxisType für Achsenbeschreibungen

AxisType, der umfangreichste von allen Subtypen, kommt zweimal vor und beschreibt je eine der beiden Achsen der Grafik mit allem, was dazugehört.

```
TYPE AxisType
  Grid AS INTEGER
  GridStyle AS INTEGER
  AxisTitle AS TitleType
  AxisColor AS INTEGER
  Labelled AS INTEGER
  RangeType AS INTEGER
  LogBase AS SINGLE
  AutoScale AS INTEGER
  ScaleMin AS SINGLE
  ScaleMax AS SINGLE
  ScaleFactor AS SINGLE
  ScaleTitle AS TitleType
  TicFont AS INTEGER
  TicInterval AS INTEGER
  TicFormat AS INTEGER
  TicDecimals AS INTEGER
END TYPE
```

*grid* ist entweder cYes oder cNo und legt fest, ob korrespondierend zu den Strichmarkierungen auf der Achse Linien durch den gesamten Grafikbereich gezogen werden sollen (ist *grid* für beide Achsen cYes, ergibt sich ein Netz, in dem die Daten abgetragen werden). *gridstyle* (Standard 1, Minimum 0, Maximum cPalLen) gibt den Linientyp für die Grid-Linien an.

*axistitle* ist die Überschrift für die betreffende Achse (eine Struktur vom Typ TitleType, siehe dort). *axiscolor* (Standard 1, Minimum 0, Maximum cPalLen) legt die Farbe für die Achse selbst und eventuell die Grid-Linien fest.

*labelled* ist entweder cYes oder cNo und bestimmt, ob die betreffende Achse beschriftet werden soll oder nicht. In unserem Beispiel hätte `labelled = cNo` auf der X-Achse bedeutet, daß keine Namen ausgegeben worden wären, und auf der Y-Achse, daß das Programm auf die Zahlen verzichtet hätte.

*rangetype* gibt an, ob die Achse linear (cLinearAxis, der Standard) oder logarithmisch (cLogAxis) sein soll. Eine logarithmische Achse ist nur bei stetigen, nicht bei diskreten Wertetypen sinnvoll, das heißt, daß es nicht gerade besser aussehen würde, wenn ich die Namens-Skala logarithmieren ließen. *logbase* gibt bei logarithmischer Skalierung die Basis des Logarithmus an; Standard ist 10.

Wenn *autoscale* auf cYes gesetzt ist, kümmert sich die Toolbox automatisch um die Folgewerte *scalemin*, *scalemax*, *scalefactor*, *scaletitle*, *ticinterval*, *ticformat* und *ticdecimals*. Anderenfalls müssen Sie diese Werte selbst einstellen.

*scalem**in* und *scalem**ax* sind der niedrigste beziehungsweise der höchste Wert, der auf der Achse dargestellt wird. *scaleg**factor* ist eine Zahl, durch die alle Werte dividiert werden, bevor sie an die Achse geschrieben werden. Üblicherweise ist *scaleg**factor* 1, bei zu hohen Werten sollte hier eine Potenz von 1000 gewählt werden. In *scaleg**title* kann dann eine Beschreibung der Teilung, beispielsweise "(In Mio.)", stehen.

Die folgenden Variablen, die sich auf sogenannte Ticks beziehen, sind nur für Achsen wichtig, die eine numerische Skala haben (in unserem Beispiel die Y-Achse).

*ticfont* enthält die Nummer des Schrifttyps, der für die Zahlen an der Achse benutzt werden soll. (Standard und Minimum ist 1, Maximum ist die Anzahl der geladenen Schriftarten). *ticinterval* ist der Abstand zwischen zwei Zahlen an der Achse, *ticformat* ist entweder cDecFormat (der Standard) für dezimale oder cExpFormat für wissenschaftliche Zahlendarstellung; *ticdecimals* beschreibt, wieviele Dezimalstellen die Zahlen haben sollen (maximal 9).

## LegendType für die Legende

LegendType ist ein Subtyp, der nur einmal benutzt wird und die Legende der Grafik beschreibt.

```
TYPE LegendType
  Legend AS INTEGER
  Place AS INTEGER
  TextColor AS INTEGER
  TextFont AS INTEGER
  AutoSize AS INTEGER
  LegendWindow AS RegionType
END TYPE
```

*legend* kann entweder cYes oder cNo sein. Kuchengrafiken haben immer eine Legende, Grafiken mit nur einer Datenreihe nie. Nur bei Grafiken mit mehreren Datenreihen kann hier festgelegt werden, ob sie eine Legende haben sollen oder nicht.

*place* gibt an, wo im Bild die Legende erscheinen soll. Die drei Möglichkeiten sind cBottom (unter der Grafik), cRight (rechts neben der Grafik) und cOverLay (in der Grafik). Bei cBottom und cRight wird das Datenfenster etwas kleiner gewählt, um Platz für die Legende zu bereiten; bei cOverLay nicht.

*textcolor* (von 0 bis cPalLen) und *textfont* (von 1 bis Anzahl der geladenen Schriftarten) bestimmen Farbe und Schriftart für den Text im Legendendenster. *autosize* ist cYes, wenn die Toolbox die Größe der Legende selbst bestimmen soll, und cNo, wenn die Angaben aus *legendwindow* übernommen werden sollen.

In *legendwindow*, einer Variable vom Typ *regiontype*, sind Koordinaten, Hintergrundfarbe etc. für die Legende angegeben. Die Koordinaten werden ignoriert, wenn *autosize* cYes ist. Ist *autosize* cNo, dann muß *legendwindow* gültige Koordinaten enthalten, und *place* muß cOverLay sein, sonst wird ein Fehler und keine Grafik erzeugt.

## ChartEnvironment - wo die Fäden zusammenlaufen

ChartEnvironment faßt schließlich und endlich alle Subtypen zusammen.

```
TYPE ChartEnvironment
  ChartType AS INTEGER
  ChartStyle AS INTEGER
  DataFont AS INTEGER
  ChartWindow AS RegionType
  DataWinodw AS RegionType
  MainTitle AS TitleType
  SubTitle AS TitleType
  XAxis AS AxisType
  YAxis AS AxisType
  Legend AS LegendType
END TYPE
```

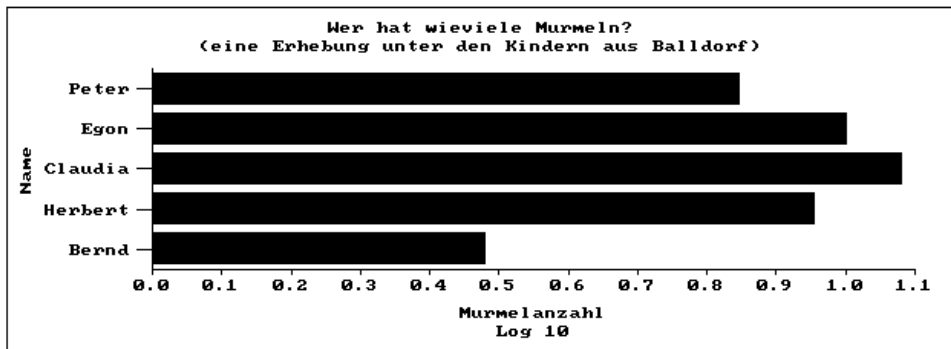
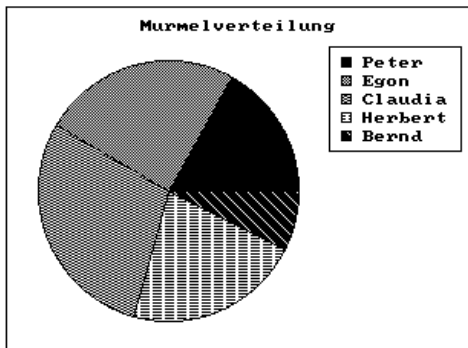
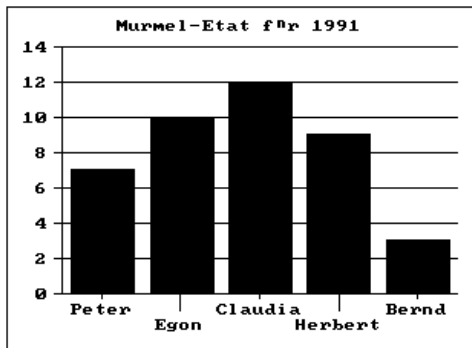
*charttype* ist cBarChart, cColumnChart, cLineChart, cScatterChart oder cPieChart (für vertikale Balken, horizontale Balken, Linien, Punkte und Kuchen). *chartstyle* beschreibt die Ausführung (siehe Referenzteil); *datafont* ist die Nummer der geladenen Schriftart, in der (bei Punkt- und Liniengrafiken) die Datenpunkte angegeben werden sollen (siehe auch Kapitel 9.2 über die Font-Toolbox).

*chartwindow* beschreibt das Fenster (siehe RegionType), das die Toolbox benutzen darf, *datawindow* ist der Bereich innerhalb *chartwindow*, in dem die Daten grafisch dargestellt werden (der Rest wird für Titel, Legende usw. verwendet). Beachten Sie, daß es keinen Sinn hat, die Größe von *datawindow* selbst zu manipulieren; sie wird beim Darstellen der Grafik jedesmal neu errechnet. *datawindow* dient lediglich dazu, daß Sie die Größe in Ihrem Programm abfragen können.

*maintitle* und *subtitle* beschreiben Haupt- und Untertitel der Grafik (siehe TitleType), *xaxis* und *yaxis* sind für die Achsen zuständig (nicht für Kuchengrafiken; siehe AxisType), *legend* kümmert sich um die Legende (siehe LegendType).



Mit diesen zusätzlichen Kenntnissen ist es nun leicht möglich, auch eine Grafik wie die folgende zu erzeugen, die schon recht anspruchsvoll aussieht.



Das Programm dazu baut auf unserem ersten Beispielpogramm 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: 'CHRTB.BI'
DIM Wert(1 TO 5) AS SINGLE, Label(1 TO 5) AS STRING
DIM Ausruecken(1 TO 5) AS INTEGER
DIM 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
```

(Fortsetzung nächste Seite)

(Fortsetzung)

```
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 = "Murmelnanzahl"
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 9-4: GRDEMO2.BAS*

Wenn Sie sich die Grafik etwas genauer anschauen, werden Sie allerdings etwas finden, das Ihr europäisches Gemüt betrüben wird. 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, der benutzt wird, wenn Sie keine Schriftart von der Diskette laden lassen. Wie Sie den Umlauten zur verdienten Geltung verhelfen, steht im Abschnitt über SetGCharSet im Referenzteil zur Font-Toolbox.

Sie können das Problem aber 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, kann aber durch die diversen Font-Variablen in den oben dargestellten Typen 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 wußten.

## Die Farb- und Musterpalette

Die Grafik-Toolbox verwaltet eine interne Farb- und Musterpalette, die aus sechzehn Einträgen besteht. Jedem Paletten-Eintrag 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 Bildschirm-Modus 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 ich wieder 16 Kombinationen erhalte.

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 beziehungsweise neu eintragen (GetPaletteDef und SetPaletteDef). Nähere Erläuterungen finden Sie im Referenzteil.

## 9.4 Die General-Toolbox

### Inhalt der Toolbox

Diese Toolbox enthält allgemeine Routinen, die einerseits von den User-Interface-Toolboxen unbedingt benötigt werden, andererseits aber auch ohne die User-Interface-Routinen durchaus von Nutzen sein können. In der vorliegenden Fassung benötigen die General-Routinen die Maus-Toolbox.

Für diese Sammlung von Funktionen und Prozeduren genügt der Referenzteil zur näheren Beschreibung. Die Routinen aus der General-Toolbox bilden nicht, wie die anderen Teile der User Interface-Toolbox, ein zusammenhängendes System, sondern stellen nur eine Kollektion hilfreicher Geister dar, die zum Beispiel Bildschirmbereiche einlesen, Rahmen zeichnen oder Farben verändern.

### Einbinden der Routinen

Wenn Sie die User-Interface-Toolbox verwenden, stehen Ihnen die General-Routinen automatisch zur Verfügung, da sie in den entsprechenden Libraries enthalten sind. Im Abschnitt "Einbinden der Routinen" der Beschreibung der User-Interface-Toolbox ist auch eine Möglichkeit beschrieben, wie Sie *nur* Maus- und General-Routinen verwenden, ohne die ganze User Interface-Toolbox mit einzubauen. Wie schon gesagt, sind die General-Routinen in der vorliegenden Fassung nicht völlig selbständig, sondern benötigen die Maus-Routinen. Wenn Sie aber nicht mit der Maus arbeiten wollen, können Sie die Funktionsaufrufe an Maus-Routinen aus GENERAL.BAS einfach streichen.

## 9.5 Die User Interface-Toolbox

### Inhalt der Toolbox

Die User Interface-Toolbox soll es ermöglichen, ein Programm schnell und einfach mit einer anspruchsvollen Benutzeroberfläche auszustatten. Inwiefern sie diesem hohen Ziel gerecht wird, mögen Sie selbst beurteilen. Mein Urteil ist ähnlich wie das zur Presentation Graphics-Toolbox: Man kann ein Programm mit Hilfe der User Interface-Routinen leicht ein wenig polieren, aber wenn man Sonderwünsche oder eigene Vorstellungen hat, wird man über kurz oder lang nicht um umfangreiche Änderungen an der Toolbox oder um die Rückkehr zu "Selbstgestricktem" herumkommen.

### Einbinden der Routinen

Die User-Interface-Routinen sind in drei Gruppen gegliedert: Maus-, Menu- und Window-Toolbox. Außerdem sind, strenggenommen, die in diesem Buch einzeln aufgeführten General-Toolbox-Routinen auch Teil des User Interface.

Zur User Interface-Toolbox gehören also:

- |                                    |                       |
|------------------------------------|-----------------------|
| 1. MOUSE.BAS + UIASM.OBJ + QBX.LIB | (unabhängig)          |
| 2. GENERAL.BAS                     | (benötigt 1.)         |
| 3. MENUE.BAS                       | (benötigt 1. und 2.)  |
| 4. WINDOW.BAS                      | (benötigt 1., 2., 3.) |

Wie Sie sehen, baut jede weitere Ausbaustufe der User Interface-Toolbox auf der vorherigen auf. Um die Sache nicht unnötig kompliziert zu machen, gibt es aber nur eine Quick Library und eine Reihe von Libraries für das separate Kompilieren, allesamt mit dem "Vornamen" UITB (User Interface-Toolbox). Diese Libraries enthalten sämtliche Dateien, die oben in der Liste aufgeführt sind.

Von nun an werde ich - der Einfachheit halber - von der General-Toolbox sprechen, wenn ich nur die unter (1) und (2) gelisteten Dateien meine, die auch ohne den Rest sinnvoll einzusetzen sind, und von der User Interface-Toolbox, wenn der ganze Komplex gemeint ist. Trotzdem werden im Referenzteil getrennt die Maus-, Menü-, Window- und General-Routinen behandelt.

Außerdem müssen Sie dafür sorgen, daß Ihr Programm die nötigen BI-Dateien als Include-Files enthält: MOUSE.BI, GENERAL.BI, MENU.BI und WINDOW.BI. Gemäß der obenstehenden Tabelle können Sie zum Beispiel WINDOW.BI weglassen, wenn Sie nur Menü- und keine Fensterroutinen benutzen wollen, nicht aber MENU.BI, wenn Sie nur Fenster- und keine Menüroutinen brauchen.

### Wenn Sie so kompilieren...

```
BC PROGRAMM /FPi /LR
BC PROGRAMM /FPi /Fs /LR
BC PROGRAMM /FPa /LR
BC PROGRAMM /FPi /Fs /LR
BC PROGRAMM /FPi /LP
BC PROGRAMM /FPi /Fs /LP
BC PROGRAMM /FPa /LP
BC PROGRAMM /FPi /Fs /LP
```

### ...benötigen Sie diese Library:

```
UITBENR.LIB
UITBEFR.LIB
UITBANR.LIB
UITBAFR.LIB
UITBENP.LIB
UITBEFP.LIB
UITBANP.LIB
UITBAFP.LIB
```

für QBX:

```
QBX PROGRAMM /L UITBEFR          UITBEFR.QLB
```

## Verändern der Toolbox

Gerade die User-Interface-Toolbox wird von Ihnen gewiß in einen oder anderen Punkt verändert werden (beispielsweise zum Eindeutschen einiger Texte). Wenn Sie Änderungen an den Routinen vornehmen, müssen Sie die User Interface-Libraries neu erstellen. Für die Quick Library geht das so:

```
BC GENERAL /Ot/LR/FPi/Fs;
BC MOUSE /Ot/LR/FPi/Fs;
BC MENU /Ot/LR/FPi/Fs;
BC WINDOW /Ot/LR/FPi/Fs;
LINK /Q GENERAL+MOUSE+MENU+WINDOW+UIASM+QBX.LIB,UITBEFR.LIB,,QBXQLB;
```

Dann entsteht eine neue Quick Library namens UITBEFR.QLB mit den geänderten Routinen.

Die UITB<sub>xyz</sub>.LIB-Libraries ändern Sie so: Führen Sie für jede der bis zu acht Libraries, die Sie erzeugen müssen, zunächst die obigen vier BC-Befehle aus, wobei Sie die Kette /LR/FPi/Fs entsprechend der Tabelle an die zu erzeugende Library anpassen (also für UITBAFP.LIB zum Beispiel /FPi /Fs /LP). Dann folgen die Befehle

```
DEL UITBxyz.LIB
LIB UITBxyz.LIB +GENERAL+UIASM+MOUSE+MENU+WINDOW+QBX.LIB;
```

wobei Sie *xyz* natürlich wieder in die Drei-Buchstaben-Kombination für die Art der Library verwandeln müssen (zum Beispiel AFP).

Hinweis: Wenn Sie ausschließlich die Maus- und General-Routinen benutzen und auf den Rest der Toolbox verzichten wollen, dann können Sie sich die Libraries wie folgt erstellen (ich nenne sie UTIL.QLB beziehungsweise UTIL<sub>xyz</sub>.LIB).

```

UTIL.QLB: BC MOUSE /Ot/LR/FPi/Fs;
          BC GENERAL /Ot/LR/FPi/Fs;
          LINK /Q MOUSE+GENERAL+UIASM+QBX.LIB,UTIL,,QBXQLB;

UTILxyz.LIB: BC MOUSE /X/Ot switches
             BC GENERAL /X/Ot switches
             LIB UTILxyz +MOUSE+GENERAL+UIASM+QBX.LIB;

```

## Allgemeine Hinweise

### 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 (siehe "Mausfunktionen", Anhang F).

### Menüroutinen

Um die Menüfunktionen der User Interface-Toolbox benutzen zu können, müssen Sie (zusätzlich zum Include-File) noch diese Zeilen an den Anfang Ihres Programmes schreiben:

```

COMMON SHARED /uitools/ GloMenu AS MenuMiscType
COMMON SHARED /uitools/ GloTitle() AS MenuTitleType
COMMON SHARED /uitools/ GloItem() AS MenuItemType
DIM GloTitle(MaxMenu) AS MenuTitleType
DIM GloItem(MaxMenu, MaxItem) AS MenuItemType

```

Das Konzept der Menü-Toolbox ist folgendes: Sie erstellen zunächst eine Menüstruktur, indem Sie eine gewisse Anzahl von Hauptmenüpunkten vorsehen, von denen jeder wiederum eine gewisse Anzahl von Menüpunkten, also Auswahlmöglichkeiten besitzt. Im Normalfall ist später - genauso wie bei QBX - immer nur die oberste Zeile mit den Namen der Menüs sichtbar, und erst wenn man mit der Maus oder der ALT-Taste einen dieser Menü-Namen auswählt, wird das Menü dazu angezeigt.

Als Programmierer müssen Sie sich, wenn einmal die Menüstruktur erstellt ist, nur noch darum kümmern, daß oft genug eine Menübearbeitungsroutine (MenuEvent, ShortCutKeyEvent) aufgerufen wird. Normalerweise ist diese Routine schon nach Millisekunden mit ihrer Arbeit fertig, und die Kontrolle geht an Ihr Programm zurück. Wenn der Benutzer jedoch, während gerade die Bearbeitungsroutine läuft, einen Ihrer Menünamen mit der ALT-Taste oder der Maus auswählt, kümmert sich die Bearbeitungsroutine darum, das entsprechende Menü

zu öffnen und den Benutzer mit Maus oder Leuchtbalken einen der Menüpunkte auswählen zu lassen.

Mit einer Abfragefunktion können Sie nach jedem Aufruf der Bearbeitungsroutine feststellen, ob der Benutzer einen Menüpunkt ausgewählt hat. Außerdem können noch Shortcut-Tasten definiert werden, also Tasten, auf die die Menübearbeitungsroutine direkt reagiert, ohne ein Menü aufzubauen. Wenn der Benutzer dann eine solche Shortcut-Taste betätigt, hat das exakt die gleichen Auswirkungen wie das Öffnen des Menüs und das anschließende Auswählen eines Menüpunkts. In QBX ist eine solche Shortcut-Taste zum Beispiel F2, die das Öffnen des View-Menüs und das Auswählen des "SUBs"-Menüpunkts erübrigt.

Die Benutzung der Menü-Toolbox funktioniert also zusammengefaßt so:

1. Toolbox initialisieren (MenuInit)
2. Menüstruktur und eventuell Shortcut-Tasten definieren (MenuSet, ShortCutKeySet, MenuPreProcess)
3. Möglichst häufig Bearbeitungsroutine aufrufen (MenuEvent und ShortCutKeyEvent oder MenuInkey\$)
4. Abfragen, ob etwas passiert ist und, wenn ja, was (MenuCheck)

Über die bereits besprochenen Funktionen hinaus gibt es dann noch Möglichkeiten, den Status eines jeden Menüpunkts zu verändern (wie "Included Lines" im View-Menü bei QBX kann ein Menüpunkt zum Beispiel markiert oder nicht markiert sein; wie "Find" im Search-Menü, wenn der Cursor im Immediate-Fenster steht, kann ein Menüpunkt entweder wählbar oder nicht wählbar sein); das Menü kann ein- und ausgeschaltet und nach einem CLS neu erstellt werden, und einiges mehr.

## Fenster-Routinen

Für die Benutzung der Fenster-Routinen ist neben der Verwendung des Include-Files WINDOW.BI die Einfügung der folgenden Zeilen in Ihr Programm unerläßlich:

```
COMMON SHARED /uitools/ GloWindow() AS windowType
COMMON SHARED /uitools/ GloButton() AS buttonType
COMMON SHARED /uitools/ GloEdit() AS EditFieldType
COMMON SHARED /uitools/ GloStorage AS WindowStorageType
COMMON SHARED /uitools/ GloWindowStack() AS INTEGER
COMMON SHARED /uitools/ GloBuffer$()
```

```
DIM GloTitle(MaxMenu) AS MenuTitleType
DIM GloItem(MaxMenu, MaxItem) AS MenuItemType
DIM GloWindow(MaxWindow) AS WindowType
DIM GloButton(MaxButton) AS ButtonType
```

*(Fortsetzung nächste Seite)*



*(Fortsetzung)*

```
DIM GloEdit(MaxEditField) AS EditFieldType  
DIM GloWindowStack(MaxWindow) AS INTEGER  
DIM GloBuffer$(MaxWindow + 1, 2)
```

#### *Listing 9-5: WINDINC.BAS*

Die Fenster-Routinen nehmen dem geplagten Programmierer in der Regel hauptsächlich die Arbeit ab, sich allzusehr um verschiedene Benutzer-Eingaben und die Reaktion darauf kümmern zu müssen. Wichtige Bestandteile des Konzepts sind Fenster (Windows), Buttons und Eingabefelder (Edit fields).

Ein Fenster ist ein (zumeist) umrandeter Bereich von nahezu beliebiger Größe, der sich wie ein Bildschirm im Bildschirm verhält. Spezielle PRINT- und LOCATE-Routinen für Fenster ermöglichen die Textausgabe in ihnen.

Es können beliebig viele (die maximale Zahl ist in einer Konstanten festgelegt, die Sie nach Herzenslust erhöhen können) Fenster geöffnet werden, die dann auf dem Bildschirm übereinanderliegen. Eines der Fenster ist immer das "aktuelle Fenster", es liegt oben auf dem "Stapel" und ist vollständig sichtbar. Auf dieses Fenster beziehen sich auch die meisten Routinen, zum Beispiel die Befehle zur Textausgabe.

Sie können jederzeit ein anderes Fenster zum aktuellen Fenster machen, indem Sie ein Fenster neu öffnen oder ein bereits vorhandenes aktivieren. Dabei wird es dann unter dem "Stapel" auf dem Bildschirm hervorgeholt.

Sie können auch verfügen, daß der Benutzer bestimmte Fenster nach seinem Gutdünken verschieben und in der Größe verändern darf, wozu allerdings eine Maus erforderlich ist.

Allen Fenstern wird eine Nummer zugeordnet, unter der sie eindeutig identifiziert werden können.

Ein Button ist - im weitesten Sinne - ein Bereich innerhalb eines Fensters, den der Benutzer mit der Maus oder mit den Tasten Tab und Enter anwählen kann. Es gibt verschiedene Arten von Buttons, zum Beispiel die Befehls-Buttons, deren Anwählen die Ausführung eines Befehls verursacht, oder die Schalter-Buttons, die nur dazu dienen, bestimmte Funktionen ein- und auszuschalten. Eine Übersicht über die Schalter-Arten finden Sie im Abschnitt über ButtonInquire im Referenzteil der User Interface-Toolbox, Seite 493.

Die meisten Buttons haben verschiedene Status, so zum Beispiel kann ein Schalter-Button entweder "ein" oder "aus" sein, und diese Unterschiede sind auch auf dem Bildschirm sichtbar.

Was die Buttons anbetrifft, so meldet die Toolbox meist nur irgendeine Aktion des Benutzers weiter, zum Beispiel "Der Benutzer hat den Button Nr. x gedrückt!", und Sie müssen dann entscheiden, was daraufhin passiert. Auf die Meldung "Der Benutzer hat die Tab-Taste gedrückt!" sollten Sie beispielsweise reagieren, indem Sie den Cursor auf den nächsten Button setzen. Wenn die Meldung "Der Benutzer hat Enter gedrückt!" eintrifft und der Cursor gerade auf einem Schalterbutton steht, sollten Sie den Status dieses Schalters ändern usw.

Auch die Buttons sind durchnummeriert, allerdings in jedem Fenster von 1 beginnend. Da mit Buttons immer nur im aktuellen Fenster gearbeitet wird (Buttons in anderen Fenstern können zwar unter Umständen sichtbar sein, sind aber nicht benutzbar), reicht diese Nummer aus, um einen Button eindeutig zu identifizieren.

Für Eingabefelder trifft weitgehend auch das zu, was eben für Buttons gesagt wurde. Eingabefelder sind sozusagen Buttons, die einen Text enthalten, den der Benutzer verändern kann. Die Editier-Routine ist in der Toolbox enthalten, so daß Sie sich nicht darum kümmern müssen, was genau passieren muß, wenn der Benutzer im Eingabefeld die Pfeil-links-Taste drückt usw.

Auch Eingabefelder sind in jedem Fenster von 1 beginnend durchnummeriert.

Die Toolbox stellt Ihnen Routinen zur Verfügung, die es erlauben, Fenster zu öffnen, zu schließen, Text in sie auszugeben und vieles mehr; weitere Routinen sind dafür verantwortlich, Buttons zu öffnen, zu schließen, anzuzeigen oder deren Status zu ändern, und wieder andere beschäftigen sich mit dem Einrichten und Abfragen von Eingabefeldern.

Ähnlich wie die Menübearbeitungsroutinen in der Menü-Toolbox gibt es auch hier eine zentrale Bearbeitungsroutine. Sie heißt WindowDo. Wenn sie aufgerufen wird, übernimmt sie die Kontrolle so lange, bis ein Ereignis eintritt - bis der Benutzer zum Beispiel auf ein anderes als das aktuelle Fenster klickt, das aktuelle Fenster verschiebt oder schließt, einen Menüpunkt auswählt, eine bestimmte Taste drückt oder einen Button anklickt. Mit einer Abfragefunktion namens Dialog können Sie danach feststellen, was passiert ist, und entsprechende Maßnahmen ergreifen.

---

## **Sektion      Aus der Praxis - IV              Probleme und Lösungen**

---

- **Praktische Algorithmik**
  - **Dateiverwaltung**
  - **Variablenverwaltung  
im Speicher**
  - **Umfangreiche  
Programmsysteme**
  - **Die Mathematik-Bibliotheken  
und der Coprozessor**
  - **Fehlerbehandlung**
  - **DOS-Interrupts und ihre  
Nutzung**
  - **Extended & Expanded  
Memory**
  - **OS/2-Programmierung**
-



# 10 Praktische Algorithmik

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.

## 10.1 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. Der Gesamtbereich wird immer weiter in zu sortierende Teilbereiche geteilt, bis die Teilbereiche nur noch ein Element umfassen - das Array ist sortiert. Quicksort ist der Array-Sortier-Algorithmus 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,
    DIM MerkR(1 TO MaxMerk) AS INTEGER
```

*(Fortsetzung nächste Seite)*

(Fortsetzung)

```
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 bear-
    ' beitet. 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 10-1: 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  $2^{\text{MaxMerk}}$  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 sogar 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 sein wird.

## 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 10-2: ISORT.BAS*

## 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
    DIM 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 10-3: 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 Anzahl von Elementen, die in sich schon sortiert sind (die Zahlenfolge 3 56 2 76 78 89 45 22 2 4 9 besteht zum Beispiel aus den vier Läufen [3 56], [2 76 78 89], [45 22] 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.

```
SUB MischSort (FileName AS STRING)

  CONST False = 0, True = -1
  CONST ExtBuffer = 10
  CONST IntBuffer = 1024
  CONST TempFileCode = "TSORT.$"

  DIM AktuellesFile AS INTEGER
  DIM TempFile(1 TO ExtBuffer) AS INTEGER
  DIM InputFile AS INTEGER, OutputFile AS INTEGER
  DIM LaufZaehler AS INTEGER
  DIM LaufEnde(1 TO ExtBuffer) AS INTEGER
  DIM 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 + LTRIM$(STR$(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
```

*(Fortsetzung nächste Seite)*

(Fortsetzung)

```
' 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 "TSORT.$" + LTRIM$(STR$(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%)
        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.)
```

(Fortsetzung nächste Seite)

(Fortsetzung)

```
        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 gezählt, wieviele Läufe gemischt wurden.
' In dem Augenblick, wo jede temporäre 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 10-4: 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 Referenzteil. `IntBuffer` sollte zwischen 512 und 4096 liegen; andere Werte sind möglich (siehe Eintrag zu `OPEN` im 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 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 (beziehungsweise das Medium, auf dem die temporären Dateien erzeugt werden) voll wird.

## 10.2 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 herausuchen.

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)

    DIM Links AS INTEGER, Rechts AS INTEGER
    DIM 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 10-5: 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 ASCII-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.

```
FUNCTION Such$ (File AS INTEGER, Gesucht AS STRING) STATIC
```

```
    CONST MaxZeilenLaenge = 80
```

```
    CONST BufferGroesse = MaxZeilenLaenge * 2 + 2
```

```
    DIM Buffer AS STRING * BufferGroesse
```

```
    DIM 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
```

*(Fortsetzung nächste Seite)*

(Fortsetzung)

```
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 10-6: 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, desto langsamer arbeitet der Algorithmus - allerdings ist er von Natur aus so fix, daß nicht viel schiefgehen kann.

## **10.3 Veränderungen an den Sortier- und Suchalgorithmen**

### **Datentypen**

Bei allen abgedruckten Algorithmen (bis auf Bucketsort) wird der Datentyp STRING gesucht beziehungsweise sortiert. Natürlich kann auch jeder andere Datentyp benutzt werden; man muß nur einige Deklarationen ändern. Bei selbst



definierten 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

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 beziehungsweise 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 beziehungsweise 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.

Die zweite: 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%`, 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 (siehe ISAM-Referenzteil). Aber seien Sie gewarnt: Wenn Sie die ISAM-Routinen nicht in einem speicherresidenten Programm und 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.

## 10.4 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:

$$n! = n \cdot (n-1) \cdot (n-2) \cdot (n-3) \dots \cdot 3 \cdot 2 \cdot 1$$

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:

$$n! = 1 \text{ für } n = 1 \quad \text{und} \quad n! = n \cdot (n-1)! \text{ für } n > 1$$

Diese Definition führt zu einer *rekursiven* Funktion:

```
FUNCTION Fakultaet#(Zahl AS INTEGER)

  IF Zahl = 1 THEN
    Fakultaet# = 1
  ELSE
    Fakultaet# = Fakultaet#(Zahl - 1) * Zahl
  END IF

END FUNCTION
```

### Listing 10-7: FAKULT.BAS

Diese Funktion ruft sich selbst auf, um die Fakultät der Vorgängerzahl zu ermitteln und multipliziert das Ganze dann mit der 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 16.4 ruft sich zum Beispiel selbst auf, um Verzeichnisse 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:

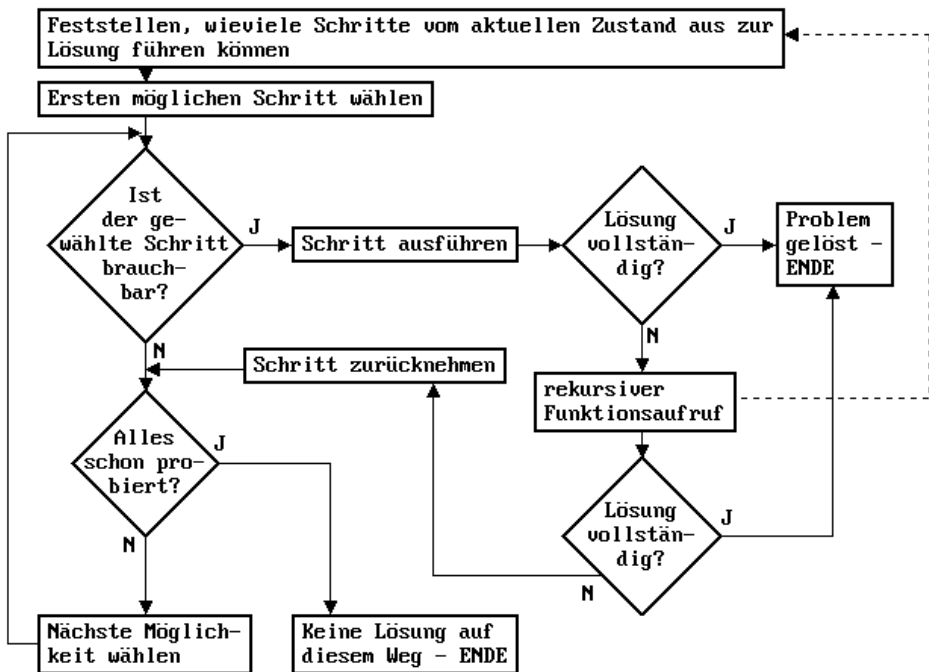


Abbildung 10-1: Ein typischer Backtracking-Algorithmus.

Die Funktion stellt zunächst fest, wieviele Schritte in Richtung der Lösung gemacht werden können (im Beispiel: Wie viele Münzen genommen werden könnten). Dabei ist es wichtig, so viele Schritte wie möglich jetzt schon auszuschießen; 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 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
  SHARED SummeGenommenerMuenzen AS INTEGER
  Erfolg = TRUE

  ' Ist das Problem gelöst?
  IF SummeGenommenerMuenzen = SollWert THEN
    EXIT FUNCTION
  END IF

  ' 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)
```

*(Fortsetzung nächste Seite)*

---

\* Niklaus Wirth: "Algorithmen und Datenstrukturen", Pascal-Version, Stuttgart 1975; ein unbedingt empfehlenswertes Buch, wenn Sie sich tiefergehender 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.

(Fortsetzung)

```
' 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 in einem Array
' o.ä. vermerken)
EXIT FUNCTION
ELSE
' die oben markierte Münze wieder "zurück-
' legen" (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
```

Das Programm ist in "Pseudo-Code" geschrieben, also nicht lauffähig. 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" einer Münze sind hier, ebenso wie die benötigten 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 setzen, 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 schneller und 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.





# 11 Dateiverwaltung

Eine große Anzahl von Programmen arbeitet in irgendeiner Weise mit einem Datenbestand, den der Benutzer verändern kann. Das müssen nicht zwangsläufig Datenbanken sein. Zum Beispiel benötigen Programme, die mit einem Drucker zusammenarbeiten, Informationen über den angeschlossenen Drucker und ermöglichen dem Benutzer häufig auch, eigene Druckertreiber zu erstellen.

Anhand eines konkreten Beispiels sollen hier verschiedene Möglichkeiten dargestellt werden, Daten auf der Diskette beziehungsweise 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 wird in diesem Kapitel nicht berücksichtigt, obwohl es natürlich für viele Anwendungen eine zumindest erwägenswerte Lösung darstellt. Mehr über ISAM erfahren Sie in Kapitel 7.

Ich betrachte als Beispiel ein Programm, das dem Benutzer ermöglicht, gegebene 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.

## 11.1 Parameterdatei

Die 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. 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 auf brutalem Wege unterbrechen, sind die Änderungen verloren. Die zweite Methode hat nicht diesen Nachteil, 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, zum Beispiel das Speichern immer dann, wenn der Benutzer das "Parameter-Ändern-Menü" verläßt. Dann sind Risiko und Zeitbedarf relativ gering.

## Wo sollte die Parameterdatei abgelegt werden?

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.

Andernfalls 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 beziehungsweise 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 läge 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, das vertragen die meisten Programme nicht. 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 Variable 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

Bei einer Parameterdatei mit konstanter Länge kann man sogar verhindern, daß der Benutzer sie beim Kopieren versehentlich "verliert" oder daß sie sich in einem anderen Verzeichnis als das Programm befindet. Man schreibt seine 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) seine 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.

Durch ausgefeilte Techniken könnte man sogar Parameterdateien variabler Länge auf diese Weise an ein Programm anhängen. Man müßte dann in den letzten zwei Zeichen des EXE-Files vermerken, wieviele Bytes vom Programmende aus gesehen zu den Parameterdaten zählen. Allerdings gibt es Schwierigkeiten bei der Verkürzung einer mit OPEN FOR BINARY geöffneten Datei, die dann eventuell von Zeit zu Zeit durchgeführt werden müßte.

Selbstverständlich muß dann, wenn das Programm gerade kompiliert und gelinkt wurde, 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 Kompilier-Vorganges.

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.

## Speicherungsart für Parameter

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 benötige ich 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, , x%
    PUT #FileNummer, , Text

END SUB
```

```
SUB GetS (FileNummer AS INTEGER, Text AS STRING)

    GET #FileNummer, , x%
    Text = SPACE$(x%)
    GET #FileNummer, , Text

END FUNCTION
```

#### *Listing 11-1: 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.

## 11.2 Druckertreiber

Eine weitere Art von Daten, die gespeichert werden wollen, sind Druckertreiber. 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.

### 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 beziehungsweise 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. Aber: Es ist schwierig, bei der heutigen Druckervielfalt den Namen eines Druckermodells eindeutig in einen achtstelligen Dateinamen zu verwandeln. Man müßte also die Gefahr in Kauf nehmen, daß die Dateinamen die Drucker nicht ausreichend beschreiben werden.

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 Druckerdatei übernehmen und damit seine eigenen Definitionen überschreiben, oder aber er bedürfte eines speziellen Konvertierprogramms... mit einzelnen Dateien ist so etwas wiederum kein Problem. Die Dateien könnten in sich so aufgebaut sein wie die Parameterdatei, die ich oben behandelt haben - BINARY mit GET und PUT. Das dürfte auch hier der effizienteste Weg sein.

### Verzeichnis vorhandener Drucker

Wie kann dem Benutzer die Auswahl eines Druckers aus einer Liste ermöglicht werden? Dazu 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 und als RANDOM-Dateien gespeichert werden. Der Zugriff erfolgt mit GET und PUT und je einem selbst-definierten Datentyp.

Für den relativ kleinen Aufgabenbereich "Druckertreiber" mag diese Lösung etwas "aufgeblasen" erscheinen. 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, über solche Dinge nachzudenken, wenn man dem Benutzer lange Wartezeiten 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.

## 11.3 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.

# 11.4 Formularbeschriftungen

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 Druckertreibern 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 ich 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. Die 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 speichert man eine Formularbeschreibung?

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 Formular) 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 FormularBeschriftungTyp
  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 ich unter der Bedingung nutzen können, 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, daß im Text nie vorkommt (zum Beispiel das ASCII-Zeichen Nr. 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 unter Umständen bewerkstelligen).
- Beim Suchen. Man kann entweder das binäre Suchen für Textdateien (vorgestellt in "Suchen", Kapitel 10.2) 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 signifikant länger, und während des Programmablaufs belegt ISAM für seine Puffer etc. 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.



## 11.5 Text-Datenbank

Um noch ein weiteres Beispiel zur Datenspeicherung vorzustellen, nehme ich an, das Formular-Programm sollte seine Benutzeroberflächen-Texte nicht direkt enthalten, sondern aus einer Datei einlesen, so daß man nur die Text-Datei 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 möglichst 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.

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, funktioniert dann so:

```
FUNCTION LiesText$(Nummer AS INTEGER)

    DIM IndexPosition AS INTEGER
    DIM Offset AS LONG, Laenge AS INTEGER
    SHARED TextFile AS INTEGER

    IndexPosition = (Nummer - 1) * 6 + 1
    GET #TextFile, IndexPosition, Offset
    GET #TextFile, , Laenge

    x$ = SPACE$(Laenge)
    GET #TextFile, Offset, x$

    LiesText$ = x$

END FUNCTION
```

### *TCOM.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 schreibt man sich am besten ein eigenes kleines Text-Umformungsprogramm, das solche Dateien aus einer gewöhnlichen Textdatei generieren kann.

# 12 Variablenverwaltung im Speicher

## 12.1 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 werden kann. Wenn auch BASIC 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 sollte die Übersichtlichkeit vorgehen.

Die Werte -1 und 0 werden auch vom Compiler verwandt, 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  $\text{Zahl1} \neq \text{Zahl2}$ ) 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 TRUE oder FALSE wird. Durch die Verwendung der Konstanten wird Ihr Programm sehr viel verständlicher (zum Thema TRUE und FALSE siehe auch "Logische und bitweise Operatoren" in Kapitel 3.2).

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 folgenden Routinen 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 angeben und das Bit, mit dem Sie arbeiten wollen. 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 hat (anstatt, konsequenterweise, 32.768), um negative Integers zu ermöglichen. In CheckBit ist der Umstand 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
    CASE ELSE
    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
    CASE ELSE
    END SELECT

END SUB
```

```
FUNCTION CheckBit%(Zahl AS INTEGER, Bit AS INTEGER)

    CheckBit% = (Zahl AND 2^(Bit - 1) <> 0)

END FUNCTION
```

### *Listing 12-1: BITMAN.BAS*

Sie werden diese Routinen zum Teil auch gebrauchen können, wenn Sie mit Interrupts arbeiten. Siehe auch die einleitende Bemerkung zu Anhang E, "Ausgewählte Interrupts".

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  $-2^{31}$ .

## 12.2 Strings mit fester und variabler Länge

Im Namen ist es nur ein kleiner Unterschied, in der Verwaltung ein Quantensprung. Während der String mit fester Länge sich in die Gesellschaft aller anderen Datentypen (numerische und selbstdefinierte Typen) einreihet, 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. In der Interpreter-Version von BASIC wird für diese Längenangabe nur ein Byte reserviert, also konnte ein String maximal 255 Zeichen lang sein. BASIC 7.1 PDS 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. Hier können Sie nur Strings mit fester Länge speichern, aber diese werden komprimiert und benötigen auf der Platte nur so viele Bytes, 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 = "" beziehungsweise 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:

```
' Programmanfang
DIM Text AS STRING * 10
PRINT LEN(Text); ASC(Text); LEN(RTRIM$(Text))
Text = ""
PRINT LEN(Text); ASC(Text); LEN(RTRIM$(Text))
```

Dieses Programm 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.

## 12.3 Far Strings

### Wozu Far Strings?

Alle bisherigen Versionen des BASIC- oder QuickBASIC-Compilers konnten Strings mit variabler Länge nur im Near-Datenbereich, dem Segment DGROUP, ablegen. Dort werden aber auch sämtliche einfachen Variablen (INTEGER, LONG, CURRENCY, SINGLE, DOUBLE), Variablen selbstdefinierten Typs, Strings mit fester Länge und alle statischen Arrays (siehe Kapitel 12.4) abgelegt. Es ist offensichtlich, daß unter diesen Umständen das Segment DGROUP, das wie jedes Segment nur 65.536 Bytes groß ist, bei bestimmten Anwendungen schlicht überfordert ist.

Die einzige Möglichkeit, den übrigen Speicher für eigene Daten zu nutzen, waren bisher die dynamischen Arrays, und auch hier nur numerische Variablen, selbstdefinierte Typen und Strings mit fester Länge.

Stringdaten konnten also nur auf kompliziertem Wege - als Strings mit fester Länge in einem dynamischen Array - außerhalb des DGROUP-Segments untergebracht werden.

Seit BASIC 7.0 ist es möglich, zu entscheiden, ob ein Programm Strings mit variabler Länge - und nur um diese geht es hier, denn Strings mit fester Länge konnten ja schon vorher ausgelagert werden - im Near- oder Far-Datenbereich speichern soll. Diese Wahl wird beim Kompilieren getroffen und muß für alle Module gleich sein, die später zu einem Programm gehören sollen.

### Near und Far Strings im Vergleich

Wählt man Near Strings, so ändert sich nichts gegenüber früher: Alle Strings liegen im DGROUP-Segment, und dieses ist entsprechend schnell voll. Jeder String benötigt zusätzlich zum eigenen Speicherplatz intern vier Bytes für Infor-

mationen: zwei für die Länge und zwei für seine Adresse. Bei großen String-Arrays können sich diese vier Bytes, die auch ein Leerstring grundsätzlich belegt, schnell bemerkbar machen.

Far Strings erfordern einen wesentlich größeren Verwaltungsaufwand. Jeder Far String belegt insgesamt 10 Bytes interner Informationen. Vier davon werden im DGROUP-Segment gespeichert (also ebensoviel wie bei Near Strings), die sechs verbleibenden sowie der gesamte String-Inhalt wandern in andere Segmente. Dabei ist die Zuordnung wie folgt:

- Alle im Hauptprogramm deklarierten Strings erhalten ein Segment.
- Alle als COMMON deklarierten Strings erhalten ein zweites Segment.
- Alle automatischen und statischen Strings (siehe Kapitel 12.5), die in Prozeduren deklariert werden, erhalten ein drittes Segment.
- Jede Prozedur erhält, wenn sie aufgerufen wird, ein neues Segment für ihre lokalen String-Arrays.

In jedem Segment werden 64 Bytes an Basis-Verwaltungsinformation sowie 6 Bytes pro String belegt.

## Far String-Speicher für Großverbraucher

Wenn Sie besonders viel String-Speicherplatz benötigen, können Sie also zunächst das String-Segment des Hauptprogramms 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 dann 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.

Hier beginnen dann allerdings die Schwierigkeiten. Wem 192 KB 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 folgende Beispiel veranschaulicht die Segmentzuteilung an Far Strings:

```

' Das erste Segment für den COMMON-Block:
' (Wir brauchen zwei Strings, um es zu füllen, denn ein
' String darf nur 32.767 Zeichen haben, und ein Segment
' hat 65.536 Bytes!)
COMMON SHARED ErstesSegment1 AS STRING
COMMON SHARED ErstesSegment2 AS STRING

' das zweite Segment für's Hauptprogramm:
DIM SHARED ZweitesSegment1 AS STRING
DIM SHARED 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
' wir nehmen nicht alle 32767 Bytes, sonst ist kein Platz
' mehr für die Verwaltungsinformationen!

ErstesSegment1 = STRING$(30000, "B")
ErstesSegment2 = STRING$(30000, "A")
ZweitesSegment1 = STRING$(30000, "S")
ZweitesSegment2 = STRING$(30000, "I")
DrittesSegment1 = STRING$(30000, "C")
DrittesSegment2 = STRING$(30000, "P")
WeiteresSegment(1) = STRING$(30000, "D")
WeiteresSegment(2) = STRING$(30000, "S")

' Hier kann nun nach Belieben mit den Strings operiert
' werden.
' Falls nötig, müßte man weitere SUBs aufrufen, die
' ihrerseits String-Arrays deklarieren. Dann müßte man
' allerdings die Variablen "WeiteresSegment" und
' "DrittesSegment" als Parameter übergeben, da sie lokal
' zu dieser Prozedur sind!

' Ende.
END SUB

```

*Listing 12-2: SEGFSTR.BAS*



Achtung: Das Segment, das hier im Beispiel "DrittesSegment" heißt, beherbergt nicht nur alle Strings aus Prozeduren und Funktionen, sondern auch temporäre Strings, wie sie zum Auswerten von String-Ausdrücken gebraucht werden (zum Beispiel bei `a$ = a$ + MID$(a$, 80)`). Es darf also nicht überladen sein, wenn unter Umständen viel Platz für temporäre Strings benötigt wird!

## Vor- und Nachteile beider String-Varianten

Zu den Vorteilen der Far Strings gehört nicht nur, daß insgesamt mehr String-Speicherplatz zur Verfügung steht, sondern auch, daß das DGROUP-Segment hier kaum belastet wird (nur durch vier Bytes Verwaltungsinformation pro String), so daß nun der Platz für einfache numerische Variablen, die nach wie vor in DGROUP abgelegt werden, größer ist. Ein Nachteil der Far Strings ist, daß die oben erwähnten Segmente schon okkupiert werden, wenn nur ein einziger kleiner String darin steht. Dadurch ist weniger Platz für andere Daten (zum Beispiel dynamische Arrays) vorhanden, die ebenfalls den Far-Datenbereich belegen können.

Hinzu kommt, daß Far Strings in der Verarbeitung langsamer sind als ihre Pendanten aus dem Near-Datenbereich (mehr als doppelt so viel Verwaltungsaufwand!). Außerdem ist der Code eines Near-String-Programmes kürzer als der eines Programmes, das Far Strings benutzt.

## Direktzugriff auf Far Strings

Wenn Sie bisher Strings mit PEEK und POKE oder mit BLOAD und BSAVE direkt manipuliert haben, müssen Sie nun bedenken, daß jeder String zusätzlich zu seiner Offset-Adresse auch noch eine Segmentadresse besitzt, die früher, als noch alle Strings in DGROUP standen, redundant war. Die Segmentadresse eines Strings kann mit den Funktionen SSEG oder SSEGADD ermittelt werden. Sie müssen dies auch dann beachten, wenn Sie Interrupts aufrufen (siehe "Datenübergabe bei Interrupt-Aufrufen" im Kapitel 16.2) oder mit Routinen arbeiten, die in anderen Sprachen programmiert sind.

## Kompatibilität zu alten Toolboxen und Quick Libraries

Sie können in Programmen, die mit der Far-String-Option kompiliert werden, im Zusammenhang mit Strings keine Routinen einsetzen, die in anderen Sprachen für frühere BASIC-Compiler geschrieben wurden. Die meisten alten Quick-Basic-Toolboxen, zum Beispiel "ADVBAS", werden bei Verwendung von Far Strings unbrauchbar. Wenn Sie Libraries benutzen, deren BASIC-Quelltext vorliegt, können Sie diese einfach neu kompilieren. Routinen in anderen Sprachen müssen erst umgeschrieben werden, wenn man Far Strings benutzen will. Bei Quick Libraries trifft dies besonders hart, da innerhalb der Entwicklungsumgebung aus-

schließlich mit Far Strings gearbeitet wird und alle alten Quick Libraries deshalb neu generiert werden müssen.

## 12.4 Statische und dynamische Felder

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 (SUB STATIC / FUNCTION STATIC) 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. So mag es manchmal sinnvoll sein, per Metabefehl ein Array statisch werden zu lassen, da Verwaltungsroutinen für dynamische Speicherzuordnung unter Umständen bis zu 5 KB zusätzlich im EXE-File benötigen.

\* "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 Compilieren 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.

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. Seit BASIC 7.1 besteht die interessante Möglichkeit, mit REDIM den bisherigen Inhalt des Arrays unverändert zu lassen (siehe Referenzteil).

## Arrays aus Strings mit variabler Länge

Ein solches Array darf maximal 64 KB umfassen und wird, egal ob statisch oder dynamisch, bei "Near Strings" in DGROUP und bei "Far Strings" im Far-Speicher abgelegt.

## 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 als statische Felder in DGROUP und als dynamische Felder im Far-Speicher abgelegt. Die Obergrenze ist auch hier 64 KB - mit zwei Ausnahmen allerdings: Ein statisches Feld wird, erstens, nie sämtliche 64 KB des DGROUP-Segments belegen können, da dieses auch für alle einfachen Variablen sowie für String- und Array-Deskriptoren benötigt wird.

## Huge Arrays

Außerdem gibt es, zweitens, eine Möglichkeit, numerische Arrays zu erzeugen, die größer als 64 KB sind. Man braucht dazu nur (beim Compiler oder bei QBX) den Switch /Ah anzugeben, 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.

## 12.5 Statische und automatische Variablen

Spätestens hier wird es vielleicht Zeit, die Nebel um das Wort "statisch" ein bißchen aufzuklären: Zunächst gibt es den eben abgehandelten Unterschied zwischen statischen und dynamischen Feldern. In diesen Bereich gehört auch der Meta-befehl \$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 Prozeduren- oder Funktionsdefinition.

Wenn im Kopf einer Prozedur oder Funktion (SUB beziehungsweise 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 beziehungsweise 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, die vor QuickBASIC 4 die einzigen waren, die es gab. Einige erinnern sich vielleicht noch an die Fehlermeldung "Missing STATIC on SUB" aus der Zeit, in der noch keine automatischen Prozeduren unterstützt wurden.

Auch heute noch sind die statischen Prozeduren durchaus sinnvoll (und einer der Punkte, in denen BASIC den meisten Compiler-Konkurrenten überlegen ist!). 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. Mehr dazu siehe im Abschnitt über selbständige Subroutinen weiter unten.

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. Ein Beispiel dazu finden Sie beim Befehl STATIC im Referenzteil.

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, ihre eigenen 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-Befehls-Interpreter 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 herumschlagen zu müssen. 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, 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.

Solche selbständigen Routinen sind immer dann besonders sinnvoll, wenn es um die Kommunikation mit einem externen Gerät oder einem Datenbestand geht. Zum Beispiel wäre es auch denkbar, daß eine Routine die alleinige Schnittstelle zur Festplatte beziehungsweise Diskette darstellt, daß man über diese Prozedur sowohl Parameterdateien einliest als auch auf die programmeigene Datensammlung zugreift, daß die Routine auch einen Datensatz aus der Sammlung sucht, wenn man nur unvollständige Angaben hat etc. Wenn nämlich die Schnittstellen eines Programms nach außen durch solche selbständigen Subroutinen kontrolliert werden, kann sich das Programm selbst als "Kernel" einzig auf seine wahre Aufgabe konzentrieren, während die Kontrollroutinen sich um die Umgebung kümmern.

Außerdem - um wieder auf das Beispiel mit der Druckerkontrollroutine zurückzukommen - hätten Sie damit 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. In vielen anderen Sprachen haben Sie solche Möglichkeiten erst, wenn Sie sich des objektorientierten Programmierens bedienen.

# 13 Umfangreiche Programmsysteme

In diesem Kapitel will ich mich mit Möglichkeiten zur Herstellung von Programmen beschäftigen, die insgesamt (inklusive des benötigten Datenspeicherplatzes) nicht mehr in den Arbeitsspeicher des Rechners passen - zumindest in die 640 KB, die DOS zur Verfügung stellt. Wie in Kapitel 17 genauer beschrieben, kann ja auch erweiterter EMS-Speicher hier nicht helfen, da er allenfalls als Zwischenspeicher für ISAM oder Overlays dient.

Für die *Entwicklung* solcher Programme ist es unbedingt empfehlenswert, EMS installiert zu haben, denn sonst kann man nicht mehr mit QBX arbeiten (es sei denn, man arbeitet mit der RUN/CHAIN-Methode, die hier an erster Stelle erläutert ist).

## 13.1 Programmsysteme mit CHAIN

Diese Methode war bisher die einzige, die es ermöglichte, Programme mit so großem Gesamtumfang zu erstellen: Man schreibt einzelne, voneinander unabhängige Programme und kompiliert beziehungsweise 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 21). 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- beziehungsweise 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 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 ("String space corrupt") 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.

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 darauf, nicht ein Modul mit Far Strings und eines mit Near Strings oder eines mit Coprozessor-Emulation und eines mit Alternate Math-Library zu erstellen.

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 mit "String Space Corrupt" 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) 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 in "BUILDRTM und Runtime-Module", Kapitel 20.



## 13.2 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.

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 EMS

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-Speicher (jenseits der 1 MB-Grenze) zu profitieren, werden unter Umständen beim Programmstart alle gerade nicht benötigten Routinen gleich in den EMS-Speicher (EMS = Expanded Memory Specification) geladen. Sie müssen dann bei Bedarf nur von dort und nicht von der Festplatte geholt werden, was einen erheblichen Zeitgewinn bedeutet. Näheres zur EMS-Nutzung siehe "Extended & Expanded Memory", Kapitel 17.

# Programmieren mit 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 garantiert nicht benötigt usw.

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.

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. Von allen umklammerten Gruppen ist immer nur eine geladen, und wenn ein Aufruf einer Funktion erfolgt, die in einem anderen Overlay-Block steht, wird dieser geladen und der alte wieder aus dem Speicher geworfen.

Das erste angegebene OBJ-File darf nicht in Klammern stehen.

Ein einzelner Overlay-Block darf nicht größer sein als 256 KB. Ein Programm darf maximal 63 Overlay-Blocks haben.

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:

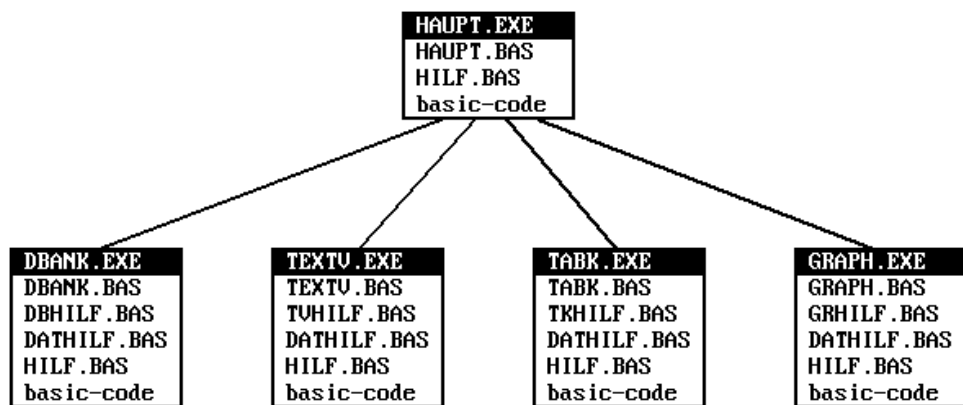
```
LINK EINS+ZWEI+(DREI)+(VIER+FUENF) ;
```

Hier bilden die Module EINS und ZWEI, die nicht mit Klammern versehen sind, den "Grundstock"; sie bleiben dauernd im Speicher. 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, so lange im

Speicher, bis eine Prozedur aus VIER oder FUENF gebraucht wird. Dann würde DREI durch VIER und FUENF überschrieben.

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. Äußerst unvernünftig wäre es also, allgemeine Routinen, die von vielen anderen aufgerufen werden, in ein Overlay zu stecken, denn dann müßte dauernd zwischen verschiedenen Overlays hin- und hergeschaltet werden, und dieses "Schalten" bedeutet Laden, und Laden heißt Warten.

Für Overlay-Programme, die unter DOS 2.1 benutzt werden sollen, gibt es die folgende Sonderregelung: Ein Overlay-Programm funktioniert nur dann unter DOS 2.1, wenn die Routinen aus der Datei OVLDOS21.OBJ zur Verfügung stehen. Ein Programm, das mit dem Standard-Runtime-Modul arbeitet, kann also nur Overlays benutzen, wenn Sie beim Installieren die Unterstützung von Overlays unter DOS 2.1 angefordert haben. Dann wird die genannte Datei in die Standard-Runtime-Module eingeschlossen. Ein Programm, das ohne Runtime-Module ("stand alone") oder mit eigenem Runtime-Modul arbeitet, ist, wenn es Overlays unter DOS 2.1 benutzen will, darauf angewiesen, daß OVLDOS21.OBJ entweder direkt beim Linken angegeben wurde oder im selbstgemachten Runtime-Modul enthalten ist.

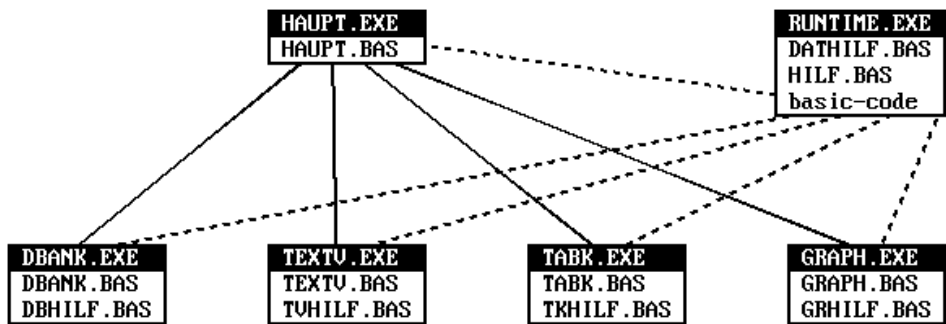


## 13.3 Die verschiedenen Konzepte im Vergleich

Abbildung 13-1: Programme, die sich mit CHAIN aufrufen.

In Abbildung 13-1 sehen Sie 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. HAUPT.BAS, DBANK.BAS, TEXTV.BAS, TABK.BAS und GRAPH.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.



AAbbildung 13-2: Programme mit CHAIN und Runtime-Modul.

Abbildung 13-2 zeigt 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.

HAUPT .EXE HAUPT .BAS HILF .BAS DATHILF .BAS basic-code	(Festbestand)
DBANK .BAS DBHILF .BAS	(1. Overlay)
TEXTV .BAS TVHILF .BAS	(2. Overlay)
TABK .BAS TKHILF .BAS	(3. Overlay)
GRAPH .BAS GRHILF .BAS	(4. Overlay)

Abbildung 13-3: Programm mit Overlays.

Abbildung 13-3 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)+(GRAPH+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.



# 14 Die Mathematik-Bibliotheken und der Coprozessor

Bei BASIC 7.1 PDS können Sie vor der Programmerstellung festlegen, mit welcher Mathematik-Library Ihr Programm ausgerüstet werden soll. Grundsätzlich gibt es zwei Möglichkeiten:

- Die "Emulator"-Library, die Fließkomma-Rechenbefehle so aufbereitet, daß ein eventuell vorhandener Coprozessor sie übernehmen kann.
- Die "Alternate Math"-Library, die so arbeitet, daß ein eventuell vorhandener Coprozessor überhaupt nicht angesprochen wird.

Ein Unterschied zwischen den beiden Bibliotheken macht sich nur bei Befehlen und Operatoren bemerkbar, die mit Fließkomma-Arithmetik arbeiten. Das sind einerseits die üblichen Rechenoperatoren +, -, \* usw., wenn sie auf SINGLE- oder DOUBLE-Zahlen angewendet werden, andererseits aber auch eine Anzahl von Befehlen, denen man es gar nicht ansieht. Eine genaue Liste aller Befehle, die Fließkomma-Arithmetik benutzen, finden Sie in Kapitel 19.2.

## 14.1 Vor- und Nachteile der Emulator-Library

Die Emulator-Library bereitet alle Fließkomma-Befehle so auf, daß ein Coprozessor sie versteht. Ist auf dem Rechner, auf dem das Programm später abläuft, 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.

Wenn Sie sicher wissen, daß das Zielsystem einen Coprozessor besitzt, können Sie durch Linken mit 87.LIB (siehe "Programmgröße und RAM-Speicherplatz", Kapitel 19.2) verhindern, daß die Emulationsroutinen 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-Library. Außerdem entstehen bei ihrer Verwendung grundsätzlich längere Programme als mit der Alternate-Version.

Beim Kompilieren ist die Verwendung der Emulator-Library die Standardeinstellung. Daß eine Library oder ein Runtime-Modul die Emulator-Mathematik

enthält, erkennt man an einem "E" innerhalb der letzten drei Buchstaben des Dateinamens (beispielsweise BRT71EFR.LIB).

## 14.2 Vor- und Nachteile der Alternate Math-Library

Die Alternate Math-Library arbeitet mit IEEE-kompatiblen Arithmetikroutinen. Dadurch wird die gesamte Rechenlast stets dem Hauptprozessor aufgebürdet, 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. Während die Emulator-Library alle Zwischenergebnisse von Berechnungen mit voller 64-Bit-Genauigkeit (für die Mantisse) berechnet, verwendet die Alternate-Library nur 24 Bits für SINGLE- und 54 Bits für DOUBLE-Berechnungen. Das macht sich zwar bei einfachen Berechnungen, die nicht viele Zwischenergebnisse erfordern, nicht bemerkbar: Die Anweisung  $X! = Y! * Z!$  werden beide Libraries gleich ausführen. Bei komplizierteren Ausdrücken - die sich durch geschickte Programmierung jedoch durchaus vermeiden lassen - wie  $X! = (Y! * Z!) + 3 * (A! + B!)$  kann es sein, daß die Ergebnisse der Alternate-Library schon etwas von den korrekten abweichen.

Außerdem kann es vorkommen, daß die von der Alternate-Library berechneten Werte transzendenter Funktionen in der letzten Dezimalstelle falsch sind.

Daß es sich um eine Library oder ein Runtime-Modul mit Alternate Math-Routinen handelt, erkennt man an einem "A" innerhalb der letzten drei Buchstaben des Dateinamens (beispielsweise "BCL71ANR.LIB").

## 14.3 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-Rechner benutzt, dürfte die Emulator-Library die bessere Wahl sein.



# 15 Fehlerbehandlung

"Error trapping" ist fast zwangsläufig Bestandteil jedes Programmes. Wer möchte schon gerne, daß sein Programm sich plötzlich mit der Meldung "Bad file name or number in line 0 at address 2345:ABCD in module LOADFILE, hit any key to return to system" oder etwas Vergleichbarem abmeldet?

Schon jeder einfache BASIC-Interpreter bietet heutzutage Möglichkeiten des Error-Trapping. In BASIC 7.1 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 und ERDEV\$.

## 15.1 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 15-1: 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.

## 15.2 Fortgeschrittene Methoden

Das zentrale Problem bei der Einrichtung flexibler Fehlerbehandlungsmethoden, die Fehler selbst korrigieren oder dem Benutzer wenigstens die Chance dazu geben, ist, daß Ihre Fehlerbehandlungsroutine immer "wissen" muß, wo der Fehler auftrat und warum, damit sie dem Benutzer entsprechende Hinweise geben oder das Programm entsprechend fortsetzen kann.

*Welcher* Fehler aufgetreten ist, läßt sich mit Hilfe von ERR und notfalls von 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.

Vier Lösungen bieten sich an. Die erste kommt deshalb nicht in Betracht, weil Sie doch wieder vor jede Zeile eine Nummer setzen müßten, um mit ERL arbeiten zu können.

Zweitens kann man eine globale Variable benutzen, die dauernd einen Code enthält, dem zu entnehmen ist, wo das Programm gerade arbeitet. Diese Variable müßte alle naselang aktualisiert werden, und das ist auch der Grund, weshalb die zweite Lösung ebenfalls abzulehnen ist. Wenn Sie jedoch eine solche Variable ohnehin schon in Ihrem Programm haben - zum Beispiel, um eine kontextsensitive Hilfstextanzeige zu ermöglichen, die ja auch immer "wissen" muß, wo gerade gearbeitet wird - können Sie diese eventuell auch verwenden, um Fehler-Orte zu ermitteln.

Die dritte und die vierte Lösung sind schon ernster zu nehmen.

Die dritte Lösung ist, 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$
  END IF

  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."
        PRINT "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 15-2: 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 Fehler-Routine 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 solcher Vorgehensweise, daß man sich regelmäßig im Programm darum kümmert, daß ein Fehler aufgetreten sein *könnte*. 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 neuen 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 (Auch das ist erst seit der Version 7 möglich). 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 vierte 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.

Wenn in einer Prozedur ein Fehler auftritt und eine globale Fehlerbehandlungsroutine aktiv ist, also eine, die im Hauptprogramm steht und mit `ON ERROR GOTO` aktiviert wurde, dann wird diese Routine zur Fehlerbehandlung aufgerufen, und mit `RESUME` und `RESUME NEXT` kann - ganz normal - zu dem Befehl, der den Fehler verursachte, oder seinem Nachfolger zurückgesprungen werden.

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.

Auch jede Prozedur, die von der besagten 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 Error-Handler 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"; STR$(ERR);"aufgetreten, aber"
  PRINT "keine Sorge, ich bin unermüdlich!"
  RESUME NEXT
```

```
SUB RoutineEins

  ON LOCAL ERROR GOTO EinsFehler
  RoutineZwei
  EXIT SUB

EinsFehler:
  PRINT "Fehler"; STR$(ERR); "! Soll ich weitermachen?"
  IF INPUT$(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 15-3: 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 vergessen. 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 RoutineZwei 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 besteht, also mehreren einzelnen .BAS-Dateien, von denen eine das Hauptprogramm und Subroutinen und alle anderen nur Subroutinen enthalten, ist die Mächtigkeit eines globalen Error-Handlers eingeschränkt. Er kann dann 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 Modulen 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 (siehe "Der Compiler BC.EXE" im Kapitel 6).

## Resumée

Lokale Error-Handler sind praktisch, wenn eine Prozedur Fehler beheben soll beziehungsweise kann, die ihre Ursache in ihr selbst haben. Die Selbständigkeit der Prozeduren wird dadurch wesentlich erhöht.

In den meisten Fällen und vor allem bei unüberschaubar großen Programmen bewährt sich eine Kombination aus Lokalen Error-Handleern und der als dritte Lösung vorgestellten Methode mit globalen Fehlervariablen, wie sie zum Beispiel auch von den Font- und Presentation Graphics-Toolboxen benutzt wird.





# 16 DOS-Interrupts und ihre Nutzung

## 16.1 Wozu Interrupts?

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 neue Funktion DIR\$ bietet auch nicht alle Möglichkeiten, zum Beispiel kann man nicht feststellen, welche Directories existieren - siehe Eintrag zu DIR\$ im Referenzteil). Teilweise handelt es sich dabei um solch elementare Routinen, daß sie für professionelle Programme unerlässlich sind.

Eine Lösung des Problems ist die Verwendung von Toolboxes oder in Assembler geschriebenen Unterprogrammen. Dies bezahlt man allerdings mit einem Verlust an Übersichtlichkeit und Struktur eines Programms und nicht selten auch mit mannigfaltigen Problemen bei der Erstellung lauffähiger Programme. Viele alte Toolboxes und vorhandene Routinen können - zumindest innerhalb QBX - nicht mehr angewandt werden, da dort jetzt Far Strings obligatorisch sind (siehe "Far Strings" im Kapitel 12.3).

Eine sehr viel elegantere Lösung sind Interrupt-Aufrufe. 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. Manche Interrupts haben viele verschiedene Funktionen. Welche davon man aufrufen möchte, muß man dem Interrupt dann in einem der Register mitteilen.

Interrupts sind nur unter DOS verfügbar; unter OS/2 treten an ihre Stelle Betriebssystemfunktionen, die wie normale Funktionen aufgerufen werden (siehe Kapitel 18).

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. Im Anhang E finden Sie eine ausführliche Beschreibung der Interrupts, die von besonderem Interesse für BASIC-Programmierer sein können.

## 16.2 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 auseinander-setzen. 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 (Details finden Sie bei der jeweiligen Interruptbeschreibung in Anhang E) in einem der Register die Segment- und in einem anderen die Offset-Adresse des Strings. Die Segmentadresse eines gewöhnlichen 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, obwohl 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. Anderenfalls ist es ziemlich sicher, daß Ihr Programm sich früher oder später mit einem *String space corrupt*-Fehler verabschiedet.

## 16.3 Wie man einen Interrupt aufruft

Eine kleine Maschinensprache-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 QBX.LIB beziehungsweise der Quick Library QBX.QLB zur Verfügung gestellt. Wenn man mit ihr in der Programmierungsumgebung QBX arbeiten will, muß man QBX mit dem Parameter /L aufrufen (siehe "Der Aufruf von QBX" in Kapitel 4.1); beim Kompilieren ohne QBX muß beim Link-Prozeß QBX.LIB als Library angegeben werden.

Diese Maschinensprache-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 QBX.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 Registervariable auslesen.

### Zum Beispiel...

Angenommen, Sie möchten mit dem Interrupt 21h (Achtung! Interruptnummern werden hier immer hexadezimal angegeben!) feststellen, welche DOS-Version gerade läuft. Im Anhang lesen Sie, daß Sie dafür als Eingabe-Register AH auf 30h setzen müssen und dann als Ausgabe in AL die Hauptnummer (vor dem Punkt)

und in AH die Unternummer (hinter dem Punkt) erhalten. Sie programmieren also:

```
' aus Bequemlichkeit die TYPE-Definitionen aus dem
' INCLUDE-File einlesen:
REM $INCLUDE:'QBX.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.)

' Interrupt aufrufen:
INTERRUPT &H21, RegEin, RegAus

' Ergebnis auswerten:
PRINT "Die DOS-Version ist";
PRINT RegAus.AX \ 256; "."; RegAus.AX MOD 256

' Ende.
```

*Listing 16-1: INT21.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:

$AX = \langle \text{Wert für AH} \rangle * 256 + \langle \text{Wert für AL} \rangle$

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 Overflow-Fehler führen würde.

Zusätzliche Informationen finden Sie am Anfang des Anhanges E, "Ausgewählte Interrupts".

## 16.4 Weitere Beispiele zur Interrupt-Nutzung

Da alle Beispiele Interrupts benutzen, muß in jedem Falle die Include-Datei QBX.BI geladen werden (`REM $INCLUDE: 'qbx.bi'`). Außerdem muß QBX mit dem Switch /L aufgerufen werden, damit es die Quick Library QBX.QLB lädt, die die Routine für Interrupt-Aufrufe enthält. Für separat kompilierte Programme muß aus demselben Grund QBX.LIB beim Linken angegeben werden.

## 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 dem neuen 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. Wegen dieser Funktion ist es notwendig, die Date-Add-On-Library zu benutzen. Sie können sie aber auch ohne weiteres streichen und dann auf die Date-Library verzichten.

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 *Anzahl* schreibt die Prozedur die Anzahl der gefundenen Dateien (bei *DirectoryRek* muß *Anzahl* vor dem Aufruf auf 0 gesetzt werden!), und in das String-Feld *FileName*( ) werden die Dateinamen eingetragen.

```
REM $INCLUDE: 'qbx.bi'
REM $INCLUDE: 'datim.bi'

SUB Directory (DirName AS STRING, Maske AS STRING, FileName() AS STRING, ~
Anzahl AS INTEGER)

    DIM Fehler AS INTEGER

    Anzahl = 0: Fehler = 0

    ' Attribut = 39 sucht alle Dateien inkl. Hidden +
    ' System + Read-Only; näheres siehe Anhang E

    FindFirstFile DirName + Maske, 39, Fehler

    DO UNTIL Fehler
        Anzahl = Anzahl + 1
        FileName(Anzahl) = GetNameFile
        FindNextFile Fehler
    LOOP

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. Anzahl zuvor auf 0 gesetzt wird, da das nicht (wie
'   bei Directory) in DirectoryRek selbst geschehen
'   kann, weil diese Prozedur sich selbst aufruft.
'
SUB DirectoryRek (DirName AS STRING, Maske AS STRING, FileName() AS ↵
STRING, Anzahl AS INTEGER)

' diese Konstante gibt an, wieviele direkte Sub-
' directories ein Verzeichnis maximal haben kann:
CONST MaxSubDirectories = 20

DIM Fehler AS INTEGER
DIM SubDirectory(MaxSubDirectories) AS STRING
DIM SubDirectoryZaehler AS INTEGER
DIM NeuDirName AS STRING

Fehler = 0

' Attribut = 39 sucht alle Dateien inkl. Hidden +
' System + Read-Only, näheres siehe Anhang E
FindFirstFile DirName + Maske, 39, Fehler
DO UNTIL Fehler
    Anzahl = Anzahl + 1
    FileName(Anzahl) = DirName + GetNameFile
    FindNextFile Fehler
LOOP

' 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) <> "." AND SubDirectoryZaehler < ↵
            MaxSubDirectories THEN
            SubDirectoryZaehler = SubDirectoryZaehler + 1
            SubDirectory(SubDirectoryZaehler) = GetNameFile
        END IF
    END IF
    FindNextFile Fehler
LOOP

' alle Dateien sind in der Liste. Rekursiv werden die
' Subdirectories abgearbeitet:

```

*(Fortsetzung nächste Seite)*

*(Fortsetzung)*

```
FOR i% = 1 TO SubDirectoryZaehler
    NeuDirName = DirName + SubDirectory(i%) + "\"
    DirectoryRek NeuDirName, Maske, FileName(), Anzahl
NEXT
```

END SUB

```
SUB FindFirstFile (Maske AS STRING, Attribut AS INTEGER, ErrorCode AS INTEGER)
```

```
' Attribut = 0 sucht alle normalen Dateien; für
' weitere Informationen siehe Anhang E
```

```
SHARED DTA AS STRING
DIM Register AS RegTypeX
DIM 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
```

*(Fortsetzung nächste Seite)*



(Fortsetzung)

```
' 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%

```
SHARED DTA AS STRING
GetAttrFile% = ASC(MID$(DTA, 22))
```

END FUNCTION

SUB GetDateFile (Month AS INTEGER, Day AS INTEGER, Year AS INTEGER)

```
SHARED DTA AS STRING
DIM Datum AS INTEGER

' Datum ist in den Bytes 25-26 der DTA gespeichert
Datum = CVI(MID$(DTA, 25, 2))
' Tag in den Bits 1-5
Day = TeilZahl(Datum, 1, 5)
' Monat in Bits 6-9
Month = TeilZahl(Datum, 6, 9)
' Jahr relativ zu 1980 in Bits 10-16
Year = 1980 + TeilZahl(Datum, 10, 16)
```

END SUB

FUNCTION GetNameFile\$

```
SHARED DTA AS STRING
GetNameFile$ = MID$(DTA, 31, INSTR(31, DTA, CHR$(0)) - 31)
```

END FUNCTION

```
FUNCTION GetSizeFile&
```

```
    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#
```

```
    GetTimeFile h%, m%, s%
```

```
    GetDateFile mm%, d%, y%
```

```
    GetTimeCode = TimeSerial(h%, m%, s%) + DateSerial(mm%, d%, y%)
```

```
END FUNCTION
```

```
SUB GetTimeFile (Hour AS INTEGER, Minute AS INTEGER, Second AS INTEGER)
```

```
    SHARED DTA AS STRING
```

```
    DIM Zeit AS INTEGER
```

```
    ' Zeitangabe steht auf Bytes 23-24 der DTA
```

```
    Zeit = CVI(MID$(DTA, 23, 2))
```

```
    ' Sekunde ist in den ersten 5 Bits gespeichert,
```

```
    ' allerdings in 2-Sekunden-Schritten
```

```
    Second = TeilZahl(Zeit, 1, 5) * 2
```

```
    ' Minute in den Bits 6-11
```

```
    Minute = TeilZahl(Zeit, 6, 11)
```

```
    ' Stunde in den Bits 12-16
```

```
    Hour = TeilZahl(Zeit, 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)
```

```
    z% = 0
```

```
    FOR i = VonBit TO BisBit
```

```
        IF Zahl AND (2 ^ (i - 1)) THEN z% = z% + 2 ^ (i - VonBit)
```

```
    NEXT
```

```
    TeilZahl = z%
```

```
END FUNCTION
```

*Listing 16-2: INHALT.BAS*

# 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 vorherigen Programm abgedruckt wurde und deshalb hier nicht mehr ausgegeben wird.

```
DECLARE FUNCTION TeilZahl% (Zahl AS INTEGER, VonBit AS INTEGER, ↵
BisBit AS INTEGER)
REM $INCLUDE: 'qbx.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 ist Bit 2 von AX gesetzt
PRINT "Coprozessor: ";
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:";
PRINT TeilZahl(Register.AX, 10, 12)

' Bits 15-16 enthalten die Anzahl der
' LPT-Schnittstellen
PRINT "Drucker-Schnittstellen:";
PRINT TeilZahl(Register.AX, 15, 16)
' Interrupt &H12: Feststellen der Speichergröße
Interrupt &H12, Register, Register
PRINT "Speicher unter 1 MB:";
PRINT Register.AX; "KB"
```

*Listing 16-3: CONFIG.BAS*

# 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 32767 sind, weil BASIC diese dann als negative darstellt, obwohl sie eigentlich zwischen 32.768 und 65.535 liegen.

```
REM $INCLUDE: 'qbx.bi'

FUNCTION FreierPlatz&(Laufwerk AS STRING)

    DIM Reg AS RegType
    Reg.AX = &h3600
    IF Laufwerk = "" THEN
        Reg.DX = 0
    ELSE
        Reg.DX = ASC(UCASE$(Laufwerk)) - 64
    END IF
    Interrupt &h21, Reg, Reg
    IF Reg.AX = -1 THEN
        FreierPlatz = -1
    ELSE
        FreierPlatz = UnsignedInt(Reg.AX) * UnsignedInt(Reg.BX)  ↵
                        * UnsignedInt(Reg.CX)
    END IF
END FUNCTION
```

```
FUNCTION UnsignedInt&(Zahl AS INTEGER)

    ' 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 Overflow-Fehler auftritt,
        ' weil Ergebnis einer INTEGER-Berechnung größer 32767
    ELSE
        UnsignedInt = Zahl
    END IF
END FUNCTION
```

*Listing 16-4: FREI.BAS*

# 17 Extended & Expanded Memory

Dieses Kapitel 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. Beide Stichworte sind nur für den DOS-Betrieb relevant, da OS/2 eine andere Speicherverwaltung benutzt.

Als 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. Expanded Memory kann in jedem PC-kompatiblen Rechner vorhanden sein, da EMS den Zusatzspeicher so verwaltet, daß auch 8086-Prozessoren ihn adressieren können. EMS-Treiber werden gewöhnlich vom Hardware-Hersteller mit dem Rechner oder der Speicherplatine ausgeliefert.

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, den Sie mit der Zeile

```
DEVICE = HIMEM.SYS /HMAMIN=63
```

in CONFIG.SYS laden können. /HMAMIN=63 sorgt dafür, daß sich in den von HIMEM zur Verfügung gestellten 64 KB kein speicherresidentes Programm einnistet, sondern der Platz für QBX frei bleibt (es sei denn, Sie laden ein speicherresidentes Programm mit mehr als 63 KB).

## 17.1 EMS und XMS innerhalb QBX

Wenn XMS vorhanden ist, kann QBX 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. Das ist alles, was QBX und BASIC PDS mit XMS anfangen können: Es bringt Ihnen also maximal 60 KB.

EMS kann bei der Arbeit mit QBX dafür genutzt werden, bestimmte Programm- und Datenbereiche, die anderenfalls den 640 KB-Standard-Arbeitsspeicher belasten würden, auszulagern.

QBX 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 QBX den Switch /Ea angibt; Subroutinen werden automatisch ausgelagert. Mit dem Switch /E:max kann man die maximale EMS-Menge festlegen, derer sich QBX bemächtigen darf; läßt man ihn weg, kann QBX unter Umständen alles zur Verfügung stehende EMS belegen.

Wenn Sie eine Quick Library benutzen, die ihrerseits Zugriffe auf das EMS unternimmt, müssen Sie überdies den Switch /Es angeben, der QBX veranlaßt, vor jedem Quick-Library-Aufruf den aktuellen Status des EMS zu sichern. Das verhindert, daß sich die Quick-Library-Routinen und QBX in die Quere kommen, kostet aber etwas Rechenzeit.

Wenn es Speicherprobleme bei der Arbeit mit QBX 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. (Speicherknappheit bei QBX: Siehe auch Switch /NOF in "Aufruf von QBX", Kapitel 4.1.)

Außerdem wird der EMS-Speicher - nicht nur in QBX - auch als ISAM-Buffer (Zwischenspeicher) benutzt, um schnellere Zugriffe zu ermöglichen (siehe auch Kapitel 7.5).

## 17.2 EMS und XMS bei kompilierten Programmen (EXE-Files)

Kompilierte Programme können EMS weder zur Auslagerung von Subroutinen noch zur Speicherung von Arrays nutzen.

Bei kompilierten Programmen, die mit Overlays arbeiten, wird EMS benutzt, um dort die Overlays abzulegen, damit beim Nachladen von einzelnen Overlays nicht jedesmal auf die Diskette beziehungsweise Festplatte zugegriffen werden muß. Darum kümmert sich der Overlay-Manager, der in Overlay-Programme vom Linker eingebunden wird, automatisch. Wenn man - zum Beispiel, um EMS für andere Zwecke freizuhalten - darauf verzichten möchte, Overlays ins EMS zu laden, muß man beim Linken das Verzicht-File NOEMS.OBJ angeben. (Verwendung von Verzicht-Files siehe Kapitel 19; mehr zu Overlays siehe Kapitel 13.2.)

Ebenso wie innerhalb QBX kann in kompilierten Programmen EMS-Speicher auch als Zwischenspeicher für ISAM dienen. Näheres dazu siehe Kapitel 7.5.

Falls Sie Overlays ins EMS auslagern lassen und mit nicht in BASIC geschriebenen Zusatz-Routinen arbeiten, die ebenfalls 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.

## 17.3 Zusammenfassung

Kompilierte Programme (.EXE-Files) können keinerlei Speicher außer den üblichen 640 KB von DOS benutzen. Es gibt höchstens Möglichkeiten, wenn man mit ISAM arbeitet, von diesen 640 KB etwas zu sparen, indem ISAM-Puffer in das EMS ausgelagert werden. Außerdem kann man, wenn man Overlays benutzt, Zeit sparen, weil Overlays ins EMS geladen werden.

In QBX wird ein Teil von QBX selbst ins XMS ausgelagert (wenn vorhanden); das spart maximal 60 KB. Außerdem wird jede Subroutine und jedes Array (wenn /Ea angegeben wurde), die/das größer als 512 und kleiner als 16.385 Bytes ist, ins EMS ausgelagert, wodurch theoretisch das ganze EMS mit Programmcode und Daten belegt werden kann.





# 18 OS/2-Programmierung

Dieses Kapitel soll keine generelle Einführung in die Programmierung unter OS/2 sein, und ohne genauere Kenntnis von OS/2 wird es Ihnen nur eingeschränkt möglich sein, BASIC-Programme für OS/2 zu erstellen. Sie finden hier nur einige Hinweise, was zu beachten ist, wenn Sie mit dem Gedanken spielen, ein Programm vielleicht später einmal auf OS/2 zu portieren.

Hinweis: Ich beschäftige mich in diesem Kapitel nur mit der Programmerstellung für den Protected Mode. Um Programme im Real-Mode, in der sogenannten "DOS-Kompatibilitätsbox", laufen zu lassen, sind keine Änderungen nötig, aber die Möglichkeiten von OS/2 werden nicht annähernd ausgeschöpft.

Der Compiler BC.EXE und die wichtigen Hilfsprogramme LINK.EXE und LIB.EXE sind Programme, die sowohl im Real Mode (also unter DOS beziehungsweise in der Kompatibilitätsbox) als auch im Protected Mode unter OS/2 laufen. QBX hingegen funktioniert ausschließlich unter DOS beziehungsweise in der Kompatibilitätsbox, nicht also unter OS/2. Die Programmer's WorkBench funktioniert unter OS/2.

Sie können - sowohl mit QBX als auch durch separates Kompilieren mit BC und LINK oder mit der PWB - EXE-Programme entweder für DOS beziehungsweise die Kompatibilitätsbox erstellen oder für OS/2. Sie können keine Programme produzieren, die, wie es der Compiler selbst tut, in beiden Umgebungen funktionieren.

# 18.1 Einschränkungen

Beim Kompilieren von Programmen für den Protected Mode unter OS/2 (mit der Compiler-Option /LP) gibt es in bezug auf die Verwendung der BASIC-Befehle einiges zu beachten:

BLOAD	Achten Sie darauf, daß Speicheradressen, auf die Sie mit BLOAD schreiben wollen, auch wirklich für Ihr Programm zugänglich (in diesem Falle beschreibbar) sind. Ist dem nicht so, erzeugt BASIC einen <i>Permission denied</i> -Fehler, oder das Betriebssystem verhindert die Operation.
BSAVE	analog zu BLOAD
CALL ABSOLUTE	analog zu BLOAD
COLOR	Im SCREEN-Modus 1 wird COLOR ignoriert.
DEF SEG	analog zu BLOAD (darf sich nur auf einen gültigen Selektor beziehen, siehe VARSEG beziehungsweise DEF SEG im Referenzteil)
INP	ist nicht verfügbar
Interrupts	Interrupts stehen selbstverständlich unter OS/2 nicht zur Verfügung, denn sie sind DOS-spezifische Funktionen. Benutzen Sie stattdessen OS/2-eigene Funktionen, die Sie wie gewöhnliche BASIC-Funktionen aufrufen können (siehe unten).
IOCTL	ist nicht verfügbar
IOCTL\$	ist nicht verfügbar (statt IOCTL und IOCTL\$ können aber OS/2-Funktionen benutzt werden)
ON PEN	ist nicht verfügbar
ON PLAY	ist nicht verfügbar
ON STRIG	ist nicht verfügbar
PALETTE	ist nicht verfügbar
PCOPY	hat keinen Sinn, da es nur eine Bildschirmseite gibt
PEEK	analog zu BLOAD
PEN	ist nicht verfügbar
PLAY	ist nicht verfügbar
POKE	analog zu BLOAD
SCREEN	Es sind nur die Modi 0, 1 und 2 erlaubt; die Parameter <i>aseite</i> und <i>vseite</i> (aktuelle beziehungsweise virtuelle Seite) werden nicht unterstützt, da es nur eine Bildschirmseite gibt.
SETMEM	hat keine Funktion
SOUND	ist nicht verfügbar
SSEG	Gibt den Selektor eines Strings zurück (nicht sein Segment)
STICK	ist nicht verfügbar
VARSEG	Gibt den Selektor einer Variable zurück (nicht ihr Segment)
WAIT	ist nicht verfügbar

## 18.2 OS/2-Betriebssystemfunktionen

OS/2 stellt den laufenden Programmen eine Vielzahl von Funktionen zur Verfügung, mehr noch als DOS mit seinen Interrupts. Diese Funktionen können Sie benutzen, wenn Sie die entsprechenden Include-Dateien laden:

Include-Datei	für...
BSEDOSFL.BI	Dateiverwaltung (Verzeichnis etc.), Gerätetreiber
BSESUBMO.BI	Mausunterstützung
BSEDOSPC.BI	OS/2-spezifische Funktionen wie zum Beispiel Kommunikation zwischen verschiedenen laufenden Programmen
BSEDOSPE.BI	Systeminformation, eingestellte Sprache, Uhrzeit und Datum

Welche Funktionen im Detail zur Verfügung stehen, kann hier nicht diskutiert werden. Sie finden die OS/2-Betriebssystemfunktionen in OS/2-Programmierhandbüchern aufgelistet.

Die Funktionen können, wenn Sie die entsprechenden Include-Dateien geladen haben, wie gewöhnliche BASIC-Subroutinen aufgerufen werden.

Dabei gibt es ein kleines Problem: OS/2 kennt einige Datentypen, die BASIC nicht kennt, und verlangt zuweilen diese Typen als Argumente zu seinen Funktionen. Es handelt sich dabei um Zeiger, das sind Adreßangaben, die sich auf irgendeine Stelle des Speichers beziehen.

Anstelle dieser Zeiger sehen die oben genannten Include-Files jeweils zwei INTEGER-Parameter vor, den ersten für das Segment, den zweiten für die Adresse innerhalb des Segments (den Offset).

Wenn OS/2 zum Beispiel einen Zeiger auf einen String verlangt, müssen Sie stattdessen SSEG und SADD benutzen, um Segment und Offset des Strings herauszufinden, und die beiden Ergebnisse an OS/2 übergeben. Bei anderen Variablen als Strings benutzen Sie VARSEG und VARPTR.

## 18.3 Runtime-Module

Wie für die unter DOS konzipierten Programme können auch OS/2-Programme Routinen aus einem Runtime-Modul benutzen. Die Runtime-Module für OS/2 heißen "Dynamic Link Libraries" und erhalten deshalb auch nicht die Extension .EXE, sondern .DLL. Sie werden, genauso wie die DOS-Runtime-Module, mit BUILDRTM erzeugt (siehe "BUILDRTM und Runtime-Module", Kapitel 20).



---

# **Sektion V      Der letzte Schliff**

---

## **. Optimierung**

---



# 19 Effiziente Programmierung

## 19.1 Die Ausführungsgeschwindigkeit eines Programms

### 80286-spezifischer Code

Verwenden Sie, wenn es möglich ist, den /G2-Switch. Die erzeugten Programme laufen dann nur auf Rechnern mit den Prozessoren 80286 und aufwärts, sind aber schneller als gewöhnlich mit diesen Prozessoren, da sie neue, schnellere Befehle verwenden, die es beim 8086/8088 noch nicht gab.

### Alternate Math-Library

Wenn das Zielsystem mit Sicherheit keinen Coprozessor besitzt, sollte die Alternate Math-Library (siehe Kapitel 14.2) benutzt werden; sie ist ein bißchen schneller als die Emulator-Library.

### Verschiedene Datentypen

Far Strings brauchen mehr Zeit als Near Strings.

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.

### GOTO und GOSUB - alt, aber rüstig

GOTO, der Befehl, der jedem Struktur-Fan Schüttelfrost beschert, 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 verschrien, da es längst durch SUB...END SUB ersetzt wurde; trotzdem ist es geringfügig schneller als sein modernes Pendant. An Stellen, wo 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 - siehe Eintrag zu CALL im 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.

## 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.

## 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$) = 0` beziehungsweise `LOOP WHILE LEN(a$) = 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.

## 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.

Diese drei Zeilen habe ich aus dem Microsoft-Handbuch zum BASIC PDS aufgeschnappt:

```
FOR i = 0 TO Length - 1
    PRINT CHR$(PEEK(i + Offset))
NEXT
```

Hier wird insgesamt Length mal zu dem sich verändernden i die Variable Offset addiert. Schreibt man stattdessen

```
FOR i = Offset TO Length - 1 + Offset
    PRINT CHR$(PEEK(i))
NEXT
```

sind nur zwei Additionen fällig (wie Sie im Referenzteil zu FOR...NEXT nachlesen können, wird der Ausdruck Length - 1 + Offset 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. Teilen Sie eine Kette IF a AND b AND c in IF a THEN IF b THEN IF c auf, dann erspart sich der Compiler unnötige Überprüfungen.

# 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 auch:

```
z$ = ""
FOR a% = 0 TO 255
    z$ = z$ + CHR$(a%)
NEXT
```

Hier wird gleich 256mal ein String an den anderen gehängt - eine Sache, die man viel einfacher und mehr als doppelt so schnell haben kann:

```
z$ = SPACE$(256)
FOR a% = 0 TO 255
    MID$(z$, a%, 1) = CHR$(a%)
NEXT
```

## Interrupts

In manchen Situationen lohnt es sich, statt eines BASIC-Befehls einen Interrupt zu benutzen. Brauchbare Interrupts sind im Anhang E aufgelistet. Zum Beispiel erzeugt der Befehl `PRINT STRING$(80,196)` erst einen temporären String, der dann Zeichen für Zeichen ausgegeben wird. Der Interrupt 10h bietet die Möglichkeit, ein bestimmtes Zeichen beliebig oft hintereinander auszugeben, ohne daß dafür ein temporärer String erzeugt werden müßte. Es ist deshalb, selbst wenn man die für den Interrupt-Aufruf nötigen Registerzuweisungen mit einbezieht, bei der Ausgabe mehrerer gleicher Zeichen schneller als der `PRINT`-Befehl.

Hinweis: Beachten Sie aber, daß man die Ausgabe von `PRINT` mit dem `>`-Zeichen in eine Datei umleiten kann, während das mit der Interrupt-Funktion nicht möglich ist; siehe dazu die Bemerkung zu `OPEN` im Referenzteil.

## Funktions- und Prozeduraufrufe

Der Switch /Ot des Compilers sorgt dafür, daß für einen Funktions- oder Prozeduraufruf weniger Code generiert wird. Das macht den Aufruf kürzer und schneller. Damit /Ot wie gewünscht funktioniert, sind aber zwei wichtige Bedingungen einzuhalten:

- Keine der Funktionen oder Prozeduren im Programm darf lokale Error-Handler, `GOSUB`- oder `ON event GOSUB`-Befehle enthalten.
- Es darf nicht mit /D oder /Fs kompiliert werden.

Wenn eine dieser Bedingungen mißachtet wird, wird das Programm zwar trotzdem funktionieren, aber die Verwendung des /Ot-Switches wird das Programm nicht verkürzen, sondern vielleicht sogar verlängern. Im Zweifel probieren Sie einfach aus, ob /Ot Ihnen ein kürzeres oder längeres Programm beschert.

## 19.2 Programmgröße und RAM-Speicherplatz

### Konstanten

Wenn überall, wo Variablen feste Werte haben, die sich im Verlauf des Programms nicht ändern, stattdessen symbolische Konstanten benutzt werden, spart das Speicherplatz im DGROUP-Segment und verkleinert auch das EXE-File, weil symbolische Konstanten bereits beim Kompilieren in Skalare verwandelt werden können (siehe Eintrag zu CONST im Referenzteil). 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.

### Dateibuffer bei sequentiellen Dateien

Kleine Dateibuffer sparen Speicherplatz im String-Bereich. Ein normaler OPEN-Befehl für sequentielle Dateien (INPUT, OUTPUT, APPEND) ohne LEN-Zusatz belegt 512 Bytes als Buffer. Durch einen LEN-Zusatz mit kleinerer Zahl kann dieser Bedarf verringert werden. Je größer der Buffer, 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 deshalb 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" am Ende des Kapitels.

## Fließkomma-Arithmetik

Die Fließkommaarithmetik-Einheit benötigt - egal, ob Sie mit Emulator-Library (/FPi) oder mit Alternate Math-Library (/FPa) arbeiten - etwa 10 KB. Sobald irgendwo im Programm Variablen beziehungsweise 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-, LONG- und zumeist auch bei CURRENCY-Zahlen (vorher mit 1E4 multiplizieren) kann man statt / auch den Integer-Divisionsoperator \ benutzen, der ohnehin um ein Vielfaches schneller arbeitet. Siehe auch Verzicht-File NOFLTIN am Ende dieses Kapitels.

## 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.

## 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

Bei Gewißheit, daß das Zielsystem einen Coprozessor besitzt, kann die Library 87.LIB beim Linken angegeben werden. Sie entspricht der Emulator-Library, kann aber den Coprozessor nicht emulieren (beim Linken /NOE angeben).

## 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, /Fs, /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, und /G2 sorgt zumeist für etwas kleinere Programme. /Ot nimmt eine Sonderrolle ein (siehe "Funktions- und Prozeduraufrufe" weiter vorne in diesem Kapitel). 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, zeigen die beiden folgenden Tabellen. Die erste enthält die relativen Größenabnahmen eines etwa 300 KB großen EXE-Files (die kompilierte Version des mitgelieferten Programms CHRTDEMO.BAS), die zweite Tabelle zeigt dieselben Daten für ein kleineres Programm.

BC-Schalter	LINK-Schalter			
	keiner	/F/PACKC	/E	beide
<b>300 KB-Programm</b>				
keiner	0%	1,3%	7,3%	8%
/G2	0,3%	1,7%	7,7%	8,3%
/S	0,8%	2,2%	7,9%	8,6%
beide	1,1%	2,4%	8,1%	8,8%
<b>65 KB-Programm</b>				
keiner	0%	1,2%	24,5%	25,2%
/G2	0,4%	1,5%	24,9%	25,6%
/S	0,1%	1,2%	24,6%	25,2%
beide	0,4%	1,5%	24,9%	25,5%

## 19.3 Verzicht-Files

Eine Anzahl von Verzicht-Files (frei übersetzt von englisch "stub files") ist im Lieferumfang des Compilers enthalten. Wenn Sie diese OBJ-Dateien beim Linken angeben, wird verhindert, daß bestimmte Routinen aus der BASIC-Library in Ihr EXE-File eingebunden werden, weil die Verzicht-Files selbst gleichnamige Routinen ohne (oder mit geringerem) Inhalt bereitstellen und das Einbinden der entsprechenden Routinen aus der Library dadurch für den Linker nicht mehr nötig erscheint.

Wenn Sie mit einem Runtime-Modul arbeiten, ist es nutzlos, beim Linken des Programms ein Verzicht-File anzugeben, da sich die Routinen aus der BASIC-Library ohnehin im Runtime-Modul befinden.

Sie können allerdings bei der Herstellung eigener Runtime-Module die Einbindung bestimmter BASIC-Routinen in das Runtime-Modul verhindern, indem Sie unter der Direktive "#OBJECTS" in der Exportliste ein oder mehrere Verzicht-files aufführen (siehe "BUILDRTM und Runtime-Module", Kapitel 20). Außerdem können Sie bei der Installation des BASIC PDS auf der Platte dem SETUP-Programm mitteilen, was in das Standard-Runtime-Modul eingeschlossen werden soll und was nicht (siehe "Libraries und Runtime-Module" in Kapitel 2.3).

Wenn Sie mit Verzicht-Files linken, müssen Sie den Switch /NOE angeben. Die folgende Tabelle enthält alle Verzicht-Files mit möglichen Einsparungen und ihrer Wirkung.

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 ("String space corrupt in line... usw."); es werden nur noch Fehlernummern, keine Texte mehr ausgegeben. Einsparung: 1,5 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

*(Fortsetzung nächste Seite)*

File	Wirkung
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
NOEVENT	Kann nur für die Herstellung von Runtime-Modulen benutzt werden; entfernt die Unterstützung jeder Art von Event Trapping.
NOEMS	Ist nur sinnvoll, wenn das Programm mit Overlays arbeitet; verhindert, daß Overlays ins EMS geladen werden.
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
NOTRNEMx	x steht für R oder P (Real oder Protected Mode); 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
TSCNIOyx	x steht für R oder P (Real oder Protected Mode), y für N oder F (Near oder Far Strings). 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. Dadurch wird die Bildschirmausgabe schneller. Siehe auch Bemerkung zu OPEN im Referenzteil. Einsparung: 3,5 KB, wenn das Programm keine Grafikbefehle enthält (sonst mehr)
NOGRAPH	Entfernt alle Grafikunterstützung. NOGRAPH ist in den TSCNIOyx-Dateien bereits enthalten, und alle folgenden Dateien 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





---

## **Sektion VI      Zusatzprogramme zum Compiler**

---

- **BUILDRTM und Runtime-Module**
  - **LIB und Libraries**
  - **Automatisierte Programm-erstellung mit NMAKE**
  - **MKKEY - Veränderung der Tastaturdefinition**
-



# 20 BUILDRTM und Runtime-Module

Dieses Kapitel zeigt zunächst an einem bereits bekannten Beispiel die praktische Verwendung eines Runtime-Moduls und erläutert, was darunter zu verstehen ist. Dann wird näher auf die Standard-Runtime-Module eingegangen, und als letztes wird das Programm BUILDRTM genau beschrieben.

## 20.1 Wozu Runtime-Module?

Vielleicht erinnern Sie sich noch an dieses Beispiel aus Kapitel 6:

HAUPT.BAS
TOOLS1.BAS
TOOLS2.BAS
TOOLS3.BAS

```
BC HAUPT;  
BC TOOLS1;  
BC TOOLS2;  
BC TOOLS3;
```

HAUPT.OBJ
TOOLS1.OBJ
TOOLS2.OBJ
TOOLS3.OBJ

Bis hierher ändere ich nichts. Die drei TOOLS-Dateien enthalten Hilfsroutinen, HAUPT.OBJ ist das Hauptprogramm. Im Original ging es jetzt so weiter:

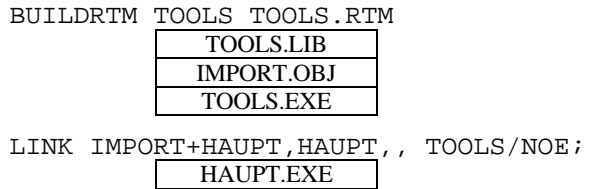
```
LIB TOOLS +TOOLS1+TOOLS2+TOOLS3;
```

TOOLS.LIB
-----------

```
LINK HAUPT,,,TOOLS;
```

HAUPT.EXE
-----------

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. Für ganz Neugierige: TOOLS.RTM ist eine sogenannte 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* von HAUPT.EXE genannt. TOOLS.EXE alleine ist nicht lauffähig. Es erscheint bei Aufruf 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 sich die TOOLS-Routinen daraus nehmen, und der Platz wird auf der Platte 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.

## 20.2 Standard-Runtime-Module

Ähnlich wie ich im obigen Beispiel macht der BASIC-Compiler es auch selbst: 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 des Compilers mit SETUP werden auf der Festplatte die Standard-Runtime-Module und ihre zugehörigen Runtime-Libraries erzeugt. Weil verschiedene Compiler-Switches untereinander inkompatibel sind, gibt es insgesamt bis zu acht verschiedene Runtime-Libraries:

Compiler-Switch	Library	Runtime-Modul
/FPa /LR	BRT71ANR.LIB	BRT71ANR.EXE
/FPa /LP	BRT71ANP.LIB	BRT71ANP.DLL
/FPa /Fs /LR	BRT71AFR.LIB	BRT71AFR.EXE
/FPa /Fs /LP	BRT71AFP.LIB	BRT71AFP.DLL
/FPi /LR	BRT71ENR.LIB	BRT71ENR.EXE
/FPi /LP	BRT71ENP.LIB	BRT71ENP.DLL
/FPi /Fs /LR	BRT71EFR.LIB	BRT71EFR.EXE
/FPi /Fs /LP	BRT71EFP.LIB	BRT71EFP.DLL

Hinweis: Sie können die beiden unter "Library" und "Runtime-Modul" genannten Dateien von BUILDRTM automatisch generieren lassen, indem Sie BUILDRTM /DEFAULT und dahinter die entsprechenden Switches angeben.

Wenn Sie jetzt beliebige Programme kompilieren und linken, ohne beim Kompilieren den Switch /O anzugeben, wird beim Linken automatisch die passende Library aus der obigen Liste gewählt. Diese Library enthält dann nicht die Funktionen selbst, sondern Verweise auf das zugehörige Runtime-Modul, und das hat zur Folge, daß das Programm später nicht ohne das Runtime-Modul läuft.

Geben Sie aber beim Kompilieren den Switch /O an (QBX tut das, wenn Sie "Stand-alone EXE File" wählen), wird beim Linken keine der Libraries aus der Liste, sondern die entsprechenden Library BCL71xyz.LIB verwendet, die nicht Verweise auf Runtime-Module enthält, sondern direkt alle benötigten Funktionen. Dadurch wird Ihr Programm um einiges länger, läuft dafür aber später unabhängig vom Runtime-Modul.

## 20.3 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), Stringspeicher-Methode (/Fs oder nichts) und Prozessormodus (/LR oder /LP) kompiliert werden.

Auch alle Programme, die später das neue RTM benutzen sollen, müssen mit diesen Optionen kompiliert werden.

Wenn alle Programme kompiliert sind, müssen Sie - mit einem beliebigen Texteditor oder mit QBX 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" auflisten, 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 BASIC-Komponentenlibraries vorhanden sind, die BUILDRTM zum Erstellen des RTMs braucht. Diese haben die Namen B71\*.LIB. Vermutlich haben Sie aber SETUP beim Installieren den Befehl gegeben, unnötige Dateien nach der Installation zu löschen; dann sind auch die B71\*.LIB-Dateien von Ihrer Platte verschwunden, da sie im normalen Compiler-Betrieb nicht benötigt werden. Sie müssen dann entweder SETUP neu laufen lassen oder mittels UNPACK.EXE, das auch auf einer der Disketten enthalten ist, alle benötigten Libraries dekomprimieren und auf die Festplatte kopieren. Da UNPACK immer nur mit einem einzelnen Librarynamen aufgerufen werden darf, gehen Sie (unter DOS) am besten wie folgt vor:

A:

(für jede Diskette, auf der B71\*.LIB-Files sind)

```
FOR %i IN (B71*.LIB) DO UNPACK %i C:\BC7\LIB
```

Statt C:\BC7\LIB müssen Sie natürlich gegebenenfalls Ihr LIB-Directory eintragen.

Wenn alle B71\*.LIB-Dateien auf der Platte sind, kann der Aufruf von BUILDRTM erfolgen.

Hinweis: BUILDRTM braucht noch einige andere Dateien, die mit BUILDRTM /H angezeigt werden, aber die werden von SETUP nicht gelöscht und müßten demzufolge noch auf Ihrer Platte sein.

*BUILDRTM switches rtm exportliste*

Als *switches* sind /LR, /LP, /FPa, /FPi und /Fs erlaubt (wie bereits erwähnt), außerdem /FPi87, den Sie verwenden sollten, wenn Sie sicher sind, daß das fertige RTM nur auf Rechnern zum Einsatz kommt, die mit einem Koprozessor ausgestattet sind. Wenn Sie weder /FPa noch /FPi angeben, wird /FPi angenommen; lassen Sie /LR und /LP weg, wählt BUILDRTM /LR, wenn es gerade im Real Mode (DOS), und /LP, wenn es im Protected Mode (OS/2) läuft.

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 oder .DLL (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* (bei /LR) oder *rtm.DLL* (bei /LP) 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 linkern, 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.

## 20.4 Runtime-Module und Verzicht-Files

Im Kapitel 19.3 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.



## 20.5 Runtime-Module und ISAM

Wenn eine der Prozeduren in Ihrem RTM oder aber das Programm selbst ISAM-Funktionen benutzt, dann müssen entweder vor dem Programmaufruf die speicherresidenten ISAM-Routinen geladen werden (PROISAMD.EXE oder PROISAM.EXE, siehe "ISAM in kompilierten Programmen", Kapitel 7.6), oder Sie müssen diese ISAM-Routinen in Ihr RTM einbinden. Dazu benötigen Sie die Dateien PROISAM.OBJ und .LIB beziehungsweise PROISAMD.OBJ und .LIB. (Die D-Version erlaubt auch das Hinzufügen/Löschen von Indizes und das Neuerstellen/Löschen von Datenbanken und ist deshalb etwas umfangreicher.) Die beiden genannten LIB-Dateien läßt SETUP normalerweise auf Ihrer Platte, während Sie für die beiden OBJ-Dateien wahrscheinlich so verfahren müssen wie für die B71\*.LIB-Files (siehe oben). Um ISAM in Ihr RTM einzubinden, geben Sie unter dem Schlüsselwort "#OBJECTS" zusätzlich PROISAMD.OBJ und PROISAMD.LIB an (beziehungsweise PROISAM.OBJ und PROISAM.LIB für den eingeschränkten Zugriff). Das RTM wird dadurch um einiges länger, stellt dann aber sämtliche ISAM-Routinen zur Verfügung.

## 20.6 Add-On-Libraries und Toolboxen in Runtime-Modulen

Ähnlich wie mit ISAM verhält es sich auch mit den mitgelieferten Add-On-Libraries und den Toolboxen. Beide liegen ja stets (unter anderem) als LIB-Datei vor. Sie können ohne weiteres Toolboxen und Add-On-Libraries in Ihr selbst erstelltes RTM packen. Sie müssen dazu nur den Namen des Toolbox- oder Add-On-LIB-Files in die Exportliste eintragen. Achten Sie dabei darauf, daß Sie für die gewählten Einstellungen (/FPi, /FPa usw.) 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" in der Exportliste unter "#LIBRARIES" anführen und darauf verzichten, Eintragungen unter "#EXPORTS" vorzunehmen. Dann stehen die entsprechenden Toolbox-Routinen jedoch nur den SUBs und FUNCTIONs zur Verfügung, die auch im RTM sind. Von außerhalb können Sie dann keine der Toolbox-Routinen benutzen. Bei dieser Methode wird auch nicht die gesamte Library in das RTM übertragen, sondern nur die Teile daraus, die wirklich benötigt werden.

## 20.7 Inkompatible Runtime-Module

Wenn bei dem Versuch, ein Programm zu starten, das mit einem selbstgebastelten RTM arbeitet, die Meldung *Incompatible runtime module* erscheint, haben Sie wahrscheinlich einen der folgenden Fehler gemacht:

- Das Programm wurde nicht mit den gleichen *switches* kompiliert wie die Routinen im RTM beziehungsweise das RTM selbst (es kommt hierbei nur auf /FPa, /FPi, /LR, /LP und /Fs an).
- 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 beim Linken IMPORT.OBJ nicht als erstes OBJ-File genannt.

# 21 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. 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 Programmes 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-Codes aus der Library heraus.

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

Sie sollten in Libraries keine OBJ-Files einbinden, die Modulcode besitzen.

## 21.1 LIB-Aufruf

Das Programm LIB wird wie folgt angewendet:

```
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 "Granularität und Inhalt einer Library" 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
+	Das angegebene OBJ-File muß auf der Platte vorhanden sein und wird an die Library angefügt. Hinter dem Befehl + kann auch statt eines OBJ-Files eine andere Library angegeben werden (die Extension LIB darf dann nicht weggelassen werden). Dann werden alle OBJ-Files 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.
-	Das angegebene OBJ-File wird aus der Library gelöscht (es muß nicht auf der Platte vorhanden sein).
--+	Das angegebene OBJ-File muß auf der Platte sein. Wenn ein OBJ-File gleichen Namens in der Library ist, wird es aus der Library gelöscht. Dann wird das angegebene File in die Library aufgenommen.
*	Das angegebene OBJ-File wird aus der Library in ein echtes OBJ-File auf der Platte kopiert (die Library bleibt dabei unverändert)
-*	Das angegebene OBJ-File wird in ein echtes OBJ-File auf der Platte verwandelt und aus der Library entfernt.

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

Switch	Bedeutung
/PA:n	Setzt die "Seitengröße" der Library auf <i>n</i> . Die Standard-Seitengröße ist 16 Bytes. Jedes Object-File, das in die Library aufgenommen wird, wird so lange mit Füllzeichen verlängert, bis es eine durch die Seitengröße teilbare Länge hat. Deshalb ist es sparsam, kleine Seitengrößen zu benutzen. Allerdings können Libraries mit größeren Seiten auch insgesamt mehr OBJ-Files aufnehmen, und die maximale Größe einer Library ist das 65.536fache der Seitengröße (bei der Standard-Seitengröße von 16 Bytes also genau 1 MB).
/NOE	Benutzen Sie diesen Switch nur, wenn LIB meldet "...try using /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 leer gelassenen oder nicht angegebenen Felder auf ihre Standardwerte gesetzt. Standard ist NUL für *listfile*, der gleiche Name wie *library* für *neulib* und "y" 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. Mehr über diese Liste im Abschnitt "Granularität und Inhalt einer Library" weiter unten in diesem Kapitel.

## Steuerungsdateien

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 Y 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 Y 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.

## 21.2 Linken mit Libraries

Sie kennen bereits die Arbeitsweise des LINK-Programms. Nehmen wir an, Sie linkten 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.

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.

Weitere Informationen über LINK finden Sie in "LINK bringt's zum Laufen" im Kapitel 6.4.

## 21.3 Granularität und Inhalt einer Library

Dem Selektionsvorgang während der LINK-Phase sind allerdings auch Grenzen gesetzt. 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.

Deshalb ist es zumeist sinnvoll, wenn Sie ohnehin mit Libraries arbeiten, möglichst jede Subroutine einzeln zu kompilieren und in die Library aufzunehmen, anstatt alle Subroutinen in wenigen BAS-Files zu kompilieren. Das hieße, um beim 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 aufgebracht werden muß, aber dafür kann später beim Herstellen von EXE-Dateien besser selektiert werden, was eingebunden wird und was nicht. Das macht sich bei allen 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 der BASIC-Libraries ist bei BASIC 7.1 höher als bei BASIC 6.0 oder QuickBASIC, so daß man mit BASIC 7.1 kleinere EXE-Dateien erstellen kann.

Die Granularität einer Library können Sie feststellen, indem Sie LIB benutzen, um eine Liste aller enthaltenen Routinen zu erstellen.

Ein Beispiel: Der Befehl LIB QBX,QBX.LST; legt eine Liste aller Routinen in QBX.LIB an und schreibt sie in die Datei QBX.LST. Diese sieht danach wie folgt aus:

ABSOLUTE.....absolute	INT86OLD.....int86old
INT86XOLD.....int86old	INTERRUPT.....intrpt
INTERRUPTX.....intrpt	
absolute	Offset: 00000010H Code and data size: cH
ABSOLUTE	
intrpt	Offset: 000000f0H Code and data size: 111H
INTERRUPT	INTERRUPTX
int86old	Offset: 000002d0H Code and data size: 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. In Großbuchstaben sind jeweils die Routinen-Namen angegeben, in Kleinbuchstaben die Namen von OBJ-Files.

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.

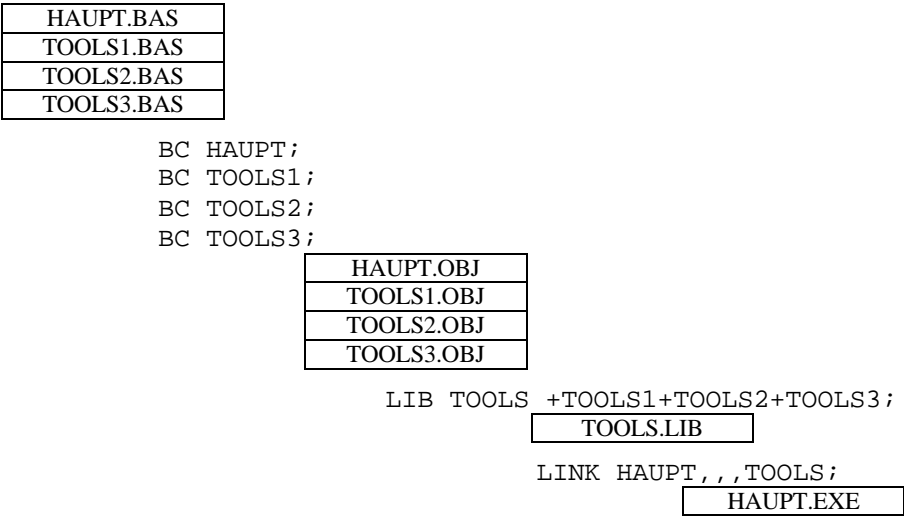


# 22 Automatisierte Programm- erstellung mit NMAKE

NMAKE, der Nachfolger von MAKE, das mit dem BASIC-Compiler 6.0 geliefert wurde, ist ein Hilfsprogramm, das die Herstellung von komplexen Programmsystemen erleichtert. Es kann automatisch feststellen, welche Dateien sich verändert haben, und daraufhin alle Befehle (zum Beispiel BC, LIB, LINK, BUILDRTM) ausführen, die benötigt werden, um das Gesamtsystem wieder auf den neuesten Stand zu bringen. NMAKE ist völlig unabhängig von BASIC. Sie können es auch für beliebige andere Zwecke benutzen. Ich werde NMAKE hier nicht erschöpfend behandeln, sondern hauptsächlich auf das näher eingehen, was beim Erstellen von Programmen mit dem BASIC 7.1 PDS nützlich ist.

## 22.1 Erste NMAKE-Anwendungen

Sie erinnern sich vielleicht noch an dieses Diagramm:



Ich habe es in Kapitel 6 konstruiert, um die Möglichkeiten von LIB vorzuführen, und im Kapitel 20 wieder aufgegriffen, um Runtime-Module zu erläutern. Nun will ich an diesem Beispiel zeigen, wozu NMAKE zu gebrauchen ist.

Zusätzlich nehme ich noch an, daß eine INCLUDE-Datei namens TOOLS.BI existiert, die mit REM \$INCLUDE in die Dateien TOOLS1.BAS, TOOLS2.BAS und TOOLS3.BAS eingebunden wird.

Sie als Entwickler müßten jetzt, wenn Sie irgendeine Änderung an einem der BAS-Files vornehmen, die entsprechenden BC-, LINK- und LIB-Befehle absetzen, um letztlich ein neues HAAPT.EXE zu erhalten. Das kann - besonders auch bei Systemen, die umfangreicher sind als das obige - recht mühselig werden. Häufig schreibt man sich dann einfach eine Batch-Prozedur, die die Aufgaben übernimmt. Wenn Sie aber eine Batch-Prozedur haben, die alles erledigt, was zur Erstellung von HAAPT.EXE nötig ist, und Sie ändern nur HAAPT.BAS, dann würde die Prozedur trotzdem sämtliche TOOLS.BAS-Dateien und die Library neu erstellen - alles Zeitverschwendung, denn die haben sich ja nicht geändert!

Um hier NMAKE einsetzen zu können, müssen Sie zunächst einmal klarstellen, welche Dateien aus welchen anderen Dateien bestehen. Dabei ergibt sich zumeist eine Kette:

```
HAAPT.EXE besteht aus TOOLS.LIB und HAAPT.OBJ
TOOLS.LIB besteht aus TOOLS1.OBJ, TOOLS2.OBJ und TOOLS3.OBJ
TOOLS1.OBJ besteht aus TOOLS1.BAS und TOOLS.BI
TOOLS2.OBJ besteht aus TOOLS2.BAS und TOOLS.BI
TOOLS3.OBJ besteht aus TOOLS3.BAS und TOOLS.BI
HAAPT.OBJ besteht aus HAAPT.BAS
```

Das wäre die Kette der Abhängigkeiten für dieses System. Ich definiere: Eine Datei ist *abhängig* von einer zweiten, wenn eine Veränderung der zweiten einen Neuaufbau der ersten notwendig macht.

NMAKE arbeitet mit einer recht umfangreichen eigenen Programmiersprache. Diese Sprache benutzt man, um zu beschreiben, wie ein gewisses Programm erstellt wird und was genau dabei zu tun ist. Ich kann nun aufgrund meiner oben dargestellten Überlegung schon ein NMAKE-Programm schreiben, das ich HAAPT.NMK nennen will:

```
HAAPT.EXE : TOOLS.LIB HAAPT.OBJ
    LINK HAAPT, , ,TOOLS;
TOOLS.LIB : TOOLS1.OBJ TOOLS2.OBJ TOOLS3.OBJ
    IF EXIST TOOLS.LIB DEL TOOLS.LIB
    LIB TOOLS +TOOLS1+TOOLS2+TOOLS3;
TOOLS1.OBJ : TOOLS1.BAS TOOLS.BI; BC TOOLS1;
TOOLS2.OBJ : TOOLS2.BAS TOOLS.BI; BC TOOLS2;
TOOLS3.OBJ : TOOLS3.BAS TOOLS.BI; BC TOOLS3;
HAAPT.OBJ : HAAPT.BAS; BC HAAPT;
```

Wie Sie sehen, ist die Syntax zur Beschreibung einer Abhängigkeit

```
programm : komponente [komponente...] ; befehl
```

oder

```
programm : komponente [komponente...]  
          befehl  
          [befehl]...
```

Der Befehl oder die Befehle, die notwendig sind, um das Programm aus seinen Komponenten neu zu erstellen, werden entweder in den Folgezeilen (um mindestens eine Leerstelle eingerückt) oder nach einem Semikolon direkt in derselben Zeile aufgeführt.

Ich speichere das NMAKE-Programm in der Datei HAUPT.NMK (normalerweise benutzt NMAKE für diese Beschreibungsdateien, die auch MAKE-Dateien genannt werden, die Extension .MAK. Weil aber auch QBX Dateien mit diesem Namen anlegt, die überhaupt nichts mit NMAKE zu tun haben, sollten Sie ihre NMAKE-Dateien lieber NMK nennen).

## 22.2 Der Aufruf von NMAKE

Die NMAKE-Befehlszeilensyntax sieht so aus:

```
NMAKE [switches] [makrodefinitionen] [programmname...]
```

oder

```
NMAKE @steuerungsdatei
```

Mit *programmname* können Sie NMAKE mitteilen, welche der in der Beschreibungsdatei erwähnten Programme es erzeugen soll. In unserem Beispiel stünden hierfür HAUPT.EXE, TOOLS.LIB, TOOLS1.OBJ, TOOLS2.OBJ, TOOLS3.OBJ und HAUPT.OBJ zur Auswahl. Wenn Sie *programmname* weglassen, wird das erste in der Beschreibungsdatei erwähnte Programm erzeugt.

*makrodefinitionen* sind beliebig viele Makrozuweisungen der Form *makroname*=*text*, die in Anführungszeichen (") eingeschlossen werden müssen, wenn sie Leerzeichen enthalten. Ich komme später darauf zurück.

Wenn Sie den Switch /F nicht benutzen, liest NMAKE seine Instruktionen (die Beschreibungsdatei) aus einer Datei namens MAKEFILE (ohne Extension) im aktuellen Directory.

Die zur Verfügung stehenden *switches* habe ich in der folgenden Tabelle aufgeführt.

Wenn Sie NMAKE nur mit dem Namen einer Steuerungsdatei aufrufen, können Sie alles, was Sie sonst in der NMAKE-Befehlszeile aufführen würden, in diese

Datei eintragen. Der Anfang einer neuen Zeile in der Datei zählt wie ein Leerzeichen auf der Befehlszeile. Eine Zeile, die in der Steuerungsdatei mit einem Backslash (\) aufhört, wird mit der darauffolgenden ohne Leerzeichen verbunden.

Switch	Bedeutung
/A	Alle Programme in der Beschreibungsdatei werden neu erstellt, egal, ob sie überholt sind oder nicht.
/C	Die NMAKE-Copyright-Meldung und Warnungen werden nicht mehr angezeigt.
/D	Zeigt das Datum der letzten Änderung für jede Datei an.
/E	Betriebssystemvariablen können nicht mehr per Makrodefinition in der Beschreibungsdatei umdefiniert werden.
/F <i>name</i>	Benutzt die angegebene Datei anstelle von MAKEFILE als Beschreibungsdatei.
/I	Ignoriert Fehler von Programmen, die NMAKE startet. NMAKE bricht normalerweise die Erstellung eines Programms ab, wenn eines der aufgerufenen Programme einen Fehlercode zurückgibt. Mit /I wird trotzdem weitergemacht.
/N	Simulationsmodus. NMAKE zeigt alle Befehle an, die es ausführen <i>würde</i> , führt sie aber nicht wirklich aus.
/NOLOGO	Zeigt keine NMAKE-Copyright-Meldung an (ähnlich /C).
/P	Gibt alle Makrodefinitionen und Abhängigkeitsbeschreibungen aus.
/Q	Mit diesem Switch gibt NMAKE einen Status-Code (der mit ERRORLEVEL in einer Batch-Prozedur abgefragt werden kann) größer 0 zurück, wenn das Programm, das es neu erstellen sollte, überholt war.
/R	Ignoriert vordefinierte und in TOOLS.INI enthaltene Ableitungsregeln.
/S	Zeigt die Befehle, die NMAKE ausführt, nicht an.
/T	Verhindert, daß die überholten Dateien wirklich aktualisiert werden, sondern setzt das Datum jeder überholten Datei auf das heutige Datum. Dadurch ist keine Datei mehr im NMAKE-Sinne "überholt", obwohl sich der Inhalt nicht verändert.
/X <i>name</i>	Schreibt alle Fehlermeldungen in die angegebene Datei <i>name</i> .

## 22.3 Die NMAKE-Funktionsweise

Ich starte NMAKE mit dem Befehl:

```
NMAKE /F HAUPT.NMK
```

Wenn ich NMAKE nur mit /F den Dateinamen der Beschreibungsdatei mitteile, nicht aber, was es eigentlich daraus erstellen soll, dann erstellt NMAKE die erste abhängige Datei aus der Liste - in unserem Falle HAUPT.EXE, genau das, was ich haben will.

NMAKEs wichtigste Funktion ist, festzustellen, ob eine abhängige Datei "überholt" (*out of date*) ist. Eine Datei ist überholt, wenn

- sie nicht existiert,
- sie älter ist als mindestens eine der Dateien, von denen sie abhängig ist oder
- eine der Dateien, von denen sie abhängig ist, überholt ist.

Die letzte Bedingung kann natürlich nur erfüllt werden, wenn das zu prüfende Programm überhaupt von Dateien abhängig ist. `TOOLS1.OBJ` wäre überholt, wenn es älter als `TOOLS1.BAS` oder `TOOLS.BI` wäre oder wenn es selbst nicht existierte - aber es könnte niemals überholt sein, weil `TOOLS1.BAS` oder `TOOLS.BI` selbst überholt sind, denn diese beiden sind keine abhängigen Dateien und deshalb niemals überholt.

Angenommen, das oben beschriebene Programmsystem existiert bereits (mit allen `OBJ`- und `LIB`-Dateien), und Sie ändern jetzt etwas an `TOOLS3.BAS`. Dann arbeitet NMAKE wie folgt:

```
Ist HAAPT.EXE überholt?
  Ist HAAPT.OBJ überholt?
    Ist HAAPT.OBJ älter als HAAPT.BAS?
      NEIN
    Nein, HAAPT.OBJ ist noch aktuell.
  Ist TOOLS.LIB überholt?
    Ist TOOLS1.OBJ überholt?
      Ist TOOLS1.OBJ älter als TOOLS1.BAS oder TOOLS.BI?
        NEIN
      Nein, TOOLS1.OBJ ist noch aktuell.
    Ist TOOLS2.OBJ überholt?
      Ist TOOLS2.OBJ älter als TOOLS2.BAS oder TOOLS.BI?
        NEIN
      Nein, TOOLS2.OBJ ist noch aktuell.
    Ist TOOLS3.OBJ überholt?
      Ist TOOLS3.OBJ älter als TOOLS3.BAS oder TOOLS.BI?
        JA
      Ja, TOOLS3.OBJ ist überholt.
      Erstelle TOOLS3.OBJ neu!
      BC TOOLS3 ;
      TOOLS3.OBJ ist nun aktuell.
  Ja, TOOLS.LIB ist überholt.
  Erstelle TOOLS.LIB neu!
  IF EXIST TOOLS.LIB DEL TOOLS.LIB
  LIB TOOLS +TOOLS1+TOOLS2+TOOLS3 ;
  TOOLS.LIB ist nun aktuell.
Ja, HAAPT.EXE ist überholt.
Erstelle HAAPT.EXE neu!
LINK HAAPT, , , TOOLS ;
HAAPT.EXE ist nun aktuell.
```

Auf diese Weise führt NMAKE nur alle Befehle durch, die wirklich notwendig sind, um die Änderungen wirksam zu machen.

## 22.4 NMAKE-Programmierdetails

### Ableitungsregeln

Betrachten wir uns noch einmal die NMK-Datei:

```
HAUPT.EXE : TOOLS.LIB HAUPT.OBJ
    LINK HAUPT, , ,TOOLS;
TOOLS.LIB : TOOLS1.OBJ TOOLS2.OBJ TOOLS3.OBJ
    IF EXIST TOOLS.LIB DEL TOOLS.LIB
    LIB TOOLS +TOOLS1+TOOLS2+TOOLS3;
TOOLS1.OBJ : TOOLS1.BAS TOOLS.BI; BC TOOLS1;
TOOLS2.OBJ : TOOLS2.BAS TOOLS.BI; BC TOOLS2;
TOOLS3.OBJ : TOOLS3.BAS TOOLS.BI; BC TOOLS3;
HAUPT.OBJ : HAUPT.BAS; BC HAUPT;
```

In der letzten Zeile steht ein Standard-Befehl, der bei größeren Systemen bestimmt sehr häufig vorkommt. Es ist der Befehl zum Umwandeln eines .BAS-Files in ein .OBJ-File unter Verwendung des Compilers.

Sie können einen solchen Befehl, der aus einem File mit einer bestimmten Extension ein File mit gleichem Namen, aber anderer Extension herstellt, abstrahieren und als Ableitungsregel schreiben:

```
.BAS.OBJ :
    BC $<;
```

Die obere Zeile muß mit einem Punkt beginnen und dann, wieder durch einen Punkt getrennt, die Ausgangs- und die Zielextension enthalten. Sie endet mit einem Doppelpunkt, und in den Folgezeilen können die Befehle angegeben werden, die zur Umwandlung benötigt werden (mindestens ein Leerzeichen eingerückt). Dabei steht das Spezialmakro \$< für den jeweiligen Dateinamen. Während des Ablaufs ersetzt NMAKE die beiden Zeichen durch den Namen des Programms, das gerade umgewandelt werden soll.

In unserem Fall könnte ich, wenn die oben genannte Ableitungsregel definiert ist, als letzte Zeile schreiben:

```
HAUPT.OBJ : HAUPT.BAS
```

Der Befehl BC müßte nicht mehr explizit angegeben werden.

Eine solche Ableitungsregel hat nicht die Folge, daß jede Datei mit der angegebenen Zielextension automatisch von der gleichnamigen Datei mit der angegebenen Ausgangsextension abhängig ist. Abhängigkeiten müssen nach wie vor

definiert werden, nur die Umwandlungsbefehle kann man sich sparen. Ausnahme hiervon ist der .SUFFIXES-Befehl, der später erläutert wird; enthielte unser NMK-Programm die Zeile ".SUFFIXES .BAS" zusammen mit der oben genannten Ableitungsregel, würde als letzte Zeile auch einfach "HAUPT.OBJ : " genügen.

Der Nachteil dieser Ableitungsregeln ist, daß Sie nur 1:1-Abhängigkeiten dadurch ausdrücken können; für die drei Zeilen, die die Abhängigkeit der TOOLS?.OBJ-Dateien beschreiben, würde die Regel nicht ausreichen, weil jede TOOLS?.OBJ-Datei von zwei Dateien (dem TOOLS.BI-File und der jeweiligen BAS-Datei) abhängig ist.

Außerdem werden Sie verschiedene Programmteile oft auch mit verschiedenen Compiler-Schaltern kompilieren wollen (zum Beispiel einen Teil mit /X, einen anderen ohne). Auch dafür können die Ableitungsregeln kaum herhalten.

## Sonderzeichen

Da die Zeichen (, ), \$, , ^, \, !, @, - und # in NMAKE unter Umständen spezielle Bedeutungen haben, müssen sie, wenn sie in einem gewöhnlichen Text oder in einem Dateinamen vorkommen, mit einem ^-Zeichen eingeleitet werden. Der Dateiname TEST^#1.EXE würde von NMAKE also als TEST#1.EXE interpretiert, während TEST#1.EXE zu einem Fehler führen würde.

## Wildcards

Durch die Verwendung der Bezeichnung TOOLS?.BAS sind wir schon beim nächsten Punkt angelangt: Wenn es schon nicht gelingt, die drei fast gleichen TOOLS-Zeilen durch eine Ableitungsregel überflüssig zu machen, vielleicht kann man sie dann wenigstens zu einer zusammenfassen? Ja. Die Lösung lautet

```
TOOLS?.OBJ : $*.BAS TOOLS.BI; BC $*;
```

Diese Zeile ersetzt exakt die drei alten Zeilen

```
TOOLS1.OBJ : TOOLS1.BAS TOOLS.BI; BC TOOLS1;  
TOOLS2.OBJ : TOOLS2.BAS TOOLS.BI; BC TOOLS2;  
TOOLS3.OBJ : TOOLS3.BAS TOOLS.BI; BC TOOLS3;
```

und zeigt auch gleich die Verwendung eines zweiten Spezialmakros, nämlich \$\*. Die Zeichen \$\* werden von NMAKE durch den Namen des gerade bearbeiteten abhängigen Files - ohne Extension - ersetzt. Ein drittes solches Makro ist \$@, das sich von \$\* nur dadurch unterscheidet, daß es die Extension beibehält.

Die Zeile mit TOOLS?.OBJ wird - mit allen Befehlen, die sie betreffen - auf sämtliche TOOLS?.OBJ-Dateien angewendet. Wenn Sie mehrere TOOLS?.OBJ-Dateien haben, aber nur 1, 2 und 3 verwenden wollen, müssen Sie die Dateien einzeln auflisten:

```
TOOLS1.OBJ TOOLS2.OBJ TOOLS3.OBJ : $*.BAS TOOLS.BI; BC $*;
```

Wenn aber nur die drei genannten TOOLS?.OBJ-Dateien überhaupt vorhanden sind, sind beide Zeilen gleichwertig.

## Makros

Makros bei NMAKE sind so etwas wie String-Variablen. Sie können in einem NMK-File Makros mit dem Befehl `makroname = string` definieren und jederzeit mit `$(makroname)` abrufen. Überall dort, wo im NMK-File `$(makroname)` vorkommt, wird stattdessen der String eingesetzt, der für das Makro definiert wurde.

Aber nicht nur die in der NMK-Datei mit dem Gleichheitszeichen definierten Makros sind vorhanden. Es gibt einige Spezialmakros, von denen ich schon drei behandelt habe; dann können Makros auch in einer Initialisierungsdatei oder über die NMAKE-Befehlszeile gesetzt werden, und schließlich sind auch alle DOS-Umgebungsvariablen als Makros verfügbar. `$(LIB)` würde also zum Beispiel durch den unter DOS mit `SET LIB=text` definierten String ersetzt.

## Die wichtigsten Spezialmakros

Makro	Bedeutung
<code>\$@</code>	Der volle Name der abhängigen Datei, die gerade erzeugt wird
<code>\$*</code> und <code>\$(@R)</code>	Der Name der abhängigen Datei, die gerade erzeugt wird, ohne Extension
<code>\$(@D)</code>	Der Name des Directories der abhängigen Datei, die gerade erzeugt wird (nur, wenn diese im NMK-File mit Directorybezeichnung angegeben ist)
<code>\$(@F)</code>	Der Name der abhängigen Datei, die gerade erzeugt wird, ohne Directory
<code>\$(@B)</code>	Der Name der abhängigen Datei, die gerade erzeugt wird, ohne Extension und Directory
<code>\$\$\$</code>	Die Liste aller Dateien, von denen die gerade bearbeitete abhängt (durch Leerzeichen getrennt)  Wenn der Zeile beziehungsweise dem Befehl, in dem <code>\$\$\$</code> vorkommt, ein Ausrufungszeichen vorangeht, wird für <code>\$\$\$</code> jeweils nur der Name <i>eines</i> abhängigen Files eingesetzt, die Zeile wird aber für <i>jedes</i> abhängige File einmal wiederholt.
<code>\$?</code>	Wie <code>\$\$\$</code> , enthält aber nur die Dateien, die überholt sind
<code>\$&lt;</code>	(nur bei Ableitungsregeln) der Name der Datei, die nach der Ableitungsregel erzeugt werden soll, ohne Extension

Die Spezialmakros `$$$` und `$?` können, ebenso wie das Makro `$@`, für das die Einschränkungen oben beschrieben sind, mit den Buchstaben R, D, F oder B ergänzt werden, um die Dateinamen einzuschränken. Dann muß das Makro allerdings in Klammern eingeschlossen werden (also zum Beispiel `$(**R)`).



# Makrodefinitionen und Ableitungsregeln in TOOLS.INI

Eine Datei namens TOOLS.INI, die entweder im aktuellen Directory oder in dem durch die DOS-Betriebssystemvariable INIT angegebenen Verzeichnis stehen muß, wird von NMAKE nach einer Zeile durchsucht, die [NMAKE] lautet. Alles, was nach dieser Zeile bis zum nächsten Wort in eckigen Klammern folgt, betrachtet NMAKE als Befehle. Sie können in diese TOOLS.INI-Datei Makrodefinitionen, Ableitungsregeln und den .SUFFIXES-Befehl (siehe weiter unten) eintragen.

## Unechte Abhängigkeiten

Ich habe in meinem Beispiel bisher nur die gewöhnliche Abhängigkeitsdefinition

```
zieldatei : quelldatei; befehle
```

verwandt. Dadurch, daß man *quelldatei* oder *befehle* weglassen kann, eröffnen sich interessante Möglichkeiten der Ablaufsteuerung einer NMK-Datei.

Wenn Sie zum Beispiel ein Programmsystem haben, das aus drei EXE-Dateien besteht, und Sie für jede Datei in einem NMK-File definieren, wie sie erstellt wird, müßten Sie NMAKE danach für jede der drei NMK-Dateien einzeln aufrufen. Vereinen Sie alle Beschreibungen in einem einzelnen NMK-File, müßten Sie zumindest auf der Befehlszeile die Namen der drei EXE-Dateien angeben, damit NMAKE weiß, was es zu tun hat.

Nun können Sie aber einfach auch als erste Zeile im NMAKE-File schreiben:

```
MACHE : PROGRAMM.EXE DATEN.EXE BILD.EXE
```

Vorausgesetzt, daß die drei genannten EXE-Programme die zu erzeugenden sind und daß ihre Erzeugung weiter unten im NMK-File beschrieben ist, wird NMAKE sich nun daran machen, das Programm "MACHE" zu erzeugen (standardmäßig wird ja immer das Programm erzeugt, das als erstes in der NMK-Datei beschrieben ist). Dazu prüft es zunächst die angegebenen EXE-Dateien und erzeugt sie neu, falls sie überholt sind. Wenn das geschehen ist, ist NMAKE fertig - ein Programm MACHE wird nie erzeugt, da Sie keinen Befehl dafür angegeben haben.

Sie könnten zum Beispiel als oberste Zeile unseres Beispiel-Files einfügen:

```
MACHE : HAUPT.EXE LISTE
```

und ans Ende:

```
LISTE : HAUPT.EXE TOOLS.LIB; !DIR $**
```

Dann würde das Programm, wenn HAUPT.EXE wie bisher erzeugt wurde, versuchen, LISTE zu erzeugen, und in diesem Zusammenhang würde ein Verzeichnis von HAUPT.EXE und TOOLS.LIB angezeigt.

# Die NMAKE-Ablaufsteuerung

NMAKE besitzt einige Programmierbefehle, die in der Hauptsache zur Ablaufsteuerung geeignet sind. Der Makro-Zuweisungsbefehl wurde bereits erwähnt; er besteht nur aus einem Gleichheitszeichen, links davon der Makroname, rechts der Text. Ein Makro wird - auch das wurde schon erwähnt - mit  $\$(makroname)$  in den Text eingefügt.

Eine erweiterte Version des Makro-Einfügebefehls lautet  $\$(makroname:altstring=neuststring)$ . Hier wird nicht das Original-Makro *makroname* eingefügt, sondern eine veränderte Version, in der jedes Auftreten von *altstring* im genannten Makro durch *neuststring* ersetzt wird. Das Original-Makro ändert sich dadurch nicht. Diese modifizierende Makro-Einfügung kann auch mit den oben beschriebenen Spezialmakros verwendet werden. Durch den Zusatz eines Bindestriches zu einem beliebigen Programmaufruf innerhalb NMAKE (zum Beispiel -LINK statt LINK) können Sie verhindern, daß NMAKE sofort abbricht, wenn dieses Programm einen Fehler meldet. Ein vor den Programmnamen gesetztes @-Zeichen verhindert, daß der Programmaufruf angezeigt wird, während NMAKE läuft.

Weitere nützliche NMAKE-Befehle sind:

!IF <i>ausdruck</i>	Führt alle Befehle bis zum !ENDIF oder !ELSEIF nur aus, wenn der angegebene Ausdruck wahr beziehungsweise ungleich 0 ist. Als Ausdruck sind Vergleiche (mit den Operatoren ==, <, >, <= und >=) erlaubt sowie ! für logisches NICHT und die Verknüpfungen && für UND und    für ODER.
!ELSE	Führt die folgenden Befehle bis zum !ENDIF aus, wenn die vorhergehenden seit dem !IF nicht ausgeführt wurden.
!IFDEF <i>makroname</i>	Führt die folgenden Befehle bis zum !ENDIF aus, wenn <i>makroname</i> definiert ist.
!IFNDEF <i>makroname</i>	Führt die folgenden Befehle bis zum !ENDIF aus, wenn <i>makroname</i> nicht definiert ist.
!ENDIF	Beendet !IF, !IFNDEF und !IFDEF.
!UNDEF <i>makroname</i>	Löscht die Definition für <i>makroname</i> .
!ERROR <i>text</i>	Gibt <i>text</i> am Bildschirm aus und bricht NMAKE ab.
!INCLUDE <i>datei</i>	Lädt die angegebene NMAKE-Programmdatei hinzu (beispielsweise, um Makrodefinitionen in einer zweiten Datei speichern zu können).
!CMDSWITCHES <i>option</i>	Schaltet NMAKE-Optionen ein oder aus. <i>option</i> kann ein + oder ein - sein, gefolgt von D, I, N oder S. Die entsprechende Option (siehe "Aufruf von NMAKE" weiter vorne in diesem Kapitel) wird ein- beziehungsweise ausgestellt. !CMDSWITCHES ohne weitere Angaben stellt alle Optionen auf den Ausgangszustand zurück.
.SUFFIXES: <i>liste</i>	Teilt NMAKE mit, daß es, wenn für ein abhängiges File keine unabhängigen angegeben sind, prüfen soll, ob das angegebene abhängige File eventuell mittels einer Ableitungsregel aus einem gleichnamigen File mit einer Extension aus der angegebenen <i>liste</i> erzeugt werden kann.

# Steuerungsdateien

Wie Sie den entsprechenden Kapiteln entnehmen können, kann man sowohl LINK als auch LIB mit einer sogenannten Steuerungsdatei benutzen, die alle Informationen enthält, die das jeweilige Programm zum Arbeiten braucht. Man benutzt diese Steuerungsdateien vornehmlich, um besser die Übersicht zu behalten und weniger Schreibarbeit zu haben. Aber es kann auch noch einen weiteren Grund dafür geben. DOS erlaubt bekanntlich nur die Eingabe von 128 Zeichen langen Zeilen. Gerade bei LINK kann es häufig vorkommen, daß ein Aufruf dieses Limit überschreitet, wenn er die Namen mehrerer OBJ- und LIB-Dateien enthält. Mit einer Steuerungsdatei kann man dieses Limit umgehen, da man in der Lage ist, die Befehlszeile in mehrere Zeilen aufzuteilen.

NMAKE gibt dem Programmierer die Möglichkeit, temporäre Steuerungsdateien zu erzeugen. NMAKE übernimmt die Erzeugung, Namensgebung und Löschung dieser Dateien; man muß nur in der Beschreibungsdatei den Inhalt einfügen.

Statt

```
LINK TEST+HILFE+GRAPHIK+ED , , ,UITBEFR+CHRTBEFR+FONTBEFR ;
```

kann man, wie im LINK-Kapitel erläutert, auch eine Datei namens TEST.CMD erstellen, die folgendes enthält (Zeilennummern weglassen)

```
(1) TEST+HILFE+GRAPHIK+ED
(2)
(3)
(4) UITBEFR+CHRTBEFR+FONTBEFR ;
```

und dann eingeben

```
LINK @TEST.CMD
```

Ein entsprechender Teil aus einer NMAKE-Beschreibungsdatei sähe vielleicht so aus:

```
TEST.EXE : HILFE.OBJ GRAPHIK.OBJ ED.OBJ
    LINK TEST+HILFE+GRAPHIK+ED , , ,UITBEFR+CHRTBEFR+FONTBEFR ;
```

und könnte folgendermaßen umformuliert werden:

```
TEST.EXE : HILFE.OBJ GRAPHIK.OBJ ED.OBJ
    LINK @<<
TEST+GRAPHIK+ED
```

```
UITBEFR+CHRTBEFR+FONTBEFR ;
<<
```

Sie fügen an der Stelle, wo gewöhnlich der Name der Steuerungsdatei stehen müßte (also hinter dem @ bei LINK), zwei Kleiner-Zeichen ein. Ab der nächsten Zeile beginnen Sie dann mit dem Inhalt ihrer Steuerungsdatei. Wenn diese kom-

plett ist, teilen Sie das NMAKE wieder mit zwei Kleiner-Zeichen auf einer neuen Zeile mit.

NMAKE ermittelt automatisch einen geeigneten Namen für die temporäre Datei, erzeugt sie mit dem angegebenen Inhalt, ruft LINK mit ihrem Namen auf und löscht sie danach wieder.

Wenn Sie einen bestimmten Namen für die temporäre Datei verwenden wollen, können Sie hinter den ersten beiden Kleiner-Zeichen diesen Namen angeben, also zum Beispiel

```
LINK @<<STEUER.CMD
...
...
<<
```

Wenn Sie möchten, daß die Datei nicht gelöscht wird, nachdem der Aufruf erfolgte, können Sie hinter die letzten beiden Kleiner-Zeichen den Befehl KEEP setzen, also zum Beispiel

```
LINK @<<STEUER.CMD
...
...
<<KEEP
```

In dem Bereich zwischen den beiden <<-Kombinationen, der den Text für die temporäre Datei enthält, können auch Makro-Einsetzungen enthalten sein, also zum Beispiel \$(LIB), wenn LIB ein definiertes Makro ist.

## 22.5 Der NMAKE-Zwilling NMK.COM

Mit dem BASIC PDS 7.1 wird auch NMK.COM mitgeliefert, ein Programm, das NMAKE sehr ähnlich ist und von der PWB benutzt wird.

NMK.COM hat gegenüber dem Original-NMAKE den Vorteil, daß es den aufgerufenen Programmen mehr vom Speicherkuchen übrig läßt, so daß ein LINK, das aus NMK heraus aufgerufen wurde, zum Beispiel schneller arbeiten kann als ein aus NMAKE heraus aufgerufenes.

Unter OS/2 brauchen Sie sich um solche Dinge nicht zu kümmern, dort steht ja meist genügend Speicher zur Verfügung. Unter DOS kann NMK allerdings zuweilen nützlich sein.

Der Nachteil von NMK ist, daß es sich nicht so viel Mühe mit der Prüfung gibt, ob ein Programm nun überholt ist oder nicht. Ein Beispiel hierfür:

```
HAUPT.EXE : TEST1.OBJ TEST2.OBJ TEST3.OBJ HAUPT.OBJ
LINK HAUPT+TEST1+TEST2+TEST3;
```

NMAKE würde, wenn es HAUPT.EXE erstellen soll, zunächst prüfen, ob TEST1.OBJ aktuell ist. Wenn nicht, würde es erstellt werden. Dann würde geprüft, ob TEST2.OBJ aktuell ist usw.

NMK würde hier zuerst für die Dateien TEST1.OBJ, TEST2.OBJ, TEST3.OBJ und HAUPT.OBJ testen, ob sie aktuell sind. Wenn nicht, werden sie in einer Liste vermerkt, und wenn alle vier Dateien überprüft sind, werden die, die nicht aktuell waren, neu erstellt.

Nehmen wir an, außer TEST1.OBJ sind alle OBJ-Dateien aktuell. Beide Programme, NMK und NMAKE, würden dann beginnen, TEST1.OBJ neu zu erstellen. Angenommen, dabei würde - weil Sie halt ein ziemlich künstlerisches NMK-File erstellt haben - TEST3.OBJ gelöscht. Dann würde NMAKE das später noch feststellen und TEST3.OBJ ebenfalls neu erstellen, während NMK, das in seiner Liste ja schon stehen hat "TOOLS3.OBJ ist o.k.", sich darum nicht mehr kümmern könnte.

Sonst gibt es zwischen den beiden Programmen keinen Unterschied. (Genaugenommen sind es gar nicht zwei Programme, denn NMK ist nur eine "Shell", die dann selbst NMAKE aufruft. Trotzdem ist das NMK-Verfahren sparsamer mit dem Speicherplatz.)



# 23 MKKEY - Veränderung der Tastaturdefinition

Falls eine der Tastenbelegungen in QBX Ihrer Gewohnheit widerspricht oder auf Ihrer Tastatur allzu arge Verrenkungen erfordert, können Sie mit MKKEY die Zuordnung von Funktionen zu Tasten in QBX ändern.

MKKEY ist ein Konvertierungsprogramm, mit dem Sie entweder eine Tastaturdefinitionsdatei in einen gewöhnlichen ASCII-Text verwandeln können oder einen Text in eine Definitionsdatei.

Wenn Sie Veränderungen vornehmen möchten, gehen Sie am besten von einer der vordefinierten Dateien aus. Diese sind

QBX.KEY	für die Standard-QBX-Tastenbelegung,
ME.KEY	für eine dem Microsoft Editor nachempfundene Tastenbelegung,
BRIEF.KEY	für eine Tastenbelegung, die mit dem BRIEF-Editor vergleichbar ist und
EPSILON.KEY	für die Tastenbelegung des EPSILON-Editors.

Konvertieren Sie eine dieser Dateien mit dem Befehl

```
MKKEY -c ba -i definitionsdatei -o textdatei
```

(wobei Sie für *definitionsdatei* den Namen derselben einsetzen und für *textdatei* den Namen des Textes, der erzeugt werden soll).

Dann können Sie mit einem beliebigen Texteditor oder auch mit Hilfe von QBX den entstandenen Text verändern.

Damit Sie den Text später wieder in ein brauchbares Tastaturdefinitionsfile konvertieren können, muß jede Zeile im Text die Syntax

```
funktion : taste
```

haben. Dadurch wird einer bestimmten Taste eine Funktion zugeordnet. Auf Groß- und Kleinschreibung kommt es dabei nicht an. Sie können die gleiche Funktion verschiedenen Tasten zuordnen, aber niemals einer Taste mehrere Funktionen.

Als *taste* sind erlaubt: Left, Right, Up, Down (die Pfeiltasten); Home, End, PgUp, PgDn, Ins, Del (die restlichen Tasten des Nummernblocks); Esc, F1 bis F12, Tab, Backspace und außerdem sämtliche Buchstaben, Ziffern und Zeichen, die auf der Tastatur vorhanden sind.

Außerdem können auch Tastenkombinationen mit CTRL gebildet werden, zum Beispiel CTRL+Right oder CTRL+W. Kombinationen mit ALT sind zwar auch möglich, geraten jedoch in Konflikt mit den QBX-Menüsystem und sind deshalb nicht sinnvoll.

Schließlich gibt es mit sämtlichen CTRL-Kombinationen noch die Möglichkeit, ihnen nach Wordstar-Manier eine Art von Menüfunktion zuzuordnen. Wenn Sie als Taste zum Beispiel "CTRL+E I" angeben, leuchtet später in QBX beim Drücken von CTRL E in der untersten Bildschirmzeile die Anzeige ^E auf; wenn nun ein I gedrückt wird, wird die Funktion ausgeführt, wird eine andere Taste gedrückt, für die keine mit "CTRL+E taste" definierte Funktion vorhanden ist, passiert nichts weiter.

Folgende Funktionen können den Tasten zugeordnet werden:

Funktionsname	Wirkung
Backspace	Löscht das Zeichen links vom Cursor.
Beep	Ein Signalton ertönt (eine sinnlose Funktion, denn dieser ertönt auch, wenn eine Taste nicht definiert ist).
BegLine	Setzt den Cursor auf die erste Spalte in der aktuellen Zeile.
BegPgm	Setzt den Cursor auf das erste Zeichen im aktuellen Fenster.
Cancel	Schließt Menüs, das Hilfsfenster und wählt "Cancel" in Dialogboxen.
Cancel2	Schließt nur Menüs.
Change	Hat dieselbe Funktion wie die Auswahl von "Change" aus dem "Search"-Menü.
CharLeft	Setzt den Cursor um eine Position nach links.
CharRight	Setzt den Cursor um eine Position nach rechts.
Copy	Wie "Copy" aus dem "Edit"-Menü.
Cut	Wie "Cut" aus dem Edit-Menü.
Del	Löscht den markierten Block beziehungsweise das Zeichen, auf dem der Cursor steht, wenn kein Block markiert ist.
DelWord	Löscht das Wort, auf dem der Cursor steht.
DoEsc	Löscht die Markierung eines Textes.
DoQuoteCharacter	Fügt das nächste Zeichen in den Programmtext ein, ohne zu prüfen, ob der Taste eine bestimmte Funktion zugeordnet ist.
DoTab	Rückt den markierten Text um eine Tabulatorposition ein oder - mit Shift - aus.
EndLine	Setzt den Cursor auf das letzte Zeichen der Zeile.
EndPgm	Setzt den Cursor auf das erste Zeichen der letzten Zeile im aktuellen Fenster.
EndScn	Setzt den Cursor auf die unterste im Augenblick sichtbare Zeile.
EraseEol	Löscht alles von der Cursorposition bis zum Ende der Zeile.
Find	Wie "Find" aus dem "Search"-Menü.
GotoBookmarkX	(für X 0-3 einsetzen) Setzt den Cursor auf die Markierung Nr. X.
HomeLine	Wie BegLine, ignoriert jedoch Leerzeichen.
HomeScn	Setzt den Cursor auf die oberste gerade sichtbare Zeile.

*(Fortsetzung nächste Seite)*



IgnoreChar	Eine Dummy-Funktion. Sie tut gar nichts, aber durch sie können Sie eine Taste "definieren", damit QBX nicht dauernd piept, wenn Sie diese Taste drücken.
KillLine	Löscht die Zeile, in der der Cursor steht, kopiert sie aber zuvor ins Clipboard.
LineDown	Bewegt den Cursor eine Zeile abwärts.
LineUp	Bewegt den Cursor eine Zeile aufwärts.
Menu	Aktiviert die Menüzeile.
Menu2	Wie Menu.
NewLine	Teilt die Zeile an der Cursorposition entzwei; alle Zeichen ab der Cursorposition werden in die nächste Zeile gesetzt und um so viele Spalten eingerückt wie die aktuelle Zeile.
NextLine	Wie LineDown, setzt den Cursor jedoch an den Zeilenanfang der nächsten Zeile.
PageDown	Blättert im aktuellen Fenster eine Seite abwärts.
PageLeft	Blättert im aktuellen Fenster eine Seite nach links.
PageRight	Blättert im aktuellen Fenster eine Seite nach rechts.
PageUp	Blättert im aktuellen Fenster eine Seite aufwärts.
ResetState	Entfernt die Markierung vom Text (wie DoEsc).
ScrollDown	Blättert eine Zeile abwärts.
ScrollUp	Blättert eine Zeile aufwärts.
SearchNext	Wie "Repeat Last Find" aus dem "Search"-Menü.
SetBookmarkX	(X durch 0-3 ersetzen) Setzt die Markierung Nr. X.
SplitLine	Wie NewLine, der Cursor bleibt aber, wo er ist.
ToggleInsertMode	Schaltet zwischen Einfüge- und Überschreibmodus hin und her.
Undo	Wie "Undo" aus dem "Edit"-Menü.
WordLeft	Springt auf den Anfang des Wortes, auf dem der Cursor steht, beziehungsweise des Wortes links davon, wenn der Cursor schon am Anfang eines Wortes ist.
WordRight	Springt auf den Anfang des Wortes rechts von dem, auf dem der Cursor steht.

Wenn Sie schließlich eine Textdatei erzeugt haben, die die von Ihnen gewünschte Tastenbelegung enthält, müssen Sie diese zunächst mit dem Befehl

```
MKKEY -c ab -i textdatei -o definitionsdatei
```

wieder in das Format einer gültigen QBX-Tastaturdefinitionsdatei bringen. Dabei meldet MKKEY, wenn Sie ungültige Funktions- oder Tastenbezeichnungen benutzt haben, und außerdem gibt es Warnungen aus, wenn Sie bestimmten Funktionen keine Taste zugeordnet haben. Diese Funktionen sind in der Regel dann in QBX nicht benutzbar, wenn sie nicht noch über ein Menü zugänglich sind.

Dann können Sie QBX aufrufen mit:

```
QBX /k:definitionsdatei
```

QBX speichert den Namen der angegebenen Definitionsdatei in QBX.INI, so daß Sie ihn in Zukunft nicht mehr angeben müssen; wohl aber muß die Definitionsdatei für QBX zugänglich sein. Ihre neuen Tastenbelegungen sind nun aktiv.

Es gibt auch Tastenbelegungen, die Sie nicht verändern können. Dazu zählen all diejenigen, die schon in den QBX-Menüs erwähnt sind, wie ALT Backspace für Undo. Sie können dann zwar anderen Tasten diese Funktion auch noch zuordnen, nicht aber der Tastenkombination ALT Backspace eine andere.

---

## **Sektion    Referenzteil VII**

---

- **Standard-BASIC-Befehle  
und Funktionen**
  - **ISAM**
  - **Date-Library**
  - **Format-Library**
  - **Finance-Library**
  - **Matrizenmathematik-  
Toolbox**
  - **Font-Toolbox**
  - **Presentation  
Graphics-Toolbox**
  - **User Interface-  
& General-Toolbox**
-



# Referenzteil Standard-BASIC

Gleich zu Anfang eine Übersicht, welche Befehle und Funktionen mit BASIC 7.0 oder 7.1 PDS neu hinzugekommen sind oder eine Änderung erfahren:

CALL .....	277
CCUR .....	281
CHDRIVE .....	282
CURDIR\$ .....	291
CVC .....	291
DECLARE .....	294
DIR\$ .....	300
END .....	305
ERR .....	308
EVENT ON/EVENT OFF .....	309
FRE .....	316
\$INCLUDE .....	322
MKC\$ .....	343
ON ERROR .....	345
REDIM .....	370
SETMEM .....	389
SHELL (Funktion) .....	391
SSEG .....	395
SSEGADD .....	396
STACK .....	396
STOP .....	400
SUB/FUNCTION .....	402
SYSTEM .....	406
TYPE .....	410

Seit wann ein Befehl unverändert existiert, ist bei jedem Eintrag angegeben; es gibt hier die Kategorien "QuickBASIC 2.0" (und früher), "QuickBASIC 3.0", "QuickBASIC 4.0", "BASIC 6.0", "BASIC 7.0 PDS" und "BASIC 7.1 PDS".

Die 190 Einträge in diesem Referenzteil sind von mir willkürlich in Gruppen eingeteilt. Diese Gruppen lauten:

Gruppe	Inhalt
DOS	Alle Befehle, die Standard-DOS- beziehungsweise OS/2-Befehle nachahmen, wie zum Beispiel CHDIR
OS/2	Befehle, die nur unter OS/2 verfügbar sind (keine Sorge, es sind nur 2)
Struktur	Befehle wie IF...END IF, die die Programmkontrollstruktur bilden
Grafik	Grafikbefehle und -funktionen
Speicher	Befehle, die zur direkten Manipulation des RAM-Speichers dienen
I/O	Befehle, die sich mit dem Datenaustausch mit externen Datenträgern und / oder Geräten befassen
Trapping	Alles, was mit Event- oder Error-Trapping zu tun hat
Standard	übrige Befehle

Außerdem gibt es eine zweite Einteilung, die die Gruppen "Befehl", "Funktion", "Metabefehl" und "Systemvariable" vorsieht.

Die Zugehörigkeit zu beiden Gruppen ist im Kopf jedes Eintrags angegeben.

Die Sortierung ist trotz der Gruppeneinteilung alphabetisch.

## ABS (Funktion)

Standard

*Seit* QuickBASIC 2.0

*Anwendung*  $x = \text{ABS}(y)$

*Nutzen* Ermittelt den Betrag seines Arguments.  $\text{ABS}(y)$  ist äquivalent zu  $y * \text{SGN}(y)$ .

*Siehe auch* SGN (390).

## ASC (Funktion)

Standard

*Seit* QuickBASIC 2.0

*Anwendung*  $x = \text{ASC}(y\$)$

*Nutzen* Gibt den ASCII-Wert (0 bis 255) des ersten Zeichens des String-Arguments zurück. *Illegal function call* bei Leerstring. Eine Tabelle aller ASCII-Zeichen finden Sie in Anhang D.3.

*Siehe auch* CHR\$ (282).

## ATN (Funktion)

Standard

*Seit* QuickBASIC 2.0

*Anwendung*  $x = \text{ATN}(y)$

*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.

*Bemerkung* » Die ATN-Funktion benutzt die Mathematik-Libraries (siehe Kapitel 19).

*Siehe auch* SIN (393), COS (290), TAN (407).

## BEEP (Befehl)

Standard

*Seit* QuickBASIC 2.0

*Anwendung* BEEP

*Nutzen* Erzeugt einen Warnton, dessen Dauer und Frequenz von Computer zu Computer unterschiedlich sind. BEEP ist äquivalent zu `PRINT CHR$(7)`.

*Siehe auch* SOUND (394), PLAY (359).

## BLOAD (Befehl)

I/O, Speicher

*Seit* QuickBASIC 2.0

*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* » Da BASIC Variablen (ausgenommen statische Arrays, siehe in Kapitel 12.4) im Speicher verschieben kann, muß man vorsichtig mit BLOAD operieren, um nicht versehentlich Programmcode oder andere Daten zu überschreiben. BLOAD darf nicht benutzt werden, um Arrays im Expanded Memory zu laden.

*Siehe auch* BSAVE (276).

## BSAVE (Befehl)

I/O, Speicher

*Seit* QuickBASIC 2.0

*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* » QBX speichert sämtliche Arrays - bis auf Arrays von Strings mit variabler Länge - 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 QBX ohne den /Ea-Switch, dann werden keine Arrays in das Expanded Memory ausgelagert.

» BASIC kann Variablen zuweilen im Speicher verschieben (ausgenommen statische Arrays, siehe "Statische und dynamische Felder" in Kapitel 12.4), so daß Sie, wenn Sie Variablen und Arrays speichern oder laden, auf jeden Fall immer unmittelbar vorher die



Adresse ermitteln sollten, selbst wenn irgendeine frühere Adresse noch bekannt ist.

*Beispiel:* Diese Programmfragmente speichern beziehungsweise laden ein ganzes Array:

```
DIM Array(1 TO 100, 1 TO 5) AS SINGLE

...

' Speichern
' laenge% bei BSAVE ist 2000, da das Array
' genau 2000 Bytes benötigt: 100 * 5 Elemente zu
' je 4 Bytes. Weil Arrays kontinuierlich im
' Speicher abgelegt werden, ist die Adresse des
' ersten Array-Elements zugleich Adresse des
' Arrays!

DEF SEG = VARSEG(Array(1, 1))
BSAVE "ARRAY.DAT", VARPTR(Array(1, 1)), 2000

...

' Laden
DEF SEG = VARSEG(Array(1,1))
BLOAD "ARRAY.DAT", VARPTR(Array(1,1))
```

*Siehe auch* BLOAD (276), DEF SEG (297), VARPTR (413), VARSEG (413).

## CALL (Befehl)

**Standard**

*Seit* BASIC 7.1 PDS

*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. Bei der herkömmlichen, für QuickBASIC 1.0 bis 3.0 gebräuchlichen Syntax (1) wird das Befehlswort CALL angegeben, und die Argumente müssen, wenn vorhanden, in runde Klammern eingeschlossen werden. Seit QuickBASIC 4.0 ist die Syntax (2) erlaubt, mit der man eigene Routinen genauso aufruft wie eingebaute BASIC-Befehle.

Die Verwendung der Syntax (2) erfordert allerdings, daß die aufgerufenen Prozeduren zuvor mit DECLARE-Anweisungen deklariert wurden (QBX erledigt das automatisch).

*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. Weder BYVAL noch SEG sind hier erlaubt. Einzelne Array-Elemente können wie normale Variablen übergeben werden. Nicht möglich ist es jedoch, aus einem mehrdimensionalen Array zum Beispiel eine einzelne Spalte o.ä. 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 - oder man benutzt das Schlüsselwort BYVAL, das vor dem betreffenden Parameter angegeben werden muß. BYVAL kann für BASIC-Prozeduren nur verwendet werden, wenn in der Prozedurdefinition mit SUB auch BYVAL angegeben wurde.

» 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 Spei-

cher. 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.

» In Zusammenhang mit CALL sind noch einige Routinen zu erwähnen, die im Lieferumfang von BASIC PDS enthalten sind:

Absolute	CALL Absolute([argumente,] offset%) Diese Routine ist in QBX.LIB beziehungsweise QBX.QLB enthalten und dient zum Aufruf einer Maschinensprache-Routine, die an einer bekannten Adresse im Speicher geladen sein muß (üblicherweise in einem Array). <i>offset%</i> ist die Offsetadresse der Routine (die Segmentadresse wird mit DEF SEG eingestellt). Alle <i>argumente</i> werden der Prozedur als Offsetadressen auf dem Stack übergeben. Da C- und Assembler-Routinen inzwischen ganz normal mit DECLARE und CALL aufgerufen werden können, wird diese Funktion nicht mehr benötigt.
Interrupt	CALL Interrupt(Nummer%, InRegs, OutRegs)
InterruptX	CALL InterruptX(Nummer%, InRegs, OutRegs) Beide Routinen dienen zum Aufruf von Interrupts und sind in den Kapiteln 16.2 und 16.3 genauer beschrieben. Sie sind in QBX.LIB beziehungsweise QBX.QLB enthalten.
Int86Old	CALL Int86Old(Nummer%, InR%(), OutR%())
Int86XOld	CALL Int86XOld(Nummer%, InR%(), OutR%()) Beide Routinen dienen ebenfalls zum Aufruf von Interrupts und sind auch in QBX.LIB beziehungsweise QBX.QLB enthalten. Sie werden nicht genauer beschrieben, da sie nur zur Kompatibilität mit QuickBASIC-Versionen vor 4.0 implementiert sind (sie ersetzen die damals benutzten Prozeduren INT86 und INT86X).
SetUEvent	CALL SetUEvent Diese Prozedur ist ohne zusätzliche Library verfügbar und im Zusammenhang mit gemischtsprachlichem Programmieren und Event-Trapping interessant. Siehe ON <i>event</i> GOSUB.
StringAddress	StringAddress(offset%) Gibt die Far-Adresse des Strings zurück, dessen Deskriptor-Offsetadresse in <i>offset%</i> angegeben ist. Stringdeskriptoren liegen alle in DGROUP, deshalb reicht die Offsetadresse aus, um den Deskriptor zu identifizieren. Einer Routine, die von BASIC aus mit einem String als Argument aufgerufen wird, wird nur die Offsetadresse des Stringdeskriptors übergeben. Sie muß StringAddress benutzen, um die Adresse des Strings zu ermitteln.

(Fortsetzung nächste Seite)

StringAssign	<p><code>StringAssign(quellAdresse&amp;, quellLaenge%, zielAdresse&amp;, zielLaenge%)</code></p> <p>Diese Routine dient dazu, Strings von BASIC in Programmteile anderer Sprachen oder umgekehrt zu übertragen. Sie wird nur von den fremdsprachlichen Programmen benutzt. <i>quellAdresse&amp;</i> und <i>zielAdresse&amp;</i> sind die Far-Adressen des Quell- beziehungsweise Zielstrings (wenn es sich um einen String fester Länge handelt) beziehungsweise die Far-Adressen des String-Deskriptors bei Strings mit variabler Länge. <i>quellLaenge%</i> und <i>zielLaenge%</i> sind die Längen der Strings, wenn es Strings mit fester Länge sind, beziehungsweise 0, wenn es sich um Strings mit variabler Länge handelt, bei denen die Länge ja im Deskriptor steht.</p> <p>Mit Hilfe dieser Prozedur kann zum Beispiel ein C-Programm auf einen String zugreifen, der mit BASIC erzeugt wurde, obwohl die Struktur des Stringdeskriptors nicht bekannt ist.</p>
StringLength	<p><code>StringLength(offset%)</code></p> <p>Diese Prozedur gibt bei Angabe der Offsetadresse des Stringdeskriptors seine Länge zurück.</p>
StringRelease	<p><code>StringRelease(offset%)</code></p> <p>Mit Hilfe dieser Prozedur kann zum Beispiel ein C-Programm auf einen String zugreifen, der mit BASIC erzeugt wurde, obwohl die Struktur des Stringdeskriptors nicht bekannt ist.</p>
StringLength	<p><code>StringLength(offset%)</code></p> <p>Diese Prozedur gibt bei Angabe der Offsetadresse des Stringdeskriptors seine Länge zurück.</p>
StringRelease	<p><code>StringRelease(offset%)</code></p> <p>Diese Routine löscht einen String variabler Länge, der mit StringAssign erzeugt wurde, wenn man ihr die Offsetadresse des Stringdeskriptors übergibt. StringRelease darf nicht für BASIC-Strings benutzt werden.</p>

Alle vier String-Routinen entspringen einzig der Tatsache, daß BASIC Strings mit variabler Länge kennt, die es in den meisten anderen Sprachen nicht gibt, und für diese einen speziellen Stringdeskriptor anlegen muß. Dieser Deskriptor ist unterschiedlich, je nachdem, ob man mit Near oder Far Strings arbeitet, und seine Struktur ist (mir zumindest) nicht bekannt. Daher benutzt man, wenn man mit anderen Sprachen auf BASIC-Strings zugreifen muß, diese vier Routinen. Innerhalb von BASIC hat keine der vier Prozeduren einen Sinn. Anwendungsbeispiele finden Sie in den Microsoft-Beispielprogrammen mit dem Kürzel MX.

*Beispiel* Siehe Beispiel bei SUB/FUNCTION.

*Siehe auch* DECLARE (294), SUB/FUNCTION (402), ON *event* GOSUB (346).

<b>CCUR (Funktion)</b>	<b>Standard</b>
------------------------	-----------------

*Seit* BASIC 7.0 PDS

*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.

*Siehe auch* CINT (283), CDBL (281), CLNG (284), CSNG (290).

<b>CDBL (Funktion)</b>	<b>Standard</b>
------------------------	-----------------

*Seit* QuickBASIC 2.0

*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 A).

*Siehe auch* CINT (283), CCUR (281), CLNG (284), CSNG (290).

<b>CHAIN (Befehl)</b>	<b>Standard</b>
-----------------------	-----------------

*Seit* QuickBASIC 2.0

*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 13.1.)

*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.
- » Wenn auf der Source-Ebene, also aus QBX heraus, andere Programme mit CHAIN aufgerufen werden, werden die entsprechenden BAS-Files geladen. Im kompilierten Zustand erwartet ein CHAIN-Befehl, daß das aufzurufende Programm als EXE vorliegt. Der `dateiname$` darf Pfad- und Laufwerksangaben enthalten.
- » Das Aufrufen von nicht in BASIC geschriebenen EXE-Programmen mit CHAIN ist überhaupt nur dann möglich, wenn man ohne Runtime-Modul (also mit dem Switch /O) arbeitet. Selbst dann besteht diese Möglichkeit nur beschränkt, d.h. viele Programme stürzen ab, wenn sie mit CHAIN aufgerufen werden. Bei Program-

men, die mit Microsoft C 6.0 erstellt wurden, ist der CHAIN-Aufruf absturzsicher.

*Siehe auch* RUN (376), COMMON (288).

## CHDIR (Befehl)

DOS

*Seit* QuickBASIC 2.0

*Anwendung* CHDIR *directoryName\$*

*Nutzen* Der Befehl wechselt das aktuelle Directory, identisch mit dem DOS-Befehl CHDIR oder CD.

*Siehe auch* CURDIR\$ (291), CHDRIVE (282), MKDIR (343), RMDIR (374).

## CHDRIVE (Befehl)

DOS

*Seit* BASIC 7.0 PDS

*Anwendung* CHDRIVE *laufwerk\$*

*Nutzen* Wechselt das aktuelle Laufwerk. Nur der erste Buchstabe von *laufwerk\$* ist relevant, und dieser muß im Bereich von A bis zum höchsten erlaubten Laufwerksbuchstaben liegen, der in CONFIG.SYS mittels `lastdrive = xx` festgelegt wird. CHDRIVE "C" hat exakt dieselbe Wirkung wie die Eingabe von C: in DOS.

*Siehe auch* CHDIR (282), CURDIR\$ (291).

## CHR\$ (Funktion)

Standard

*Seit* QuickBASIC 2.0

*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.3.

*Siehe auch* ASC (275).

<i>Seit</i>	QuickBASIC 2.0
<i>Anwendung</i>	$x\% = \text{CINT}(y)$
<i>Nutzen</i>	Wandelt einen beliebigen numerischen Wert in einen INTEGER-Wert um. Der erlaubte Bereich ist -32.768 bis 32.767.
<i>Bemerkung</i>	» CINT, FIX und INT sind ähnliche Funktionen. Die Unterschiede sind im Abschnitt über INT zusammengefaßt.
<i>Siehe auch</i>	CCUR (281), CDBL (281), CLNG (284), CSNG (290), FIX (313), INT (327).

**CIRCLE (Befehl)****Grafik**

<i>Seit</i>	QuickBASIC 2.0
<i>Anwendung</i>	<pre> CIRCLE [STEP] (x, y), radius,         [, [farbe] [, [von] [, [bis]         [, verhaelt]]]] </pre>
<i>Nutzen</i>	<p>Zeichnet einen Kreis beziehungsweise eine Ellipse oder ein Segment derselben auf den Bildschirm.</p> <p><i>STEP</i> gibt wie bei allen Grafikbefehlen an, daß die Koordinaten relativ vom momentanen Grafikkursor aus zu verstehen sind.</p> <p>(x, y) sind die Koordinaten des Mittelpunkts. Auch für Ellipsen wird nur ein Mittelpunkt angegeben.</p> <p><i>radius</i> ist der Radius für den Kreis beziehungsweise die Ellipse. Er wird in derselben Einheit gemessen wie die Koordinaten; diese ist gewöhnlich ein Pixel, kann aber mit dem WINDOW-Befehl geändert werden.</p> <p><i>farbe</i> ist das Farbattribut, das dem Kreis beziehungsweise der Ellipse zugeordnet werden soll (siehe auch SCREEN, COLOR und PALETTE).</p> <p><i>von</i>, <i>bis</i> sind der Start- und der Endwinkel (im Bogenmaß). Bei Weglassen der Angaben werden 0 und 2Ö eingesetzt, so daß ein Vollkreis (beziehungsweise eine Vollenipse) entsteht. Alle Werte zwischen -2Ö und 2Ö sind erlaubt. Negative Werte haben zur Folge, daß der Radius als Linie zum Start- beziehungsweise Endpunkt abgetragen wird. Wählt man für beide Winkelangaben negative Werte, entsteht ein Kreissegment, das leicht mit PAINT gefüllt werden kann.</p> <p><i>verhaelt</i> 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 <math>v = 4/3</math></p>

\* `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.

*Bemerkung* » Der CIRCLE-Befehl benutzt die Mathematik-Libraries (siehe Kapitel 19.2).

*Siehe auch* PAINT (354), SCREEN (378), COLOR (286), PALETTE (355), VIEW (414), WINDOW (419), LINE (334).

## CLEAR (Befehl)

Standard

*Seit* QuickBASIC 4.0

*Anwendung* CLEAR [, , *stack*]

*Nutzen* CLEAR schließt alle Dateien, setzt alle Variablen auf 0 beziehungsweise 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* » Zwei Kommata vor dem *stack*-Parameter sind notwendig, um die Kompatibilität zum BASIC-Interpreter aufrechtzuerhalten.

» CLEAR wird automatisch bei einem RUN-Befehl aufgeführt, und die Stack-Manipulationen können im BASIC PDS eleganter mit dem STACK-Befehl und der gleichnamigen Funktion erledigt werden.

» CLEAR darf keinesfalls innerhalb von SUBs oder FUNCTIONS benutzt werden (illegal function call) und ist auch in Subroutinen, die mit GOSUB aufgerufen wurden, fehl am Platze, da nach CLEAR kein RETURN mehr möglich ist.

*Siehe auch* STACK (396), SETMEM (389), ERASE (307).

## CLNG (Funktion)

Standard

*Seit* QuickBASIC 2.0

*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.

*Siehe auch* CINT (283), CDBL (281), CCUR (281), CSNG (290).



<i>Seit</i>	QuickBASIC 2.0
<i>Anwendung</i>	<code>CLOSE [[#]dateinummer [, [#]dateinummer]] ...</code>
<i>Nutzen</i>	Schließt Dateien. Ohne Argumente benutzt, werden alle Dateien geschlossen, ansonsten nur die angegebenen.
<i>Bemerkung</i>	» CLEAR, END, RESET, RUN und SYSTEM schließen ebenfalls alle Dateien.
<i>Siehe auch</i>	OPEN (348), RESET (372).

**CLS (Befehl)****Standard**

<i>Seit</i>	QuickBASIC 3.0
<i>Anwendung</i>	<code>CLS [0 1 2]</code>
<i>Nutzen</i>	<p>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 beziehungsweise VIEW PRINT) definiert ist. Außerdem ist relevant, ob mit KEY ON die Funktionstasten-Anzeige eingeschaltet wurde oder nicht.</p> <p><b>CLS im Grafikmodus ohne Grafik-Viewport:</b> Ganzer Bildschirm wird gelöscht; Funktionstastenanzeige wird, falls eingeschaltet, regeneriert; Cursor wird auf Zeile 1, Spalte 1 gesetzt.</p> <p><b>CLS im Grafikmodus mit Grafik-Viewport:</b> Definierter Grafik-Viewport wird gelöscht.</p> <p><b>CLS im Textmodus (SCREEN 0):</b> 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.</p> <p><b>CLS 0</b> 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.</p> <p><b>CLS 1</b> 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.</p> <p><b>CLS 2</b> Löscht nur den Text-Viewport, läßt in jedem Falle die unterste Zeile unberührt und setzt den Cursor auf Zeile 1, Spalte 1.</p>

*Bemerkung* » Falls die Funktionstasten-Anzeige nicht mit KEY ON eingeschaltet wurde, erzeugt CLS stattdessen eine Leerzeile.

*Siehe auch* VIEW (414), VIEW PRINT (415), KEY ON (330), SCREEN (378), WIDTH (417).

## COLOR (Befehl)

**Standard**

*Seit* QuickBASIC 4.0

*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 *Illegal function call*-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.

*Siehe auch* PALETTE (355), SCREEN (378).

*Seit* QuickBASIC 2.0

*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 nur COM STOP, nicht aber COM OFF wirksam ist, werden alle auftretenden Aufrufe zwischengespeichert und unmittelbar nach dem nächsten COM ON ausgeführt.

*Siehe auch* EVENT ON/OFF (309), ON event GOSUB (346).

**COMMAND\$ (Funktion)****DOS**

*Seit* QuickBASIC 2.0

*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 (nur mit UCASE, also unter Vernachlässigung von Umlauten).

*Bemerkung* » Um Programme, die mit COMMAND\$ arbeiten, in QBX laufen zu lassen, kann der Wert für COMMAND\$ entweder mit der QBX-Option /CMD oder mit "Modify COMMAND\$" aus dem "Run"-Menü gesetzt werden.

» Die folgende Funktion - die allerdings erst ab DOS 3.0 benutzt werden kann - gibt die Befehlszeile völlig unverfälscht zurück, also ohne das Abtrennen von Leerzeichen und ohne die Umwandlung in Großbuchstaben. Sie benutzt einen Interrupt. Programme, die sie benutzen, müssen mit der Library QBX.LIB gelinkt beziehungsweise mit der Quick Library QBX.QLB in QBX geladen werden.

```

REM $INCLUDE: 'qbx.bi'

FUNCTION DetailCommand$

    DIM Reg AS RegType, Laenge AS INTEGER

    Reg.AX = &H6200
    Interrupt &H21, Reg, Reg
    DEF SEG = Reg.BX
    Laenge = PEEK(128)
    DetailCommand$ = SPACE$(Laenge)
    FOR i% = 1 TO Laenge
        MID$(DetailCommand$, i%, 1) = CHR$(PEEK(128 +
i%))
    NEXT

END FUNCTION

```

## COMMON (Befehl)

Standard

*Seit* QuickBASIC 2.0

*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 (siehe Kapitel 13.1) oder wenn verschiedene Source-Codes (\*.BAS) als ein großes Programm gemeinsame Variablen haben sollen. Letzteres kann auch ausgedehnt werden auf Libraries und Quick Libraries; in den BI-Files zu den Toolboxes 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 Fehlern wie *String space corrupt* und ähnlichen.

*Siehe auch* DIM (299), SHARED (390), CHAIN (281).

## CONST (Befehl)

**Standard**

*Seit* QuickBASIC 3.0

*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. Die Verarbeitung durch den Compiler ist jedoch völlig verschieden von der der Variablen. Symbolische Konstanten werden bereits während des Kompilierens im ganzen Programm durch die entsprechenden Zahlen beziehungsweise Zeichenketten oder Adressen ersetzt. Das spart - verglichen mit Variablen - Speicherplatz und Verarbeitungszeit.

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. Daraus folgt, daß es keine zwei gleichnamigen Konstanten verschiedenen Typs geben kann.

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.

*Bemerkung* » Konstanten, die im Hauptprogramm definiert werden, sind global für das ganze Programm (nicht jedoch für andere, gleichzeitig geladene Programme in QBX oder für dazugelinkte Module außerhalb QBX). Konstanten, die in Prozeduren definiert werden, sind lokal für die betreffenden Prozeduren.

*Beispiel* (zur automatischen Typzuweisung)

```
CONST a = 10, b = 77417, c& = 10
```

Hier erhält a automatisch den Typ INTEGER, da er der kleinste ist, der die Zahl 10 repräsentieren kann. b wird dementsprechend LONG, und ebenso c, obwohl für c eigentlich ein INTEGER gereicht hätte. Dafür sorgt das angegebene &-Zeichen.

## COS (Funktion)

Standard

*Seit* QuickBASIC 2.0

*Anwendung*  $x = \text{COS}(y)$

*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).

*Bemerkung* » Die COS-Funktion benutzt die Mathematik-Libraries (siehe Kapitel 19.2).

*Siehe auch* ATN (275), SIN (393), TAN(407).

## CSNG (Funktion)

Standard

*Seit* QuickBASIC 2.0

*Anwendung*  $x! = \text{CSNG}(y)$

*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).

*Siehe auch* CINT (283), CDBL (281), CLNG (284), CCUR (281).

## CSRLIN (Funktion)

Standard

*Seit* QuickBASIC 2.0

*Anwendung*  $x = \text{CSRLIN}$

*Nutzen* Gibt zurück, in welcher Bildschirmzeile sich der Textcursor gerade befindet.

*Siehe auch* VIEW PRINT (415), POS (362), LOCATE (337).

## CURDIR\$ (Funktion)

DOS

*Seit* BASIC 7.0 PDS

*Anwendung*  $x\$ = \text{CURDIR\$ } [(laufwerk\$)]$

*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.

*Bemerkung* » CURDIR\$ ohne Argument gibt dieselbe Information aus, die der DOS-Befehl CD ohne Argument ausgibt.

*Siehe auch* CHDIR (282), CHDRIVE (282).

## CVx (Funktionen)

Standard

*Seit* QuickBASIC 2.0

*Anwendung*  $x\% = \text{CVI}(y\$)$

$x\& = \text{CVL}(y\$)$  (seit 4)

$x! = \text{CVS}(y\$)$

$x\# = \text{CVD}(y\$)$

$x@ = \text{CVC}(y\$)$  (seit 7)

*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* » Die zugehörigen MKx-Funktionen bewirken das Gegenteil: Sie wandeln Zahlen in Code-Strings um. Wenn man nicht mehr mit

FIELD arbeitet, benötigt man diese Funktionsgruppen zumeist nicht mehr.

» Der Compiler-Switch /MBF (siehe Kapitel 6.3) beeinflusst diese Funktionen.

*Siehe auch* CVxMBF (292), MKx\$ (343), FIELD (311).

## CVxMBF (Funktionen)

Standard

*Seit* QuickBASIC 3.0

*Anwendung*  $x! = \text{CVSMBF}(y\$)$

$x\# = \text{CVDMBF}(y\$)$

*Nutzen* Diese beiden Funktionen entschlüsseln wie ihre Nachfolger CVS und CVD codierte SINGLE- beziehungsweise 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 coprozessor kompatibel zu werden. Die Zahlen wurden nicht mehr im Microsoft-eigenen Binärformat, sondern im weitverbreiteten IEEE-Format gespeichert. Will man heute alte Random-Access-Dateien aus der "MBF-Zeit" lesen, muß man mit FIELD arbeiten und sich der CVxMBF-Funktionen bedienen, die identisch sind mit den CVx-Funktionen von damals. Für andere Zwecke sollten diese Funktionen, wie auch ihre Gegenstücke MKxMBF\$, nicht mehr benutzt werden.

*Siehe auch* MKxMBF\$ (344), CVx (291), FIELD (311).

## DATA (Befehl)

Standard

*Seit* QuickBASIC 2.0

*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 (der Versuch, eine Zeichenkette mittels READ in eine numerische Variable einzulesen, führt zu einem *Syntax error*).

Stringkonstanten können in Anführungszeichen stehen; enthalten sie keine Kommata und Doppelpunkte, ist das jedoch nicht notwendig.

Eine DATA-Zeile kann beliebig viele Konstanten enthalten. Ob Sie zum Beispiel zehn DATA-Zeilen mit je zwei Konstanten oder eine



DATA-Zeile mit 20 Konstanten angeben, spielt keine Rolle, bis auf die Tatsache, daß die letztgenannte Verfahrensweise ein paar Bytes im EXE-Programm spart.

*Bemerkung* » Symbolische Konstanten, die in DATA-Zeilen stehen, werden nicht als symbolische Konstanten ausgewertet, sondern wie Text behandelt.

» DATA-Zeilen dürfen innerhalb QBX nur im Hauptprogramm, nicht aber in Subroutinen stehen. Deplazierte DATA-Zeilen werden automatisch ins Hauptprogramm verschoben; man sollte das jedoch besser von Hand erledigen, da QBX Zeilenlabels, die zum Beispiel eine Zeile vor dem betreffenden DATA-Block stehen (und in einer RESTORE-Anweisung gebraucht werden könnten) nicht mit-verschiebt.

*Siehe auch* READ (369), RESTORE (372).

DATE\$ (Systemvariable)		Standard
<i>Seit</i>	QuickBASIC 2.0	
<i>Anwendung</i>	<code>x\$ = DATE\$</code> <code>DATE\$ = x\$</code>	
<i>Nutzen</i>	DATE\$ enthält stets das Datum der Systemuhr im Format <i>mm-tt-jjjj</i> . Man kann in diese Variable ein neues Systemdatum eintragen, das dann entweder die Form <i>mm-tt-jj</i> oder <i>mm-tt-jjjj</i> (auch Schrägstriche als Trennzeichen) haben muß.	
<i>Bemerkung</i>	<p>» Da DATE\$ einen <i>Illegal function call</i> 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.</p> <p>» Zu beachten ist der Unterschied zwischen Systemuhr und Echtzeituhr. Die Systemuhr wird beim Booten von der Echtzeituhr gesetzt und läuft von da an unabhängig; eine Änderung an DATE\$ ist also auch nur bis zum nächsten Bootvorgang wirksam.</p> <p>» Die Date-Toolbox stellt interessante Möglichkeiten zur Verarbeitung von Uhrzeit und Datum zur Verfügung.</p>	
<i>Siehe auch</i>	TIME\$ (408), TIMER (408).	

*Seit* BASIC 7.1 PDS

*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.

QBX fügt automatisch alle benötigten DECLARE-Zeilen in ein Programm ein; wenn Sie mit mehreren Modulen arbeiten, müssen Sie aber Handarbeit leisten, weil QBX nicht weiß, welche Module später zusammengefügt 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.

Das Vorhandensein eines DECLARE-Befehls führt auch dazu, daß der Compiler Anzahl und Typ der Argumente prüft, die einer Prozedur übergeben werden sollen, und eventuelle Unstimmigkeiten meldet.

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).

In der Tat ist ein DECLARE-Befehl für BASIC-Prozeduren normalerweise völlig identisch mit dem zugehörigen Prozedurkopf, nur daß vorn ein DECLARE davorsteht und hinten das STATIC fehlt, falls die Prozedur eines besaß.

Arrays werden einfach durch eine offene und eine geschlossene runde Klammer angezeigt; auf Wunsch kann darin die Anzahl der Dimensionen angegeben werden.

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 allerdings nicht vom Compiler geprüft, und daher ist es ratsam, immer die vollständige DECLARE-Anweisung zu benutzen.

Die Optionen CDECL und ALIAS sind nur für externe Prozeduren (getrennt kompiliert beziehungsweise 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 und hinten der bei FUNCTION-Deklarationen vorhandene Typenbezeichner gestrichen 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- beziehungsweise FUNCTION-Kopf - entweder mit einem Typenbezeichner oder mit einer AS-Formulierung festlegen. Fehlt beides, wird der Standard-Typ benutzt (üblicherweise SINGLE, siehe aber DEFxxx-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.

» Strings mit fester Länge sind in DECLARE-Anweisungen ebenso wie in Prozedurköpfen nicht erlaubt.

*Siehe auch* CALL (277), SUB/FUNCTION (402), DEFxxx (297).

DEF FN (Befehl)	Standard
<i>Seit</i>	QuickBASIC 2.0
<i>Anwendung</i>	<pre>(1) DEF FNfunktionsname [(parameterliste)] =     ausdruck (2) DEF FNfunktionsname [(parameterliste)]     ...     END DEF</pre>
<i>Nutzen</i>	<p>DEF FN ist eine altertümliche Methode zur Funktionsdefinition, die inzwischen von der weitaus mächtigeren FUNCTION...END FUNCTION-Konstruktion ersetzt wurde und nur noch in Ausnahmefällen ihre Berechtigung hat.</p>

DEF FN in der ersten Syntax wurde schon vom BASIC-Interpreter unterstützt. Hier wird innerhalb einer einzigen Zeile dem Namen der Funktion, dem immer das FN vorangestellt sein muß, ein Ausdruck zugewiesen, zum Beispiel: DEF FNQuadrat&(Zahl%) = Zahl% \* Zahl%. In Klammern können dabei beliebig viele Argumente angegeben werden; Funktionen ohne Argumente benötigen keine Klammern.

Die zweite Syntax stammt aus der Zeit von QuickBASIC 2, in der es noch keine FUNCTIONS gab. Hier kann die Funktion beliebig viele

Zeilen umfassen, beginnend mit DEF FN und endend mit END DEF; dazwischen sind auch EXIT DEF-Befehle erlaubt, die die Funktion vorzeitig abbrechen. Der Funktionswert wird zugewiesen, indem man innerhalb der Definition irgendwo den Befehl `FNfunktionsname = ausdruck` anbringt. Sollte die Funktion verlassen werden, ohne daß ein Funktionswert zugewiesen wurde, so ist dieser 0 beziehungsweise ein Leerstring.

*FNfunktionsname* sollte wie eine gewöhnliche Variable einen Typenbezeichner enthalten, der festlegt, welchen Typs die Funktion ist. Im Gegensatz zu den mit FUNCTION definierten Funktionen ist es hier möglich, mehrere gleichnamige, aber verschieden typisierte Funktionen gleichzeitig zu benutzen (zum Beispiel FNAnzahl%, FNAnzahl\$ und FNAnzahl@).

*Bemerkung* » Eine DEF FN-Funktion kann nur im Hauptprogramm definiert werden, und sie muß definiert sein, bevor sie zum ersten Mal aufgerufen wird. QBX öffnet für DEF FN-Funktionen kein eigenes Fenster. Sämtliche Variablen, die im Hauptprogramm verfügbar sind (also lokale Variablen des Hauptprogramms und alle globalen Variablen), stehen auch der Funktion zur Verfügung. Die *parameterliste* entspricht der gängigen Variablenvereinbarung bei SUB-Befehlen (entweder AS-Formel oder Typenbezeichner, sonst wird der Standard-Datentyp genommen); jedoch dürfen an DEF FN-Funktionen keine Arrays, keine Variablen selbstdefinierten Typs und keine Strings mit fester Länge übergeben werden. Alle Parameter werden - im Unterschied zu FUNCTION - *by value* übergeben, als Werteparameter also, die die Funktion nicht manipulieren kann.

» Variablen, die in einer DEF FN-Funktion benutzt werden und nicht Teil der *parameterliste* sind, sind zugleich lokale Variablen des Hauptprogramms. Variablen, die wirklich lokal für diese Funktion sind, kann man - nur in der zweiten Syntax - mittels des STATIC-Befehls definieren.

» Bei Aufrufen an eine DEF FN-Funktion prüft der Compiler automatisch Parametertypen und -anzahl.

» DEF FN-Funktionen können nur in dem Modul benutzt werden, in dem sie definiert sind.

» DEF FN-Funktionen sind gewöhnlich in der Ausführung etwas langsamer als ihre Kolleginnen von FUNCTION.

» Keine gewöhnliche Variable darf als erste zwei Buchstaben ihres Namens FN haben.

*Siehe auch* SUB/FUNCTION (402), STATIC (397).

## DEFxxx (Befehle)

Standard

*Seit* QuickBASIC 2.0

*Anwendung* DEF CUR *bereich* [, *bereich*]... (Seit 7)  
DEF DBL *bereich* [, *bereich*]...  
DEF INT *bereich* [, *bereich*]...  
DEF LNG *bereich* [, *bereich*]... (Seit 4)  
DEF SNG *bereich* [, *bereich*]...  
DEF STR *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 DEFxxx-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 DEFxxx-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.

*Siehe auch* DIM (299).

## DEF SEG (Befehl)

Speicher

*Seit* QuickBASIC 2.0

*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 DGROUPE gesetzt. Der erlaubte Bereich für *segment* ist 0 bis 65.535.

Im Protected Mode bezieht sich DEF SEG nicht auf Segmente, sondern auf Selektoren (siehe dritte Bemerkung).

*Bemerkung* » (für den Real Mode/DOS): Der kühle Rechner könnte denken: 65.536 Segmente gibt es, in jedem können mit PEEK 65.536 Speicherstellen abgefragt werden, das macht einen Gesamtspeicher von genau 4 Gigabyte. Falsch gerechnet: Von den vier hexadezimalen Stellen der Segmentadresse sind drei redundant. Die Erhöhung der Speicherstelle um 16 ist unter DOS identisch mit der Erhöhung der Segmentadresse um 1, das heißt, daß `DEF SEG = 0: PRINT PEEK(16)` zum Beispiel identisch ist mit `DEF SEG = 1: PRINT PEEK(0)`. So ergibt sich ein Gesamtspeicher von genau 1 MB. Es hat natürlich keinen Sinn, Speicherstellen anzusprechen, die nicht existieren (das Abfragen solcher Speicherstellen ergibt sinnlose Werte, der Versuch, Daten hineinzuschreiben, mißlingt - es wird aber kein Fehler erzeugt). Bei einem Rechner, der 640 KB Hauptspeicher hat, ist die höchste Adresse die Speicherstelle 655.359, also zum Beispiel `DEF SEG = 40959` und `PEEK(15)`.

» Der Direktzugriff auf Speicherstellen (mit PEEK, POKE, DEF SEG, BLOAD, BSAVE) kann nicht auf das Expanded Memory ausgedehnt werden. QBX kann jedoch Arrays in das EMS auslagern, mit der Konsequenz, daß sich diese Arrays dann dem direkten Zugriff entziehen. Bei kompilierten Programmen ist das nicht möglich. Das Weglassen des Switches /Ea verhindert es auch in QBX. Siehe dazu "Extended & Expanded Memory" in Kapitel 17.

» Bei der OS/2-Programmierung (*protected mode*) muß darauf geachtet werden, daß man nur einen gültigen Selektor mit DEF SEG anspricht, da sonst Fehler auftreten (*Permission denied* oder ein Betriebssystemfehler). DEF SEG darf sich nur auf einen gültigen Selektor beziehen, obwohl die Gültigkeit der angegebenen Adresse bei DEF SEG noch nicht geprüft wird. Ein Selektor im Protected Mode ist vergleichbar mit einem Segment im Real Mode; im Real Mode wird jedoch die Far-Adresse einfach durch lineare Verknüpfung der Segment- und Offsetadresse (siehe erste Bemerkung) gebildet, während der Selektor im Protected Mode ein Zeiger auf eine Tabelle von Segmentdeskriptoren ist, mit deren Hilfe die Adresse berechnet wird. Diese Tabelle enthält auch Informationen, welche Speicherbereiche von welchem Prozeß beschrieben und gelesen werden dürfen und welche nicht.

» Funktionen, die die Segmentadressen von Variablen und Strings ermitteln, sind SSEG (beziehungsweise SSEGADD) für Strings und VARSEG für Variablen. VARSEG macht die externe Funktion PTR86 überflüssig, die in älteren Versionen des Compilers benutzt wurde.

*Siehe auch* BLOAD (276), BSAVE (276), PEEK (357), POKE (362), VARSEG (413), VARPTR (412), SSEG (395), SADD (377), SSEGADD (396).

*Seit* QuickBASIC 4.0

*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. Siehe dazu "Statische und dynamische Felder" in Kapitel 12.4.

### **Globale Variablen mit SHARED**

Wenn Sie hinter DIM das Wort SHARED angeben (es gilt für die ganze Zeile, also 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, verbessert aber die Übersicht.

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.

Anstatt die Variable `Laenge&` im Programm zu benutzen, können Sie also auch `DIM Laenge AS LONG` schreiben und in Zukunft auf den Typenbezeichner `&` verzichten. Das ist mehr Schreibarbeit, aber übersichtlicher und fehlersicher: Hätten Sie sonst einmal aus Versehen statt `Laenge&` nur `Laenge` geschrieben, wäre das in Compilers Augen eine zweite Variable gewesen. Mit `DIM` kann Ihnen das nicht passieren. Wenn Sie sich die Beispielprogramme in diesem Buch ansehen, sehen Sie, daß ich Typenbezeichner fast ausschließlich für temporäre Variable (wie Schleifenvariablen) benutze.

*Beispiel* Einige Beispiele für mögliche `DIM`-Befehle:

```
DIM a(99, 1 TO 50) AS INTEGER
DIM SHARED Zeile(-50 TO 50) AS STRING
DIM Datum AS DateType
DIM SHARED z
DIM Text AS STRING
DIM a AS INTEGER, b AS INTEGER, c AS INTEGER
DIM x, y, z, Faktor AS DOUBLE
' Achtung: bei diesem letzten Befehl wird nur
' Faktor als DOUBLE deklariert; die Variablen
' x, y und z erhalten, weil kein Typ für sie
' angegeben ist, den Standard-Datentyp. Dieser
' ist SINGLE, wenn nicht mit DEFxxx etwas
' anderes vereinbart wurde.
```

*Siehe auch* `COMMON` (288), `STATIC` (397), `SHARED` (390), `REDIM` (370), `ERASE` (307).

## DIR\$ (Funktion)

DOS

*Seit* BASIC 7.0 PDS

*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 *illegal function call*.



- Bemerkung** » Werden in *directorymaske\$ Laufwerk* und Pfad nicht angegeben, so wird jeweils das aktuelle Laufwerk beziehungsweise 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 beziehungsweise 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 (siehe Kapitel 16).
- » 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...`

**Siehe auch** FILES (313), CURDIR\$ (291).

<b>DO...LOOP (Befehl)</b>	<b>Struktur</b>
---------------------------	-----------------

**Seit** QuickBASIC 3.0

**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, wie man leicht erkennt, 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.

*Beispiel*

Diese Schleife wartet auf das Drücken einer Taste:

```
DO
  a$ = INKEY$
LOOP UNTIL LEN(a$)
```

Es ist nicht nötig, UNTIL LEN(a\$) > 0 zu schreiben, da die Bedingung als erfüllt gilt, wenn ihr Wert ungleich 0 ist.

Diese Schleife liest die Datei TEST.TXT ein und zeigt ihren Inhalt auf dem Bildschirm:

```
OPEN "TEST.TXT" FOR INPUT AS #1
DO UNTIL EOF(1)
  LINE INPUT #1, a$ : PRINT a$
LOOP
CLOSE 1
```

*Siehe auch* WHILE...WEND (417), FOR...NEXT (314).

## DRAW (Befehl)

**Grafik**

*Seit* QuickBASIC 2.0

*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 <i>n</i> Einheiten aufwärts
D [n]	Zeichne <i>n</i> Einheiten abwärts
L [n]	Zeichne <i>n</i> Einheiten nach links
R [n]	Zeichne <i>n</i> Einheiten nach rechts
E [n]	Zeichne diagonal <i>n</i> Einheiten auf/rechts
F [n]	Zeichne diagonal <i>n</i> Einheiten ab/rechts
G [n]	Zeichne diagonal <i>n</i> Einheiten ab/links
H [n]	Zeichne diagonal <i>n</i> Einheiten auf/links
M <i>x,y</i>	Zeichne zum Punkt <i>x, y</i> (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 +/- <i>x,y</i>	Zeichne relativ vom aktuellen Punkt <i>x</i> Einheiten in <i>x</i> - und <i>y</i> Einheiten in <i>y</i> -Richtung (je nachdem, ob - oder + angegeben wird, werden <i>x</i> und <i>y</i> von den aktuellen Koordinaten abgezogen oder zu ihnen addiert).
A <i>n</i>	Zeichnung um $n * 90^\circ$ drehen (0 $\frac{3}{4}n$ $\frac{3}{4}3$ )
TA <i>n</i>	Zeichnung um $n^\circ$ drehen (-360 $\frac{3}{4}n$ $\frac{3}{4}360$ )
C <i>n</i>	Vordergrundfarbe (Zeichenfarbe) auf <i>n</i> setzen (siehe SCREEN für gültige Werte)
P <i>f, r</i>	Fläche ab aktuellem Punkt in Farbe <i>f</i> füllen; Füllgrenzen sind Linien der Farbe <i>r</i>
S <i>n</i>	Skalierung <i>n</i> setzen. <i>n</i> = 1 ist normal (1 DRAW-Einheit = 1 Pixel beziehungsweise 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 <i>n</i> bedeutet mehr Pixel pro Einheit.
X <i>adr</i> \$	Führt einen weiteren DRAW-String aus, dessen Adreßcode mit VARPTR\$ ermittelt und in <i>adr</i> \$ eingetragen werden muß (siehe Bemerkung).
= <i>adr</i> \$	Kann anstelle einer Zahl ( <i>n, x, y, f, g</i> in obigen Beispielen) gesetzt werden; <i>adr</i> \$ 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 beziehungsweise = und dem dazugehörigen VARPTR\$ darf sich *kein* Leerzeichen befinden.

» Der DRAW-Befehl benutzt die Mathematik-Libraries. Sein unbeachteter Einsatz kann deshalb das EXE-File unnötig um etwa 10 KB verlängern (siehe "Programmgröße und RAM-Speicherplatz" in Kapitel 19.2).

» Es gibt zwei Möglichkeiten, Variablen in einen DRAW-String einzufügen. Die einfachere Methode ist, einfach das Pluszeichen zur Stringverknüpfung zu benutzen - dann benötigt man weder den Befehl "=" noch "X": DRAW "L30 U" + STR\$(Auf%) + "R 30" oder DRAW "L30 " + Rest\$ sind Beispiele dafür. Die zweite Methode ist der Zugriff über die Adresse der einzusetzenden Vari-

ablen: DRAW "L30 U=" + VARPTR\$(Auf%) + "R 30" oder DRAW "L30 X" + VARPTR\$(Rest\$) haben dieselbe Funktion wie die obigen Beispiele. Die VARPTR\$-Methode ist recht umständlich und höchstens in zeitkritischen Fällen brauchbar, wie das folgende Beispiel zeigen soll.

*Beispiel* (zum Unterschied der Plus- und der VARPTR\$-Methode)

```
' Die Plus-Methode zeichnet einen Kreis
Linie$ = "U60 R30 BL30 BD60"
FOR a% = 1 TO 360
    DRAW "TA" + STR$(a%) + Linie$
    ' Jedesmal muß der String neu gebildet wer-
    ' den! (Lesen Sie Kapitel 19, und Sie wis-
    ' sen, daß das viel Zeit kostet!)
NEXT

' Die VARPTR$-Methode zeichnet einen Kreis
Linie$ = "TA=" + VARPTR$(a%) + "U60 R30 BL30
BD60"
FOR a% = 1 TO 360
    DRAW Linie$
    ' Der String enthält nur die Adresse von a%,
    ' also werden Änderungen von a% automatisch
    ' übernommen!
    ' Die Gefahr dabei ist allerdings, daß sich
    ' die Adresse von a% irgendwann einmal än-
    ' dert, und dann klappt's nicht mehr.
NEXT
```

*Siehe auch* LINE (334), PSET (366), CIRCLE (283), PAINT (354), SCREEN (378), COLOR (286).

## **\$DYNAMIC (Metabefehl)**

**Standard**

*Seit* QuickBASIC 2.0

*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).

*Bemerkung* » Mehr über dynamische und statische Arrays finden Sie im Kapitel 12.4.

*Siehe auch* \$STATIC (397), DIM (299).

*Seit* BASIC 7.0 PDS

*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).

Die optionale Variable *errorlevel%* kann benutzt werden, um einen Beendigungs-Code an das aufrufende Programm (zumeist das Betriebssystem, also DOS oder OS/2) 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.

END schließt alle Dateien; die Kontrolle wird an QBX oder an DOS zurückgegeben, je nachdem, ob das Programm innerhalb QBX oder selbständig lief.

*Bemerkung* » END ist identisch mit SYSTEM.

*Siehe auch* STOP (400), SYSTEM (406), DEF FN (295), IF (320), SELECT CASE (387), SUB/FUNCTION (402), TYPE (410).

**ENVIRON\$ (Funktion)****DOS**

*Seit* QuickBASIC 2.0

*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. Dann hat man alle Systemvariablen eingelesen.

*Bemerkung* » In der Tat ist die Kombination aus dem DOS-Befehl SET und der BASIC-Funktion ENVIRON\$ durchaus brauchbar, um eine Kommunikation zwischen Batchprozeduren und BASIC-Programmen aufzubauen. Dabei kann es jedoch vorkommen, daß ein SET-Befehl von DOS mit dem Kommentar *Out of environment* space abgelehnt wird. Dann ist der Speicher für Systemvariablen voll. Abhilfe schafft die Zeile "SHELL = C:\COMMAND.COM /P /E:1024" in CONFIG.SYS, wobei anstelle von C:\ eventuell ein anderer Pfad und anstelle von 1024 eine beliebige Größe des Systemvariablen-speichers eingesetzt werden kann (/E:512 ist Standard, /E:1024 dürfte für die meisten Anwendungen reichen).

*Siehe auch* ENVIRON (306), SHELL (392).

## ENVIRON (Befehl)

DOS

*Seit* QuickBASIC 2.0

*Anwendung* ENVIRON *variablenzuordnung*\$

*Nutzen* Verändert beziehungsweise löscht eine bestehende oder erzeugt neue Betriebssystemvariablen. Der String *variablenzuordnung*\$ muß die Form "VARIABLE=Zuordnung" haben; der Name der Variablen (zum Beispiel PATH) muß in Großbuchstaben geschrieben sein, und links vom Gleichheitszeichen dürfen keine Leerzeichen angegeben werden, da diese sonst dem Variablennamen zugeordnet werden. Läßt man die Zuordnung weg (ENVIRON "PATH=") 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 Variablen-tabelle 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 Out of memory-Fehler auftritt.

*Siehe auch* ENVIRON\$ (305), SHELL (391).

*Seit* QuickBASIC 2.0

*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).

*Beispiel* Siehe Beispiel zu DO...LOOP.

*Siehe auch* LOC (336), LOF (338), OPEN (348), CLOSE (285), DO...LOOP (301).

**ERASE (Befehl)****Standard**

*Seit* QuickBASIC 4.0

*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. (Siehe Kapitel 12.4.)

*Bemerkung* » Beim Verlassen von Prozeduren werden alle Arrays, die in der Prozedur als automatische Variablen definiert wurden, automatisch gelöscht - ein ERASE-Befehl ist nicht notwendig. Zur Erläuterung der automatischen Variablen siehe "Statische und automatische Variablen" in Kapitel 12.5.

*Siehe auch* DIM (299), REDIM (370).

*Seit* QuickBASIC 4.0

*Anwendung*  $x = \text{ERDEV}$   
 $x\$ = \text{ERDEV\$}$

*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.

*Siehe auch* ERR, ERL (308), ON ERROR (345)

**ERR, ERL (Systemvariablen)****Standard**

*Seit* BASIC 7.0 PDS

*Anwendung*  $x = \text{ERR}$   
 $x = \text{ERL}$   
 $\text{ERR} = x$

*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 "Alle Fehlermeldungen auf einen Blick", Anhang C), 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. Existiert auch eine solche Nummer nicht (in kompilierten Programmen wird das ganze Programm nach einer Nummer durchsucht, innerhalb QBX wird nur innerhalb der Prozedur gesucht, in der der Fehler auftrat), dann ist ERL 0.

*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. (Siehe Kapitel 15).

*Siehe auch* ERDEV, ERDEV\$ (308), ON ERROR (345).

<b>ERROR (Befehl)</b>		<b>Standard</b>
<i>Seit</i>	QuickBASIC 2.0	
<i>Anwendung</i>	ERROR <i>fehlernummer</i>	
<i>Nutzen</i>	Simuliert das Auftreten eines Fehlers. Der erlaubte Bereich für <i>fehlernummer</i> ist 1 bis 255. Ein durch diesen Befehl verursachter "Pseudo-Fehler" steht einem echten Fehler in nichts nach. Die ERR-Variable wird auf <i>fehlernummer</i> gesetzt, ERL auf die nächstliegende Zeilennummer (siehe ERL), und falls eine Fehlerbehandlungsroutine aktiv ist, wird diese aufgerufen (falls nicht, meldet das Programm den Fehler und geht zurück ins DOS beziehungsweise in QBX).	
<i>Bemerkung</i>	» ERROR kann benutzt werden, um die Fehlerbehandlungsroutinen auf Dinge wie zum Beispiel falsche Eingaben o.ä., die keine Fehler im Sinne des Compilers sind, auszudehnen. Dazu bedient man sich am besten eigener, von BASIC ungenutzter Fehler-Codes (Anhang C beschreibt alle in PDS 7.1 benutzten Fehlercodes).	
<i>Siehe auch</i>	ERR, ERL (308), ON ERROR (345), RESUME (372).	

<b>EVENT ON, EVENT OFF (Metabefehle)</b>		<b>Trapping</b>
<i>Seit</i>	BASIC 7.0 PDS	
<i>Anwendung</i>	EVENT ON EVENT OFF	
<i>Nutzen</i>	<p>Diese Befehle schalten das Event-Trapping aus beziehungsweise 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.</p> <p>Wenn mit den Compiler-Schaltern /W oder /V gearbeitet wird, erzeugt der Compiler für jedes Label beziehungsweise 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.</p>	

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. Dazu das folgende Beispiel:

*Beispiel* (zur Teilung in EVENT ON- und EVENT OFF-Blocks)

```
...  
EVENT ON  
GOTO ZweiterTeil  
  
ErsterTeil:  
PRINT "Dies ist der erste Programmteil."  
PRINT "Bitte warten Sie, ich rechne ein wenig."  
FOR a% = 1 TO 1000  
    b# = b # + a% * 2 + SQR(CDBL(a%))  
NEXT  
RETURN  
  
ZweiterTeil:  
PRINT "Es läuft der zweite Programmteil."  
EVENT OFF  
    ' dieser EVENT OFF-Befehl verhindert  
    ' nicht, daß während des gesamten  
    ' ersten Teils Event-Tests stattfinden,  
    ' da sich der erste Teil vom Compiler  
    ' aus gesehen in einem EVENT ON-Block  
    ' befindet!  
GOSUB ErsterTeil  
EVENT ON  
...
```

*Siehe auch* ON...GOSUB (348).

## EXIT (Befehl)

Struktur

*Seit* QuickBASIC 4.0

*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.

*Siehe auch* DEF FN (295), DO...LOOP (301), FOR...NEXT (314), SUB/FUNCTION (402).

## EXP (Funktion)

Standard

*Seit* QuickBASIC 2.0

*Anwendung*  $x = \text{EXP}(y)$

*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.

*Siehe auch* LOG (338).

## FIELD (Befehl)

I/O

*Seit* QuickBASIC 2.0

*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. *byte* ist jeweils die Länge eines Feldes, *buffer\$* der Name der Stringvariablen, die zuzuordnen ist. Die Gesamtlänge aller ver-

einbarten Felder darf die Satzlänge der Datei (`OPEN FOR RANDOM LEN = satzlänge`) nicht überschreiten und wird üblicherweise identisch mit ihr sein. Normalerweise wird ein `FIELD`-Befehl direkt nach einer `OPEN FOR RANDOM`-Anweisung ausgeführt. Danach sorgt jeder `GET`-Befehl, der sich auf die Datei bezieht, dafür, daß der angegebene Datensatz aus der Datei in die bei `FIELD` definierten Buffer-Variablen übertragen wird, und ein `PUT`-Befehl bewirkt das Gegenteil. Sobald die Datei geschlossen wird, werden alle `FIELD`-Variablen mit Leerstrings überschrieben.

*Bemerkung* » Solange eine Variable Buffer-Variable des `FIELD`-Befehls ist, sollte auf sie nur mit `LSET`, `RSET` oder einfachen Zuweisungen zugegriffen werden, da sonst die Gefahr besteht, daß die Adresse der Variablen geändert wird und sie nicht mehr in den Datenpuffer der Datei zeigt.

» Moderne Programme arbeiten nicht mehr mit der `FIELD`-Anweisung, die noch aus der Zeit stammt, da es keine selbstdefinierten Datentypen gab. Praktischer ist es, statt eines Satzes von `FIELD`-Variablen eine Variable selbstdefinierten Typs zu benutzen. Man verzichtet völlig auf `FIELD` und gibt stattdessen den Namen der Variablen, aus der beziehungsweise in die Daten übertragen werden sollen, bei `PUT` beziehungsweise `GET` mit an. Vgl. hierzu den Abschnitt "Eine Parameterdatei" in Kapitel 11.1.

» Jede `FIELD`-Anweisung setzt eventuelle vorhergehende `FIELD`-Befehle, die sich auf dieselbe Datei bezogen, außer Kraft.

*Siehe auch* `GET` (317), `PUT` (366), `LSET` (340), `RSET` (375), `OPEN` (348).

## FILEATTR (Funktion)

DOS, I/O

*Seit* QuickBASIC 4.0

*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-Unterrouinen von Nutzen sein, wird in BASIC aber sonst nicht benötigt.

*Bemerkung* » `FILEATTR` kann auch auf ISAM-Dateien angewandt werden. Siehe ISAM-Referenzteil.

*Siehe auch* `OPEN` (348).

*Seit* QuickBASIC 2.0

*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 Directories ausgegeben. Die Ausgabeform ist identisch mit dem Befehl DIR /W in DOS (also nur Dateinamen, keine Größe, kein Datum und keine Uhrzeit). Die Ausgabe kann nicht beeinflußt werden.

*Bemerkung* » Vermeiden Sie den FILES-Befehl, wenn es möglich ist. Er ist zu unflexibel, um in modernen Programmen sinnvoll eingesetzt zu werden. (Man denke nur an einen etwaigen Rahmen auf dem Schirm, den FILES zwangsläufig überschreiben würde.) Benutzen Sie besser die neue BASIC-Funktion DIR\$ oder die in diesem Buch enthaltenen Directory-Routinen, die mit Interrupts arbeiten (siehe "Weitere Beispiele zur Interrupt-Nutzung", Kapitel 16.4).

» FILES zeigt nur Dateien an, die DOS auch anzeigen würde, also keine, bei denen das Hidden- oder das Systemattribut gesetzt ist.

» FILES zeigt auch Subdirectories an, aber so, daß man sie nicht von Dateien ohne Extension unterscheiden kann.

*Siehe auch* DIR\$ (300).

**FIX (Funktion)****Standard**

*Seit* QuickBASIC 2.0

*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.

*Siehe auch* CINT (283), INT (327).

*Seit* QuickBASIC 4.0

*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, da bei korrekter Programmierung eindeutig sein muß, auf welches FOR sich ein NEXT bezieht. Trotzdem empfiehlt es sich, diese redundante Angabe zu machen, um Flüchtigkeitsfehler beim Programmieren zu vermeiden.

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.

» Die Variable *zähler* kann innerhalb der Schleife verändert werden, obwohl das eine Sünde an der Struktur ist. Zum vorzeitigen Verlassen (früher setzte man einfach gewaltsam *zähler* auf *ende* + 1) kann nun EXIT FOR benutzt werden.

» *ende* + *schrittweite* darf nie größer sein als der maximal zulässige Wert für den Typ von *zähler* (beziehungsweise, bei negativer *schrittweite*, nie kleiner als der minimale Wert). Der Befehl FOR a% = 1 TO 32767 (die Schrittweite ist 1, wenn STEP weggelassen wird) führt zu einem Absturz des kompilierten Programms beziehungsweise zu einem *Overflow* in QBX, da die Zählervariable hier bis auf 32.767 + 1 erhöht wird, und 32.768 fällt aus dem Bereich für INTEGER-Variablen.

» *anfang* und *ende* werden, wenn es sich um Variablen oder Ausdrücke handelt, nur am Anfang der Schleife einmal berechnet. Dadurch ist in manchen Fällen ein DO...LOOP-Befehl besser

geeignet als FOR...NEXT. Dies ist vor allem bei String-Manipulationen zu berücksichtigen, bei denen *ende* die Länge des Strings ist. Dazu das folgende Beispiel:

*Beispiel*

(zur Änderung von *ende* während der Schleife)

```
' diese Prozedur soll hinter jeden Punkt
' im String ein Leerzeichen einfügen:
SUB PunktMitLeerzeichen (Zeile AS STRING)

    FOR a% = 1 TO LEN(Zeile)
        IF MID$(Zeile, a%, 1) = "." THEN
            Zeile = LEFT$(Zeile, a%) + " " +
MID$(Zeile, a% + 1)
        END IF
    NEXT

END SUB

' Sie funktioniert aber nicht, weil LEN(Zeile)
' nur am Anfang der Schleife ermittelt wird und
' so die Längenänderung des Strings während der
' Schleife unberücksichtigt bleibt:

a$ = "Dies.ist.ziemlich.punktiert.!"
PunktMitLeerzeichen a$
PRINT a$

' Dabei kommt heraus:

Dies. ist. ziemlich. punktiert.!

' Hinter dem letzten Punkt wurde kein Leer-
' zeichen eingefügt.
' Um solche Fehler zu vermeiden, hätte man
' statt FOR...NEXT ein DO...LOOP einsetzen
' können, bei dem die Länge jedesmal aufs neue
' ermittelt wird.
```

*Siehe auch* DO...LOOP (301), EXIT (311), WHILE...WEND (417).

*Seit* BASIC 7.0 PDS

*Anwendung*  $x = \text{FRE}(\text{argument})$

*Nutzen* Stellt fest, wieviel Speicher einer bestimmten Kategorie zur Verfügung steht. Die folgende Tabelle beschreibt die Möglichkeiten:

<i>argument</i>	<b>Ergebnis</b>
-1	(in DOS:) Der verbleibende Platz im Far-Speicher in Bytes (in OS/2:) immer 2.147.483.647
-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 ( <i>Feature unavailable</i> -Fehler tritt auf, wenn kein EMS vorhanden ist)
x\$	(bei near strings:) freier Platz im DGROUP-Segment (bei Far Strings:) freier Platz in dem Segment, in dem sich x\$ befindet (siehe "Far Strings" in Kapitel 12.3)
""	(bei Near Strings:) freier Platz im DGROUP-Segment (bei Far Strings:) freier Platz für temporäre Strings
0	(bei Near Strings:) freier Platz im DGROUP-Segment (bei Far Strings:) <i>Illegal function call</i>

Jedes *argument*, das hier nicht genannt ist, führt zu einem *Illegal function call*.

*Bemerkung* » FRE(-3) ist nur innerhalb von QBX sinnvoll; in mit BC kompilierten Programmen funktioniert es zwar ohne weiteres, aber da diese Programme EMS ohnehin nur durch Overlays oder ISAM nutzen können (siehe "Extended & Expanded Memory", Kapitel 17) und beides auch ohne EMS funktioniert, ist es relativ gleichgültig, wieviel oder wiewenig EMS noch frei ist.

» FRE(-2) kann dazu benutzt werden, am Ende eines Programms festzustellen, wie hoch der maximale Stack-Bedarf während des Laufs war.

*Siehe auch* SETMEM (389), STACK (396).



## FREEFILE (Funktion)

I/O

*Seit* QuickBASIC 4.0

*Anwendung* `x% = FREEFILE`

*Nutzen* Gibt die kleinste noch unbenutzte Dateinummer (eine Nummer, unter der noch keine Datei mit OPEN geöffnet ist) zurück.

*Bemerkung* » Die Funktion FREEFILE ist besonders praktisch, wenn man Subroutinen schreibt, die unabhängig vom aufrufenden Programm sein sollen und die selbst mit Dateien arbeiten. Dann können diese sich nämlich mittels FREEFILE eine noch freie Dateinummer suchen und sicher sein, dem Hauptprogramm nicht in die Quere zu kommen.

» Es kann (unter DOS) trotz allem vorkommen, daß beim Öffnen einer Datei unter einer Nummer, die FREEFILE zurückgegeben hat, ein Fehler auftritt. Es dürfen nicht mehr Dateien geöffnet sein, als DOS Handles zur Verfügung stellt (`FILES = xx` in `CONFIG.SYS`). Speicherresidente Programme und QBX beziehungsweise BC selbst verbrauchen unter Umständen auch File-Handles, so daß bei zu kleiner FILES-Zahl leicht ein *Too many files*-Fehler auftreten kann.

» 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.

*Siehe auch* OPEN (348).

## GET (für Dateien) (Befehl)

I/O

*Seit* QuickBASIC 4.0

*Anwendung* `GET [#]dateinummer[, [satznummer], [satzvariable]]`

*Nutzen* Liest Daten aus einer Datei, die FOR RANDOM oder FOR BINARY geöffnet wurde. Der Unterschied zwischen beiden Dateiartern 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 so viele 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 CVx 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. Die Anwendung von Leerstrings ("" ) führt nicht zu einem Fehler, ist aber denkbar uneffektiv (sinnlos).

*Siehe auch* PUT (für Dateien) (366), OPEN (348), LOC (336), SEEK (386).

## GET (für Grafik) (Befehl)

**Grafik**

*Seit* QuickBASIC 2.0

*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) beziehungsweise 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 *Illegal function call*-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.

» Die oben genannte Reihenfolge der Elementspeicherung ändert sich auch beim Compiler-Switch /R nicht.

*Siehe auch* PUT (für Grafik) (367), DIM (299).

## GOSUB (Befehl)

## Struktur

*Seit* QuickBASIC 2.0

*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 beziehungsweise einem Zeilenlabel verzweigen, das in derselben Code-Einheit ist, also nicht vom Hauptprogramm in ein SUB oder umgekehrt, nicht von einem SUB in ein 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 beziehungsweise 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.

*Siehe auch* CLEAR (284), ON *event* GOSUB (346), SUB/FUNCTION (402).

## GOTO (Befehl)

Struktur

*Seit* QuickBASIC 2.0

*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.

*Siehe auch* GOSUB (319).

## HEX\$ (Funktion)

Standard

*Seit* QuickBASIC 2.0

*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  $-2^{31} \frac{3}{4} \times < 2^{31}$  führen zu einem *overflow*.

*Siehe auch* OCT\$ (345).

## IF...THEN...ELSE (Befehl)

Struktur

*Seit* QuickBASIC 2.0

*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 einzeliligen 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 (siehe dazu Kapitel 3.2, "Operatoren und Ausdrücke").

*Bemerkung* » Wenn Sie nach Syntax (1) mehrere IF-Befehle mit ELSE in eine Zeile packen, müssen Sie genau aufpassen, um keinen Fehler zu machen. Die Zeile

```
IF A = 0 THEN IF B = 0 THEN BEEP ELSE PRINT "OK"
```

würde einen Signalton ertönen lassen, wenn A und B 0 sind, und sie würde "OK" ausgeben, wenn A = 0 und B ungleich 0 ist. Ist aber A ungleich 0, würde diese Zeile überhaupt nichts tun. Wenn Sie wollen, daß "OK" ausgegeben wird, wenn A ungleich 0 ist, muß die Zeile lauten:

```
IF A = 0 THEN IF B = 0 THEN BEEP ELSE ELSE PRINT "OK"
```

Dann würde allerdings wiederum nichts passieren, wenn B ungleich 0 und A = 0 wäre.

» Vielfach ist es besser, eine lange Konstruktion nach Syntax (2) mit vielen ELSEIFs durch eine SELECT-CASE-Anweisung zu ersetzen.

*Siehe auch* SELECT CASE (387).

## \$INCLUDE (Metabefehl)

Standard

*Seit* BASIC 7.0 PDS

*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.

Da \$INCLUDE ein Metabefehl ist, muß ihm ein REM oder ein Apostroph (') vorangehen. Es ist wichtig, daß hinter \$INCLUDE sofort ein Doppelpunkt, dann ein Apostroph, der Dateiname und wieder ein Apostroph folgen; es dürfen keine zusätzlichen Leerzeichen vorhanden sein.

Mehrere \$INCLUDE-Prozesse können verschachtelt sein, das heißt, die Datei, die Sie mit \$INCLUDE laden, kann wiederum eine andere Datei mit \$INCLUDE aufrufen. Die maximale Verschachtelungstiefe ist 5.

*Bemerkung* » Wenn eine mit \$INCLUDE angeforderte Datei nicht im aktuellen Verzeichnis steht, suchen sowohl QBX 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
```

Dieses Verhalten ist mit der Version 7.0 implementiert worden.

## INKEY\$ (Funktion)

Standard

*Seit* QuickBASIC 2.0

*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 (einige davon jedoch mit GetShiftState und alle mit INP).

Die Codes für alle Tasten und Tastenkombinationen sind in einer Tabelle in Anhang D.2 aufgelistet.

*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.

» Wenn Sie ISAM in Ihrem Programm benutzen, wird nach 65.535 aufeinanderfolgenden INKEY\$-Abfragen, die alle kein Zeichen zurückgaben, weil der Tastaturpuffer leer war, automatisch ein CHECKPOINT-Befehl ausgeführt.

» Wenn INKEY\$ in einem kompilierten Programm benutzt wird, kann mit dem Kleinerzeichen oder dem Pipe-Symbol (!) die Standardeingabe umgeleitet werden, so daß das Programm keine Zeichen mehr von der Tastatur, sondern aus einer Datei liest. Davon wird auch INKEY\$ betroffen. Wird jedoch bei einer solchen Umleitung versucht, ein Zeichen mit INKEY\$ einzulesen, wenn bereits keines mehr in der Standardeingabe-Datei vorhanden ist, bricht das Programm auf der Stelle - ohne Fehlermeldung - ab. Außerdem ist es nicht möglich, erweiterte Zeichencodes aus einer Datei zu lesen; diese würden dann als zwei einzelne Zeichen, CHR\$(0) und ein anderes, interpretiert.

» 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\$(0) zurückgegeben wird.

*Beispiel* Dieses Programm zeigt den ASCII-Code jeder Taste an, die Sie drücken; ist es ein erweiterter Code, dem also ein CHR\$(0) vorangeht, gibt das Programm den Code invers aus.

```
DO
  a$ = INKEY$
  IF a$ <> "" THEN
    IF LEN(a$) = 2 THEN COLOR 0, 7
    PRINT ASC(RIGHT$(a$, 1))
    COLOR 7, 0
  END IF
LOOP
```

*Siehe auch* INPUT\$ (325), GetShiftState (515), INP (324), CHECKPOINT (422).

*Seit* QuickBASIC 2.0

*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 die spezifische "Sprache" der Chips kennen, also in welcher Reihenfolge welche Daten gesendet und empfangen werden. Außerdem können einige Chips nicht mit der Geschwindigkeit kommunizieren, wie es Ihnen von BASIC aus möglich ist, so daß Sie Warteschleifen benutzen müssen, um festzustellen, wann der Chip wieder bereit ist usw. Die Chip-Adressen sind auf normalen PCs und ATs nicht dieselben. Fazit: Sie sollten alles tun, um die Direktadressierung der Systemkanäle zu vermeiden.

*Beispiel* Dieses Beispiel fragt auf einem AT den Tastaturchip ab, um festzustellen, welche Taste gedrückt oder losgelassen wurde. Der Tastaturchip meldet den Scancode (siehe Anhang D.1) einer Taste, der unabhängig davon ist, ob Shift oder irgendeine andere Taste dazu gedrückt wurde. Wenn die Taste gedrückt wird, schickt der Chip den gewöhnlichen Scancode, wenn sie losgelassen wurde, addiert er zum Scancode 128. Für erweiterte Tasten (beispielsweise Enter auf dem Nummernblock) werden zusätzliche Codes übermittelt. Da auf dem Input-Kanal 96 die zuletzt vom Chip gesendete Information so lange "liegenbleibt", bis eine neue kommt, brauche ich eine Variable `AlteTaste%`, um prüfen zu können, ob sich etwas geändert hat.

```
AlteTaste% = INP(96)
DO
  Taste% = INP(96)
  IF Taste% <> AlteTaste% THEN
    IF Taste% > 128 THEN
      PRINT "losgelassen: ";
    ELSE
      PRINT "gedrückt: ";
    END IF
    PRINT "Taste"; Taste%
```

(Fortsetzung nächste Seite)



(Fortsetzung)

```
        AlteTaste% = Taste%  
    END IF  
LOOP
```

Siehe auch OUT (354), WAIT (416).

## INPUT\$ (Funktion)

Standard, I/O

Seit QuickBASIC 2.0

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.

Bemerkung » INPUT\$ liest, wenn auf sequentielle Dateien angewandt, auch Sonderzeichen wie die Neue-Zeile-Codekombination CHR\$(13) + CHR\$(10) (im Gegensatz zu INPUT und LINE INPUT).

» Wenn bei einem kompilierten Programm die Standardeingabe mittels des Kleiner- oder des Pipe-Zeichens (|) umgeleitet wird, liest INPUT\$ ohne *dateinummer* nicht mehr von der Tastatur, sondern aus der neuen Standardeingabe-Datei. Wenn diese Datei jedoch zu Ende ist, führt der Versuch, ein weiteres Zeichen mit INPUT\$ zu lesen, zu einem *Input past end*-Fehler.

» Tasten, für die ein Key-Trapping (siehe KEY) aktiv ist, werden von INPUT\$ ignoriert. Trotzdem wird, solange INPUT\$ auf eine Taste wartet, die Trapping-Routine nicht aufgerufen, sondern erst, wenn INPUT\$ ein Zeichen erhalten hat.

Siehe auch INKEY\$ (322), INPUT (326), LINE INPUT (336).

*Seit* QuickBASIC 2.0

*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 (wichtig bei INPUT in letzter Bildschirmzeile, um Scrollen zu vermeiden). 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.

Führende Leerzeichen in der Benutzereingabe beziehungsweise der Datei werden abgeschnitten, und Anführungszeichen (") werden als Anfang beziehungsweise Ende eines String-Inhalts erkannt.

Wenn Sie mit dem Verzicht-File NOEDIT.OBJ linken, kann der Benutzer, um seine Eingabe zu machen, nur die Backspace- und die Enter-Taste benutzen. Sonst stehen ihm zur Verfügung: Pfeil links, Pfeil rechts, CTRL-Pfeil links, CTRL-Pfeil rechts (zum jeweils nächsten Wort), Home (zum Anfang der Eingabe), End (zum Ende der Eingabe), CTRL End (alles bis zum Ende der Eingabe löschen), Insert (Einfügemodus ein- oder ausschalten), Del (Zeichen unter dem Cursor löschen), Esc (ganze Eingabe löschen), CTRL T (Funktionstastenanzeige ein- oder ausschalten) und CTRL C oder CTRL Break, um das Programm abubrechen.

*Bemerkung* » Wenn Sie Daten aus einer Datei lesen, die nicht zum benutzten Datentyp kompatibel sind, kann es einen Overflow-Fehler geben; wenn Sie den Inhalt eines Strings in eine numerische Variable lesen wollen, wird diese einfach 0.

- » Für Tastatur-Eingaben ist LINE INPUT zumeist besser geeignet.
- » Wenn Sie bei einer Tastatur-Eingabe keine Eingabeaufforderung verwenden, wird normalerweise einfach ein Fragezeichen angezeigt. Um auch das zu verhindern, müssen Sie statt INPUT a\$ schreiben: INPUT "", a\$.
- » Während einer INPUT-Warteschleife ist jede Art von Event-Trapping kurzfristig suspendiert.

*Siehe auch* LINE INPUT (336), INPUT\$(325).

## INSTR (Funktion)

**Standard**

*Seit* QuickBASIC 2.0

*Anwendung*  $x\% = \text{INSTR}([anfang\%,] string\$, teilstring\$)$

*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.

## INT (Funktion)

**Standard**

*Seit* QuickBASIC 2.0

*Anwendung*  $x = \text{INT} (y)$

*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.

*Siehe auch* CINT (283), INT (327).

## IOCTL\$ (Funktion)

DOS

*Seit* QuickBASIC 2.0

*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; die Blocktreiber (A:, B:, C: etc.) und die Gerätetreiber LPTx, COMx, SCRn, CONS und PIPE können das nicht.

» Sie werden diesen Befehl kaum benötigen, es sei denn, Sie arbeiten mit spezieller Treibersoftware wie zum Beispiel CD-ROM-Laufwerkstreibern. Dann können Sie dem entsprechenden Treiberhandbuch entnehmen, wie die Status-Meldungen des Treibers zu entschlüsseln sind.

*Siehe auch* IOCTL (328).

## IOCTL (Befehl)

DOS

*Seit* QuickBASIC 2.0

*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* » IOCTL kann nur mit Gerätetreibern benutzt werden, die IOCTL-Strings verarbeiten; die Blocktreiber (A:, B:, C: etc.) und die Gerätetreiber LPTx, COMx, SCRn, CONS und PIPE können das nicht.

» Sie werden diesen Befehl kaum benötigen, es sei denn, Sie arbeiten mit spezieller Treibersoftware wie zum Beispiel CD-ROM-Laufwerkstreibern. Dann können Sie dem entsprechenden Treiberhandbuch entnehmen, welche Befehle an den Treiber geschickt werden können und welche Wirkung sie haben.

» Für den Zugriff auf IOCTL-Gerätetreiber stellt DOS außerdem eine Reihe von Interrupt-Funktionen zur Verfügung, die hier nicht beschrieben werden, da man sie zu selten benötigt.

*Siehe auch* IOCTL\$ (328).

## KEY (zur Belegung der Funktionstasten) (Befehl)    Standard

*Seit*                QuickBASIC 2.0

*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 beziehungsweise 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.

*Bemerkung*        » Die Funktionstastenanzeige in der untersten Bildschirmzeile kann auch mit CTRL T innerhalb eines INPUT-Aufrufs ein- und ausgeschaltet werden.

*Beispiel*                Dieses Programm liest Zahlen ein, bis der Benutzer QUIT eingibt oder F1 drückt:

```
KEY 1, "QUIT" + CHR$(13)
' (CHR$(13) simuliert Enter)

PRINT "Geben Sie Zahlen ein!"
PRINT "Ende mit QUIT oder F1."
DO
    LINE INPUT x$
    IF x$ = "QUIT" THEN EXIT DO
    Zaehler = Zaehler + 1
    Wert(Zaehler) = VAL(x$)
LOOP
```

*Siehe auch*    INKEY\$ (322), INPUT\$ (325), INPUT(326), LINE INPUT (336).

*Seit* QuickBASIC 2.0

*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(*nummer*) GOSUB spezifizierte Routine gesprungen. KEY(*nummer*) OFF schaltet das Key-Trapping ab, während KEY(*nummer*) STOP es nur unterbindet. Wenn nach einem KEY(*nummer*) STOP-Befehl wieder ein KEY(*nummer*) ON folgt, wird nachträglich noch auf einen Tastendruck reagiert, den der Benutzer eventuell in der Zwischenzeit getätigt hat, während bei KEY(*nummer*) OFF die Taste wieder ihren ganz gewöhnlichen Status erhält. Eine Taste, deren Key-Trapping nur mit KEY(*nummer*) STOP unterbunden ist, wird bei INKEY\$ und INPUT\$ ignoriert, während eine Taste, deren Key-Trapping mit KEY(*nummer*) OFF abgeschaltet wurde, bei INKEY\$ oder INPUT\$ ihren üblichen Tastencode zurückgibt.

Folgende *nummern* sind möglich:

<i>nummer</i>	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 D.1; 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 etc.

Das Beispiel zeigt einige mögliche Kombinationen.

*Bemerkung* » Indem Sie mittels dieser Befehle ein Key-Trapping für die Tastenkombination CTRL-ALT-Del etablieren, können Sie sogar verhindern, daß der Rechner durch den Druck auf diese Tasten neu gebootet wird. Beachten Sie aber, daß man, wenn Sie nur ein Trapping programmieren, immer noch zum Beispiel mit Shift-CTRL-ALT-Del oder mit Num Lock-CTRL-ALT-Del etc. booten kann. Um jede Möglichkeit abzufangen, müßten Sie insgesamt 16 Tastenkombinationen abfangen, wenn Sie davon ausgehen, daß jemand auf die Idee kommen könnte, wenn Num Lock und Caps Lock an sind, die Kombination Shift-CTRL-ALT und die zusätzliche Del-Taste der erweiterten Tastatur zu drücken.

» Während einer Tastatureingabe mit INPUT\$, INPUT oder LINE INPUT sind Key-Trapping-Routinen nicht aktiv.

*Beispiel*

Die Kombination...	bewirkt Trapping für...
CHR\$(0) + CHR\$(37)	K
CHR\$(8) + CHR\$(37)	ALT K
CHR\$(12) + CHR\$(37)	CTRL-ALT-K
CHR\$(64) + CHR\$(37)	K, wenn Caps Lock an ist

```

KEY 15, CHR$(12) + CHR$(51) ' CTRL-ALT-;
KEY 16, CHR$(12) + CHR$(52) ' CTRL-ALT-:
ON KEY(1) GOSUB FlGedrueckt
ON KEY(15) GOSUB Semikolon
ON KEY(16) GOSUB Doppelpunkt
KEY(0) ON 'schaltet alle drei ein

DO 'Endlosschleife
  PRINT "Ich würfle"; INT(RND * 6) + 1
LOOP

FlGedrueckt:
  PRINT "HILFE:"
  PRINT "Das Programm simuliert das Würfeln"
  PRINT "mit einem 6seitigen Würfel."
  PRINT "Mit CTRL-ALT-Semikolon oder CTRL-"
  PRINT "ALT-Doppelpunkt können Sie zwischen"
  PRINT "40- und 80-Zeichen-Modus umschalten."
  PRINT "Drücken Sie eine Taste!"
  DO: LOOP UNTIL LEN(INKEY$)
  RETURN

Semikolon: WIDTH 40: RETURN
Doppelpunkt: WIDTH 80: RETURN

```

Siehe auch ON event GOSUB (346), EVENT ON/OFF (309).

## KILL (Befehl)

DOS

Seit QuickBASIC 2.0

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.

Bemerkung » Im Gegensatz zum DEL-Befehl in DOS fragt KILL niemals "Are you sure? (Y/N)" o.ä. nach, wenn Sie zum Beispiel mit KILL "\*.\*" alle Dateien löschen.



## LBOUND (Funktion)

Standard

*Seit* QuickBASIC 4.0

*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.

*Siehe auch* DIM (299), UBOUND (411), OPTION BASE (353).

## LCASE\$ (Funktion)

Standard

*Seit* QuickBASIC 4.0

*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 Buchstaben mit den ASCII-Codes 65-90, also A-Z, in Kleinbuchstaben verwandelt werden.

LCASE\$ wandelt außer den genannten 26 Buchstaben keine um, so daß Umlaute jedweder Art unberücksichtigt bleiben.

*Siehe auch* UCASE\$ (411).

## LEFT\$ (Funktion)

Standard

*Seit* QuickBASIC 2.0

*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.

*Siehe auch* RIGHT\$ (374), MID\$ (342).

## LEN (Funktion)

Standard

*Seit* QuickBASIC 4.0

*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.

## LET (Befehl)

Standard

*Seit* QuickBASIC 2.0

*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 beziehungsweise 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.

*Siehe auch* LSET (340).

## LINE (Befehl)

Grafik

*Seit* QuickBASIC 2.0

*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 Position des Grafikcursors genommen (der Grafikcursor wird nach einem CLS- oder SCREEN-Befehl auf die Bildschirmmitte gesetzt; CIRCLE setzt ihn auf den Kreismittelpunkt, PSET, PRESET, DRAW und LINE setzen ihn auf den im jeweiligen Befehl zuletzt angesprochenen Punkt). 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 Linie) wäre dementsprechend  $2^0 + 2^2 + 2^4 + \dots + 2^{14}$ . Eine deutlich gestrichelte Linie könnte zum Beispiel erreicht werden, indem man von 16 Punkten immer 8 hintereinander wegläßt; der Linientyp dafür wäre also  $2^0 + 2^1 + \dots + 2^7 = 255$ . Eine große Einschränkung ist leider, daß man nur eine 16-Bit-Maske benennen kann, die dann immer wiederholt wird. Deshalb ist es zum Beispiel nicht möglich, eine Linie zu zeichnen, bei der immer 16 Pixel gezeichnet und 16 ausgelassen werden.

Natürlich können auch ungleichmäßige Linientypen benutzt werden, zum Beispiel fünf Punkte auslassen, drei zeichnen, wieder fünf auslassen und wieder drei zeichnen. Wenn Ihr Muster allerdings nicht in 16 Bits paßt, ist die optische Wirkung unter Umständen unschön.

Beachten Sie darüberhinaus, 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. Sie müßten, um das zu erreichen, zunächst die

gewünschte Linie als durchgezogene in einer anderen Farbe zeichnen.

» 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.

*Siehe auch* CIRCLE (283), PSET (366), PRESET (363), DRAW (302), SCREEN (378), PALETTE (355), COLOR (286).

LINE INPUT (Befehl)	Standard, I/O
---------------------	---------------

*Seit* QuickBASIC 2.0

*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.

*Bemerkung* » Alles weitere siehe INPUT.

*Siehe auch* INPUT (326).

LOC (Funktion)	I/O
----------------	-----

*Seit* QuickBASIC 4.0

*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).

LOC kann nicht auf die Geräte SCRN:, CONS:, LPTx: und KYBD: (siehe OPEN) angewendet werden.

Siehe auch EOF (307), LOF (338), OPEN (348).

## LOCATE (Befehl)

Standard

Seit QuickBASIC 2.0

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 beziehungsweise 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).

Im Protected Mode werden die Werte 1-16 benutzt.

Bemerkung » Experimentieren Sie nicht zuviel mit der Cursorgrößen-Einstellung, da sie erfahrungsgemäß auf verschiedenen Systemen ziemlich unterschiedliche Folgen haben kann (aller Theorie zum Trotz). Wenn *start* größer *stop* ist, teilt sich der Cursor üblicherweise in zwei Hälften - aber nicht auf VGA-Systemen.

Siehe auch CLS (285).

## LOCK, UNLOCK (Befehle)

I/O

Seit QuickBASIC 4.0

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) beziehungsweise dieses Byte (bei BINARY-Dateien) geschützt; setzen Sie noch *TO ende* hinzu, werden alle Datensätze beziehungsweise 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.

» LOCK und UNLOCK können nur auf gewöhnliche Dateien, nicht auf ISAM-Dateien oder Geräte (wie LPT1:, COM1: etc) angewandt werden.

» Mit OPEN können Sie ebenfalls fremden Zugriff auf Dateien verhindern, auch getrennt nach Schreib- und Lesezugriffen.

*Siehe auch* OPEN (348).

## LOF (Funktion)

I/O

*Seit* QuickBASIC 2.0

*Anwendung*  $x\& = \text{LOF}(\text{dateinummer})$

*Nutzen* Gibt die Größe einer Datei 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.

Bei ISAM-Datenbanken wird die Anzahl der Datensätze in der Datenbank zurückgegeben.

*Siehe auch* BOF (421), EOF (307), LOC (336).

## LOG (Funktion)

Standard

*Seit* QuickBASIC 2.0

*Anwendung*  $x = \text{LOG}(\text{zahl})$

*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  $\text{Log}_{10} = \text{LOG}(x) / \text{LOG}(10)$  berechnen.

Siehe auch EXP (311).

<b>LPOS (Funktion)</b>	<b>Standard</b>
------------------------	-----------------

*Seit* QuickBASIC 2.0

*Anwendung*  $x = \text{LPOS}(\text{druckernummer})$

*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:.

*Bemerkung* » LPOS ist eine der unbrauchbarsten Funktionen. Theoretisch entspricht der Funktionswert zwar der aktuellen Druckkopfposition, aber nur so lange, wie jedes an den Drucker gesandte Zeichen den Druckkopf auch um eine Stelle vorwärts bewegt - und diese Voraussetzung ist beispielsweise für Steuersequenzen, die gar nicht gedruckt werden, und für TAB-Zeichen, die den Druckkopf meist um mehr als ein Zeichen bewegen, nicht erfüllt.

Siehe auch WIDTH (417).

<b>LPRINT (Befehl)</b>	<b>Standard</b>
------------------------	-----------------

*Seit* QuickBASIC 2.0

*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 eine neue Zeile angefangen. Das sorgt insbesondere bei Grafikausdrucken für Schwierigkeiten, da für eine Grafikzeile weit mehr als 80 Zeichen an den Drucker gesendet werden - Steuerzeichen, die nicht als Buchstaben gedruckt werden. Darum kümmert sich LPRINT aber nicht (wie auch LPOS), und deshalb

würde eine Grafikzeile alle 80 Zeichen durch CHR\$(13) und CHR\$(10) unterbrochen. Sie können den WIDTH-Befehl benutzen, um das zu verhindern, oder aber der ersten Bemerkung Folge leisten.

*Siehe auch* PRINT (363), LPOS (339), WIDTH (417).

## LSET (Befehl)

**Standard**

*Seit* QuickBASIC 4.0

*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.

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.

Betrachten Sie dazu das Beispiel:

*Beispiel* (zu Variablen von verschiedenem selbstdefinierten Typ):

```
TYPE Adressel
    Name AS STRING * 20
    Vorname AS STRING * 20
    Ort AS STRING * 20
END TYPE

TYPE Adresse2
    Name AS STRING * 20
    Vorname AS STRING * 20
    Ort AS STRING * 20
    Telefon AS STRING * 25
END TYPE

TYPE VollerName
    Name AS STRING * 40
END TYPE
```

(Fortsetzung nächste Seite)



(Fortsetzung)

```
TYPE Adresse3
    Name AS STRING * 20
    Vorname AS STRING * 20
    Straße AS STRING * 20
    Ort AS STRING * 20
END TYPE

DIM A1 AS Adresse1, A2 AS Adresse2
DIM A3 AS Adresse3, A4 as VollerName

A1.Name = "Halmbach"
A1.VorName = "Gerhard"
A1.Ort = "Lauenburg"

' Das geht, hier wird das Telefon-Feld von A2
' nicht verändert und bleibt somit leer:
LSET A2 = A1

' Das gibt zwar keinen Fehler, führt aber zu
' dem Ergebnis, daß im Straße-Feld von A3 das
' steht, was bei A2 im Ort-Feld stand, und im
' Ort-Feld von A3 sich die ersten zwanzig
' Zeichen des Telefon-Feldes von A2
' wiederfinden:

LSET A3 = A2

' Dieser Befehl wird zur Folge haben, daß im
' Name-Feld von A4 sich Name und Vorname von A3
' - mit entsprechender Anzahl Leerzeichen da-
' zwischen - befinden, während der Rest von A3
' verschluckt wird:
LSET A4 = A3

' Und hier schließlich kehren Name und Vorname
' wieder heil in A1 zurück, nur der Ort bleibt
' auf der Strecke, weil er ja schon in A4
' keinen Platz mehr fand:
LSET A1 = A4
```

*Siehe auch*    RSET (375), LSET (340).

## LTRIM\$ (Funktion)

Standard

*Seit* QuickBASIC 4.0

*Anwendung*  $x\$ = \text{LTRIM\$}(y\$)$

*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. Siehe dazu "Strings mit fester und variabler Länge" in Kapitel 10, Abschnitt 2.

» LTRIM\$ ist zuweilen für die Ausgabe von Zahlen sinnvoll - siehe Bemerkung zu PRINT.

*Siehe auch* RTRIM\$ (375), PRINT (363).

## MID\$ (Funktion und Befehl)

Standard

*Seit* QuickBASIC 2.0

*Anwendung* (1)  $x\$ = \text{MID\$}(\text{stringausdruck}, \text{anfang} [, \text{laenge}])$   
(2)  $\text{MID\$}(x\$, \text{anfang} [, \text{laenge}]) = \text{stringausdruck}$

*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.

Sowohl in Syntax (1) als auch in Syntax (2) darf *anfang* nie kleiner als 1 werden.

*Bemerkung* » Wie der Abschnitt "String-Verknüpfungen" im Kapitel 19.1, "Effiziente Programmierung", zeigt, kann durch den geschickten Einsatz von MID\$ viel Zeit gespart werden.

*Siehe auch* LEFT\$ (333), RIGHT\$ (374).

## MKDIR (Befehl)

DOS

*Seit* QuickBASIC 2.0

*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.

» Beachten Sie, daß Sie ein Verzeichnis erst erstellen können, wenn das übergeordnete Verzeichnis schon existiert. MKDIR "C:\DATEN\TEXTE" funktioniert nur, wenn das Verzeichnis C:\DATEN schon existiert. Ansonsten müßte es erst mit MKDIR "C:\DATEN" erstellt werden.

*Siehe auch* CHDIR (282), CURDIR\$ (291), RMDIR (374).

## MKx\$ (Funktionen)

Standard

*Seit* QuickBASIC 2.0

*Anwendung* x\$ = MKI\$(y%)

x\$ = MKL\$(y&) (seit 4)

x\$ = MKS\$(y!)

x\$ = MKD\$(y#)

x\$ = MKC\$(y@) (seit 7)

*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 CVx-Funktionen bewirken das Gegenteil: Sie wandeln Code-Strings wieder in Zahlen um. Wenn man nicht mehr mit FIELD arbeitet, sondern die moderneren TYPE-Deklarationen

zur Verarbeitung von Random-Access-Dateien benutzt, benötigt man diese Funktionsgruppen zumeist nicht mehr.

» Der Compiler-Switch /MBF beeinflußt die Funktionen MKS\$ und MKD\$. Siehe dazu Kapitel 6.3, "Der Compiler BC.EXE".

*Siehe auch* MKxMBF\$ (344), CVx (291), FIELD (311).

MKxMBF\$ (Funktionen)		Standard
<i>Seit</i>	QuickBASIC 3.0	
<i>Anwendung</i>	$x\$ = \text{MKSMBF}\$(y!)$ $x\$ = \text{MKDMBF}\$(y\#)$	
<i>Nutzen</i>	Diese beiden Funktionen codieren wie ihre Nachfolger MKS\$ und MKD\$ SINGLE- beziehungsweise 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.	
<i>Bemerkung</i>	» Von QuickBASIC 2 auf QuickBASIC 3 wurde die interne Repräsentation der Fließkommazahlen geändert, um coprozessor kompatibel zu werden. Die Zahlen wurden nicht mehr im Microsoft-eigenen Binärformat, sondern im weitverbreiteten IEEE-Format gespeichert. Will man heute noch Random-Access-Dateien im Format der "MBF-Zeit" beschreiben (eventuell, um kompatibel zu alten Programmen zu bleiben), muß man mit FIELD arbeiten und sich der MKxMBF\$-Funktionen bedienen, die identisch sind mit den MKx\$-Funktionen von damals. Für andere Zwecke sollten diese Funktionen, wie auch ihre Gegenstücke CVxMBF, nicht mehr benutzt werden.	
<i>Siehe auch</i>	CVxMBF (292), MKx\$ (343), FIELD (311).	

NAME (Befehl)		DOS
<i>Seit</i>	QuickBASIC 2.0	
<i>Anwendung</i>	(1) NAME <i>dateiname1\$</i> AS <i>dateiname2\$</i> (2) NAME <i>directory1\$</i> AS <i>directory2\$</i>	
<i>Nutzen</i>	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. <i>dateiname1\$</i> muß der Name einer vorhandenen Datei sein, <i>dateiname2\$</i> darf noch nicht vorhanden, muß aber gültig sein. <i>dateiname2\$</i> muß auf demselben Laufwerk wie <i>dateiname1\$</i> sein. Wenn <i>dateiname2\$</i> eine andere Directory-Angabe als <i>dateiname1\$</i> 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.

*Bemerkung* » Im Gegensatz zum DOS-Befehl REN sind auch Umbenennungen auf anderen Laufwerken als dem aktuellen möglich, so zum Beispiel NAME "A:COMMAND.COM" AS "A:COMMAND.ALT", wenn man sich auf Laufwerk C: befindet.

## OCT\$ (Funktion)

Standard

*Seit* QuickBASIC 2.0

*Anwendung* `x$ = OCT$(y)`

*Nutzen* OCT\$ gibt ihr Argument in oktaler Schreibweise zurück. Obwohl eine Oktalzahl nur Ziffern enthält, wird ein String erzeugt.

*Bemerkung* » Wer braucht schon Oktalzahlen?

*Siehe auch* HEX\$ (320).

## ON ERROR (Befehl)

Trapping

*Seit* BASIC 7.0 PDS

*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 (zur Definition der Begriffe Modul- und Prozedurcode siehe Kapitel 4.4).

`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 LOCAL hinter ON. Lokale Error-Handler gibt es nur in Prozeduren und Funktionen, 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 (wie das in allen früheren BASIC-Versionen der Fall war).

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.

*Bemerkung* » Klärendes zum Thema Fehlerbehandlung finden Sie in Kapitel 13.

*Siehe auch* ERL, ERR (308), ERDEV, ERDEV\$ (308).

## ON event GOSUB (Befehl)

Trapping

*Seit* QuickBASIC 2.0; UEVENT und SIGNAL seit BASIC 6.0

*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.

Wenn Sie für *zeilennummer/-label* 0 einsetzen, wird die Event-Trapping-Routine deaktiviert.

Folgende *events* sind möglich:

COM( <i>n</i> )	Ruft die Trapping-Routine auf, wenn am durch <i>n</i> spezifizierten Kommunikations-Port Zeichen anliegen.
KEY( <i>n</i> )	Ruft die Trapping-Routine auf, wenn die Taste <i>n</i> gedrückt wird. Zu möglichen Tastennummern siehe KEY.
PEN	Ruft die Trapping-Routine auf, wenn mit dem Lichtstift (Lightpen) auf eine Bildschirmposition gezeigt wird.
PLAY( <i>n</i> )	Ruft die Trapping-Routine auf, wenn die <i>n</i> te Note gespielt wurde und nun nur noch <i>n</i> -1 Noten zu spielen sind. ON PLAY GOSUB wirkt nur, wenn der PLAY-Befehl im Hintergrund arbeitet (siehe Befehl "MB" bei PLAY). <i>n</i> darf nicht größer als 32 sein.
SIGNAL( <i>n</i> )	Ruft die Trapping-Routine auf, wenn ein OS/2-Prozeßsignal gesetzt wird. <i>n</i> ist die Signalnummer (1 für Druck auf CTRL C, 2 wenn die Datenverbindung zwischen zwei Programmen, die "pipe connection", unterbrochen wird, 3, wenn ein Programm beendet wird, 4 für Druck auch CTRL Break, 5, 6 und 7 für die selbstdefinierbaren Prozeßsignale A, B und C, die mit der OS/2-Funktion DOSFLAGPROCESS gesetzt werden können).
STRIG( <i>n</i> )	Ruft die Trapping-Routine auf, wenn ein Feuerknopf an einem der beiden Joysticks gedrückt wurde. <i>n</i> 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( <i>n</i> )	Ruft die Trapping-Routine alle <i>n</i> Sekunden auf. <i>n</i> 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.

*Bemerkung* » Konsultieren Sie die Einträge zu den zugehörigen Event-Befehlen für weitere Informationen.

*Siehe auch* COM (287), KEY (329), PEN (358), SIGNAL (393), STRIG (401), TIMER (409), UEVENT (411), EVENT ON/OFF (309).

*Seit* QuickBASIC 2.0

*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 beziehungsweise -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 *Illegal function call*.

*Bemerkung* » SELECT CASE eignet sich in den meisten Fällen besser, um unterschiedliche Verzweigungen auszulösen.

*Siehe auch* GOSUB (319), GOTO (320), SELECT CASE (387).

**OPEN (Befehl)****I/O**

*Seit* QuickBASIC 4.0

*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:, LPTx: und CONS: benutzt.

Die Syntax (2) wird ausschließlich verwendet, um Kommunikationsschnittstellen zu öffnen.

BASIC 7.1 PDS 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 <i>LEN=satzlaenge</i> eine große Rolle, da auf RANDOM-Dateien nur satzweise zugegriffen werden kann. Standard für <i>satzlaenge</i> ist hier 128 Bytes. RANDOM-geöffnete Dateien 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 senden 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 <i>n</i>	setzt den Empfangspuffer auf eine Größe von <i>n</i> 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 <i>n</i> ist 32.767.
TB <i>n</i>	setzt den Sendepuffer auf eine Größe von <i>n</i> 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 <i>n</i> ist 32.767.
RS	verhindert, daß die RTS-Leitung beim OPEN-Befehl unter Spannung gesetzt wird.
CD <i>n</i>	verursacht einen Timeout-Fehler (ERDEV = 130), wenn die DCD-Leitung (Carrier Detect) der Schnittstelle <i>n</i> Millisekunden lang keine Spannung führt. Wenn CD oder der Parameter <i>n</i> weggelassen wird oder <i>n</i> 0 ist, wird die DCD-Leitung nicht beachtet. <i>n</i> liegt zwischen 0 und 65.535.
CS <i>n</i>	verursacht einen Timeout-Fehler (ERDEV = 128), wenn die CTS-Leitung (Clear To Send) der Schnittstelle <i>n</i> Millisekunden lang keine Spannung führt. Wenn CS oder der Parameter <i>n</i> weggelassen wird, wird eine Sekunde gewartet; ist <i>n</i> 0, wird der Status der CTS-Leitung ignoriert. <i>n</i> liegt zwischen 0 und 65.535.
DS <i>n</i>	verursacht einen Timeout-Fehler (ERDEV = 129), wenn die DSR-Leitung (Data Set Ready) der Schnittstelle <i>n</i> Millisekunden lang keine Spannung führt. Wenn DS oder der Parameter <i>n</i> weggelassen wird, wird eine Sekunde gewartet; ist <i>n</i> 0, wird der Status der DSR-Leitung ignoriert. <i>n</i> liegt zwischen 0 und 65.535.
OP <i>n</i>	verursacht einen Timeout-Fehler, wenn nach <i>n</i> Millisekunden nicht alle Kommunikationsleitungen aktiv sind. <i>n</i> 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 <i>n</i> 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 LPTx: (nur OUTPUT). Eine Ausgabe in eine solche Datei entspricht der Ausgabe mit LPRINT. Wenn Sie an den Gerätenamen ein BIN anfügen (LPTx: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 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 TSCNIOxx 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 TSCNIOxx überhaupt nicht sichtbar.

Der LOCATE-Befehl wirkt auf alle drei Arten der Zeichenausgabe, nur dann nicht, wenn in eine Datei umgeleitet wird.

» OPEN dient auch zum Öffnen von ISAM-Datenbanken. Mehr dazu siehe im ISAM-Referenzteil.

*Beispiel*

```
OPEN "DATA.QRY" FOR RANDOM AS #1 LEN = 45
```

Öffnet DATA.QRY für Random Access mit der Dateinummer 1 und der Satzlänge 45.

```
OPEN "DATA.QRY" FOR RANDOM AS #1 LEN = LEN(Adr)
```

Öffnet DATA.QRY für Random Access mit der Dateinummer 1 und einer Satzlänge, die so groß ist, daß die Variable Adr hineinpaßt. Solche Formulierungen werden meist benutzt, wenn Adr einen selbstdefinierten Datentyp hat und DATA.QRY eine Adreßdatei ist. Wenn am Datentyp später eine Änderung gemacht wird, muß am Programm nichts mehr geändert werden, weil der OPEN-Befehl sich automatisch anpaßt (nichtsdestotrotz wären eventuell mit der alten Satzlänge gespeicherte Daten natürlich nicht mehr brauchbar).

```
OPEN "KUNDEN.DAT" LOCK READ WRITE FOR INPUT AS #1  
LEN = 20000
```

Öffnet KUNDEN.Dat für sequentielles Lesen (zumeist mit INPUT oder LINE INPUT) als Datei Nr. 1. Kein anderer Prozeß im Netzwerk darf die Datei lesen oder schreiben, solange sie geöffnet ist. Als Dateipuffer werden 20.000 Bytes bereitgestellt.

```
OPEN "COM1:9600,N,8,1" FOR RANDOM AS #1
```

Öffnet die Schnittstelle COM1: mit 9600 Baud, 8 Datenbits, einem Stopbit und keiner Parität für Random Access als Datei Nummer 1.

OPEN wird auch im Beispiel zu DO...LOOP verwendet.

*Siehe auch* LEN (334), WIDTH (417), LOC (336), LOF (338), CLOSE (285), DO...LOOP (301).

## OPTION BASE (Befehl)

**Standard**

*Seit* QuickBASIC 2.0

*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.

*Siehe auch* DIM (299).

## OUT (Befehl)

Standard

<i>Seit</i>	QuickBASIC 2.0
<i>Anwendung</i>	OUT <i>kanal</i> , <i>byte</i>
<i>Nutzen</i>	Sendet auf einem Hardware-I/O-Kanal ein Byte. <i>kanal</i> ist die Kanalnummer (0 bis 65.535), <i>byte</i> das zu sendende Byte (0 bis 255).
<i>Bemerkung</i>	» 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!
<i>Siehe auch</i>	INP (324), WAIT (416).

## PAINT (Befehl)

Grafik

<i>Seit</i>	QuickBASIC 2.0
<i>Anwendung</i>	PAINT [STEP] ( <i>x</i> , <i>y</i> ) [, [ <i>fuellen</i> [, [ <i>rahmenfarbe</i> [, <i>hintergrund</i> \$]]]]
<i>Nutzen</i>	<p>Füllt einen begrenzten Bereich auf dem Bildschirm von einem Punkt ausgehend mit einer Farbe oder einem Muster aus. (<i>x</i>, <i>y</i>) sind die Koordinaten des Punktes, von dem aus gefüllt werden soll. Wenn STEP angegeben wird, werden die Koordinaten relativ zur Position des Grafikcursors verstanden.</p> <p><i>fuellen</i> kann entweder ein String oder eine Zahl sein. Wenn Sie eine Zahl angeben, wird vom angegebenen Punkt aus in der Farbe <i>fuellen</i> gefüllt. Geben Sie einen String an, wird vom aktuellen Punkt aus mit dem Muster <i>fuellen</i> gefüllt. <i>fuellen</i> 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: Ein einfaches Programm zur Erstellung schwarz-weißer Musterstrings ist im Lieferumfang des BASIC PDS enthalten und heißt EDPAT.BAS. Musterstrings für PAINT, die für fast alle Grafikzwecke ausreichen müßten, lassen sich mit den Routinen GetPattern\$ und MakeChartPattern\$ aus der Presentation Graphics-Toolbox erzeugen (siehe dort). Experimentieren Sie ruhig ein wenig mit diesem Stringparameter, es kann ja nichts schaden.</p> <p><i>rahmenfarbe</i> ist die Farbe, in der das Objekt gezeichnet wurde, das PAINT füllen soll. Wenn beim Füllen Linien der Farbe <i>rahmenfarbe</i> erreicht werden, hört PAINT dort mit dem Füllen auf. Wenn <i>rahmenfarbe</i> weggelassen wird, wird die gleiche Farbe wie <i>fuellen</i></p>

angenommen. Wenn auch diese weggelassen wird, ist es die aktuelle Grafikfarbe.

*hintergrund\$* ist ein String, der ein Muster angibt, das beim Füllen übergangen werden soll. *hintergrund\$* ist für solche Zwecke gedacht, in denen eine bereits gefüllte Fläche neu gefüllt werden soll, und kann dann eingesetzt werden, um zu verhindern, daß PAINT einen Teil der Füllung für einen Rand hält und mit dem Füllen aufhört. Der Einsatz von *hintergrund\$* ist allerdings so kompliziert und auch sehr beschränkt, daß es sehr viel besser ist, den neu zu füllenden Bereich einfach zu löschen und neu zu füllen.

*Bemerkung* » PAINT setzt den Grafikcursor auf die ihm übergebenen Koordinaten.

*Beispiel* Dieses Programm zeichnet zunächst konzentrische Kreise und füllt diese dann verschiedenfarbig, so daß konzentrische Ringe entstehen.

```
DIM Farbe AS INTEGER

SCREEN 12

FOR i% = 40 TO 240 STEP 40
    Farbe = Farbe + 1
    ' Kreis in Weiß (Farbe 7)
    CIRCLE (320, 240), i%, 7
    ' Füllen mit Farbe, Weiß wird als Grenzfarbe
    ' angegeben
    PAINT (300 + i%, 240), Farbe, 7
NEXT
```

*Siehe auch* SCREEN (378), GetPattern\$ (477), MakeChartPattern\$ (480).

## PALETTE (Befehl)

Standard

*Seit* QuickBASIC 4.0

*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*. Das erste Farbattribut erhält also dann die Farbe *array(index)* usw.

Wenn im Array die Zahl -1 vorkommt, wird die dem betreffenden Farbattribut zugeordnete Farbe nicht geändert.

*Bemerkung* » Der PALETTE-Befehl ist nicht nur im Grafikmodus verwendbar, er kann auch frische Farben in den Textmodus bringen. Während üblicherweise nur 16 Standard-Vordergrundfarben und 8 Hintergrundfarben verfügbar sind, die in allen Programmen gleich aussehen, können Sie durch PALETTE zwar auch nur maximal 16 verschiedene Vorder- und 8 verschiedene Hintergrundfarben benutzen, aber bei EGA-, VGA- und MCGA-Karten können Sie aus 64 Farben auswählen, die Sie diesen Attributen zuordnen.

» Eine Übersicht über die Anzahl der verfügbaren Farbattribute und Farben bei verschiedenen Bildschirmen, Grafikkarten und Bildschirm-Modi finden Sie im Abschnitt über SCREEN.

*Siehe auch* SCREEN (378), COLOR (286).

PCOPY (Befehl)	Standard
----------------	----------

*Seit* QuickBASIC 4.0

*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. (Siehe dazu aber auch GetBackground und PutBackground in der General-Toolbox.)

*Bemerkung* » Eine ausführliche Beschreibung der einzelnen Bildschirmmodi mit der Anzahl der verfügbaren Bildschirmseiten finden Sie beim SCREEN-Befehl.

*Siehe auch* SCREEN (378), GetBackground (514), PutBackground (516).



*Seit* QuickBASIC 2.0

*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.

*Bemerkung* » QBX speichert sämtliche Arrays - bis auf Arrays von Strings mit variabler Länge - unter Umständen im Expanded Memory, wenn sie kleiner als 16 KB sind. Solche im Expanded Memory befindlichen Arrays können nicht mit PEEK und POKE bearbeitet werden. Im Zweifelsfalle starten Sie QBX ohne den /Ea-Switch, dann werden keine Arrays in das Expanded Memory ausgelagert.

» PEEK kann zwar keinen Schaden anrichten, da es lediglich Speicherstellen ausliest, ihr Komplement POKE aber ist schon eher geeignet, um Daten zu zerstören oder ein Programm durch unachtsame Anwendung abstürzen zu lassen. Vorsicht also mit diesen Befehlen!

*Beispiel* Dieses kleine Programm ermittelt die Adresse eines gegebenen Strings und wandelt dann alle seine Buchstaben in Großbuchstaben um. Dabei werden drei Umlaute berücksichtigt, die Liste ließe sich aber beliebig fortsetzen. Das Beispiel erhebt nicht den Anspruch, die schnellstmögliche Lösung zu sein, ist aber schneller als eine Prüfung mit `SELECT CASE ASC(MID$(...))`.

```
SUB Gross (Text AS STRING)
  DEF SEG = SSEG(Text)
  FOR i% = SADD(Text) TO SADD(Text)+LEN(Text)
    SELECT CASE PEEK(i%)
      ' Kleinbuchstaben:
      CASE 97 TO 122: POKE i%, PEEK(i%) AND 95
      ' die drei Umlaute:
      CASE 129: POKE i%, 154
      CASE 132: POKE i%, 142
      CASE 148: POKE i%, 153
      CASE ELSE
      END SELECT
    NEXT
  END SUB
```

*Siehe auch* POKE (362), DEF SEG (297), BLOAD (276), BSAVE (276).

*Seit* QuickBASIC 2.0

*Anwendung*  $x = \text{PEN}(n)$

*Nutzen* Gibt den Status des Lichtstiftes zurück. Die Zahl  $n$  (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 (siehe "Lichtgriffel emulieren" im Anhang F), dann klappt das nicht so ganz.

Bei $n=...$	ist der Funktionswert...
0	-1, wenn der Lichtstift seit dem letzten PEN-Aufruf betätigt wurde, 0, wenn nicht
1	die x-Koordinate, an der der Stift zum letzten Mal gedrückt wurde
2	die y-Koordinate, an der der Stift zum letzten 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 beziehungsweise -spalten zurückgegeben.

*Siehe auch* ON event GOSUB (346), PEN (358)

*Seit* QuickBASIC 2.0

*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.

*Siehe auch* ON event GOSUB (346), EVENT ON/OFF (309), PEN (358).

## PLAY (Funktion)

Standard

*Seit* QuickBASIC 2.0

*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.

*Siehe auch* `PLAY` (359).

## PLAY (für Event-Trapping) (Befehle)

Trapping

*Seit* QuickBASIC 2.0

*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` (346), `PLAY` (Funktion) (359), `PLAY` (für Tonerzeugung) (359).

## PLAY (für Tonerzeugung) (Befehl)

Standard

*Seit* QuickBASIC 2.0

*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 <i>n</i>	Spielt die Note Nr. <i>n</i> ( <i>n</i> im Bereich 1 bis 84, oder 0 für Pause). Die Note wird mit der durch L festgelegten Länge gespielt.
L <i>n</i>	Legt die Länge für Noten fest, denen keine Längenangabe folgt. <i>n</i> ist die Notenlänge (1 für ganze Note bis 64 für Vierundsechzigstelnote).
O <i>n</i>	Wählt die Oktave <i>n</i> ( <i>n</i> 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 <i>n</i>	Setzt das Spieltempo. <i>n</i> 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.
Xadr\$	Führt einen weiteren PLAY-String aus, dessen Adresscode mit VARPTR\$ ermittelt und in adr\$ eingetragen werden muß.
=adr\$	Kann anstelle einer Zahl ( <i>n</i> 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 beziehungsweise = und dem dazugehörigen VARPTR\$ darf sich kein Leerzeichen befinden.

» Bitte lesen Sie zu den Befehlen = und X die Bemerkung und das Beispiel beim DRAW-Befehl, die für PLAY ebenso gelten.

**Siehe auch** PLAY (Funktion) (359), ON *event* GOSUB (346), SOUND (394).

*Seit* QuickBASIC 4.0

*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.

<i>verfahren</i>	<b>Funktion</b>
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 wirklich um absolute Pixelkoordinaten (Punkt 0,0 in der oberen linken Ecke).

Wenn Sie WINDOW nicht benutzt haben, sind Pixel-Koordinaten und logische Koordinaten identisch, diese Funktion ist dann irrelevant.

*Siehe auch* VIEW (414), WINDOW (419).

*Seit* QuickBASIC 2.0

*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:

<i>funktion</i>	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

*Siehe auch* PMAP (361), VIEW (414), WINDOW (419), PSET (366).

POKE (Befehl)	Speicher
---------------	----------

*Seit* QuickBASIC 2.0

*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).

*Bemerkung* » Siehe Bemerkungen zu PEEK.

» In OS/2 darf POKE nur angewendet werden, um auf Speicherstellen zu schreiben, auf die Ihr Programm Schreibzugriff hat. Anderenfalls wird entweder ein *Permission denied*-Fehler oder ein Betriebssystemfehler hervorgerufen.

*Beispiel* Siehe Beispiel zu PEEK.

*Siehe auch* PEEK (357), DEF SEG (297), BLOAD (276), BSAVE (276).

POS (Funktion)	Standard
----------------	----------

*Seit* QuickBASIC 2.0

*Anwendung*  $x = \text{POS}(n)$

*Nutzen* POS gibt die Bildschirmspalte zurück, in der sich der Textcursor befindet. Der Funktionswert liegt also zwischen 1 und 80 beziehungsweise zwischen 1 und 40 bei Bildschirm-Modi mit 40 Zeichen. *n* ist ein Dummy-Argument, das jeden beliebigen Wert haben kann.

*Siehe auch* CSRLIN (291), LOCATE (337).

## PRESET (Befehl)

Grafik

*Seit* QuickBASIC 2.0

*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.

*Siehe auch* PSET (366), POINT (361).

## PRINT (Befehl)

Standard, I/O

*Seit* QuickBASIC 2.0

*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.

Nun zu den möglichen Formatierzeichen, die *format\$* bei Zahlenformatierungen enthalten darf:

Zeichen	Bedeutung
#	Ziffern-Platzhalter. Wird - anders als bei den Routinen aus der Format-Add-On-Library - stets gefüllt. 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, wo 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. Hier bedeuten die Sonderzeichen folgendes:

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 beziehungsweise 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 (ein Carriage Return- und ein Linefeed-Zeichen, `CHR$(13)` und `CHR$(10)`, werden ausgegeben, wodurch auch in einer Datei eine neue Zeile beginnt).

Wenn ein *ausdruck* nicht mehr auf die aktuelle Zeile paßt, wird er nicht in der Mitte durchschnitten, sondern vollständig auf die nächste Zeile umgebrochen. Bei Dateien ist die Zeilenlänge unbeschränkt.

*Bemerkung* » Wenn Sie nur PRINT oder nur PRINT #dateinummer, schreiben, wird eine Leerzeile auf dem Bildschirm beziehungsweise in die Datei ausgegeben.

» Wenn Sie PRINT #dateinummer benutzen, um in eine mit OPEN FOR RANDOM geöffnete Datei zu schreiben, wird - egal, ob FIELD benutzt wurde oder nicht - der Dateipuffer mit der Ausgabe des PRINT-Befehls überschrieben. PRINT alleine schreibt nichts in die Datei; es muß erst ein PUT ausgeführt werden, damit der Puffer wirklich in die Datei geschrieben wird. Es dürfen nicht mehr Bytes geschrieben werden, als die Satzlänge es zuläßt, sonst tritt ein *FIELD buffer overflow* auf.

» Mit einem *format\$* für Zahlen können keine Strings formatiert werden und umgekehrt. Ein angegebener *format\$* wird für alle *ausdrücke* benutzt. Wenn Sie also zuerst eine formatierte Zahl und danach, auf derselben Zeile, einen String ausgeben möchten, müssen Sie zwei PRINT-Befehle dazu benutzen und den ersten mit einem Semikolon enden lassen.

» Wie oben beschrieben, werden Zahlen ohne USING mit angehängtem, positive Zahlen auch mit führendem Leerzeichen ausgegeben. Um das angehängte Leerzeichen zu vermeiden, können Sie statt PRINT zahl schreiben: PRINT STR\$(zahl). Dabei bleibt allerdings immer noch das führende Leerzeichen bei positiven Zahlen erhalten. PRINT MID\$(STR\$(zahl), 2) schneidet das erste Zeichen grundsätzlich ab, auch wenn es - bei negativen Zahlen - ein

Minuszeichen ist. PRINT LTRIM\$(STR\$(zahl)) gibt negative Zahlen mit führendem Minuszeichen aus, entfernt aber sämtliche Leerzeichen.

Beispiel (zur numerischen Formatierung)			
format\$	für -4.5	für 0.6666	für 1150.8
"####.##"	-4.50	0.67	1150.80
"DM ####.##"	DM -4.50	DM 0.67	DM 1150.80
"**####"	***_5	****1	*1151
"\$####.###"	\$ -4.500	\$ 0.667	\$1150.800
"+#.####^ ^^"	-4.5000E+00	+6.6660E-01	+1.1508E+03
"#,###.##"	-4.50	0.67	1,150.80
"##+"	5-	1+	%1151+
"##-"	5-	1	%1151

Siehe auch LPRINT (339), OPEN (348), Formatx\$ (440), LEFT\$ (333).

PSET (Befehl)

Grafik

Seit

QuickBASIC 2.0

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.

Bemerkung

» Genaueres über die verschiedenen Grafikmodi, ihre Farben und Auflösungen finden Sie bei SCREEN.

Siehe auch

PRESET (363), POINT (361), PMAP (361), SCREEN (378).

PUT (für Dateien) (Befehl)

I/O

Seit

QuickBASIC 4.0

Anwendung

PUT [#]dateinummer[, [satznummer], [satzvariable]]

Nutzen

Schreibt Daten in eine Datei, die FOR RANDOM oder FOR BINARY geöffnet wurde. Der Unterschied zwischen beiden Dateiararten 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  $\text{LOC}(\text{dateinummer}) + 1$  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. Sie darf nicht länger sein als die Satzlänge, mit der die Datei geöffnet wurde. Bei BINARY-Dateien wird die angegebene *satzvariable* geschrieben, also  $\text{LEN}(\text{satzlänge})$  Bytes. Wird eine *satzvariable* von numerischem Typ angegeben, hat das denselben Effekt, als würde ein String angegeben, der mittels MKx\$ aus dieser Variable erzeugt wurde.

*Bemerkung* » Arrays sind nicht als *satzvariable* zugelassen.

*Siehe auch* GET (für Dateien) (317), OPEN (348), LOC (336), FIELD (311).

## PUT (für Grafik) (Befehl)

**Grafik**

*Seit* QuickBASIC 2.0

*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, und damit die Position, an die der Bereich kopiert 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, wenn sie angegeben werden, das Array-Element bezeichnen, bei dem die Übertragung in den Bildschirmspeicher beginnen soll.

*modus* ist die Art und Weise, in der der Bereich auf den Bildschirm kopiert wird:

<i>modus</i>	<b>Wirkung</b>
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.

*Bemerkung*   » Wenn auf dem Schirm (beziehungsweise im definierten Viewport, falls VIEW benutzt wurde) nicht genügend Platz vorhanden ist, um das Bild aus dem Array abzubilden, gibt es einen *Illegal function call*-Fehler.

» Es ist möglich, Bilder, die in einem bestimmten Bildschirmmodus mit GET in ein Array gelesen wurden, in einem anderen Modus wieder auszugeben. Das führt jedoch, je nachdem, um welche Modi es sich handelt, zu Verzerrungen und/oder Farbveränderungen.

*Siehe auch*   GET für Grafik (318).

<b>RANDOMIZE (Befehl)</b>	<b>Standard</b>
---------------------------	-----------------

*Seit*           QuickBASIC 2.0

*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.

*Bemerkung*   » Geeignete Möglichkeiten für diese Aufgabe sind zum Beispiel ein Zähler, der zählt, wie oft das Programm aufgerufen wurde, oder eine

Zahl, die aus dem Systemdatum abgeleitet wird, oder - besonders häufig verwendet - einfach die Systemvariable TIMER.

*Siehe auch* RND (374), TIMER (409).

## READ (Funktion)

Standard

*Seit* QuickBASIC 2.0

*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.

Der Versuch, Daten zu lesen, wenn der DATA-Zeiger nirgendwohin zeigt (wenn alle DATA-Konstanten schon gelesen sind), führt zu einem *Out of data*-Fehler.

Das Einlesen von DATA-Konstanten in Variablen mit READ gleicht einer Reihe von einfachen Zuweisungen und unterliegt denselben Beschränkungen. Die Kombination DATA 10: READ a% ist identisch mit a% = 10. Sie dürfen also nicht eine DATA-Konstante 60.000 in eine INTEGER-Variable einlesen (deren Datenbereich geht nur bis 32.767) usw. Eine Ausnahme stellt hier nur das Lesen von numerischen Konstanten in Stringvariablen dar, das keinen Fehler verursacht. Während a\$ = 50 kein gültiger Befehl ist, funktioniert DATA 50: READ a\$ problemlos - nämlich genauso wie a\$ = "50".

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.

*Beispiel* Hier sehen Sie einen Ausschnitt aus einem Programm, in dem mit Hilfe der Prozedur FarbInit ein globales Array namens Farbe mit sinnvollen Farbwerten für verschiedene Bildschirme gefüllt wird. Das Feld *Farbe* enthält in der ersten Dimension die vier verwendeten "Farben" Normal, Invers, Hervorgehoben und Unterstrichen; in der zweiten Dimension wird zwischen Vorder- und Hintergrundfarbe unterschieden.

```

DIM SHARED Farbe(1 TO 4, 1 TO 2) AS INTEGER

MonoData:
  DATA 7,0,0,7,1,0,23,0
BWData:
  DATA 7,0,0,7,15,0,23,0
LCDDData:
  DATA 7,0,0,7,0,7,23,0
ColorData:
  DATA 7,1,1,7,14,7,23,0

SUB FarbInit(Modus AS STRING)

  SELECT CASE UCASE$(Modus)
  CASE "BW":      RESTORE BWData
  CASE "LCD":     RESTORE LCDDData
  CASE "COLOR":   RESTORE ColorData
  CASE ELSE:      RESTORE MonoData
  END SELECT

  FOR i% = 1 TO 4
    FOR j% = 1 TO 2
      READ Farbe(i%, j%)
    NEXT
  NEXT

END SUB

```

*Siehe auch* DATA (292), RESTORE (372).

## REDIM (Befehl)

**Standard**

*Seit* BASIC 7.1 PDS

*Anwendung* REDIM [PRESERVE] [SHARED] *variable(array-bereich)*  
 [AS *typ*] [, *variable(array-bereich)*  
 [AS *typ*]]...

*Nutzen* REDIM redimensioniert 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.

Es hat keinen Sinn, ein Array, das mit DIM SHARED deklariert wurde, mit REDIM ohne SHARED neu zu deklarieren, weil das Array in jedem Fall eine globale Variable bleibt.

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.

*Beispiel* (zu den verschiedenen REDIM-Fällen):

```
REM $DYNAMIC

DIM a(199, 20) AS INTEGER
' Ohne REM $DYNAMIC wäre das ein statisches
' Array, und REDIM könnte nicht benutzt werden!
' ...
' Dieser Befehl ist erlaubt, denn ich ändere
' nicht die Anzahl der Dimensionen, sondern nur
' die höchste Dimension:
REDIM PRESERVE a(199, 25) AS INTEGER

' Ich kann andere Dimensionen als die höchste
' nur verändern, wenn ich auf PRESERVE ver-
' zichte und so das Array löschen lasse:
REDIM a(299, 20) AS INTEGER

' Die folgenden Befehle sind ungültig:
REDIM a(299, 20) AS LONG
' (weil ich den Typ geändert habe)

REDIM PRESERVE a(299, 20) AS INTEGER
' (weil ich mit PRESERVE versucht habe, eine
' andere als die höchste Dimension zu ändern)

REDIM PRESERVE a(199, 10 TO 30) AS INTEGER
' (weil ich die untere Grenze der höchsten
' Dimension mit PRESERVE ändern wollte)
```

*Siehe auch* DIM (299), ERASE (307).

## REM (Befehl)

**Standard**

*Seit* QuickBASIC 2.0

*Anwendung* REM *bemerkung*  
' *bemerkung*

*Nutzen* Leitet eine Bemerkung ein. *bemerkung* ist ein beliebiger Text, der vom Compiler nicht beachtet wird. Alle Zeichen von REM oder dem Apostroph bis zum Ende der Zeile werden der Bemerkung zugerechnet. Wenn der REM-Befehl am Ende einer Programmzeile steht, muß er, wie jeder andere Befehl, mit einem Doppelpunkt ab-

gegrenzt werden. Bei der Apostroph-Form ist das nicht nötig; allerdings darf diese Form nicht nach DATA-Zeilen benutzt werden.

*Bemerkung* » REM wird auch zur Einleitung der Metabefehle \$INCLUDE, \$DYNAMIC und \$STATIC benutzt.

» REM-Befehle schlagen sich in keiner Weise im erzeugten OBJ- oder EXE-Code nieder.

*Siehe auch* \$DYNAMIC (304), \$INCLUDE (322), \$STATIC (397).

## RESET (Befehl)

I/O

*Seit* QuickBASIC 4.0

*Anwendung* RESET

*Nutzen* RESET schließt alle offenen Dateien. Die Wirkung ist dieselbe wie die von CLOSE ohne Parameter.

*Siehe auch* CLOSE (285).

## RESTORE (Befehl)

Standard

*Seit* QuickBASIC 2.0

*Anwendung* RESTORE [*zeilennummer*/*-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-Konstante im Modul.

*Beispiel* Siehe Beispiel zu READ.

*Siehe auch* DATA (292), READ (369).

## RESUME (Befehl)

Trapping

*Seit* QuickBASIC 2.0

*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, denn die Ausführung von `RESUME`, ohne daß zuvor ein Fehler auftrat, ruft selbst einen Fehler hervor (*Resume without error*).

- Bemerkung* » Jede Form von `RESUME` außer `RESUME zeilennummer/-label` erfordert, daß das Programm mit `/X` kompiliert wird.
- » `RESUME` und `RESUME NEXT` können, wenn sie in einem lokalen Error-Handler vorkommen, die Programmausführung nur beim nächsten Befehl in der eigenen Prozedur fortsetzen. Das gleiche gilt für globale Error-Handler fremder Module. Näheres dazu finden Sie im Kapitel 15.

*Siehe auch* `ON ERROR` (345), `ERR`, `ERL` (308).

## RETURN (Befehl)

## Struktur, Trapping

*Seit* QuickBASIC 2.0

*Anwendung* `RETURN [zeilennummer/-label]`

*Nutzen* `RETURN` alleine kehrt hinter den zuletzt ausgeführten `GOSUB`-Befehl beziehungsweise 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.

RETURN mit Zeilenangabe ist zwar auch in Event-Handling-Routinen erlaubt, allerdings haben Sie dann keine Möglichkeit, herauszufinden, von wo die Routine aufgerufen wurde, und so auch keine Chance, das Programm an dieser Stelle wieder fortzusetzen.

*Siehe auch* GOSUB (319), SUB/FUNCTION (402), ON *event* GOSUB (346), ON...GOSUB (348).

## RIGHT\$ (Funktion)

Standard

*Seit* QuickBASIC 2.0

*Anwendung*  $x\$ = \text{RIGHT\$}(y\$, \text{zeichen})$

*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; es wird dann weder ein Fehler generiert, noch füllt RIGHT\$ den String mit Leerzeichen auf.

*Siehe auch* LEFT\$ (333), MID\$ (Befehl) (342).

## RMDIR (Befehl)

DOS

*Seit* QuickBASIC 2.0

*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.

*Bemerkung* » RMDIR ist in der Funktion identisch mit dem gleichnamigen DOS-Befehl.

*Siehe auch* CHDIR (282), CURDIR\$ (291), MKDIR (343).

## RND (Funktion)

Standard

*Seit* QuickBASIC 2.0

*Anwendung*  $x! = \text{RND}[(\text{argument})]$

*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.

Siehe auch RANDOMIZE (368).

## RSET (Befehl)

Standard

Seit QuickBASIC 4.0

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 » RSET wird benutzt, um Daten in einen mit FIELD definierten Puffer zu schreiben, weil dabei nicht die Gefahr besteht, zu viele Daten in den Puffer zu schreiben.

» REST kann nicht, wie LSET, benutzt werden, um zwei verschiedene Variablen selbstdefinierten Typs einander zuzuordnen.

Siehe auch LSET (340).

## RTRIM\$ (Funktion)

Standard

Seit QuickBASIC 4.0

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. Siehe dazu "Strings mit fester und variabler Länge" in Kapitel 12.2.

» ISAM führt beim Speichern von Strings automatisch ein RTRIM\$ aus, so daß nie Leerzeichen am rechten Rand gespeichert werden.

Siehe auch LTRIM\$ (342)

*Seit* QuickBASIC 2.0

*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. RUN ist der einzige Befehl, bei dem nicht statt Zeilennummern auch Zeilenlabels verwendet werden können. 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, weil das aufgerufene Programm das aufrufende im Speicher überschreibt. Dadurch steht dem aufgerufenen Programm allerdings unter Umständen signifikant mehr Speicher zur Verfügung als beim SHELL-Befehl.

*Bemerkung* » In allen Versionen außer PDS 7.0 ist es möglich, mit dem RUN-Befehl auch Programme aufzurufen, die nicht die Extension COM oder EXE haben (es müssen zwar COM- oder EXE-Programme sein, aber man kann sie umbenennen). PDS 7.0 erlaubt aufgrund eines Fehlers nur COM und EXE; PDS 7.1 ist hier aber wieder tolerant.

» Leider gibt es keine Möglichkeit, dem aufgerufenen Programm für die Befehlszeile Informationen zu übergeben, d.h. COMMAND\$ wird im aufgerufenen Programm nie Zeichen enthalten können.

» Wenn Sie in der zweiten Syntax keine Extension im Dateinamen angeben, wird für kompilierte Programme automatisch EXE angenommen, während in der Entwicklungsumgebung QBX versucht würde, eine Datei mit der Extension BAS zu laden.

» 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.

*Siehe auch* CHAIN (281), SHELL (392).

*Seit* BASIC 6.0

*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.

*Bemerkung* » Sie werden die Funktion SADD häufig benötigen, wenn Sie Interrupts aufrufen oder Daten an Routinen übergeben wollen, die in anderen Sprachen geschrieben sind.

» Die Adresse eines Strings kann von BASIC laufend verändert werden. Ermitteln Sie die Adresse deshalb stets unmittelbar, bevor Sie sie verwenden.

» Für andere Variablen als Strings kann die Adresse mit VARPTR ermittelt werden.

*Beispiel* SADD wird in fast allen Beispielen zu Interruptaufrufen verwendet; außerdem im Programmbeispiel zu PEEK.

*Siehe auch* SSEG (395), SSEGADD (396), DEF SEG (297), PEEK (357), POKE (362).

## **SCREEN (Funktion)**

**Standard**

*Seit* QuickBASIC 2.0

*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 (zum Beispiel TRUE, also -1), wird stattdessen der Farbcode des angegebenen Zeichens ermittelt. Mit "Farbcode" ist hier eine Kombination aus Vorder- und Hintergrundfarbe eines Zeichens im Textmodus gemeint, die im Abschnitt über AttrBox im Referenzteil zur User Interface/General-Toolbox genauer beschrieben wird.

Im Grafikmodus gibt die SCREEN-Funktion keine Farbcodes zurück, und als ASCII-Code wird 32 zurückgegeben, wenn an der angegebenen Position nicht ganz exakt ein Buchstabe steht (es reicht schon, daß sich ein Pixel durch PSET o.ä. geändert hat).

*Bemerkung* » Um größere Mengen von Zeichen vom Bildschirm abzulesen, bedienen Sie sich besser der Funktion GetBackground aus der General-Toolbox.

*Siehe auch* GetBackground (514).

## SCREEN (Befehl)

**Grafik**

*Seit* QuickBASIC 4.0

*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) beziehungsweise 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.ä. 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 unterstützten 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	<i>mit CGA-Bildschirm:</i> wie CGA, jedoch auch im 80x25-Modus acht Bildschirmseiten, wenn die Karte mit mehr als 64 KB Bildschirmspeicher ausgestattet ist (das ist üblich).
EGA	<i>mit Monochrom-Bildschirm:</i> 80x25 Zeichen (vier oder vier 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	<i>mit EGA- oder Multisync-Bildschirm:</i> 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 Kunden 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 Farbattributen, 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 Farbattributen, denen mit PALETTE folgende 9 "Farben" zugeordnet werden können: 0 = schwarz, 1 = blinkend schwarz/weiß, 2 = blinkend schwarz/leuchtend weiß, 3 = blinkend 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 Farbattributen, 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 Farbattributen, denen mit PALETTE 262.144 Farben zugeordnet werden können.
-----	---

## Modus 13 - Grafik 320x200 Punkte

MCGA	40x25 Zeichen; eine Bildschirmseite; 256 Farbattributen, 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	hellviolett	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* » Anmerkungen wie "vier Bildschirmseiten" oder "16 Farbattribute" bedeuten stets, daß die Bildschirmseiten von 0 bis 3 beziehungsweise die Farbattribute von 0 bis 15 benutzt werden können, wo es nicht anders vermerkt ist.

Eine Ausnahme machen die VGA- und MCGA-Karten dort, wo 262.144 Farben möglich sind. Der Wertebereich dieser Farben reicht von 0 bis 4.144.959, aber nur 262.144 von diesen möglichen Werten sind gültig. Ein gültiger Farbwert errechnet sich aus der Formel  $\text{farbwert} = 65.536 * \text{blauwert} + 256 * \text{grünwert} + \text{rotwert}$ , 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:  $\text{farbwert} = (\text{blauwert} \text{ AND } 1) + 2 * ((\text{grünwert} \text{ AND } 1) + 2 * ((\text{rotwert} \text{ AND } 1) + (\text{blauwert} \text{ AND } 2)) + 2 * ((\text{grünwert} \text{ AND } 2) + 2 * (\text{rotwert} \text{ AND } 2)))$ . Anders

ausgedrückt: Bit 1 und 4 des Farbwertes repräsentieren den *blauwert*, Bit 2 und 5 den *grünwert* und Bit 3 und 6 den *rotwert*. Das höhere der beiden Bits wird jeweils von dem Kontrastregler am Bildschirm beeinflusst, so daß ein *blauwert* von 3 identisch mit einem *blauwert* von 1 und ein *blauwert* von 2 identisch mit einem *blauwert* von 0 ist, wenn der Kontrastregler auf kleinsten Kontrast gestellt ist. Farben, die nur die Farbwerte 0 und 1 beinhalten, bei denen also das höhere Bit immer 0 ist, werden durch Betätigen des Kontrastreglers nicht beeinflusst.

» 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  $\text{graustufe} = 0,11 \cdot \text{blauwert} + 0,59 \cdot \text{grünwert} + 0,3 \cdot \text{rotwert}$ . 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 256 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 hier abgedruckte Funktion 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 (ein Monochrombildschirm, der Farbwerte in Graustufen umrechnet, wird hier allerdings als Farbbildschirm diagnostiziert).

```

FUNCTION Grafikkarte$

  DIM MaxSeite AS INTEGER, Display AS STRING

  ON LOCAL ERROR RESUME NEXT
  SCREEN 12
  IF ERR = 0 THEN
    GOSUB Seiten
    GOSUB Bildschirm
    Grafikkarte$ = "VGA mit" + STR$(MaxSeite * 64) + "
                  " KByte Bildschirmspeicher, " + Display
  ELSE
    ERR = 0: SCREEN 11
  
```

(Fortsetzung nächste Seite)

*(Fortsetzung)*

```
IF ERR = 0 THEN
    Grafikkarte$ = "MCGA"

ELSE

    ERR = 0: SCREEN 7
    IF ERR = 0 THEN
        GOSUB Seiten
        GOSUB Bildschirm
        Grafikkarte = "EGA mit" + STR$(MaxSeite * 64) +
            " KByte Bildschirmspeicher, " + Display

    ELSE

        ERR = 0: SCREEN 4
        IF ERR = 0 THEN
            Grafikkarte = "Olivetti oder AT&T"

        ELSE

            ERR = 0: SCREEN 3
            IF ERR = 0 THEN
                SCREEN 3, , 1, 1
                IF ERR = 0 THEN
                    Grafikkarte = "Hercules, 2 Bildschirmseiten"
                ELSE
                    Grafikkarte = "Hercules, 1 Bildschirmseite"
                END IF
            ELSE

                ERR = 0: SCREEN 2
                IF ERR = 0 THEN
                    Grafikkarte = "CGA"
                ELSE
                    Grafikkarte = "MDPA oder bei Hercules " +
                        "vergessen, MSHERC.COM zu laden"
                END IF
            END IF
        END IF
    END IF
END IF

SCREEN 0

EXIT FUNCTION
```

*(Fortsetzung nächste Seite)*

(Fortsetzung)

Seiten:

```
MaxSeite = 0: ERR = 0
FOR a% = 0 TO 7
    SCREEN 8, , a%, a%
    IF ERR = 0 THEN MaxSeite = MaxSeite + 1 ELSE EXIT FOR
NEXT
RETURN
```

Bildschirm:

```
ERR = 0: SCREEN 10
IF ERR THEN
    Display = "Monochrombildschirm"
ELSE
    Display = "Farbbildschirm"
END IF
RETURN
```

END FUNCTION

Siehe auch PALETTE (385), COLOR (286).

## SEEK (Funktion)

I/O

Seit QuickBASIC 4.0

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, nur 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, während LOC hier eine altertümliche 128-Byte-Block-Aufteilung vornimmt.

SEEK kann nicht auf COMx:, LPTx:, SCRn:, CONS:, KYBD: oder auf ISAM-Dateien angewandt werden (in diesen Fällen wird 0 als Funktionswert zurückgegeben).

Beispiel Siehe Beispiel zum SEEK-Befehl.

Siehe auch LOC (336), LOF (338), SEEK (Befehl) (386).

*Seit* QuickBASIC 4.0

*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, so daß die Bezeichnung "sequentielle Datei" eigentlich unzutreffend wird.

*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.

*Beispiel* Dieses Programm zweckentfremdet eine sequentielle Datei, die eigentlich nur von der ersten bis zur letzten Zeile kontinuierlich gelesen werden kann. Hier wird es möglich, bereits gelesene Zeilen nochmals einzulesen, indem zu jeder Zeile die Byte-Position in einem Array gemerkt wird. Allerdings sind maximal MaxZeilen Zeilen erlaubt.

```
CONST MaxZeilen = 2000

DIM ZeilenAnfang(1 TO MaxZeilen) AS LONG
DIM ZeilenNummer AS INTEGER, Zeile AS STRING
DIM Datei AS STRING

ZeilenNummer = 1
ZeilenAnfang(1) = 1

LINE INPUT "Welche Datei? "; Datei

OPEN Datei FOR INPUT AS #1

DO UNTIL EOF(1)
    LINE INPUT #1, Zeile
    PRINT Zeile
    PRINT "      (v)orwärts (r)ückwärts (e)nde"
    DO: a$ = UCASE$(INKEY$)
    LOOP UNTIL a$ = "V" OR a$ = "R" OR a$ = "E"
    IF a$ = "R" THEN
        ' an die Position, an der die letzte
        ' Zeile stand, zurückspringen:
        ZeilenNummer = ZeilenNummer - 1
    
```

(Fortsetzung nächste Seite)

(Fortsetzung)

```
IF ZeilenNummer < 1 THEN ZeilenNummer = 1
SEEK #1, ZeilenAnfang(ZeilenNummer)
ELSEIF a$ = "V" THEN
' Position der Zeile, die gleich gelesen
' wird, merken:
ZeilenNummer = ZeilenNummer + 1
ZeilenAnfang(ZeilenNummer) = SEEK(1)
ELSE
CLOSE : SYSTEM
END IF
LOOP
```

*Siehe auch* SEEK (Funktion) (385), GET (für Dateien) (317), PUT (für Dateien) (366).

## SELECT CASE (Befehl)

**Struktur**

*Seit* QuickBASIC 3.0

*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.

*prüfausdruck* wird in den meisten Fällen der Name einer Variable sein; hier kann aber auch ein Funktionsaufruf oder eben ein Ausdruck wie Anzahl\*2 stehen.

*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):

<i>ausdruck</i>	<b>IF <i>prüfausdruck</i> = <i>ausdruck</i></b>
<i>ausdruck1</i> TO <i>ausdruck2</i>	IF <i>ausdruck1</i> <= <i>prüfausdruck</i> AND <i>prüfausdruck</i> <= <i>ausdruck2</i>
IS <i>operator</i> <i>ausdruck</i>	IF <i>prüfausdruck</i> <i>operator</i> <i>ausdruck</i>

In der letztgenannten Form mit IS sind als *operator* zulässig: =, <, <=, >, >= und <>.

Das Beispiel wird die eventuell nun bestehende Verwirrung klären.

*Beispiel*

Diese Beispiele zeigen verschiedene SELECT CASE-Konstruktionen. Statt Befehlen ist in den REM-Zeilen angegeben, welchem IF...THEN-Befehl die jeweilige Formulierung entspricht.

```
SELECT CASE Laenge%
CASE IS < 1, IS > 10
    REM IF Laenge% < 1 OR Laenge% > 10 THEN ...
CASE 1 TO 4
    REM ELSEIF Laenge%>=1 AND Laenge%<=4 THEN...
CASE 5 TO 9
    REM ELSEIF Laenge% >= 5 AND Laenge% <= 9 THEN ...
CASE 10
    REM ELSEIF Laenge% = 10 THEN ...
END SELECT
```

```
SELECT CASE UCASE$(a$)
CASE "A"
    REM IF UCASE$(a$) = "A" THEN ...

CASE "T", "V" TO "Z"
    REM IF INSTR(UCASE$(a$), "TVWXYZ") THEN ...
    REM oder:
    REM IF UCASE$(a$)="T" OR (UCASE$(a$) >= "V"
    REM    AND UCASE$(a$) <= "Z"
```

(Fortsetzung nächste Seite)



(Fortsetzung)

```
CASE IS > UCASE$(b$)
    REM ELSEIF UCASE$(a$) > UCASE$(b$) THEN ...

CASE ELSE
    REM ELSE...
END SELECT
```

Siehe auch IF...THEN...ELSE (320).

## SETMEM (Funktion)

Speicher

Seit BASIC 7.0 PDS

Anwendung  $x\% = \text{SETMEM}(\text{farspeicheraenderung})$

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 DGROUPE 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 `malloc` 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.

Wenn Sie für *farspeicheraenderung* eine Zahl angeben, die so groß oder so klein ist, daß bei Ihrer Addition zum gerade benutzten Far-Speicher ein ungültiger Wert entstünde, wird automatisch die größte beziehungsweise kleinste mögliche Zahl gewählt, die noch zu einem erlaubten Ergebnis führt. Der Befehl `PRINT SETMEM(-655360)` würde den Far-Speicher also auf das absolute Minimum reduzieren, während `PRINT SETMEM(655360)` ihn auf das Maximum setzen würde, so, wie die Einstellung beim Programmstart und nach jedem RUN- oder CLEAR-Befehl wieder gewählt wird.

Bemerkung » `PRINT SETMEM(0)` ergibt die Menge an Far-Speicher, die augenblicklich belegt ist, da die Zahl 0 logischerweise keine Änderungen an der belegten Menge vornimmt.

» In OS/2 ist die Funktion bedeutungslos.

*Beispiel* Dieses Programm spielt ein wenig mit dem Speicher herum und benutzt auch die Funktion FRE(-1), die die Menge an freiem Far-Speicher zurückgibt, also dem Teil, den BASIC zwar für sich belegt hat, in dem es aber noch keine Daten untergebracht hat:

```
PRINT "Programmstart. Das Programm belegt"
PRINT "jetzt"; SETMEM(0); "Bytes Far-Speicher;"
PRINT "davon sind aber"; FRE(-1); "Bytes frei."
PRINT "Die Hälfte des noch freien Far-Spei-"
PRINT "chers reserviere ich jetzt für andere"
PRINT "Zwecke. Sie stehen BASIC dann nicht"
PRINT "mehr zur Verfügung."
PRINT "Jetzt sind nur noch";
PRINT SETMEM(-FRE(-1)\2); "Bytes"
PRINT "belegt; davon sind"; FRE(-1); "Bytes";
PRINT "noch frei."
PRINT "Ich kann auch allen Speicher wieder"
PRINT "für BASIC okkupieren:"
PRINT "So. jetzt sind wieder"; SETMEM(655350);
PRINT "Bytes frei."
```

*Siehe auch* FRE (316), STACK (396), CLEAR (284).

## SGN (Funktion)

**Standard**

*Seit* QuickBASIC 2.0

*Anwendung*  $x = \text{SGN}(y)$

*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.

## SHARED (Befehl)

**Standard**

*Seit* QuickBASIC 4.0

*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 nicht im Hauptprogramm definiert sein. Es wäre auch denkbar, den SHARED-Befehl nur in einigen Prozeduren zu benutzen, damit diese gemeinsam einige Variablen benutzen können, ohne daß diese im Hauptprogramm vereinbart oder benutzt werden.

*Beispiel* Siehe Beispiel zu STATIC.

*Siehe auch* DIM (299), COMMON (288), STATIC (397).

## **SHELL (Funktion)**

**OS/2**

*Seit* BASIC 7.0 PDS

*Anwendung* `x = SHELL(programmaufruf$)`

*Nutzen* Ruft unter OS/2 ein anderes Programm auf. *programmaufruf\$* enthält den Namen des Programms, das aufgerufen werden soll, und eventuell auch Parameter (eine Befehlszeile), die an dieses Programm übergeben werden sollen.

Als Funktionswert wird die Identifikationsnummer des hiermit aufgerufenen Prozesses zurückgegeben. Diese Identifikationsnummer wird für einige OS/2-Betriebssystemfunktionen, die den aufgerufenen Prozeß betreffen, benötigt.

Der Unterschied zwischen der Funktion SHELL und dem gleichnamigen Befehl ist, daß die Funktion nur unter OS/2 benutzt werden kann, während der Befehl auch für DOS anwendbar ist. Wenn in OS/2 mit der Funktion SHELL ein Programm aufgerufen wird, laufen von da an das aufgerufene und das aufrufende Programm parallel (Multitasking), während die Verwendung des Befehls SHELL die Ausführung des aufrufenden Programms so lange anhält, bis das aufgerufene Programm beendet ist.

*Siehe auch* SHELL (Befehl) (392).

*Seit* QuickBASIC 2.0

*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 bei DOS, CMD.EXE bei OS/2) in den Speicher und übergibt diesem mit der /C-Option den *befehl*\$. Dadurch wird erreicht, daß jeder Befehl oder Programmaufruf, den man unter DOS beziehungsweise OS/2 direkt eingeben könnte, auch mit SHELL aufgerufen werden kann. Einziges Handicap beim Ausführen von Programmen als "Tochterprozeß" ist, daß den so ausgeführten Programmen relativ wenig Speicher zur Verfügung steht, weil das BASIC-Programm selbst im Speicher ist.

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.

» Der Unterschied zur Funktion SHELL, die nur unter OS/2 eingesetzt werden kann, ist, daß bei der Funktion das BASIC-Programm nicht bis zum Ende des aufgerufenen Programms angehalten wird.

*Beispiel* Der Befehl

```
SHELL "CHKDSK /F < chkinp > chkoutp"
```

wird unter DOS ausgeführt als

```
COMMAND /C CHKDSK /F < chkinp > chkoutp
```

woraufhin der neue Befehlsprozessor den Befehl

```
CHKDSK /F
```

ausführt, die Eingaben für CHKDSK aus chkinp liest und die Ausgaben von CHKDSK in chkoutp schreibt.

*Siehe auch* SHELL (Funktion) (391).

## SIGNAL (Befehl)

OS/2, Trapping

<i>Seit</i>	BASIC 6.0
<i>Anwendung</i>	SIGNAL ( <i>n</i> ) ON SIGNAL ( <i>n</i> ) OFF SIGNAL ( <i>n</i> ) STOP
<i>Nutzen</i>	Aktiviert, deaktiviert oder unterbricht das Event-Trapping für OS/2-Betriebssystemsignale. Solange das Trapping nur mit SIGNAL( <i>n</i> ) STOP suspendiert ist, werden Signale zwar erkannt, aber die Trapping-Routine wird erst nach dem nächsten SIGNAL( <i>n</i> ) ON wieder aufgerufen.
<i>Bemerkung</i>	» Die Liste der erlaubten Signalnummern <i>n</i> finden Sie bei ON event GOSUB.
<i>Siehe auch</i>	ON event GOSUB (346).

## SIN (Funktion)

Standard

<i>Seit</i>	QuickBASIC 2.0
<i>Anwendung</i>	$x = \text{SIN}(y)$
<i>Nutzen</i>	Gibt den Sinus des Arguments <i>y</i> zurück. Das Ergebnis wird mit doppelter Genauigkeit berechnet, wenn es sich bei <i>y</i> um eine Zahl vom Typ LONG, DOUBLE oder CURRENCY handelt.  <i>y</i> wird als Winkel im Bogenmaß erwartet. Wenn ein Winkel in Grad vorliegt, muß dieser zunächst durch 180/π (57,2957795130824) geteilt werden.
<i>Bemerkung</i>	» Die SIN-Funktion benutzt die Mathematik-Libraries. Ihr unbedachter Einsatz kann deshalb das EXE-File unnötig um etwa 10 KB verlängern (siehe "Programmgröße und RAM-Speicherplatz" in Kapitel 19.2).
<i>Siehe auch</i>	COS (290), TAN (407), ATN (275).

## SLEEP (Befehl)

Standard

<i>Seit</i>	BASIC 6.0
<i>Anwendung</i>	SLEEP <i>sekunden</i>
<i>Nutzen</i>	Hält das Programm für <i>sekunden</i> Sekunden an. Die Pause wird sofort beendet, wenn eine Taste gedrückt wird oder ein Ereignis eintritt, für das eine Event-Trapping-Routine aktiv ist. <i>sekunden</i> muß eine Ganzzahl sein.
<i>Bemerkung</i>	» SLEEP beeinflusst den Tastaturpuffer nicht. Wenn Zeichen im Tastaturpuffer stehen, wartet SLEEP trotzdem, bis die Zeit abgelaufen ist.

fen oder eine Taste gedrückt worden ist. Die eventuell gedrückte Taste wird in jedem Fall in den Tastaturpuffer aufgenommen.

*Siehe auch* ON event GOSUB (346).

## SOUND (Befehl)

Standard

*Seit* QuickBASIC 2.0

*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 (*dauer* = 182 entspräche also 10 Sekunden).

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.

*Siehe auch* PLAY (359), BEEP (275).

## SPACE\$ (Funktion)

Standard

*Seit* QuickBASIC 2.0

*Anwendung* x\$ = SPACE\$(*anzahl*)

*Nutzen* Erzeugt einen String mit *anzahl* Leerzeichen. *anzahl* darf zwischen 1 und 32.767 liegen.

*Bemerkung* » SPACE\$(*anzahl*) ist identisch mit STRING\$(*anzahl*, 32).

*Siehe auch* SPC (394), STRING\$ (402).

## SPC (Funktion)

Standard

*Seit* QuickBASIC 2.0

*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.

Die SPC-Funktion sorgt dafür, daß an der Cursorposition *anzahl* Leerzeichen eingefügt werden (0  $\frac{3}{4}$  *anzahl*  $\frac{3}{4}$  32.767). Sie ist fast

identisch mit SPACE\$. Im Unterschied zu SPACE\$ ist SPC jedoch, wie gesagt, in der Verwendung eingeschränkt. Außerdem erzeugt SPC nie mehr Leerzeichen, als in eine Zeile passen; wenn *anzahl* größer als die Zeilenbreite ist, werden nur  $\text{anzahl} \bmod \text{zeilenbreite}$  Leerzeichen ausgegeben. Der dritte Unterschied zu SPACE\$ ist, daß SPC die Leerzeichen, die es ausgibt, in der Mitte umbrechen kann, wenn ein Zeilenende erreicht wird, während alle Strings, darunter auch SPACE\$, von PRINT gleich in die nächste Zeile gesetzt werden.

*Siehe auch* SPACE\$ (394), TAB (406).

<b>SQR (Funktion)</b>	<b>Standard</b>
-----------------------	-----------------

*Seit* QuickBASIC 2.0

*Anwendung*  $x = \text{SQR}(y)$

*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 (338).

<b>SSEG (Funktion)</b>	<b>Speicher</b>
------------------------	-----------------

*Seit* BASIC 7.0 PDS

*Anwendung*  $x\% = \text{SSEG}(\text{variable}\$)$

*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.

*Bemerkung* » Sie werden die Funktion SSEG häufig benötigen, wenn Sie Interrupts aufrufen oder Daten an Routinen übergeben wollen, die in anderen Sprachen geschrieben sind.

» Die Adresse eines Strings kann von BASIC laufend verändert werden. Ermitteln Sie die Adresse deshalb stets unmittelbar, bevor Sie sie verwenden.

» Für andere Variablen als Strings kann das Segment mit VARSEG ermittelt werden.

*Beispiel* SSEG wird in fast allen Beispielen zu Interrupt-Aufrufen verwendet; außerdem im Programmbeispiel zu PEEK.

*Siehe auch* SADD (377), SSEGADD (396), DEF SEG (297), PEEK (357), POKE (362).

## SSEGADD (Funktion)

Speicher

*Seit* BASIC 7.0 PDS

*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.

» Die Adresse eines Strings kann von BASIC laufend verändert werden. Ermitteln Sie die Adresse deshalb stets unmittelbar bevor Sie sie verwenden.

*Siehe auch* SADD (377), SSEG (395), VARPTR (412), VARSEG (413).

## STACK (Funktion und Befehl)

Speicher

*Seit* BASIC 7.0 PDS

*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.

» Der Befehl CLEAR kann ebenfalls benutzt werden, um die Stack-Größe zu verändern.

» Die Standardgröße für den Stack ist 3 KB unter DOS und 3,5 KB unter OS/2; die minimale Größe sind 325 Bytes unter DOS und 825 Bytes unter OS/2.

» Mit der Kombination STACK STACK aus Befehl und Funktion wird der Stack auf die maximale Größe gesetzt.

*Siehe auch* CLEAR (284).



## \$STATIC (Metabefehl)

Standard

*Seit* QuickBASIC 2.0

*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.

*Bemerkung* » Mehr über dynamische und statische Arrays finden Sie im Kapitel 12.4.

*Siehe auch* \$DYNAMIC (304), DIM (299).

## STATIC (Befehl)

Standard

*Seit* QuickBASIC 4.0

*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 (so, als wäre die Prozedur mit STATIC vereinbart, siehe SUB/FUNCTION).

Außerdem setzt der STATIC-Befehl eventuell gültige globale Variablen oder Konstanten außer Kraft (siehe Beispiel).

Als dritte Funktion kann der STATIC-Befehl auch innerhalb einer DEF FN-Definition benutzt werden, die sich ja immer im Modulcode befindet, um Variablen lokal zur DEF FN-Funktionsdefinition zu machen. Ohne STATIC sind alle Variablen, die in der DEF FN-Funktion benutzt wurden, auch im ganzen anderen Modulcode benutzbar.

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.

### Beispiel

Dieses Beispiel macht den Versuch, mit allen Unklarheiten betreffs globaler und lokaler, automatischer und statischer Variablen aufzuräumen.

```
' Zwei Variablen lokal für den Modulcode:
DIM Haupt1 AS STRING
DIM Haupt2 AS STRING
' Zwei Variablen global:
' (DIM und COMMON sind hier äquivalent)
DIM SHARED Haupt3 AS STRING
COMMON SHARED Haupt4 AS STRING

CALL Prozedur
CALL Prozedur
END

SUB Prozedur

    STATIC Haupt3 AS INTEGER
    SHARED Haupt2 AS STRING
    DIM Sub1 AS INTEGER

    ' durch diese drei Befehle wurde
    ' erreicht, daß die Prozedur
    ' 1. die globale Variable Haupt3 nicht
    '    benutzt, sondern eine eigene, lokale
    '    gleichen Namens besitzt, die ihren
    '    Wert zwischen zwei Aufrufen behält
    '    und von der globalen Variablen un-
    '    abhängig ist
    ' 2. die eigentlich lokal für den Modulcode
    '    vereinbarte Variable Haupt2 trotzdem
    '    benutzt, als wäre sie global
    ' 3. eine eigene lokale Variable namens Sub1
    '    erhält, die sich von Haupt3 dadurch
    '    unterscheidet, daß sie ihren Wert
    '    nicht zwischen zwei Aufrufen behält
    '    (sie ist eine automatische Variable)

    WRITE Haupt2, Haupt3, Haupt4, Sub1
    IF Haupt2 = "" THEN Haupt2 = "1"
    IF Haupt3 = 0 THEN Haupt3 = 1
    IF Haupt4 = "" THEN Haupt4 = "1"
    IF Sub1 = 0 THEN Sub1 = 0
```

(Fortsetzung nächste Seite)

(Fortsetzung)

```
' beim ersten Aufruf wird "",0,"",0
' ausgegeben. Beim zweiten Aufruf lautet
' die Ausgabe: "1",1,"1",0
' Haupt4 behält ihren Wert, weil sie global
' ist; Haupt2 ist nicht global, wurde aber
' als SHARED vereinbart und behält deshalb
' auch ihren Wert; Haupt3 ist zwar lokal zur
' Prozedur, aber STATIC, und behält deshalb
' ihren Wert, und nur Sub1 behält ihren Wert
' nicht, weil sie eine gewöhnliche lokale
' Variable einer automatischen Prozedur ist.
```

END SUB

Siehe auch DIM (299), COMMON (288), SHARED (390).

## STICK (Funktion)

Standard

Seit QuickBASIC 2.0

Anwendung  $x = \text{STICK}(\text{argument})$

Nutzen Erfragt den Status eines Steuerknüppels (Joystick). Der zurückgegebene Funktionswert richtet sich nach dem *argument*:

<i>argument</i>	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.

» Falls Ihnen bisher nur die Joysticks von Home- und Spielcomputern bekannt waren: Diese Computer verwenden *digitale* Joysticks, das heißt, die Joysticks können nur die Signale "auf", "ab", "links", "rechts" und "Feuer" an den Computer senden. Joysticks für IBM-Rechner sind im Gegensatz dazu *analoge* Geräte, die die exakte Stellung des Steuerknüppels melden - deshalb die Koordinatenangaben.

*Siehe auch* STRIG (Funktion) (401), ON event GOSUB (346).

## STOP (Befehl)

**Standard**

*Seit* BASIC 7.0 PDS

*Anwendung* STOP [*exitcode*]

*Nutzen* STOP hat keinen Nutzen. In der QBX-Umgebung wirkt es wie ein Breakpoint (siehe "Terminologie" in Kapitel 4.6); in einem kompilierten Programm hat es dieselbe Wirkung wie END oder SYSTEM, mit dem Unterschied, daß eine Meldung wie "*STOP in line xxx of module yyy at address zzz - Hit any key to return to system*" angezeigt wird.

Geben Sie einen *exitcode* an, so wird - genau wie bei END und SYSTEM - dieser an das Betriebssystem zurückgegeben, wo man dann (zum Beispiel in einer Batchprozedur) feststellen kann, welcher Exit-Code benutzt wurde.

*Siehe auch* END (305), SYSTEM (406).

## STR\$ (Funktion)

**Standard**

*Seit* QuickBASIC 2.0

*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 Funktionen Formatx\$ sind in vielen Fällen bequemer einzusetzen als STR\$.

*Siehe auch* VAL (412), Formatx\$ (440).

*Seit* QuickBASIC 2.0

*Anwendung*  $x = \text{STRIG}(\text{argument})$

*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):

<i>argument</i>	<b>Funktion</b>
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

*Bemerkung* » Die Stellung der Steuerknüppel wird mit der Funktion STICK abgefragt.

*Siehe auch* STICK (399).

*Seit* QuickBASIC 2.0

*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 (346), STRIG (Funktion) (401).

## STRING\$ (Funktion)

Standard

*Seit* QuickBASIC 2.0

*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 *Illegal function call!*). *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 Zeichnen von Linien usw.:

```
PRINT "Ü"; STRING$(78, "-"); "¿"
```

Bedenken Sie aber, daß dabei erst ein temporärer String erzeugt werden muß, was in zeitkritischen Anwendungen ein Nachteil sein kann (hier ist es dann eventuell sinnvoller, Zeichen mittels eines Interrupts auszugeben; siehe Anhang E).

*Siehe auch* SPACE\$ (394).

## SUB/FUNCTION (Befehl)

Struktur

*Seit* BASIC 7.1 PDS

*Anwendung* (1) `SUB prozedurname [parameter] [STATIC]`  
(2) `FUNCTION funktionsname [parameter] [STATIC]`  
...  
(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 (beziehungsweise dem FUNCTION- und END FUNCTION-Befehl) ist Prozedurcode.

*prozedurname* ist ein beliebiger gültiger Name für die Prozedur beziehungsweise Funktion. Bei FUNCTION darf (und sollte) dieser

Name einen Typenbezeichner enthalten, denn dies ist die einzige Möglichkeit, festzulegen, von welchem Variablentyp eine Funktion sein soll, sieht man einmal von DEFxxx-Befehlen ab.

*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. Strings mit fester Länge können nicht in der Liste auftauchen; wenn allerdings eine Prozedur aufgerufen wird und man als Argument einen String mit fester Länge angibt, wird dieser in einen gewöhnlichen String umgewandelt, bevor er an die Prozedur weitergegeben wird.

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 eigentliche Variable immer noch 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.

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. Zu statischen Variablen siehe Kapitel 12.5, "Statische und automatische Variablen".

Prozeduren und Funktionen können sich selbst rekursiv aufrufen, nicht aber rekursiv definiert werden. Das heißt, daß nicht innerhalb einer SUB...END SUB-Definition eine weitere Prozedur mit SUB...END SUB definiert werden kann, wie das zum Beispiel in Pascal möglich ist. Rekursive Aufrufe sind aber möglich, also kann innerhalb der Definition SUB TestSub...END SUB durchaus der

Aufruf TestSub vorkommen. Zum diesem Thema siehe Kapitel 10.4, "Rekursive Programmierung".

Mit dem Befehl EXIT SUB beziehungsweise EXIT FUNCTION kann die Prozedur/Funktion jederzeit verlassen werden.

Für Funktionen gilt, daß der Funktionswert mittels einer gewöhnlichen Zuweisung (Gleichheitszeichen) zugeordnet wird.

Alle Variablen, die innerhalb einer Funktion oder Prozedur benutzt und vereinbart werden, sind lokale Variablen, die ausschließlich dieser Prozedur beziehungsweise Funktion zur Verfügung stehen. Ausnahmen sind die Parametervariablen, Variablen, die innerhalb der Prozedur mit SHARED vereinbart sind, und Variablen, die im Modulcode mit COMMON SHARED oder DIM SHARED vereinbart sind. Siehe dazu Kapitel 3.4, "Prozeduren, Funktionen, Parameter, globale Variablen".

Auch für Labels und Zeilennummern gilt die lokale Zugehörigkeit zur Prozedur beziehungsweise Funktion. Sie können nicht mit GOSUB oder GOTO eine Zeile anspringen, die außerhalb der Prozedur liegt (Ausnahmen: ON ERROR GOTO, ON event GOSUB).

#### *Beispiel*

(einige durcheinandergewürfelte und kommentierte Beispiele zur Parameterübergabe):

```
SUB DruckWas(a$, b$)
    PRINT a$
    PRINT b$
END SUB

' Die beiden Stringkonstanten heißen in der
' Prozedur a$ und b$.
DruckWas "Dies ist ein", "kurzer Text"

' Innerhalb der Prozedur wird dieser b$
' a$ heißen:
b$ = "Dies ist ein"
' Und dieser a$ b$:
a$ = "kurzer Text"
DruckWas a$, b$

' Man kann übrigens auch schreiben:
CALL DruckWas (a$, b$)

FUNCTION QuerSumme&(Text AS STRING)
    FOR i% = 1 TO LEN(Text)
        ' Ich würde gern schreiben:
        ' QuerSumme& = QuerSumme& + ...
        ' aber das geht nicht, weil dabei
        ' dauernd die Funktion QuerSumme
        ' rekursiv aufgerufen würde!
        z& = z& + ASC(MID$(Text, i%, 1))
    NEXT
```

*(Fortsetzung nächste Seite)*



(Fortsetzung)

```
    QuerSumme& = z&
    ' Ein z& = 0 am Anfang kann ich mir sparen,
    ' weil es eine automatische Variable ist, die
    ' auf 0 gesetzt wird (genauso, wie wenn ich
    ' in einem gewöhnlichen Programm z& zum
    ' ersten Mal benutze)
END FUNCTION

PRINT QuerSumme&("Halli-Hallo")

SUB MachEsGross(a$)
    a$ = UCASE$(a$)
    PRINT a$
END SUB

Text$ = "dies ist ein kleiner text"
MachEsGross Text$
PRINT Text$
' Weil a$ in MachEsGross ein Variablenparameter
' ist, wird die Änderung, die in der Prozedur
' an a$ gemacht wird, auch mit Text$ durchge-
' führt (a$ ist sozusagen nur ein verkleideter
' Text$). Es wird also zweimal auf dem Schirm
' erscheinen:
' DIES IST EIN KLEINER TEXT

SUB LassEsKlein(BYVAL a$)
    a$ = UCASE$(a$)
    PRINT a$
END SUB

Text$ = "dies bleibt ein kleiner text"
LassEsKlein Text$
PRINT Text$

' Diesmal ist a$ in LassEsKlein ein Wertepara-
' meter. Für ihn wird eine temporäre Variable
' erzeugt, in der Prozedur gelingt das Umwan-
' deln in Großbuchstaben, und es erscheint
' DIES BLEIBT EIN KLEINER TEXT
' Dann aber, zurück im Hauptprogramm, hat sich
' der Wert von Text$ nicht geändert. Es
' erscheint
' dies bleibt ein kleiner text
```

*Siehe auch* CALL (277), DECLARE (294).

## SWAP (Befehl)

Standard

*Seit* QuickBASIC 2.0

*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.

## SYSTEM (Befehl)

Standard

*Seit* BASIC 7.0 PDS

*Anwendung* SYSTEM [*errorlevel%*]

*Nutzen* Beendet ein Programm. Die optionale Variable *errorlevel%* kann benutzt werden, um einen Beendigungs-Code an das aufrufende Programm (zumeist das Betriebssystem, also DOS oder OS/2) zurückzugeben. In Batch-Dateien kann dieser Code dann mit IF ERRORLEVEL... abgefragt werden. Schwere Fehler, die nicht vom Programm abgefangen werden, setzen diese Variable auf -1.

SYSTEM schließt alle Dateien; die Kontrolle wird an DOS beziehungsweise OS/2 zurückgegeben.

*Bemerkung* » SYSTEM ist identisch mit END.

*Siehe auch* STOP (400), END (305).

## TAB (Funktion)

Standard

*Seit* QuickBASIC 2.0

*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 benutzt werden kann und, anstatt einen Funktionswert zurückzugeben, den Cursor an 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 (beziehungsweise ausgegebenen) Zeichen nicht mehr direkt auf die Cursorposition schließen läßt, wenn Sie also zum Beispiel Zeichen drucken, die nicht sichtbar sind (siehe Beispiel).

*Bemerkung* » Exaktere Positionierung können Sie durch Einsatz von SPACES erreichen; dann muß allerdings auch mehr gerechnet werden.

*Beispiel* (zur Unvollkommenheit von TAB):

```
' Dieses Programm funktioniert anstandslos;  
' die Summe wird wirklich ab Spalte 50 ausge-  
' geben.  
  
LPRINT  
LPRINT "Summe:"; TAB(50); "285,56 DM"  
  
' Hier, wo die Zeichenkombination CHR$(27)  
' + "-1" benutzt wird, um das Unterstreichen  
' anzuschalten, erfolgt die Ausgabe jedoch ab  
' Spalte 47, weil die drei Zeichen von TAB mit-  
' gezählt werden, aber den "Cursor", also den  
' Druckerkopf in diesem Falle, nicht bewegen:  
  
LPRINT  
LPRINT CHR$(27); "-1";  
LPRINT "Summe:"; TAB(50); "285,56 DM";  
LPRINT CHR$(27); "-0"
```

*Siehe auch* SPC (394), SPACE\$ (394).

## TAN (Funktion)

**Standard**

*Seit* QuickBASIC 2.0

*Anwendung*  $x = \text{TAN}(y)$

*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  $180/\pi$  (57,2957795130824) geteilt werden.

*Bemerkung* » Die TAN-Funktion benutzt die Mathematik-Libraries. Ihr unbeachteter Einsatz kann deshalb das EXE-File unnötig um etwa 10 KB verlängern (siehe "Programmgröße und RAM-Speicherplatz" in Kapitel 19.2).

*Siehe auch* SIN (393), COS (290), ATN (275).

## TIME\$ (Systemvariable)

Standard

*Seit* QuickBASIC 2.0

*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.

*Bemerkung* » Zu beachten ist der Unterschied zwischen Systemuhr und Echtzeituhr: Die Systemuhr wird beim Booten von der Echtzeituhr gesetzt und läuft von da an unabhängig; eine Änderung an TIME\$ ist also auch nur bis zum nächsten Bootvorgang wirksam.  
» Die Date-Toolbox enthält interessante Routinen zum Verarbeiten von Datum und Uhrzeit.

*Siehe auch* DATE\$ (293), TIMER (409).

## TIMER (Funktion)

Standard

*Seit* QuickBASIC 2.0

*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.  
» Wenn der Inhalt von TIME\$ verändert wird, ändert sich der Inhalt von TIMER entsprechend. TIMER kann allerdings nicht direkt auf einen bestimmten Wert gesetzt werden.

*Beispiel* Diese Prozedur verursacht eine Pause von 0,5 Sekunden (mit dem Pausen-Befehl SLEEP kann nur für ganze Sekunden pausiert werden):

```
SUB KurzePause  
  
    x = TIMER  
    DO UNTIL TIMER > x + 0.5  
    LOOP  
  
END SUB
```

*Siehe auch* TIME\$ (408).

<i>Seit</i>	QuickBASIC 2.0
<i>Anwendung</i>	TIMER ON TIMER OFF TIMER STOP
<i>Nutzen</i>	Schaltet das Event-Trapping für die Systemuhr ein (bevor es wirksam wird, muß noch ein ON TIMER( <i>n</i> ) GOSUB-Befehl ausgeführt werden), schaltet es aus oder unterbricht es. Weitere Erklärungen finden Sie bei ON <i>event</i> GOSUB.
<i>Siehe auch</i>	ON <i>event</i> GOSUB (346), TIMER (Funktion) (408).

<i>Seit</i>	QuickBASIC 2.0
<i>Anwendung</i>	TRON TROFF
<i>Nutzen</i>	<p>Diese Befehle wurden im alten BASIC-Interpreter benutzt, um den Programmablauf am Bildschirm zu verfolgen. Während des Programmablaufs wurde, wenn TRON aktiv war, immer die Zeilennummer, die gerade ausgeführt wurde, am Bildschirm ausgegeben. Dieselbe Funktion haben die Befehle heute noch, aber nur, wenn Sie beim Kompilieren den Switch /D angeben, der das Programm unter Umständen signifikant verlängert. Bedingung ist außerdem, daß Ihr Programm ausreichend Zeilennummern enthält (die ebenfalls die Programmgröße unnötig erhöhen), denn die Ausführung von Zeilen ohne Nummer wird nicht angezeigt (auch Zeilenlabels nützen nichts).</p> <p>Innerhalb von QBX ist ein TRON-Befehl im Programm gleichwertig mit dem Aktivieren von "Trace On" im "Debug"-Menü. TROFF deaktiviert den Trace-Modus wieder, und so können die beiden Befehle in QBX durchaus sinnvoll sein, um interessante Programmpassagen "live" mitzerleben. Man müßte sonst dauernd von Hand den Trace-Modus ein- und ausschalten.</p>

*Seit* BASIC 7.0 PDS

*Anwendung* TYPE *typename*  
          *element*[(*index*)] AS *typ*  
          ...  
END TYPE

*Nutzen* Vereinbart einen selbstdefinierten Typ. Eine solche TYPE-Vereinbarung sollte immer möglichst weit am Programmanfang stehen.

*typename* ist der Name des neuen Typs. Die Erfahrung hat gezeigt, daß es praktisch ist, den Namen immer auf "..typ" enden zu lassen, denn wenn Sie einen Typ namens "Adresse" definierten, könnten Sie später keine Variable namens "Adresse" benutzen, die diesen Typ hat, weil dann ein Name doppelt aufträte. Das führt zu Verwirrungen, also besser ein "AdressenTyp" und eine Variable "Adresse" von diesem Typ.

*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 (hier dürfen nicht, wie sonst meistens, statt AS auch Typenbezeichner benutzt werden) 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 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 dürfen.

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.

*Beispiel* Siehe Beispiele in Kapitel 3.3, "Arrays und Records".

*Siehe auch* DIM (299).

## UBOUND (Funktion)

Standard

*Seit* QuickBASIC 4.0

*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. Bei Dimensionierungen mit einer TO-Angabe, wie DIM z%(-180 TO 180), ist die obere Grenze die Zahl rechts von TO; bei einfachen Dimensionierungen wie DIM k%(99) ist die obere Grenze die in Klammern angegebene Zahl.

*Siehe auch* DIM (299), LBOUND (333).

## UCASE\$ (Funktion)

Standard

*Seit* QuickBASIC 4.0

*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 Buchstaben mit den ASCII-Codes 97-122, also a-z, in Großbuchstaben verwandelt sind.

UCASE\$ wandelt außer den genannten 26 Buchstaben keine um, so daß Umlaute jedweder Art unberücksichtigt bleiben.

*Siehe auch* LCASE\$ (333).

## UEVENT (Befehle)

Trapping

*Seit* BASIC 6.0

*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.

*Siehe auch* ON *event* GOSUB (346).

## VAL (Funktion)

Standard

*Seit* QuickBASIC 2.0

*Anwendung*  $x = \text{VAL}(y\$)$

*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.

*Beispiel* Einige Strings und der dazugehörige VAL-Wert:

"844"	844
"844.34"	844.34
"844.34.56"	844.34
" - . 5"	-0.5
" 12 A, 13 B"	12
"1-2"	1
"Hallo!"	0

*Siehe auch* STR\$ (400).

## VARPTR (Funktion)

Speicher

*Seit* QuickBASIC 2.0

*Anwendung*  $x = \text{VARPTR}(\text{variable})$

*Nutzen* Gibt die (Offset-)Adresse einer beliebigen Variable zurück. Lediglich für Strings ist die Verwendung von VARPTR eingeschränkt; für Far Strings ist die Funktion unbrauchbar, mit Near Strings gibt sie die Adresse des *Stringdeskriptors* und nicht des Strings selbst zurück. 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 und es darüberhinaus gleichgültig ist, ob man mit Near oder Far Strings arbeitet.



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.

» BASIC verschiebt Variablen im Speicher. Ermitteln Sie die Adresse einer Variablen erst unmittelbar, bevor Sie sie verwenden wollen.

*Beispiel* Siehe Beispiel zu BSAVE.

*Siehe auch* VARSEG (413), SSEG (395), SADD (377), BSAVE (276), BLOAD (276), PEEK (357), POKE (362).

## VARPTR\$ (Funktion)

Speicher

*Seit* QuickBASIC 2.0

*Anwendung* `x$ = VARPTR$(variable)`

*Nutzen* VARPTR\$ gibt einen als drei Bytes langer String codierten Zeiger auf die *variable* zurück. VARPTR\$ sollte ausschließlich zur Verwendung in Verbindung mit PLAY und DRAW dienen (siehe dort).

*Siehe auch* PLAY (359), DRAW (302).

## VARSEG (Funktion)

Speicher

*Seit* QuickBASIC 2.0

*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.ä. auf die Variable zugreift, muß man mit `DEF SEG = segmentadresse` die Segmentadresse der Variable einstellen, da bei den Befehlen selbst immer nur die Offsetadresse angegeben wird.

*Bemerkung* » Die komplette Adresse einer Variable besteht aus der Segment- und der Offsetadresse. Die Offsetadresse erhält man mit VARPTR.

» BASIC verschiebt Variablen im Speicher. Ermitteln Sie die Adresse einer Variablen erst unmittelbar, bevor Sie sie verwenden wollen.

*Siehe auch* VARPTR (412), BSAVE (276), BLOAD (276), PEEK (357), POKE (362).

## Grafik

Anwendung VIEW [[SCREEN] (x1, y1)-(x2, y2)  
[ , [farbe] [ , rahmen]]]

Wenn Sie alle Angaben weglassen und nur den Befehl VIEW benutzen, wird der ganze Bildschirm zum Viewport.

*Bemerkung* » Die Befehle RUN und SCREEN führen jeweils auch ein VIEW aus, so daß nach Benutzung dieser Befehle der ganze Bildschirm wieder Grafik-Viewport ist.

*Seit* QuickBASIC 4.0

*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 *illegal function call*, 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.

*Bemerkung* » Mit Hilfe der Window-Toolbox und ihrer speziellen Print-Routinen ist es möglich, Fenster zu benutzen, die, im Gegensatz zu einem Text-Viewport, auch spaltenweise begrenzt werden können.

*Beispiel* Dieses Programm, das als Prototyp für verschiedene Anwendungen gedacht ist, spaltet den Bildschirm in zwei Text-Bereiche und reagiert auf jede Eingabe im unteren Fenster mit einer Ausgabe im oberen.

```
DECLARE SUB Meldung (Text AS STRING)
DIM Befehl AS STRING

' Oberes Fenster
COLOR 7, 1: CLS

' Unteres Fenster
VIEW PRINT 21 TO 25
COLOR 1, 7: CLS

' Schleife wird natürlich in einem
' ernsthaften Programm umfangreicher:
DO
    LINE INPUT Befehl
    Meldung "Befehl "+ Befehl+ " " eingegeben!"
LOOP

SYSTEM
```

```

SUB Meldung (Text AS STRING)

    DIM UntenSpalte AS INTEGER
    DIM UntenZeile AS INTEGER
    STATIC ObenSpalte AS INTEGER
    STATIC ObenZeile AS INTEGER
    UntenSpalte = POS(0) ' Position des Cursors
    UntenZeile = CSRLIN  ' in unterem Fenster
                        ' merken

    ' Oberes Fenster aktivieren:
    VIEW PRINT 1 TO 20: COLOR 7, 1

    ' Wenn die Prozedur zum ersten Mal
    ' aufgerufen wird, enthalten die statischen
    ' Variablen noch 0:
    IF ObenSpalte = 0 THEN
        OpenSpalte = 1
        ObenZeile = 1
        LOCATE 1, 1
    ELSE
        ' Sonst wird der Cursor auf die letzte
        ' Position gesetzt
        LOCATE ObenZeile, ObenSpalte
    END IF

    ' Text wird ausgegeben
    PRINT Text

    ' Neue "letzte Position" wird gemerkt
    ObenSpalte = POS(0)
    ObenZeile = CSRLIN

    ' unteres Fenster aktivieren
    VIEW PRINT 21 TO 25: COLOR 1, 7
    ' Cursor korrekt in unteres Fenster setzen
    LOCATE UntenZeile, UntenSpalte

END SUB

```

Siehe auch WINDOW (419).

## WAIT (Befehl)

Standard

*Seit* QuickBASIC 2.0

*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 übrig-

bleibt, der nicht 0 ist, wird das Programm fortgesetzt, sonst wird weiter gewartet.

Der Befehl

WAIT *k*, *a*, *x*

läßt sich auch schreiben als:

DO

*z* = INP(*a*)

LOOP UNTIL ((*z* XOR *x*) AND *a*) > 0

Die untere Umschreibung ist von der Funktion her identisch mit dem WAIT-Befehl, hat den Nachteil, daß sie etwas länger ist, und den Vorteil, daß man zusätzlich eine Zeitprüfung einbauen könnte, damit das Programm nicht unter Umständen ewig wartet (= abstürzt), wenn der Hardware-Kanal nicht so handelt wie erwartet.

*Bemerkung* » Siehe Bemerkung zu INP.

*Siehe auch* INP (324), OUT (354).

## WHILE...WEND (Befehl)

**Struktur**

*Seit* QuickBASIC 2.0

*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.

*Bemerkung* » Die DO...LOOP-Schleife bietet zusätzlich die Möglichkeit, die Schleife mit EXIT DO zu verlassen, was bei WHILE...WEND nicht möglich ist.

*Siehe auch* DO...LOOP (301).

## WIDTH (Befehl)

**Standard**

*Seit* QuickBASIC 4.0

*Anwendung* (1) WIDTH [*spalten*] [, *zeilen*]

(2) WIDTH {LPRINT|#*dateinummer*|*geraet*\$},  
*zeilenbreite*

*Nutzen* Setzt die Breite des Bildschirms oder einer Datei beziehungsweise eines Geräts.

Am Bildschirm (Syntax 1) macht sich die 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.

**Bemerkung** » WIDTH #*dateinummer* hat auf gewöhnliche Dateien keine Wirkung.

» Für die Zeichenbreite auf dem Drucker: siehe Bemerkung zu LPOS.

**Siehe auch** LPOS (339).

*Seit* QuickBASIC 2.0

*Anwendung* WINDOW [[SCREEN] (x1, y1)-(x2, y2)]

*Nutzen* Mit dem WINDOW-Befehl läßt sich das übliche Pixel-Koordinatensystem 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 (das nichts mit dem SCREEN aus dem VIEW-Befehl zu tun hat), 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. Probleme kann es nur noch geben, wenn Sie mit PRINT und mit Buchstaben arbeiten, deren Höhe bei der CGA-Auflösung 8, bei EGA aber 14 Pixel ist. Wenn Sie solche Probleme aber - zum Beispiel durch Einführung von Konstanten - schon bei der Programmerstellung berücksichtigen, werden Sie später keine Schwierigkeiten haben, das Programm an andere Grafikmodi anzupassen.

» Wenn ein VIEW-Befehl aktiv ist, dann bezieht sich der WINDOW-Befehl nur auf den damit definierten Grafik-Viewport.

» Die Auflösungen der verschiedenen Grafikmodi finden Sie bei SCREEN; sie spielen für den WINDOW-Befehl aber direkt keine Rolle.

*Siehe auch* SCREEN (378), VIEW (414).

## WRITE (Befehl)

I/O, Standard

*Seit* QuickBASIC 2.0

*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 beziehungsweise in die Datei ausgegeben. Strings werden dabei automatisch in Anführungszeichen eingeschlossen.

Eine neue Zeile wird begonnen, wenn alle Elemente der Liste auf den Schirm beziehungsweise in die Datei geschrieben wurden.

Wird *liste* weggelassen, so schreibt WRITE eine Leerzeile auf den Bildschirm oder in die Datei.

*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. In der Tat lassen sich die Daten, die WRITE in eine Datei schreibt, mit einem INPUT-Befehl, der völlig gleich lautet, wieder einlesen.

» Wenn Sie WRITE #dateinummer benutzen, um in eine mit OPEN FOR RANDOM geöffnete Datei zu schreiben, wird - egal ob FIELD benutzt wurde oder nicht - der Dateipuffer mit der Ausgabe des WRITE-Befehls überschrieben. WRITE alleine schreibt nichts in die Datei; es muß erst ein PUT ausgeführt werden, damit der Puffer wirklich in die Datei geschrieben wird. Es dürfen nicht mehr Bytes geschrieben werden, als die Satzlänge es zuläßt, sonst tritt ein FIELD buffer overflow auf.

*Beispiel* Das Beispiel zu STATIC benutzt den WRITE-Befehl.

*Siehe auch* PRINT (363), INPUT (326).



# Referenzteil ISAM

Dieser Referenzteil enthält alle Befehle, die auf ISAM-Datenbanken angewendet werden können. Die meisten davon sind ausschließlich für den ISAM-Gebrauch geeignet; einige Befehle sind allerdings auch schon aus dem Standard-BASIC-Referenzteil bekannt, weil sie sowohl für ISAM als auch für gewöhnliche Dateiverwaltung eingesetzt werden.

## BEGINTRANS (Befehl)

ISAM

*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 einen CLOSE- oder COMMITTRANS-Befehl 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.

*Beispiel* Siehe Prozedur DatenbankOeffnen im ISAM-Beispielprogramm (Kapitel 7.4).

*Siehe auch* COMMITTRANS (423), SAVEPOINT (431).

## BOF (Funktion)

ISAM

*Anwendung*  $x\% = \text{BOF}(\text{dateinummer})$

*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. Dann wird BOF 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 BOF immer TRUE.

### Beispiel

Diese Prozedur gibt - ich setze eine ISAM-Datei und eine Datenstruktur nach dem Muster meines Demoprogrammes in Kapitel 7.4 voraus - alle Namen in umgekehrter alphabetischer Reihenfolge aus. Dabei wird einfach die normale Sortierfolge der Indexliste "NameVorname" rückwärts ausgegeben, und BOF wird gebraucht, um festzustellen, wann man das Ende (oder richtiger: den Anfang) der Liste erreicht hat.

```
SUB RueckwaertsListe

    SETINDEX 1, "NameVorname"
    MOVELAST 1
    DO UNTIL BOF(1)
        RETRIEVE 1, Zugriff
        AnzeigeDatensatz Kurz
        MOVEPREVIOUS
    LOOP

END SUB
```

Siehe auch EOF (426).

## CHECKPOINT (Befehl)

ISAM

*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.

Bei INPUT- und INKEY\$-Anweisungen wird von BASIC aus automatisch in regelmäßigen Abständen ein CHECKPOINT-Befehl ausgeführt (siehe dort). Wenn Sie mit /D kompilieren, wird CHECKPOINT zudem nach jedem DELETE-, INSERT- und UPDATE-Befehl automatisch aufgerufen.

*Bemerkung* » Die Tatsache, daß mit ROLLBACK Operationen innerhalb einer Transaktion zurückgenommen werden können, wird von CHECKPOINT in keiner Weise beeinflusst.

## CLOSE (Befehl)

ISAM

**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 Standard-Referenzteil.

**Siehe auch** CLOSE (285), COMMITTRANS (423).

## COMMITTRANS (Befehl)

ISAM

**Anwendung** COMMITTRANS

**Nutzen** Beendet die gerade laufende Transaktion. COMMITTRANS darf nicht benutzt werden, wenn keine Transaktion läuft; andernfalls wird ein *Illegal function call*-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. Dafür ist CHECKPOINT vorgesehen.

**Beispiel** Siehe ISAM-Beispielprogramm (85).

**Siehe auch** BEGINTRANS (421), CLOSE (423).

## CREATEINDEX (Befehl)

ISAM

**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 *Duplicate value for unique 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 Felder 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.

» Der CREATEINDEX-Befehl ist nur mit der erweiterten ISAM-Library verfügbar, also mit dem speicherresidenten PROISAMD.EXE beziehungsweise der SETUP-Einstellung "Database Creation and Access".

*Beispiel* Siehe Prozedur DatenbankOeffnen im ISAM-Beispielprogramm (Kapitel 7.4).

*Siehe auch* DELETEINDEX (425), SETINDEX (433), GETINDEX\$ (427).

## DELETE (Befehl)

ISAM

*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 *no current record* 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.

*Beispiel* Siehe Prozedur HauptMenue im ISAM-Beispielprogramm (Kapitel 7.4).

*Siehe auch* INSERT (427), UPDATE (434), ROLLBACK (431).

## DELETEINDEX (Befehl)

ISAM

*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.

» Der DELETEINDEX-Befehl ist nur mit der erweiterten ISAM-Library verfügbar, also mit dem speicherresidenten PROISAMD.EXE beziehungsweise der SETUP-Einstellung "Database Creation and Access".

*Siehe auch* CREATEINDEX (423), SETINDEX (433).

## DELETETABLE (Befehl)

ISAM

*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 diese trotzdem weiterhin.
- » Der DELETETABLE-Befehl ist nur mit der erweiterten ISAM-Library verfügbar, also mit dem speicherresidenten PROISAMD.EXE beziehungsweise der SETUP-Einstellung "Database Creation and Access".
- Siehe auch* OPEN (428).

## EOF (Funktion)

ISAM

*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.

*Beispiel* Siehe ISAM-Beispielprogramm (Kapitel 7.4).

*Siehe auch* BOF (421).

## FILEATTR (Funktion)

ISAM

*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 Standard-Referenzteil.

*Siehe auch* FILEATTR (312).

## GETINDEX\$ (Funktion)

ISAM

**Anwendung**  $x\$ = \text{GETINDEX\$}(\text{dateinummer})$

**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 (433), CREATEINDEX (423).

## INSERT (Befehl)

ISAM

**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.

**Beispiel** Siehe ISAM-Beispielprogramm (85).

**Siehe auch** UPDATE (434), DELETE (425), CREATEINDEX (423).

## LOF (Funktion)

ISAM

**Anwendung**  $x = \text{LOF}(\text{dateinummer})$

**Nutzen** Auf ISAM-Datenbanken angewendet, gibt LOF die Anzahl der Datensätze in der Datenbank zurück, die unter der Nummer *dateinummer* geöffnet wurde.

**Bemerkung** » Details zu LOF siehe im Standard-Referenzteil.

**Siehe auch** LOF (338).

**Anwendung**    `MOVEFIRST [#]dateinummer`

`MOVELAST [#]dateinummer`

`MOVENEXT [#]dateinummer`

`MOVEPREVIOUS [#]dateinummer`

**Nutzen**        Den Dateizeiger in einer ISAM-Datenbank verschieben. *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 eines SETINDEX-Befehls 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 der schon Fall ist, ignoriert ISAM den Befehl, ohne einen Fehler zu verursachen.

**Beispiel**        Siehe ISAM-Beispielprogramm (Kapitel 7.4) und Beispiel zu BOF.

**Siehe auch**    BOF (421), EOF (426), SEEKxx (432), SETINDEX (433).

**OPEN (Befehl)****ISAM**

**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 (siehe dazu das Beispiel). 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 zugreift.

*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 Standard-Referenzteil.
- » Es können maximal 13 ISAM-Datenbanken zugleich geöffnet sein (siehe auch "Anzahl gleichzeitig geöffneter ISAM-Dateien und -Datenbanken" in Kapitel 7.7).

*Beispiel* (zum Öffnen der Datenbank mit verschiedenen Typen):

```
TYPE GesamtTyp
    Name AS STRING * 40
    Personalnummer AS INTEGER
    Gehaltskonto AS STRING * 15
END TYP

TYPE OhneNamen
    Gehaltskonto AS STRING * 15
    Personalnummer AS INTEGER
END TYPE

TYPE FehlerTyp
    MitarbeiterName AS STRING * 10
    Gehaltskonto AS STRING * 15
END TYPE
```

(Fortsetzung nächste Seite)

(Fortsetzung)

```
' wir gehen davon aus, daß die Datenbank nicht
' existiert. Dann wird sie mit dem Befehl
OPEN "MITARB.ISM" FOR ISAM GesamtTyp "Personal" AS #1
' neu erzeugt.
CLOSE

' Der folgende Befehl ist möglich, da der Typ
' OhneNamen eine Teilmenge der Elemente von
' GesamtTyp, mit dem die Datenbank erstellt
' wurde, enthält. Wenn die Datenbank so
' geöffnet wird, müssen natürlich auch die
' RETRIEVE-, INSERT- und UPDATE-Befehle
' Variablen vom Typ OhneNamen benutzen, und es
' wird unmöglich, auf die Namen zuzugreifen.
OPEN "MITARB.ISM" FOR ISAM OhneNamen "Personal" AS #1
CLOSE
' Ein dritter Versuch wird fehlschlagen:
OPEN "MITARB.ISM" FOR ISAM FehlerTyp "Personal" AS #1
(Fortsetzung nächste Seite)
' Dieser Befehl ist nicht erlaubt, weil
' FehlerTyp keine Untermenge von GesamtTyp ist.
' Zwar hat FehlerTyp Elemente, die sich auch in
' GesamtTyp wiederfinden, aber bei ISAM kommt
' es auf die Feldnamen an - und ein Feld namens
' "MitarbeiterName" ist in GesamtTyp nicht
' vorhanden.
```

Ein weiteres Beispiel zur OPEN-Anwendung finden Sie in der Prozedur DatenbankOeffnen im ISAM-Beispielprogramm (Kapitel 7.4).

Siehe auch OPEN (428), CLOSE (423).

## RETRIEVE (Befehl)

ISAM

**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 *No current record*-Fehler.

**Beispiel** Siehe ISAM-Beispielprogramm (Kapitel 7.4).

**Siehe auch** UPDATE (434), INSERT (427).

*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.

*Beispiel* Siehe Proz. HauptMenue im ISAM-Beispielprogramm (Kapitel 7.4).

*Siehe auch* BEGINTRANS (421), COMMITTRANS (423), SAVEPOINT (431).

**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.

*Beispiel* Siehe ISAM-Beispielprogramm (Kapitel 7.4).  
*Siehe auch* BEGINTRANS (421), COMMITTRANS (423), ROLLBACK (431).

## SEEKxx (Befehle)

ISAM

*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 Index aktiv ist, der mit CREATEINDEX 1, "DemoIndex", 0, "Name", "Vorname", "PLZ", "Ort" erstellt wurde, 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 SEEKxx-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* » SEEKxx-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 SEEKxx-Befehle keinen praktischen Nutzen, weil sie immer nur den ersten passenden Datensatz aus der Datei suchen.

*Beispiel* Siehe Prozeduren SucheDatensatz und DatenListe im ISAM-Beispielprogramm (Kapitel 7.4).

*Siehe auch* MOVExxx (428), CREATEINDEX (423), SETINDEX (433), TEXTCOMP (433).

**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 CREATE-INDEX erstellten) Index sein. Wenn Sie einen Leerstring ("" ) für *indexname*\$ verwenden, wird der Null-Index aktiv, in dem die Daten unsortiert sind.

**Beispiel** Siehe ISAM-Beispielprogramm (Kapitel 7.4).

**Siehe auch** CREATEINDEX (423); GETINDEX\$ (427), DELETEINDEX (425).

**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 7.4).

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 "Veränderungen an den Sortier- und Suchalgorithmen" (Kapitel 10.3).

**Beispiel** Die beiden folgenden Funktionen ermöglichen es, auch längere Strings mit TEXTCOMP-Methoden zu vergleichen. LongTEXTCOMP können Sie in Ihren Programmen anstelle von TEXTCOMP einsetzen; LongTEXTCOMPx ist eine nur intern benutzte Routine, die sich rekursiv aufruft und so längere Strings in TEXTCOMP-gerechte Häppchen unterteilt, wenn sie sich nicht in den ersten 255 Zeichen unterscheiden. Die Zwei-Prozeduren-Architektur ist nötig, um den RAM-Speicher nicht mit zu vielen temporären Strings zu belasten. Bei Strings von mehreren tausend Zeichen kann es trotzdem vorkommen, daß der String-Speicherplatz für diese Prozeduren nicht ausreicht.

```

DECLARE FUNCTION LongTEXTCOMP% (string1 AS STRING,
                                string2 AS STRING)
DECLARE FUNCTION LongTEXTCOMPx% (string1 AS STRING,
                                offset1 AS INTEGER, string2 AS STRING, offset2 AS INTEGER)

FUNCTION LongTEXTCOMP% (string1 AS STRING, string2 AS STRING)
    LongTEXTCOMP% = LongTEXTCOMPx(string1, 1, string2, 1)
END FUNCTION

FUNCTION LongTEXTCOMPx% (string1 AS STRING, offset1 AS INTEGER,
                        string2 AS STRING, offset2 AS INTEGER)

    ltc% = TEXTCOMP(MID$(string1, offset1, 255),
                    MID$(string2, offset2, 255))
    IF ltc% = 0 THEN
        IF LEN(string1) > 254 + offset1
            OR LEN(string2) > 254 + offset2 THEN
            ltc% = LongTEXTCOMPx(string1, offset1 + 255,
                                string2, offset2 + 255)
        END IF
    END IF
    LongTEXTCOMPx = ltc%
END FUNCTION

```

Siehe auch SEEKxx (432).

## UPDATE (Befehl)

ISAM

**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 *No current record* erzeugt.

**Beispiel** Siehe ISAM-Beispielprogramm (Kapitel 7.4).

**Siehe auch** INSERT (427), DELETE (425), CREATEINDEX (423).

# Referenzteil Date-Library

## FUNCTION DateSerial

Date

*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 (*jahr%* = 2078, *monat%* = 12, *tag%* = 32 wäre also eine ungültige Kombination, da sie einen Tag im Jahre 2079 beschreibt). Ungültige Angaben verursachen einen *Illegal function call*-Fehler.

*Bemerkung*    » Die von dieser Funktion errechneten reinen Datums-Zeitcodes sind immer ganzzahlig, denn der Datums-Teil wird, wie in Kapitel 8.2 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.

*Siehe auch*    *DateValue* (435), *TimeSerial* (437).

## FUNCTION DateValue

Date

*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, sogar, wenn die Monatsnamen ausgeschrieben sind, bei deutscher Formatierung aber versagt. "30.12.1999" als *text\$* wäre schon Grund genug für die Funktion, einen *Illegal function call* zu generieren und die Flinte ins Korn zu werfen. In *text\$* enthaltene Zeitangaben werden ignoriert, wenn sie gültig sind, und führen zu einem *Illegal function call*, wenn sie ungültig sind.

*Bemerkung* » Wie auch `DateSerial` gibt diese Funktion nur ganzzahlige Funktionswerte zurück (siehe dort).

*Siehe auch* `DateSerial` (435), `TimeValue` (438).

## FUNCTION Day

Date

*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.

*Siehe auch* `Month` (436), `Year` (439).

## FUNCTION Hour

Date

*Anwendung* `x% = Hour(zeitcode#)`

*Nutzen* Ermittelt die zum angegebenen `zeitcode#` gehörige Stunde (0-23). Es spielt keine Rolle, ob `zeitcode#` ein reiner Uhrzeit-Zeitcode oder ein vollständiger Zeitcode ist. Handelt es sich aber um einen reinen Datums-Zeitcode, wird der Funktionswert 0 zurückgegeben.

*Siehe auch* `Minute` (436), `Second` (437).

## FUNCTION Minute

Date

*Anwendung* `x% = Minute(zeitcode#)`

*Nutzen* Ermittelt die zum angegebenen `zeitcode#` gehörige Minute (0-59). Es spielt keine Rolle, ob `zeitcode#` ein reiner Uhrzeit-Zeitcode oder ein vollständiger Zeitcode ist. Handelt es sich aber um einen reinen Datums-Zeitcode, wird der Funktionswert 0 zurückgegeben.

*Siehe auch* `Hour` (436), `Second` (437).

## FUNCTION Month

Date

*Anwendung* `x% = Month(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.

*Siehe auch* `Day` (436), `Year` (439).



*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 FormatD\$-Routine aus der Format-Add-On-Library ist es ein Leichtes, das aktuelle Datum im deutschen Format auszugeben: Der Befehl dafür heißt:

`PRINT FormatD$(Now, "d.m.yy")`

*Siehe auch* DateValue (435), TimeValue (438), FormatD\$ (440).

**FUNCTION Second**

*Anwendung* `x% = Second(zeitcode#)`

*Nutzen* Ermittelt die zum angegebenen *zeitcode#* gehörige Sekunde (0-59). Es spielt keine Rolle, ob *zeitcode#* ein reiner Uhrzeit-Zeitcode oder ein vollständiger Zeitcode ist. Handelt es sich aber um einen reinen Datums-Zeitcode, wird der Funktionswert 0 zurückgegeben.

*Siehe auch* Hour (436), Minute (436).

**FUNCTION TimeSerial**

*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. Sie können aber durchaus für die drei Parameter Werte angeben, die außerhalb des üblichen Bereiches liegen, zum Beispiel 1 Uhr, -10 Minuten und 100 Sekunden. Diese Angabe wäre zum Beispiel äquivalent zu 0 Uhr, 51 Minuten und 40 Sekunden.

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.

*Beispiel* (zur Überschreitung der Tagesgrenze)

```
PRINT Hour(TimeSerial(10, 47, 23))
```

ergibt 10.

```
PRINT Hour(TimeSerial(-10, 47, 23))
```

ergibt 14, denn der angegebene Zeitpunkt liegt um 14:47:23 Uhr am Vortag.

```
PRINT Day(TimeSerial(10, 47, 23))
```

Ergibt 30, denn die von TimeSerial zurückgegebene Zahl hat keinen Datumsanteil (also 0), und der Tag mit dem Datums-Zeitcode 0 ist der 30. Dezember 1899.

```
PRINT Day(TimeSerial(-10, 47, 23))
```

Ergibt 29, denn durch die Überschreitung der 1-Tag-Grenze hat der TimeSerial-Funktionswert hier als Datumsanteil -1, und der Tag mit dem Datums-Zeitcode -1 ist der 29. Dezember 1899.

*Siehe auch* DateSerial (435), TimeValue (438).

FUNCTION TimeValue	Date
<p><i>Anwendung</i> <code>x# = TimeValue(text\$)</code></p> <p><i>Nutzen</i> 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).</p> <p>TimeValue erkennt auch 12-Stunden-Zeitangaben mit AM oder PM korrekt.</p> <p>In <code>text\$</code> enthaltene Datumsangaben werden ignoriert, wenn sie gültig sind, und führen zu einem <i>Illegal function call</i>, wenn sie ungültig (oder nicht in amerikanischer MM/DD-Notation) sind.</p> <p><i>Bemerkung</i> » Aufgrund der Tatsache, daß nur gültige Zeitangaben erkannt werden, kann diese Funktion nur reine Uhrzeit-Zeitcodes zwischen 0 und 0,99999 zurückgeben.</p> <p><i>Siehe auch</i> DateValue (435), TimeSerial (437).</p>	

## FUNCTION Weekday

Date

*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).

*Siehe auch*    Day (436).

## FUNCTION Year

Date

*Anwendung*    `x% = Year(zeitcode#)`

*Nutzen*        Ermittelt das zum angegebenen `zeitcode#` gehörige Jahr (1753-2078). Es spielt keine Rolle, ob `zeitcode#` ein reiner Datums-Zeitcode oder ein vollständiger Zeitcode ist.

*Siehe auch*    Day (436), Month (436).

# Referenzteil Format-Library

FUNCTION FormatX\$	Format
--------------------	--------

Anwendung    x\$ = FormatI\$(zahl%, format\$)  
                 x\$ = FormatL\$(zahl&, format\$)  
                 x\$ = FormatC\$(zahl@, format\$)  
                 x\$ = FormatS\$(zahl!, format\$)  
                 x\$ = FormatD\$(zahl#, format\$)

Nutzen        Wandelt eine beliebige Zahl nach festgelegtem Format in einen String um. Der letzte Buchstabe des Funktionsnamens gibt an, für welchen Datentyp sie geeignet ist: I = INTEGER, L = LONG, C = CURRENCY, S = SINGLE und D = DOUBLE. *zahl* enthält jeweils die zu formatierende Zahl, und *format\$* ist die Formatbeschreibung.

Alle Zeichen, die keine besondere Bedeutung gemäß den folgenden Tabellen haben, können in *format\$* 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 <i>format\$</i> befinden, wird gerundet. Hat die Zahl mehr Vorkommastellen, als sich Nullen vor dem Dezimalpunkt in <i>format\$</i> 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 <i>format\$</i> 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 <i>format\$</i> benutzen.

(Fortsetzung nächste Seite)

%	Der Wert wird mit 100 multipliziert, und ein Prozentzeichen wird an der Stelle eingefügt, an der es in format\$ steht.
,	Erwirkt, daß an der Stelle, an der es in format\$ 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  $\geq 0$  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.

Die FormatD\$-Routine besitzt außerdem Sonderfunktionen für die Darstellung von Zeitcodes (siehe Kapitel 8.2).

Zeichen	Bedeutung
d	Tag als Nummer ohne führende Null anzeigen
dd	Tag als Nummer mit führender Null anzeigen
ddd	Tag als englische 3-Buchstaben-Abkürzung anzeigen
dddd	Tag als vollen englischen 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 format\$ 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.)

*Beispiel*      Siehe Kapitel 8.1.  
*Siehe auch*    SetFormatCC (442).

## **SUB SetFormatCC**

**Format**

*Anwendung*    SetFormatCC *landescode%*

*Nutzen*        Setzt den Landescode, der alle FormatX\$-Funktionen betrifft. Der Landescode ist identisch mit der internationalen Telefonvorwahl, also 1 für USA, 49 für Deutschland, 33 für Frankreich etc.

*Bemerkung*    » Es reicht völlig aus, die zwei Landescodes 1 (für USA) und 49 (für Deutschland) zu benutzen, da der einzige Einfluß der Routine die Vertauschung des Tausender- und des Dezimalzeichens ist und es damit nur zwei Möglichkeiten gibt. Bei *landescode% = 1* ist der Punkt das Dezimal- und das Komma das Tausenderzeichen, bei *landescode% = 49* ist es umgekehrt.

*Siehe auch*    FormatX\$ (440).

# Referenzteil Finance-Library

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.

## FUNCTION DDB

Finance

*Anwendung*    `x# = DDB(awert#, rwert#, dauer%, periode%, fehler%)`

*Englisch*      Double-Declining Balance 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, das heißt, die degressive Abschreibung stellt immer einen gewissen Prozentsatz des gegenwärtigen Wertes dar, während die lineare Abschreibung immer ein Prozentsatz des Anschaffungswertes ist.

» Offenbar wegen eines Fehlers in der Library berechnet die Funktion in einer Laufzeit *n* nur *n-1* Abschreibungen. Deshalb müssen Sie *dauer%* immer um 1 höher setzen als gewünscht.

*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, 5, 3, x%)` und erhält DM 1.440,- (Als *dauer%* wurde 5 eingesetzt, siehe Bemerkung!).

*Siehe auch*    SLN (451), SYD (451).

**Anwendung**  $x\# = FV(zins\#,dauer\#,rate\#, anfang\#, typ\#, fehler\%)$

**Englisch** Future Value

**Nutzen** Errechnet den künftigen Wert 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 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 - ja, was nun eigentlich? Der künftige Wert dieses Geschäfts entspricht einer Zahlung von DM 3.596,94, das heißt im Klartext, 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, die 8% Zinsen will, höhere jährliche Beträge hätten zahlen müssen.

**Siehe auch** NPV (447), PV (449), Rate (450).



**Anwendung**  $x\# = \text{IPmt}(\text{zins}\#, \text{periode}\%, \text{dauer}\%,$   
 $\text{anfang}\#, \text{ende}\#, \text{typ}\%, \text{fehler}\%)$

**Englisch** Interest Payment

**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 ( $0 \leq \text{zins}\# < 1$ ); *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- 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.

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 abgezahlt 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 Zahlung auf die Kapitalsumme liefert, und diese Zahlung zur Zinszahlung addieren, oder Sie benutzen einfach die Funktion `Pmt`, die Ihnen sofort das gewünschte Ergebnis liefert. 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` (448), `PPmt` (449).

**Anwendung**  $x\# = \text{IRR}(\text{cashflow}\#(), \text{anzahl}\%, \text{schaeztung}\#, \text{fehler}\%)$

**Englisch** Internal Rate of Return

**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 *schaeztung#* müssen Sie das ungefähre Ergebnis der Berechnung angeben, von dem ausgehend IRR dann das genaue Ergebnis iteriert. Liegt *schaeztung#* zu weit neben dem richtigen Ergebnis, wird nach 20 Iterationen *fehler%* auf 1 gesetzt und die Berechnung abgebrochen.

**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 (446), NPV (447), Rate (450).

**FUNCTION MIRR****Finance**

**Anwendung**  $x\# = \text{MIRR}(\text{cashflow}\#(), \text{anzahl}\%, \text{fzins}\#, \text{izins}\#, \text{fehler}\%)$

**Englisch** Modified Internal Rate of Return

**Nutzen** Errechnet den internen Zinsfuß einer Reihe periodischer Cash Flows (Einnahmen und Ausgaben). 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, und deshalb muß bei IRR keiner von beiden angegeben werden. Weil MIRR nach einem anderen Prinzip funktioniert als IRR, entfällt die Angabe eines Schätzwertes bei MIRR.

**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 ersten 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 90.000,-, 130.000,-, 160.000,-, 170.000,-, 140.000,-, 150.000,-, 145.000,- und 130.000,-. Diese können mit einem Ertrag von 11 Prozent reinvestiert beziehungsweise angelegt werden. Mit diesem Programm...

```
REM $INCLUDE: 'financ.bi'
DIM CFlow#(1 TO 10)

DATA -500000,-20000,90000,130000,160000
DATA 170000,140000,150000,145000,130000
FOR a% = 1 TO 10: READ CFlow#(a%): 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).

Siehe auch IRR (445), NPV (447), Rate (450)

## FUNCTION NPer

Finance

Anwendung  $x\% = \text{NPer}(\text{zins}\#, \text{zahlung}\#,$   
 $\text{anfang}\#, \text{ende}\#, \text{typ}\#, \text{fehler}\%)$

Englisch Number of Periods

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 in diesem Falle eine negative Zahl angegeben werden, da Sie ja jedes Jahr Geld ausgeben.

*Siehe auch* FV (443), IPmt (444), Pmt (448), PPmt (449), PV (449), Rate (450).

## FUNCTION NPV

Finance

*Anwendung* `x# = NPV(zins#, cashflow#(), anzahl%, fehler%)`

*Englisch* Net Present Value

*Nutzen* Die Funktion errechnet den Gegenwartswert einer Reihe periodischer Cash Flows, die in der Zukunft auftreten werden, auf der Basis eines festen Abzinsungssatzes.

*zins#* ist dieser Abzinsungssatz, *cashflow#()* ist ein Feld 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 Abzinsungssatz auf ihn ja gar keinen Einfluß hat.

*Beispiel* Sie haben sich verpflichtet, an eine Firma im nächsten 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, das 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 (443), IRR (445), PV (449).

**Anwendung**  $x\# = \text{Pmt}(\text{zins}\#, \text{dauer}\%,$   
 $\text{anfang}\#, \text{ende}\#, \text{typ}\%, \text{fehler}\%)$

**Englisch** Payment

**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 verwandten 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 (444), PPmt (449).

**FUNCTION PPmt****Finance**

**Anwendung**  $x\# = \text{PPmt}(\text{zins}\#, \text{periode}\%, \text{dauer}\%,$   
 $\text{anfang}\#, \text{ende}\#, \text{typ}\%, \text{fehler}\%)$

**Englisch** Principal Payment

**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 ( $0 \leq \text{zins}\# < 1$ ); *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 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.

*Siehe auch* IPmt (444), Pmt (448).

## Finance

*Siehe auch* NPV (447), FV (443).

## Finance

Frederik Ramm: Microsoft BASIC PDS - 450 -

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 und 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 (443), PV (449), NPV (447), Pmt (448).

## FUNCTION SLN

Finance

*Anwendung* `x# = SLN(awert#, rwert#, dauer%, fehler%)`

*Englisch* Straight-line depreciation

*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 TRUE 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 (443), SYD (451).

<i>Anwendung</i>	<code>x# = SYD(awert#, rwert#, dauer%, periode%, fehler%)</code>
<i>Englisch</i>	Sum-of-year's digits depreciation
<i>Nutzen</i>	<p>Errechnet die arithmetisch-degressive (digitale) Abschreibung eines Vermögenswertes für eine bestimmte Periode seiner Abschreibungsdauer.</p> <p><i>awert#</i> ist der Anschaffungswert, <i>rwert#</i> der Restwert, der nach der Abschreibungsdauer verbleibt. <i>dauer%</i> ist die Abschreibungsdauer in Perioden; <i>periode%</i> 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.</p> <p><i>fehler%</i> wird von der Funktion auf 1 gesetzt, wenn die Berechnung nicht ausgeführt werden konnte; ansonsten ist diese Variable 0.</p>
<i>Bemerkung</i>	» Die Funktion ist vom Aufruf her identisch mit DDB, hat aber nicht den gleichen Fehler wie DDB, so daß als <i>dauer%</i> hier die wirkliche Laufzeit angegeben werden kann.
<i>Beispiel</i>	vgl. das Beispiel zu DDB; aufgrund des anderen Berechnungsverfahrens hat SYD natürlich ein anderes Ergebnis.
<i>Siehe auch</i>	DDB (443), SLN (451).



# Referenzteil

## Matrizenmathematik-Toolbox

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 Entschlüsselung in den Abschnitten über die einzelnen Funktionen beschrieben ist.

<b>FUNCTION MatAddx</b>	<b>Matrizenmathematik</b>
-------------------------	---------------------------

*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 denselben Dimensionen sein; je nach verwendeter 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 wird zurückgegeben, wenn *matrix1* und *matrix2* nicht dieselbe Dimension haben. Jeder positive Funktionswert bedeutet einen gewöhnlichen BASIC-Fehler (zum Beispiel 7 = *Out of memory*).

*Siehe auch*    MatSubx (453).

<b>FUNCTION MatSubx</b>	<b>Matrizenmathematik</b>
-------------------------	---------------------------

*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*. *matrix1* und *matrix2* müssen Arrays mit denselben Dimensionen sein; je nach verwendeter 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 wird zurückgegeben, wenn *matrix1* und *matrix2* nicht dieselbe Dimension haben. Jeder positive Funktionswert bedeutet einen gewöhnlichen BASIC-Fehler (zum Beispiel 7 = *Out of memory*).

**Siehe auch** MatAddx (453).

## FUNCTION MatMultx

## Matrizenmathematik

**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  $m \cdot n$  Elementen, *matrix2* hat  $n \cdot k$  Elemente, und *matrix3* schließlich enthält  $m \cdot k$  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 (zum Beispiel 7 = *Out of memory*).

**Siehe auch** MatInvx (455).

**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 genauso viele 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 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 MATB.BI zwei Konstantendefinitionen, Seps! und Deps#, die bestimmen, ab welcher Größe eine SINGLE-Zahl (Seps!) beziehungsweise 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 MatInvx und MatSEqnx.

**Siehe auch** MatSubx (453).

**FUNCTION MatInvx****Matrizenmathematik**

**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, also ebenso viele Zeilen enthalten, wie sie Spalten hat.

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 MatDetx.

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 (454).

## FUNCTION MatSEqnx

## Matrizenmathematik

**Anwendung**  $x\% = \text{MatSEqnC}(\text{matrix@}(), \text{vektor@}())$   
 $x\% = \text{MatSEqnS}(\text{matrix!}(), \text{vektor!}())$   
 $x\% = \text{MatSEqnD}(\text{matrix\#}(), \text{vektor\#}())$

**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, d.h. es werden scheinbar unsinnige Werte hineingeschrieben.

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 MatDetx.

**Beispiel** Angenommen, Sie haben das folgende Gleichungssystem:

$$\text{I. } 7b - 2a - 8 + 13c = 12$$

$$\text{II. } a + b = 3 - c$$

$$\text{III. } 8b + 15 = 5a$$

Bevor Sie es in MatSEqnx 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:

$$\text{I*} \quad -2a + 7b + 13c = 20$$

$$\text{II*} \quad 1a + 1b + 1c = 3$$

$$\text{III*} \quad -5a + 8b + 0c = -15$$

Jetzt können Sie MatSEqn aufrufen:

```
REM $INCLUDE: 'MATB.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)
```

Ein ganzes Programm, das diese Funktion benutzt, wird im Kapitel 9.1 vorgestellt.

*Siehe auch* MatDetx (455).

# Referenzteil Font-Toolbox

## SUB DefaultFont

## Presentation Graphics

*Anwendung* DefaultFont *segment%, offset%*

*Nutzen* Diese Routine ermittelt die Segment- und die Offsetadresse der internen Schriftart, die in das EXE-File eingebunden wurde (beziehungsweise bei QBX 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 CHRTASM.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 CHRTASM.OBJ in die Font-Quick Library eingebaut wird. Darüberhinaus müssen Sie, wenn Sie ein EXE-File erstellen, beim Linken CHRTASM gesondert aufführen oder in die Font-Libraries einschließen. Näheres siehe im Kapitel 9.2.

*Siehe auch* RegisterMemFont (466).

## SUB GetFontInfo

## Font

*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:

*fontnum* ist 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* ist 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 verschiedene Screen-Modi erleichtern soll (siehe Kapitel 9.2).

*pixwidth* und *pixheight* geben Breite und Höhe eines Buchstabens beziehungsweise Zeichens an; *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* ist 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* schließlich ist der Name der FON-Datei, aus der der Font gelesen wurde, und *facename* der Name des Schrifttyps (bei den mitgelieferten Dateien "Helv", "Tms Rmn" oder "Courier" beziehungsweise "IBM" für die interne Schriftart).

*Bemerkung* » *ascent-leading* ist die Höhe eines Großbuchstabens in Pixeln.

*Siehe auch* `GetRFontInfo` (460), `LoadFont` (462).

## FUNCTION GetGTextLen

Font

**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 Schriftart 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.

» Wenn der Text nicht in normaler Richtung, sondern gedreht ausgegeben wird (SetGTextDir), ist GetGTextLen immer noch die Textlänge vom ersten zum letzten Zeichen, obwohl diese Länge dann auf dem Bildschirm unter Umständen vertikal ins Gewicht fällt.

**Siehe auch** OutGText (465).

## SUB GetMaxFonts

Font

**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 beziehungsweise registriert sind.

» Benutzen Sie SetMaxFonts, um die maximale Zahl der registrier- und ladbaren Fonts zu setzen.

» Wenn Sie SetMaxFonts nicht benutzen, sind maximal 10 Fonts registrier- und ladbar.

**Siehe auch** GetTotalFonts (461), SetMaxFonts (469), RegisterFonts (465), LoadFont (462).

## SUB GetRFontInfo

Font

**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 (458), LoadFont (462).

## SUB GTextWindow

Font

*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 (419), OutGText (465).

## SUB GetTotalFonts

Font

*Anwendung* GetTotalFonts *register%*, *loaded%*

*Nutzen* GetTotalFonts ermittelt, wieviele Schriftarten zum Zeitpunkt des Aufrufs geladen (*loaded%*) und registriert (*register%*) 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 (460), SetMaxFonts (469).

## FUNCTION LoadFont

Font

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. Leerzeichen können sich an beliebiger Stelle befinden; nur beim T-Befehl sind sie relevant.

Eine einzelne Fontbeschreibung wieder setzt sich aus einem oder mehreren Schriftart-Auswahlkriterien - hier kurz Befehle genannt - zusammen. Diese sind:

N <i>nummer</i>	Lädt von allen registrierten Fonts den Font Nr. <i>nummer</i> . 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 <i>punkt</i>	Wählt nur eine Schriftart mit der Größe <i>punkt</i> 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. Siehe Tabelle in Kapitel 9.2.
H <i>höhe</i>	Wählt nur eine Schriftart mit der Schrifthöhe <i>höhe</i> 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 beziehungsweise 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.

(Fortsetzung nächste Seite)

M <i>modus</i>	(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 <i>name</i>	(KEIN Leerzeichen zwischen T und <i>name</i> , auch wenn es im Handbuch anders steht!) Lädt nur einen Font vom Schrifttyp <i>name</i> . <i>name</i> entspricht dem Element <i>facename</i> aus dem Typ FontInfo, der bei GetFontInfo näher erklärt ist. Möglich sind bei den mitgelieferten Schriften "Courier", "Tms Rmn", "Hely" 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 <i>name</i> müssen entweder ein Schrägstrich und die nächste Beschreibung kommen, oder <i>fontbezeichnung</i> \$ muß zu Ende sein). Außerdem müssen Sie für <i>name</i> 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 9.2 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 *fontbezeichnung*\$ zu Ende sein, was zur Folge hat, daß von allen geladenen Fonts immer nur einer mit dem T-Befehl beschrieben worden sein kann.

» 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. Die Daten aller

geladenen Fonts werden in einem Array abgespeichert, so daß Sie den Compiler-Switch /Ah benutzen müssen, wenn Sie mehr als 64 KB an Font-Daten laden möchten.

*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 den-
' selben 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 beziehungsweise 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 (465), RegisterMemFont (466).

*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, wäre *text\$* noch länger gewesen (Der Punkt *x!*, *y!* ist also ein Pixel in Schreibrichtung von der oberen rechten Ecke des letzten geschriebenen Zeichens entfernt).

*Siehe auch*    LoadFont (462), SelectFont (466), GetGTextLen (460), GTextWindow (461), SetGTextColor (468), SetGTextDir (487).

**FUNCTION RegisterFonts****Font**

*Anwendung*    `x% = RegisterFonts(dateiname$)`

*Nutzen*        Registriert alle Schriftarten, die in der angegebenen Datei *dateiname\$* enthalten sind. Die eventuell schon früher registrierten Schriftarten bleiben registriert (im Gegensatz zum Laden von Schriftarten mit LoadFont, das alle bisher geladenen Fonts aus dem Speicher löscht). 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 (466), LoadFont (462).

## FUNCTION RegisterMemFont

Font

*Anwendung*    `x% = RegisterMemFont(segment%, offset%)`

*Nutzen*        RegisterMemfont registriert eine interne Schriftart (eine, die ins EXE-File eingebunden ist). Sie können nur eine einzige Schriftart ins EXE-File einbinden, also ist der Funktionswert von RegisterMemFont, die Anzahl der registrierten Fonts, 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 zu suchen ist.

Um diese beiden Adressen zu ermitteln, können Sie die Routine DefaultFont benutzen.

*Bemerkung*    » Um Mißverständnissen vorzubeugen: Sie können nicht einfach eine der FON-Dateien als interne Schriftart in das EXE-File einbinden. Vielmehr müssen die Daten für die interne Schriftart in CHRTASM.OBJ enthalten sein. Wenn Sie sie unbedingt ändern wollen, müssen Sie die Daten in CHRTASM.ASM ändern - das ist nicht einfach - und diese Datei dann mit einem Makroassembler neu assemblieren.

» Bei der internen Schriftart ist ein Fehler in der Zeichenzuordnung gemacht worden, der bei SetGCharSet beschrieben wird.

*Siehe auch*    DefaultFont (458), SetGCharSet (467).

## SUB SelectFont

Font

*Anwendung*    `SelectFont fontnummer%`

*Nutzen*        Macht einen der geladenen Fonts zum aktuellen. *fontnummer%* ist eine Zahl zwischen 1 und der Anzahl geladener Fonts (wenn *fontnummer%* zu groß ist, wird der zu ladende Font durch eine Modulo-Division der angegebenen *fontnummer%* durch die Anzahl der geladenen Fonts ermittelt).

*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 (462).

*Anwendung* SetGCharSet zeichensatz%

*Nutzen* Microsoft Windows benutzt einen Zeichensatz, der sich etwas 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 Proportional-schriften 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 CHRTASM 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 Presentation Graphics-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.

*Beispiel* Dieses Programm zeigt alle Zeichen einer Schriftart im Windows- und IBM-Zeichensatz an.

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

SCREEN 11

a% = RegisterFonts("HELVE.FON")
a% = LoadFont("n5")

FOR a% = 0 TO 6
  z$ = SPACE$(32)
  FOR b% = 1 TO 32
    MID$(z$, b%, 1)=CHR$((a%+1) * 32+b%-1)
  NEXT
  SetGCharSet cIBMChars
  c% = OutGText(5, a% * 60, "IBM")
  c% = OutGText(120, a% * 60, z$)
```

(Fortsetzung nächste Seite)

(Fortsetzung)

```
SetGCharSet cWindowsChars
c% = OutGText(5, a% * 60 + 25, "Windows")
c% = OutGText(120, a% * 60 + 25, z$)
NEXT
```

Dabei entsteht diese Anzeige:

IBM	!"#\$%&'()*+,-./0123456789;<=>?
Windows	!"#\$%&'()*+,-./0123456789;<=>?
IBM	@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
Windows	@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
IBM	`abcdefghijklmnopqrstuvwxyz{ }~!
Windows	`abcdefghijklmnopqrstuvwxyz{ }~!
IBM	ÇüëääâäçêëëëïïÄÅÉæÆôöðûüÿÖÜç£¥pf
Windows	IIIIIIIIIIIIII'IIIIIIIIIIII
IBM	áíóúñÑ <sup>20</sup> ¿_¬½¼¡«»____!_____
Windows	ıç£¤¥¦§¨©ª«¬®¯°±²³´µ¶·¸¹º»¼½¾¿
IBM	
Windows	ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞß
IBM	_ß_¶__µ_____Ø____±_____°··_n²__
Windows	àáâãäåæçèéêëëïïðñòóôõö÷øùúûüýþÿ

Siehe auch OutGText (465), RegisterMemFont (466).

## SUB SetGTextColor

Font

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 (465).



## SUB SetGTextDir

Font

*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 (465).

## SUB SetMaxFonts

Font

*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 (460).

## SUB UnRegisterFonts

Font

*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 (469), RegisterFonts (465).

# Referenzteil

## Presentation Graphics-Toolbox

### SUB AnalyzeChart

### Presentation Graphics

*Anwendung* `AnalyzeChart env, name$( ), wert!( ), anzahl%`

*Nutzen* AnalyzeChart funktioniert genauso wie Chart, mit dem Unterschied, daß AnalyzeChart keine Grafik auf den Bildschirm zeichnet. AnalyzeChart berechnet nur aufgrund der gegebenen Werte verschiedene Teile der Variable *env* neu, zum Beispiel obere und untere Grenze für die Zahlen an der Achse oder die Größe des Datenfensters.

*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, für die die Berechnungen gemacht 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 AnalyzeChart funktionieren kann, muß ein Grafikmodus mit ChartScreen eingestellt werden.

» Es ist nicht notwendig, die AnalyzeChart-Routine zu benutzen; Chart berechnet die benötigten Daten üblicherweise selbst. Verwenden Sie AnalyzeChart nur, wenn Sie Grafiken nach eigenem Geschmack manipulieren wollen.

» Diese Routine macht es Ihnen leicht, auch kompliziertere Parameter, wie zum Beispiel die Grenzen der Achsenbeschriftung, zu verändern. Sie rufen zunächst DefaultChart auf, lassen sich dann von der Toolbox mit AnalyzeChart einen Vorschlag geben, an dem Sie dann "von Hand" Änderungen machen können, setzen die entsprechenden Auto-Parameter auf cNo (*AutoScale* im AxisType, *AutoSize* im LegendType), damit Chart nicht wieder alles neu berechnet und Ihre Änderungen hin sind, und rufen schließlich die Chart-Routine auf.

*Siehe auch* Chart (472), ChartScreen (475), DefaultChart (476).

*Anwendung* `AnalyzeChartMS env, name$(), wert!(),  
anzahl%, von%, bis%, reihe$()`

*Nutzen* AnalyzeChartMS stellt alle Berechnungen an, die ChartMS vor der Ausgabe der Grafik auch erledigen muß. Allerdings wird hier keine Grafik gezeichnet, sondern nach der Berechnung der Werte aufgehört.

*env* ist eine beschreibende Variable vom Typ ChartEnvironment, in die auch die Ergebnisse der Berechnungen eingetragen werden. Die Beschreibungen der weiteren Parameter können Sie dem Abschnitt über ChartMS entnehmen.

*Bemerkung* » AnalyzeChartMS funktioniert nur, wenn ein mit ChartScreen eingestellter Grafikmodus aktiv ist.

» Über die Einsatzmöglichkeiten der Analyze-Routinen siehe die Bemerkung bei AnalyzeChart.

*Siehe auch* AnalyzeChart (470), ChartMS (473), ChartScreen (475), DefaultChart (476).

*Anwendung* `AnalyzePie env, name$(), wert!(),  
ausrück%(), anzahl%`

*Nutzen* AnalyzePie funktioniert genau wie ChartPie (die Parameter sind identisch); AnalyzePie zeichnet indessen keine Grafik, sondern berechnet nur Teile der Beschreibungsvariable *env* aufgrund der gegebenen Werte neu.

*Bemerkung* » Setzen Sie vor der Benutzung mit ChartScreen einen Grafikmodus.

» Sorgen Sie für die korrekte Initialisierung von *env* mittels DefaultChart.

» Über die Einsatzmöglichkeiten der Analyze-Routinen siehe die Bemerkung bei AnalyzeChart.

*Siehe auch* AnalyzeChart (470), ChartPie (474), ChartScreen (475), DefaultChart (476).

## SUB AnalyzeScatter

## Presentation Graphics

**Anwendung** `AnalyzeScatter env, wertx!(), werty!(), anzahl%`

**Nutzen** AnalyzeScatter errechnet die Details zu den Werten für eine Punktgrafik, ohne sie sofort zu zeichnen. Die Parameter sind wie bei ChartScatter: *env* eine Variable vom Typ ChartEnvironment, die das Aussehen der Grafik bestimmt und die berechneten Werte aufnimmt, *wertx!()* und *werty!()* die Punktkoordinaten und *anzahl%* die Gesamtzahl der Werte.

Details über die Punktgrafik finden Sie bei ChartScatter.

**Bemerkung** » *wertx!()* und *werty!()* sollten als untere Bereichsgrenze 1 haben.  
» Sie müssen zuvor *env* korrekt initialisieren (benutzen Sie DefaultChart) und mit ChartScreen einen Grafikmodus setzen.  
» Über die Verwendungsmöglichkeiten der Analyze-Routinen siehe AnalyzeChart.

**Siehe auch** AnalyzeChart (470), ChartScatter (474), ChartScreen (475), DefaultChart (476).

## SUB AnalyzeScatterMS

## Presentation Graphics

**Anwendung** `AnalyzeScatterMS env, wertx!(), werty!(), anzahl%,  
von%, bis%, reihe$()`

**Nutzen** AnalyzeScatterMS errechnet die detaillierten Angaben zu einer Punktgrafik mit mehreren Reihen, ohne sie zu zeichnen. Die Parameter sind identisch mit denen von ChartScatterMS.

**Bemerkung** » Es gelten dieselben Hinweise wie bei ChartScatterMS.

**Siehe auch** ChartScatterMS (475), AnalyzeScatterMS (470).

## SUB Chart

## Presentation Graphics

**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).

**Beispiel** Siehe Kapitel 9.3.

**Siehe auch** AnalyzeChart (470), ChartMS (473), ChartScreen (475), DefaultChart (476).

## SUB ChartMS

## Presentation Graphics

**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 Type 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.

**Bemerkung** » Auch hierfür muß zuvor ein Grafikmodus mit ChartScreen eingestellt werden.

» Die ChartMS-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).

**Siehe auch** AnalyzeChartMS (470), Chart (472), ChartScreen (475), DefaultChart (476).

## SUB ChartPie

## Presentation Graphics

**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.

*Bemerkung* » Setzen Sie vor der Benutzung mit `ChartScreen` einen Grafikmodus.

» Sorgen Sie für die korrekte Initialisierung von *env* mittels `DefaultChart`.

*Beispiel* Siehe Kapitel 9.3.

*Siehe auch* `AnalyzePie` (471), `Chart` (472), `ChartScreen` (475), `DefaultChart` (476).

## SUB ChartScatter

## Presentation Graphics

*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.

*Bemerkung* » *wertx!( )* und *werty!( )* sollten als untere Bereichsgrenze 1 haben.

» Sie müssen zuvor *env* korrekt initialisieren (benutzen Sie `DefaultChart`) und mit `ChartScreen` einen Grafikmodus setzen.

» Die `ChartScatter`-Routine erkennt fehlende Werte, wenn Sie ihnen die Konstante `cMissingValue` zuordnen. Ein Wert, der als *wertx!* oder als *werty!* die Konstante `cMissingValue` enthält, wird nicht gezeichnet.

*Siehe auch* `AnalyzeScatter` (470), `ChartScreen` (475), `DefaultChart` (476).

**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 "Die Farb- und Musterpalette" in Kapitel 9.3) 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** » Sie müssen zuvor *env* korrekt initialisieren (benutzen Sie DefaultChart) und mit ChartScreen einen Grafikmodus setzen.  
» Die ChartScatterMS-Routine erkennt fehlende Werte, wenn Sie ihnen die Konstante cMissingValue zuordnen. Ein Wert, der als *wertx!* oder als *werty!* die Konstante cMissingValue enthält, wird nicht gezeichnet.  
» Die Felder *wertx!* und *werty!* sollten in der ersten Dimension als untere Grenze 1 haben.

**Siehe auch** ChartScatter (474), AnalyzeScatterMS (472).

**Anwendung** `ChartScreen modus%`

**Nutzen** Setzt einen SCREEN-Modus für nachfolgende Grafikausgaben. Wenn Sie ChartScreen verwenden, wird die Benutzung des SCREEN-Befehls überflüssig, es sei denn, Sie wollen einige der speziellen Optionen (aktive Seite, virtuelle Seite o.ä.) des SCREEN-Befehls benutzen. In diesem Falls 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 Standard-Referenzteil).

*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 (378).

## SUB DefaultChart

## Presentation Graphics

*Anwendung* DefaultChart *env*, *grafiktyp*, *variation*

*Nutzen* Setzt einige Elemente der Beschreibungsvariable *env* (vom Typ ChartEnvironment) entsprechend der angegebenen Parameter *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 zeichnen

*variation* = cNoPercent: Kreissegmente unbeschriftet lassen

*Bemerkung* » cPlain, cLines und cPercent haben den Wert 1; cStacked, cNoLines und cNoPercent sind 2. cBar, cColumn, cLine, cScatter und cPie haben der Reihe nach die Werte 1 bis 5. Natürlich können Sie auch direkt mit diesen Zahlen arbeiten; das Programm ist jedoch wesentlich leichter lesbar, wenn Sie die Konstanten verwenden.



**Anwendung** `GetPaletteDef farbe%(), linie%(),  
muster$(), punkt%(), rand%()`

**Nutzen** Liest den Inhalt der Farb- und Musterpalette der Presentation Graphics-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 Nummern der Zeichen, die zur Punktdarstellung bei Linien- und Punktgrafiken benutzt werden, und in *rand%( )* die Linientypen für Ränder und Gridlinien zu jedem Paletteneintrag zurückgegeben.

Die Linientypen sind als INTEGER-Zahlen angegeben, so, wie man auch beim LINE-Befehl ein Linientyp-Argument benutzt. Die Füllmuster sind Strings, wie man sie für den PAINT-Befehl benötigt.

**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.

» Über Sinn und Verwendungsweise der Palette siehe "Die Farb- und Musterpalette" in Kapitel 9.3.

**Siehe auch** SetPaletteDef (481), ResetPaletteDef (481).

## FUNCTION GetPattern\$

## Presentation Graphics

**Anwendung** `x$ = GetPattern$(bitspropixel%, musternummer%)`

**Nutzen** Eine korrekte Füllmusterdefinition, wie sie zum Beispiel beim PAINT-Befehl benutzt wird, wie sie aber auch 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 beigelegt 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).

*Bemerkung* » Bei *bitspropixel%* = 8 wird ungeachtet der Variable *musternummer%* immer nur ein Muster zurückgegeben.

*Beispiel* Dieses Programm benutzt die Funktionen *GetPattern\$* und *MakeChartPattern\$*, um Füllmuster zu erzeugen, die dann als Argument zum PAINT-Befehl "mißbraucht" werden.

```
'$INCLUDE: 'chrtb.bi'
'$INCLUDE: 'fontb.bi'
DIM Muster AS STRING, GeruestMuster AS STRING
DIM MusterNummer AS INTEGER, Zeile AS INTEGER
DIM Spalte AS INTEGER

' Ohne ChartScreen läuft gar nichts
ChartScreen 11

FOR Zeile = 0 TO 3
  FOR Spalte = 0 TO 3
    MusterNummer = Zeile * 4 + Spalte + 1

    IF MusterNummer <= cPalLen THEN
      ' farbloses "Mustergerüst" holen
      GeruestMuster = GetPattern$(1, MusterNummer)

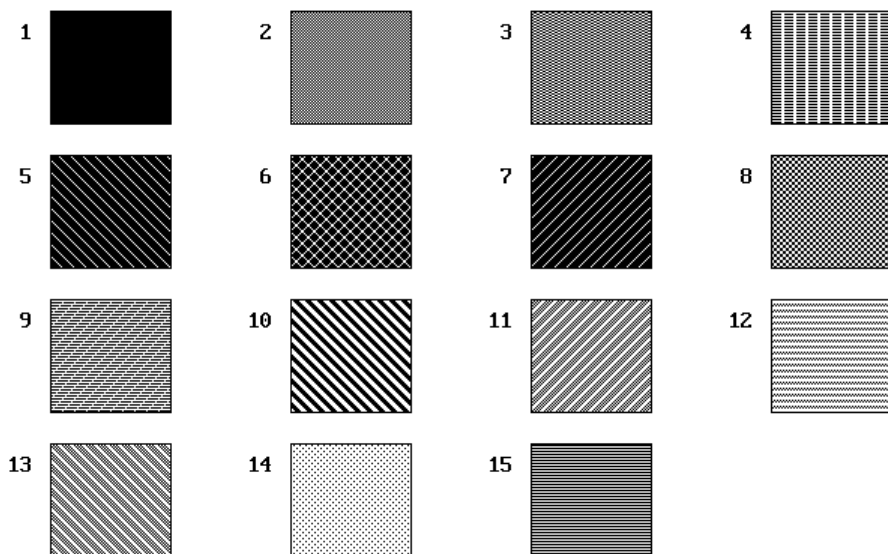
      ' daraus eine gültige Musterbezeichnung basteln
      Muster = MakeChartPattern$(GeruestMuster, 7, 0)

      ' ein Kästchen malen...
      LINE (Spalte * 160 + 60, Zeile * 96 + 15)
        -(Spalte * 160 + 140, Zeile * 96 + 90), 7, B

      ' ... und ausfüllen
      PAINT STEP(-1, -1), Muster, 7

      ' Nummer dranschreiben
      LOCATE Zeile * 6 + 2, Spalte * 20 + 5
      PRINT USING "##"; MusterNummer
    END IF
  NEXT
NEXT
```

Es erzeugt die folgende Mustertabelle (für den SCREEN-Modus 1 mit *bitspropixel%* = 2 ist sie etwas anders).



Siehe auch      `MakeChartPattern$` (480), `PAINT` (354), `GetPaletteDef` (477),  
`SetPaletteDef` (481).

## SUB LabelCharH

## Presentation Graphics

**Anwendung**    `LabelCharH 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 Schriftart, die benutzt werden soll (es muß natürlich eine der geladenen Schriften sein).

*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. Bei ungültigen *font%*-Nummern

wird der erste geladene Font benutzt, und wenn keine geladen ist, wird auf die interne Schriftart zurückgegriffen.

*Bemerkung* » Da der Text mit der Font-Toolbox ausgegeben wird, können Sie die SetGCharSet-Routine benutzen, um den Zeichensatz zu verstellen. Wenn Sie die interne Schriftart verwenden, wird das zur Darstellung von Umlauten sogar unumgänglich sein (siehe dort).

*Siehe auch* LabelChartV (480), SetGCharSet (467).

## SUB LabelChartV

## Presentation Graphics

*Anwendung* LabelChartV *env*, *x%*, *y%*, *font%*, *color%*, *text\$*

*Nutzen* Gibt einen beliebigen Text vertikal entlang einer Grafik aus. Die Benutzung ist ansonsten identisch mit der LabelChartH: *env* ist die übliche ChartEnvironment-Variable, die 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 Schriftart, die benutzt werden soll. *color%* verweist auf den Eintrag in der Farb- und Musterpalette, mittels dessen dem Text eine Farbe zugeordnet werden soll, und *text\$* ist der Text, der ausgegeben wird.

*Bemerkung* » Details siehe bei LabelChartH.

*Siehe auch* LabelChartH (479).

## SUB MakeChartPattern\$

## Presentation Graphics

*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 Presentation Graphics-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, und *vordergrund%* und *hintergrund%* als Vorder- und Hintergrundfarben. Diese Farben sollten natürlich für den Bildschirmmodus, in dem das Muster benutzt wird, gültige Farbattribute sein.

*Bemerkung* » Was MakeChartPattern\$ genau tut, ist folgendes: Eine Musterbeschreibung, die nur Punkt-An- und Punkt-Aus-Informationen enthält, wird in eine Musterbeschreibung überführt, die jedem Punkt eine Farbe zuordnet. Dabei erhalten die "An"-Punkte die Farbe *vordergrund%* und die "Aus"-Punkte die Farbe *hintergrund%*.

*Siehe auch*   `GetPattern$` (477), `SetPaletteDef` (481), `GetPaletteDef` (477).

## Presentation Graphics

*Siehe auch*    [SetPaletteDef](#) (481), [GetPaletteDef](#) (477).

## Presentation Graphics

*Siehe auch* [GetPaletteDef](#) (477), [ResetPaletteDef](#) (481).

# Referenzteil

## User Interface-Toolbox

### Menü-Routinen

FUNCTION MenuCheck	User Interface/Menü
--------------------	---------------------

*Anwendung*    `x% = MenuCheck(funktion%)`

*Nutzen*        Prüft, ob ein Menüpunkt ausgewählt wurde. *funktion%* kann entweder 0, 1 oder 2 sein und bestimmt die Tätigkeit, die die Funktion ausführt:

Wert	Funktion von MenuCheck
0	Prüft, ob seit dem letzten MenuCheck(0)-Aufruf ein Menüpunkt ausgewählt wurde. Wenn ja, wird als Funktionswert die Nummer des Menüs zurückgegeben; wenn nicht, ist der Funktionswert 0. MenuCheck(0) setzt außerdem die globale Variable, die mit MenuCheck(1) abgefragt wird.
1	Kann nur unmittelbar nach einem MenuCheck(0)-Aufruf benutzt werden und gibt die Nummer des Menüpunktes zurück, der gewählt wurde (Die Nummer des Menüs wird ja schon durch MenuCheck(0) ermittelt).
2	Gibt TRUE zurück, wenn seit dem letzten MenuCheck(0)-Aufruf ein Menüpunkt ausgewählt wurde, sonst FALSE. MenuCheck(2) ändert keine Variablen.

*Bemerkung*    » MenuCheck(2) erscheint auf den ersten Blick überflüssig, weil seine Funktion fast der von MenuCheck(0) gleicht. In manchen Fällen ist es jedoch notwendig, zu prüfen, ob ein Menüpunkt gewählt wurde, ohne gleich die Nummer wissen zu wollen - zum Beispiel, um gegebenenfalls in eine Spezialroutine zu verzweigen, die die Menübehandlung übernimmt. Eine Zeile wie

```
IF MenuCheck(0) THEN GOTO Spezialroutine
```

würde hier jedoch nicht weiterhelfen, denn wenn die "Spezialroutine" nun mit MenuCheck(0) abfragen will, welches Menü gewählt wurde, wird das fehlschlagen, da der MenuCheck(0)-Aufruf in der o.g. Zeile schon die Variable zurückgesetzt hat. Für solche Zwecke ist MenuCheck(2) zu benutzen; wenn Sie die 0 in der Zeile durch eine 2 ersetzen, wird die "Spezialroutine" wie geplant funktionieren.

*Beispiel* Siehe Beispiel bei MenuItemToggle.

*Siehe auch* MenuEvent (483), MenuInkey\$ (484), MenuItemToggle (485), ShortCutKeyEvent (490).

## SUB MenuColor

User Interface/Menu

*Anwendung* MenuColor *farben*

*Nutzen* MenuColor setzt die Farben, in denen MenuShow die Menüs darstellt. Als *farben* müssen, durch Komma getrennt, sieben Farbwerte in dieser Reihenfolge angegeben werden (in Klammern jeweils der Default-Wert):

- » Die Vordergrundfarbe für normale Menüeinträge (0)
- » Die Hintergrundfarbe für normale Menüeinträge (7)
- » Die Vordergrundfarbe für den hervorgehobenen Buchstaben im Menüeintrag (15)
- » Die Vordergrundfarbe für Menüeinträge, die im Augenblick nicht wählbar sind (8)
- » Die Vordergrundfarbe für normale Menüeinträge, wenn der Leuchtbalken auf ihnen steht (7)
- » Die Hintergrundfarbe für normale Menüeinträge, wenn der Leuchtbalken auf ihnen steht (0)
- » Die Vordergrundfarbe für den hervorgehobenen Buchstaben in einem Menüeintrag, auf dem der Leuchtbalken steht (15)

Der erlaubte Bereich ist 0 bis 15 für die Vordergrund- und 0 bis 7 für die Hintergrundfarben.

*Bemerkung* » Wenn das Menü schon auf dem Bildschirm angezeigt wird und Sie dann die Farben ändern, müssen Sie MenuShow aufrufen, damit die Änderungen auch sichtbar werden.

*Siehe auch* MenuShow (489).

## SUB MenuEvent

User Interface/Menu

*Anwendung* MenuEvent

*Nutzen* MenuEvent prüft, ob der Benutzer im Augenblick entweder eine ALT-Tastenkombination drückt oder mit der Maus in die oberste Bildschirmzeile klickt. Falls eines von diesen beiden Ereignissen eintritt, ruft MenuEvent eine interne Prozedur auf, die die entsprechenden Menüs öffnet und anzeigt, den Leuchtbalken bewegt usw., bis der Benutzer eine Wahl getroffen hat.

Ob der Benutzer etwas derartiges getan hat oder nicht, kann nicht mit MenuEvent festgestellt werden. Dafür ist MenuCheck da.

*Bemerkung* » MenuEvent ist die einzige Prozedur, die dem Benutzer die Möglichkeit gibt, einen Menüpunkt aufzurufen. Sie muß häufig genug aufgerufen werden, so daß jederzeit ein Menü gewählt werden kann. Würde beispielsweise nur jede Sekunde einmal MenuEvent aufgerufen, dann würde eine ALT-Tastenkombination beziehungsweise eine Maustaste, die der Benutzer zwischen den beiden Aufrufen drückt, ignoriert.

*Siehe auch* MenuCheck (482), MenuInkey\$ (484).

## SUB MenuInit

User Interface/Menu

*Anwendung* MenuInit

*Nutzen* Diese Prozedur muß aufgerufen werden, bevor irgendeine der anderen Menü-Routinen benutzt werden kann. Sie initialisiert einige globale Felder, die die Menü-Daten aufnehmen, und macht mit MouseInit die Maus-Routinen nutzbar.

*Bemerkung* » Ein erneuter Aufruf von MenuInit löscht alle Menü-, Farb- und Shortcut-Definitionen.

*Siehe auch* MouseInit (511), WindowInit (504).

## FUNCTION MenuInkey\$

User Interface/Menu

*Anwendung* x\$ = MenuInkey\$

*Nutzen* MenuInkey\$ funktioniert genau wie die eingebaute Funktion INKEY\$: Sie gibt einen Leerstring zurück, wenn kein Zeichen von der Tastatur kommt (oder im Tastaturpuffer steht), ansonsten liest sie ein Zeichen und gibt dieses zurück.

Im Gegensatz zum Original-INKEY\$ ruft MenuInkey\$ überdies noch MenuEvent und ShortCutKeyEvent auf, beides Routinen, die möglichst häufig aufgerufen werden müssen, um einen reibungslosen Menü-Betrieb zu gewährleisten (siehe die jeweiligen Einträge).

Wenn eine dieser beiden Prozeduren ihrerseits feststellt, daß der Benutzer eine Menü-Wahl trifft, dann gibt MenuInkey\$ die Zeichenkette "menu" zurück. Der String, der gewöhnlich zurückgegeben wird, ist ja maximal 2 Bytes lang, also kann man im Programm leicht auf diese Ausnahme reagieren. Mit der Funktion MenuCheck läßt sich dann prüfen, welche Wahl der Benutzer traf.

*Bemerkung* » In vielen Programmen besteht die Hauptschleife aus einer Routine, die wartet, bis der Benutzer eine bestimmte Taste drückt. Hier muß man, wenn man auf die User-Interface-Toolbox umrüsten will, einfach nur INKEY\$ durch MenuInkey\$ ersetzen, und schon ist



gewährleistet, daß die Menü-Routinen ausreichend schnell auf Benutzeranforderungen reagieren können.

*Beispiel* Siehe Beispiel bei MenuItemToggle.

*Siehe auch* MenuCheck (482), MenuEvent (483), MenuItemToggle (485), ShortCutKeyEvent (490).

## SUB MenuItemToggle

User Interface/Menu

*Anwendung* MenuItemToggle *menue%*, *punkt%*

*Nutzen* Schaltet den Status eines Menüpunkts von 1 (anwählbar beziehungsweise ausgeschaltet) auf 2 (angewählt beziehungsweise eingeschaltet) und zurück. Ein angewählter Menüpunkt wird mit einer Markierung dargestellt.

*menue%* ist die Nummer des Menüs, in dem sich der Menüpunkt befindet, und *punkt%* ist die Nummer des Menüpunkts.

Benutzen Sie diese Routine, wenn Sie in Ihren Menüs einen "Schalter" haben (wie zum Beispiel "Included Lines" in der QBX-Oberfläche), der entweder an oder aus sein kann. Sie brauchen dann nur, wenn der betreffende Menüpunkt ausgewählt wird, MenuItemToggle aufzurufen. Wenn der Menüpunkt "eingeschaltet" war, wird er nun "ausgeschaltet" und umgekehrt.

*Bemerkung* » Verwechseln Sie "ausgeschaltet", eine Bezeichnung, die ich hier für einen ganz gewöhnlichen Menüpunkt verwende, der jederzeit gewählt werden kann, aber im Augenblick keine Markierung hat, nicht mit "nicht wählbar". Nicht wählbare Menüpunkte können, wie der Name schon sagt, weder mit Maus noch mit einer Tastenkombination ausgewählt werden und sind in einer anderen Farbe dargestellt.

» Alles, was Sie mit MenuItemToggle machen können, geht auch mit MenuSetState oder MenuSet - mit MenuItemToggle ist es aber kürzer.

*Beispiel* In dieser Schleife wird laufend MenuInkey\$ aufgerufen, um die Tastatur zu überwachen. Wenn der Benutzer eine Menü-Auswahl trifft, wird mit MenuCheck festgestellt, was er gewählt hat. Die dritte Auswahl im zweiten Menü sei "Automatisch Speichern". Wenn der Benutzer diesen Punkt wählt, der die Funktion eines Schalters haben soll, wird die entsprechende Darstellung auf dem Bildschirm mit MenuItemToggle gemacht, und intern wird die Variable AutoSave von TRUE auf FALSE oder umgekehrt gesetzt.

```

DO
  a$ = MenuInkey$
  IF a$ = "menu" THEN
    MenuNummer = MenuCheck(0)
    MenuPunkt = MenuCheck(1)
    IF MenuNummer = 2 AND MenuPunkt = 3 THEN
      MenuItemToggle 2, 3
      AutoSave = NOT AutoSave
    ELSE
      ' andere Auswahlen verarbeiten
    END IF
  ELSE
    ' andere Tasten verarbeiten
  END IF
LOOP

```

*Siehe auch* MenuSetState (489).

## SUB MenuOff

User Interface/Menu

*Anwendung* MenuOff

*Nutzen* Schaltet die Verarbeitung von Menü-Ereignissen ab. ALT-Tastenkombinationen, Mausklicks in der obersten Bildschirmzeile und Shortcut-Tasten wirken nicht mehr auf die Menüs; MenuCheck(0) ist immer 0, und MenuInkey\$ funktioniert wie ein gewöhnliches INKEY\$.

*Bemerkung* » Die Menüstruktur im Speicher bleibt vollständig erhalten, und auch die Menüleiste, die auf dem Bildschirm steht, wird nicht verändert.

» Benutzen Sie diese Routine, wenn Sie während einer bestimmten Operation das Anwählen eines Menüpunkts unmöglich machen wollen. Solange MenuEvent und ShortCutKeyEvent nicht aufgerufen werden, ist das allerdings ohnehin nicht möglich.

*Siehe auch* MenuOn (486).

## SUB MenuOn

User Interface/Menu

*Anwendung* MenuOn

*Nutzen* Schaltet die Verarbeitung von Menü-Ereignissen wieder ein, nachdem sie mit MenuOff unterbrochen wurde.

*Bemerkung* » MenuOn muß nicht am Anfang des Programms aufgerufen werden; MenuInit schaltet die Menüverarbeitung automatisch ein.

*Siehe auch* MenuOff (486), MenuInit (484).

*Anwendung* MenuPreProcess

*Nutzen* MenuPreProcess führt einige Berechnungen durch, die Arbeitsgrundlage der meisten anderen Menüprozeduren sind. MenuPreProcess muß immer dann aufgerufen werden, wenn Sie grundlegende Änderungen an Ihrer Menüstruktur vorgenommen haben. Das heißt:

» nach jedem MenuSet-Aufruf (wenn mehrere MenuSet-Aufrufe unmittelbar aufeinander folgen, reicht ein MenuPreProcess nach dem letzten)

» nach jedem MenuSetState, wenn damit ein Menüpunkt den Status -1 (nicht angezeigt) erhält oder ein Menüpunkt mit Status -1 einen anderen Status erhält. Auch hier reicht einziger Aufruf nach einer Gruppe von MenuSetState-Befehlen.

MenuPreProcess braucht nicht aufgerufen zu werden, wenn Sie zum Beispiel nur mit MenuSetState einen Menüpunkt auf den Status 0 (nicht wählbar) gesetzt oder mit MenuItemToggle einen Schalter umgeschaltet haben.

*Siehe auch* MenuSet (487), MenuSetState (489).

**SUB MenuSet****User Interface/Menu**

*Anwendung* MenuSet *menue%*, *punkt%*, *status%*, *text\$*, *wahltaste%*

*Nutzen* Setzt (oder verändert) einen Menüpunkt in einem beliebigen Menü. Diese Prozedur wird benutzt, um Menüs zu erstellen.

*menue%* ist die Nummer des Menüs, in dem ein Menüpunkt erstellt werden soll (von 1 bis MaxMenu). *punkt%* ist die Nummer des Menüpunktes innerhalb des Menüs. Menüpunkt 1 erscheint als erster (oberster) im Menü. Menüpunkt 0 ist der Name des Menüs, der in der ersten Bildschirmzeile erscheint.

*status%* ist der Status für den Menüpunkt - 0 steht für nicht wählbar, 1 für wählbar und 2 für gewählt (siehe auch MenuItemToggle). Der Status -1 bedeutet, daß dieser Menüpunkt überhaupt nicht angezeigt wird und demzufolge auch nicht wählbar ist (ein gewöhnlicher nicht wählbarer Menüpunkt wird andersfarbig angezeigt).

*text\$* ist der Text des Menüpunktes beziehungsweise bei *punkt%* = 0 der Name des Menüs. Wenn Sie statt eines Textes einen Bindestrich (-) angeben, wird an der benannten Stelle eine Linie durch das Menü gezogen. Die maximale Länge für Menünamen ist 15, für Menüpunkte ist sie 30 Zeichen.

*wahltaste%* ist eine Zahl, die mindestens 1 und maximal die Länge von *text\$* sein muß; sie gibt an, welcher Buchstabe aus *text\$* her-

vorgehoben dargestellt wird, so daß man damit diesen Menüpunkt auswählen kann (nicht zu verwechseln mit den Shortcut-Tasten).

*Bemerkung* » Der Status -1 kann mit MenuSetState jederzeit auf einen anderen Status geändert werden. Auf diese Weise können Sie in Ihren Menüs Menüpunkte verstecken, die nur in bestimmten Situationen überhaupt sichtbar werden (zum Beispiel: je höher die Zugriffsberechtigung eines Benutzers, desto mehr Menüpunkte sieht er).

» Auch wenn *text\$* ein Bindestrich ist, dürfen Sie niemals in *wahltaste%* eine 0 schreiben - das verträgt die Toolbox nicht.

» Die Toolbox verarbeitet es nicht korrekt, wenn *wahltaste%* auf einen Umlaut weist. Er wird dann zwar hervorgehoben, aber weil die eingebaute UCASE\$-Routine keine kleinen Umlaute in große umwandeln kann, kann es Schwierigkeiten beim Auswählen eines solchen Menüpunkts geben.

» Wenn Sie ungültige Parameter benutzen, kann es Ihnen passieren, daß die Toolbox den Lautsprecher Ihres Computers in nervenzerfetzenden Tonlagen plärren läßt - ein Programmierer bei Microsoft hat offenbar einige SOUND-Befehle in der Toolbox (MENU.BAS) vergessen. Sie sollten diese am besten löschen und die Toolbox neu kompilieren.

*Beispiel* Dieses kleine Programm erzeugt drei Menüs und demonstriert dabei die meisten Möglichkeiten von MenuSet und ShortCutKeySet. Zur Lauffähigkeit fehlen diesem Beispiel allerdings die REM \$INCLUDE-Zeilen und die COMMON- und DIM-Befehle, die zur Benutzung der Menü-Routinen benötigt werden (siehe Kapitel 9.5).

```
MenuInit

MenuSet 1, 0, 1, "Schreibtisch", 1
MenuSet 1, 1, 1, "Information F1", 1
' an Stelle 2 hat das erste Menü eine Linie:
MenuSet 1, 2, 0, "-", 1
MenuSet 1, 3, 1, "Wecker", 1
MenuSet 1, 4, 1, "Kalender", 1
MenuSet 1, 5, 1, "Rechner", 1

MenuSet 2, 0, 1, "Datei", 1
MenuSet 2, 1, 1, "Laden...", 1
MenuSet 2, 2, 1, "Speichern", 1
MenuSet 2, 3, 1, "Speichern als...", 11
MenuSet 2, 4, 1, "Drucken...", 1
MenuSet 2, 5, 0, "-", 1
MenuSet 2, 6, 1, "Exit          CTRL-X", 2

MenuSet 3, 0, 1, "Ändern", 1
MenuSet 3, 1, 1, "Einfügen", 1
```

(Fortsetzung nächste Seite)

(Fortsetzung)

```
MenuSet 3, 2, 1, "Löschen", 1
MenuSet 3, 3, 1, "Verschieben", 1
' Dieser Menüpunkt wird nicht dargestellt:
MenuSet 3, 4, -1, "Alles löschen", 1

' Menü 2, Menüpunkt 6, soll mit CTRL-X erreichbar
sein
ShortCutKeySet 2, 6, CHR$(24)
' Menü 1, Menüpunkt 1, mit F1
ShortCutKeySet 1, 1, CHR$(0) + CHR$(59)

MenuPreProcess
MenuShow
```

*Siehe auch* MenuSetState (489), ShortCutKeySet (490).

## SUB MenuSetState

User Interface/Menu

*Anwendung* MenuSetState *menue%*, *punkt%*, *status%*

*Nutzen* Verändert den Status eines Menüpunkts. Die Parameter, die angegeben werden, gleichen den ersten dreien von MenuSet. Details siehe dort.

*Bemerkung* » Wenn Sie MenuSetState benutzen, um einen Menüpunkt in den Status -1 (nicht angezeigt) oder aus dem Status -1 wieder in einen anderen zu versetzen, müssen Sie danach jeweils MenuPreProcess aufrufen; bei anderen Status-Änderungen ist das nicht notwendig.

» Alles, was Sie mit MenuSetState machen können, geht auch mit MenuSet - mit MenuSetState ist es aber kürzer.

» Auch auf MenuSetState trifft die Bemerkung bezüglich der SOUND-Befehle bei MenuSet zu.

*Siehe auch* MenuSet (487), MenuItemToggle (485).

## SUB MenuShow

User Interface/Menu

*Anwendung* MenuShow

*Nutzen* Zeigt die Menüzzeile (die oberste Bildschirmzeile) an. MenuShow muß mindestens einmal, nach der Definition der Menüpunkte mit MenuSet und der Berechnung mit MenuPreProcess, eingesetzt werden; danach kann es jederzeit aufgerufen werden, wenn beispielsweise der Bildschirm gelöscht wurde, oder wenn Sie mit MenuColor die Menüfarben geändert haben.

*Beispiel* Siehe Beispiel zu MenuSet.

*Siehe auch* MenuSet (487), MenuPreProcess (487).

## SUB ShortCutKeyDelete

User Interface/Menu

*Anwendung* ShortCutKeyDelete *menue%*, *punkt%*

*Nutzen* Löscht den zu einem bestimmten Menüpunkt existierenden Shortcut-Eintrag. *menu%* bestimmt die Nummer des Menüs, *punkt%* die Nummer des Menüpunkts darin.

*Bemerkung* » Zur Wirkung von Shortcut-Tasten siehe ShortCutKeySet.

*Siehe auch* ShortCutKeySet (490), ShortCutKeyEvent (490).

## SUB ShortCutKeyEvent

User Interface/Menu

*Anwendung* ShortCutKeyEvent *taste\$*

*Nutzen* Prüft, ob die angegebene Taste als Shortcut-Taste irgendeines Menüpunktes eingetragen ist. Wenn ja, hat das dieselbe Wirkung, als hätte der Benutzer diesen Menüpunkt mit Tastatur oder Maus auf dem normalen Wege angewählt. Mit der Funktion MenuCheck kann dann abgefragt werden, ob etwas gewählt wurde und, wenn ja, was.

*Bemerkung* » MenuInkey\$ ruft automatisch für jede Taste, die es verarbeitet, ShortCutKeyEvent auf.

*Siehe auch* MenuInkey\$ (484), ShortCutKeySet (490), ShortCutKeyDelete (490).

## SUB ShortCutKeySet

User Interface/Menu

*Anwendung* ShortCutKeySet *menue%*, *punkt%*, *taste\$*

*Nutzen* Ordnet einem bestimmten Menüpunkt eine Shortcut-Taste zu. Es können nur solche Menüpunkte durch eine Shortcut-Taste aufgerufen werden, deren Status größer als 0 ist. Zuordnen können Sie eine Shortcut-Taste allerdings jedem beliebigen Menüpunkt - sogar nicht-existenten oder Linien.

*menue%* und *punkt%* identifizieren den Menüpunkt, dem die Taste zuzuweisen ist, und *taste\$* enthält ein oder zwei Bytes, die genau die Shortcut-Taste identifizieren (Sie müssen als *taste\$* genau das angeben, was INKEY\$ liefert, wenn diese Taste gedrückt wird). Tastenkombinationen mit der ALT-Taste sind nicht als *taste\$* erlaubt.

*Bemerkung* » Maximal 100 Shortcut-Tasten dürfen definiert werden. Wenn Sie für einen Menüpunkt, der schon eine hat, eine zweite Shortcut-Taste definieren, wird die alte überschrieben. Wenn Sie dieselbe Shortcut-Taste mehreren Menüpunkten zuordnen, ist nur die jeweils letzte Zuordnung gültig.

» Sinnvollerweise sollte die Shortcut-Tastendefinition gleichzeitig mit oder kurz nach der Menüdefinition mit MenuSet erfolgen.

*Beispiel* Siehe Beispiel bei MenuSet.

*Siehe auch* MenuSet (487), ShortCutKeyEvent (490), ShortCutKeyDelete (490).

# Window-Routinen

## FUNCTION Alert

## User Interface/Window

**Anwendung** `x% = Alert(layout%, text$, koordinaten, buttons)`

**Nutzen** Die Funktion zeigt ein Fenster mit mindestens einem und maximal drei Buttons auf dem Bildschirm an und wartet so lange, bis der Benutzer einen dieser Buttons betätigt.

Im Fenster wird der Text *text\$* in dem durch *layout%* spezifizierten Stil ausgegeben:

<i>layout%</i>	Stil
1	Linksbündig. Textzeilen, die länger sind als die Fensterbreite, werden abgeschnitten.
2	Linksbündig. Textzeilen, die länger sind als die Fensterbreite, werden auf die nächste Zeile umgebrochen.
3	Linksbündig. Wie Stil 2, jedoch wird hier nur an Leerzeichen umgebrochen.
4	Zentriert. Textzeilen, die länger sind als die Fensterbreite, werden abgeschnitten.

Sie können zwar nur einen *text\$* angeben, aber dieser kann beliebig viele Zeilen enthalten, deren Trennung Sie durch ein | -Zeichen (ASCII 124) kenntlich machen müssen. Jedes | -Zeichen in *text\$* bedeutet den Anfang einer neuen Zeile im Fenster.

*koordinaten* sind die vier Zahlen *y1%*, *x1%*, *y2%*, *x2%* (wie auch beim LOCATE-Befehl wird die y-Koordinate zuerst genannt), die in Bildschirmzeilen beziehungsweise -spalten gemessen werden. Sie geben an, wie groß das Fenster ist und wo es sich befindet. Das Fenster muß mindestens drei Zeilen haben und so breit sein, daß die Buttons nebeneinander Platz haben. Spalten und Zeilen werden (anders als bei LOCATE) von 0 beginnend gezählt; da der Rahmen jedoch um die angegebenen Koordinaten herum gezeichnet wird, ist der kleinste Wert für *x1%* 1 und für *y1%* 2 (wegen der Menüleiste).

*buttons* sind drei Strings *b1\$*, *b2\$* und *b3\$*, die die Texte für die drei Buttons enthalten. Wenn Sie nicht für alle Buttons Text angeben (sondern "" stattdessen), werden entsprechend weniger Buttons erstellt; lassen Sie alle drei Texte leer, dann erstellt die Alert-Funktion automatisch einen "OK"-Button.

Als Funktionswert gibt die Funktion Alert die Nummer des gedrückten Buttons zurück oder 0, wenn ein Fehler auftrat (zum Beispiel, wenn das Fenster zu klein war).



*Bemerkung* » Die Alert-Funktion kann dort eingesetzt werden, wo der Benutzer gezwungenermaßen auf eine Anfrage reagieren muß. Während eine Alert-Box auf dem Schirm ist, kann der Benutzer nichts anderes tun, als einen der Buttons in ihr auszuwählen.

*Beispiel* Das Programm öffnet in der Mitte des Schirms eine Alert-Box mit den zwei Buttons "Äpfel" und "Bananen":

```
text$ = "Entscheiden Sie sich endlich, "  
text$ = text$ + "was Sie kaufen wollen!"  
b1$ = "Äpfel" : b2$ = "Bananen" : b3$ = ""  
  
SELECT CASE Alert(3, text$, 10, 26, 16, 55, b1$,  
b2$, b3$)  
CASE 0  
    PRINT "PROGRAMMFEHLER!"  
CASE 1  
    PRINT "Aha, Äpfel also."  
CASE 2  
    PRINT "So. Bananen. Auch gut!"  
END SELECT
```

*Siehe auch* WindowOpen (505).

## SUB ButtonClose

User Interface/Window

*Anwendung* ButtonClose *buttonnummer%*

*Nutzen* Schließt den Button Nr. *buttonnummer%*. Er wird vom Bildschirm gelöscht. Wenn *buttonnummer% = 0* ist, werden alle Buttons des aktuellen Fensters geschlossen.

Wenn Sie ein Fenster mit WindowClose schließen, werden alle Buttons darin automatisch geschlossen.

*Siehe auch* EditFieldClose (497), ButtonOpen (494), WindowClose (501).

## FUNCTION ButtonInquire

User Interface/Window

*Anwendung* *x%* = ButtonInquire(*buttonnummer%*)

*Nutzen* Gibt den aktuellen Status für den Button Nr. *buttonnummer%* zurück.

Bei den verschiedenen Button-Typen hat der Status verschiedene Bedeutungen. Aus der folgenden Tabelle können Sie das Aussehen aller Button-Typen und die jeweilige Bedeutung des Buttonstatus entnehmen.

Typ	Beschreibung/Status
1	Befehlsbutton <div> <div>&lt; Status 1 &gt;</div> <div> <div>⏏ Status 2 ⏏</div> <div>⏏ Status 3 &gt;</div> </div> </div>
2	Schalterbutton <div> <div>[ ] Status 1</div> <div>[X] Status 2</div> </div>
3	Auswahlbutton <div> <div>( ) Status 1</div> <div>(◆) Status 2</div> </div>
4	Flächenbutton (nicht sichtbar, Status immer 0)
6	<div> <div> <div>↑</div> <div>vertikaler Verschiebepfeil</div> <div> <div>← Status = Markenposition</div> <div>↓</div> </div> </div> </div>
7	<div> <div> <div>←</div> <div>Horizontaler Verschiebepfeil</div> <div>→</div> </div> <div> <div>↑</div> <div>Status = Markenposition</div> </div> </div>

Siehe auch ButtonSetState (495), ButtonOpen (494).

## SUB ButtonOpen

User Interface/Window

**Anwendung** ButtonOpen *nummer%*, *status%*, *text\$*, *groesse*, *typ%*

**Nutzen** Öffnet im aktuellen Fenster einen Button vom Typ *typ%* mit dem Text *text\$* und der Button-Nummer *nummer%*. Der Button erhält den Status *status%*, der allerdings jederzeit wieder verändert werden kann.

Eine Übersicht über die verschiedenen Button-Typen und ihre Stati finden Sie bei ButtonInquire. Der Button-Typ 4 (Flächenbutton) ist für den Benutzer überhaupt nicht sichtbar; es kann lediglich eine bestimmte Reaktion auf das Klicken mit der Maus in einem Flächenbutton programmiert werden.

*text\$* ist nur für die Button-Typen 1, 2 und 3 relevant (in jedem Fall muß das Argument aber angegeben werden).

*groesse* sind die vier Zahlen *y1%*, *x1%*, *y2%*, *x2%*; die ersten beiden geben die obere linke Ecke des Buttons an (relativ zur oberen linken Ecke des Fensters!), die letzten beiden stehen für die untere rechte Ecke, werden aber nur für die Button-Typen 4, 6 und 7 gebraucht.

*Bemerkung* » Der Button-Typ 5 existiert nicht. Diese Nummer ist für späteren Gebrauch reserviert.

» In Fenstern mit variabler Größe können nur Buttons vom Typ 6 oder 7 (Verschiebepalken) geöffnet werden. Achtung! Das Programm bricht mit einer Fehlermeldung ab, wenn Sie versuchen, in einem solchen Fenster einen anderen Button zu öffnen. Vielleicht sollten Sie die Toolbox dahingehend ändern, daß das Programm nicht gleich abbricht.

*Siehe auch* ButtonClose (493), ButtonInquire (493).

## **SUB ButtonSetState**

**User Interface/Window**

*Anwendung* ButtonSetState *nummer%*, *status%*

*Nutzen* Setzt einen neuen Status für einen Button im aktuellen Fenster. *nummer%* ist die Nummer des Buttons (wie bei ButtonOpen angegeben), *status%* ist der neue Status (Liste siehe bei ButtonInquire).

Der neue Status wird sofort entsprechend auf dem Bildschirm dargestellt.

*Siehe auch* ButtonOpen (494), ButtonInquire (493).

## **SUB ButtonToggle**

**User Interface/Window**

*Anwendung* ButtonToggle *nummer%*

*Nutzen* Versetzt den angegebenen Button Nr. *nummer%* in den Status 2, wenn er sich im Status 1 befand, und in den Status 1, wenn er sich im Status 2 befand.

Sie können diese Funktion benutzen, um den Status der Button-Typen 1, 2 oder 3 von "gewählt" auf "nicht gewählt" und umgekehrt zu ändern.

*Bemerkung* » Die Prozedur läßt sich - aufgrund unzureichender Programmierung der Toolbox - auch auf Buttons vom Typ 4, 6 oder 7 anwenden. Das sollte allerdings vermieden werden, da es keine sinnvollen Resultate ergibt.

» Eine Tabelle der verschiedenen Button-Typen und -Status steht bei ButtonInquire.

*Siehe auch* ButtonSetState (495), ButtonOpen (494), ButtonInquire (493).

**Anwendung** `x% = Dialog(funktion%)`

**Nutzen** Hier haben wir es wieder einmal mit einer Universalfunktion zu tun, die so ziemlich jede Information liefern kann, die das Programmiererherz begehrt. Dialog informiert darüber, was der Benutzer während der letzten WindowDo-Schleife getan hat.

Mit dem Parameter *funktion%* teilt man der Funktion mit, was man wissen will.

Bei *funktion% = 0* spuckt Dialog erst einmal aus, was passiert ist:

Ergebnis	Ereignis
0	nichts
1	Ein Button wurde gedrückt; mit Dialog(1) können Sie herausfinden, welcher.
2	Ein Eingabefeld wurde gewählt; mit Dialog(2) können Sie herausfinden, welches.
3	Ein anderes Fenster als das aktuelle wurde mit der Maus angeklickt (das geht nur, wenn das aktuelle Fenster kein exklusives ist); Dialog(3) gibt die Nummer dieses anderen Fensters zurück.
4	Der Benutzer hat das aktuelle Fenster geschlossen (nur bei schließbaren Fenstern).
5	Der Benutzer hat die Größe des aktuellen Fensters verändert (nur bei Fenstern mit variabler Größe). Es muß deshalb eventuell neu erstellt werden. Benutzen Sie WindowCols und WindowRows, um die neue Größe festzustellen.
6	Benutzer hat ENTER gedrückt.
7	Benutzer hat die TAB-Taste gedrückt.
8	Benutzer hat Shift-Tab gedrückt.
9	Benutzer hat ESC gedrückt.
10	Benutzer hat » (Pfeil auf) gedrückt.
11	Benutzer hat » (Pfeil ab) gedrückt.
12	Benutzer hat <- (Pfeil links) gedrückt.
13	Benutzer hat -> (Pfeil rechts) gedrückt.
14	Benutzer hat die Leertaste gedrückt.
15	Das aktuelle Fenster wurde bewegt (nur bei verschiebbaren Fenstern).
16	Benutzer hat Home (Pos 1) gedrückt.
17	Benutzer hat End (Ende) gedrückt.
18	Benutzer hat PgUp (Bild ») gedrückt.
19	Benutzer hat PgDn (Bild ») gedrückt.
20	Ein Menüpunkt wurde ausgewählt (nur, wenn die Menüroutinen in Gebrauch sind und nicht mit MenuOff abgeschaltet wurden). Benutzen Sie MenuCheck, um festzustellen, welcher es war.

Der Wert 0 kommt nur vor, wenn Sie Dialog aufrufen, ohne zuvor WindowDo benutzt zu haben, denn WindowDo wird nicht verlassen, wenn nichts passiert oder wenn Sie ein ungültiges Argument benutzen.

Da Sie die oben aufgeführten Werte sehr häufig in umfangreichen SELECT-CASE-Anweisungen abfragen werden, wenn Sie die Routine benutzen, ist es sinnvoll, Konstanten für diese Zahlen zu definieren.

Nun zu den Spezialfunktionen:

<i><b>funktion%</b></i>	<b>Funktionswert von Dialog</b>
1	Die Nummer des Buttons, der gedrückt wurde. Wenn es ein Button vom Typ 4 war, benutzen Sie Dialog(17) und Dialog(18) für nähere Informationen; bei Buttons vom Typ 6 oder 7 gibt Dialog(19) mehr Details.
2	Die Nummer des Eingabefelds, das gewählt wurde.
3	Die Nummer des anderen Fensters, das gewählt wurde.
17	Die Zeile innerhalb des Flächenbuttons, auf der geklickt wurde beziehungsweise der Cursor steht.
18	Die Spalte innerhalb des Flächenbuttons, auf der geklickt wurde beziehungsweise der Cursor steht.
	Die Funktionen 17 und 18 geben die Klick-Position nur an, wenn in einem Flächenbutton geklickt wurde. Die Positionsangabe ist relativ zur oberen linken Ecke des aktuellen Fensters.
19	(nur, wenn ein Verschiebepalken gewählt wurde) Gibt -2 zurück, wenn auf das "Abwärts"-Symbol eines vertikalen oder das "Rechts"-Symbol eines horizontalen Verschiebepalkens geklickt wurde; -1, wenn "Aufwärts" oder "Links" gewählt wurde, und eine positive Zahl, wenn direkt auf eine Stelle innerhalb des Balkens geklickt wurde. Die positive Zahl ist dann 1 für ganz linke beziehungsweise oberste Position und kann maximal so groß sein, wie der Balken lang (oder hoch) ist.

*Siehe auch* WindowDo (503), ButtonSetState (495).

## **SUB EditFieldClose**

**User Interface/Window**

*Anwendung* EditFieldClose *nummer%*

*Nutzen* Analog zu ButtonClose entfernt diese Prozedur ein Eingabefeld aus dem aktuellen Fenster und dem Speicher. Sein Inhalt geht dabei verloren. *nummer%* ist die Nummer des Eingabefeldes, das entfernt werden soll. Bei *nummer%* = 0 werden alle Eingabefenster aus dem aktuellen Fenster gelöscht.

Wenn Sie ein Fenster mit WindowClose schließen, werden alle Eingabefelder darin automatisch geschlossen.

*Siehe auch* ButtonClose (493), EditFieldOpen (498), WindowClose (501).

## FUNCTION EditFieldInquire\$

User Interface/Window

**Anwendung** `x$ = EditFieldInquire$(nummer%)`

**Nutzen** Gibt - analog zu ButtonInquire - den Text zurück, der in einem bestimmten Eingabefeld des aktuellen Fensters gerade steht. *nummer%* ist die Nummer dieses Feldes.

**Bemerkung** » Ein Eingabefeld umfaßt maximal 255 Zeichen.

**Siehe auch** EditFieldOpen (498), EditFieldClose (497), WindowDo (503).

## SUB EditFieldOpen

User Interface/Window

**Anwendung** `EditFieldOpen nummer%, text$, position,  
farbe, breite%, laenge%`

**Nutzen** Öffnet im aktuellen Fenster unter der Nummer *nummer%* ein Eingabefeld. *text\$* enthält den Text, der anfangs zum Verändern im Eingabefeld bereitstehen soll (nicht länger als *laenge%*).

*position* sind die beiden Zahlen *y%*, *x%*, die Zeile und Spalte für das Eingabefeld (relativ zur oberen linken Fensterecke) angeben.

*farbe* sind die Zahlen *vordergrund%* und *hintergrund%*, die die Farbe des einzugebenden Textes festlegen (*vordergrund%* von 0 bis 15, *hintergrund%* von 0 bis 7). Wenn Sie für beide Werte 0 angeben, wird die normale Textfarbe des aktuellen Fensters benutzt.

*breite%* ist die Breite des Eingabefeldes; *laenge%* ist die maximale Anzahl von Zeichen, die in diesem Eingabefeld eingegeben werden können (darf nicht größer als 255 sein).

**Bemerkung** » Eingabefelder können nicht in Fenstern mit variabler Größe geöffnet werden. Achtung! Wenn Sie versuchen, das zu tun, erscheint unkontrolliert eine Fehlermeldung. Sie sollten vielleicht die Toolbox ändern, so daß das nicht mehr passieren kann.

» Wenn ein Eingabefeld mit gleicher *nummer%* schon existiert, wird dieses gelöscht.

» Die Prozedur EditFieldOpen zeichnet keinen Rahmen um das neue Eingabefeld. Benutzen Sie dazu WindowBox.

**Siehe auch** EditFieldInquire (498), EditFieldClose (497), WindowBox (501).

## FUNCTION ListBox

User Interface/Window

**Anwendung** `x% = ListBox(text$( ), anzahl%)`

**Nutzen** Die Routine zeigt ein Fenster in der Mitte des Bildschirms an, das einen OK- und einen Cancel-Button besitzt. In diesem Fenster werden die ersten Elemente aus dem String-Feld *text\$( )* angezeigt,

Element Nr. 1 mit einem Leuchtbalken. Der Benutzer kann mit der Maus oder den Pfeiltasten den Leuchtbalken und den Ausschnitt, den er sieht, verschieben. *anzahl%* ist die Gesamtanzahl der Elemente in *text\$( )*; ListBox geht davon aus, daß das erste Element in *text\$( )* den Index 1 hat.

Solange das ListBox-Fenster auf dem Bildschirm ist, kann der Benutzer nichts anderes tun, als in der Liste "herumzulaufen" oder "OK" oder "Cancel" zu wählen.

Der Funktionswert der ListBox-Routine ist 0, wenn der Benutzer "Cancel" gewählt hat, und anderenfalls die Nummer des Elements aus *text\$( )*, das er wählte.

Die maximale Breite des dargestellten Textes ist 15 Zeichen. (Siehe aber Bemerkung hierzu.)

*Bemerkung* » Sie sollten, um die Funktion flexibler zu machen, auf jeden Fall dem Rat der Toolbox-Programmierer folgen und die WINDOW.BI-Datei sowie die Toolbox selbst (WINDOW.BAS) ändern, so daß man die Breite dieser List-Box beliebig wählen kann. Dazu müssen Sie das Apostroph vor der Zeile

```
'FUNCTION ListBox (text$( ), MaxRec, BoxWidth)
```

in WINDOW.BAS löschen und statt dessen vor die Zeile

```
FUNCTION ListBox (text$( ), MaxRec)
```

ein Apostroph setzen; außerdem muß die Zeile

```
BoxWidth = 14
```

gelöscht und die DECLARE FUNCTION-Anweisung in WINDOW.BI entsprechend der neuen Funktionsdefinition geändert werden. Dann ändert sich der Aufruf dieser Funktion in *x% = ListBox(text\$( ), anzahl%, breite%)*, und Sie können die Breite der List-Box in den Grenzen 14 bis 55 beliebig festlegen.

» Wenn Sie ohnehin schon beim Ändern der Routine sind: Fügen Sie vor die WindowOpen-Zeile am Anfang der Prozedur noch ein:

```
IF LBOUND(text$( )) < 1 THEN title$ = text$(0)
```

und schreiben Sie als letztes in der WindowOpen-Zeile statt "" besser *title\$*. Durch diese Änderung ist es nun möglich, als 0tes Element des *text\$( )*-Arrays eine Überschrift für die List-Box zu übergeben.

*Beispiel* Das folgende Programmfragment benutzt die in Kapitel 16.4 vorgestellte Directory-Routine, um mittels der List-Box eine Datei auswählen zu lassen.

```

DIM DateiName(0 TO 999) AS STRING
DIM DateiAnzahl AS INTEGER, Auswahl AS INTEGER

Directory "*.*", DateiName, DateiAnzahl
' diese Zeile hat nur Sinn, wenn Sie die unter
' "Bemerkungen" vorgeschlagene zweite Änderung
' an der ListBox-Routine gemacht haben:
DateiName(0) = "Dateiliste"

Auswahl = ListBox(DateiName(), DateiAnzahl)
IF Auswahl = 0 THEN
    PRINT "Na gut, dann eben nichts von allem!"
ELSE
    PRINT "Und sie haben wirklich ";
    PRINT DateiName(Auswahl); " gewählt?"
END IF

```

Siehe auch WindowOpen (505).

## FUNCTION MaxScrollLength

User Interface/Window

*Anwendung* `x% = MaxScrollLength(buttonnummer%)`

*Nutzen* Gibt die Anzahl möglicher Positionen der Marke in einem horizontalen oder vertikalen Verschieberegler (Button-Typen 6 und 7) zurück. Die Markenposition wird bei ButtonOpen und bei ButtonSetState gesetzt; sie ist mindestens 1 (ganz oben beziehungsweise ganz links) und höchstens Buttonbreite (beziehungsweise -höhe) - 2 (an beiden Enden des Balkens steht ein Pfeil). Dieser Maximalwert, Buttonbreite - 2, wird von der Funktion MaxScrollLength zurückgegeben. Als Argument übergibt man ihr *buttonnummer%*, die Nummer des Buttons im aktuellen Fenster, dessen maximale Markenposition man abfragen will.

Wenn die angegebene *buttonnummer%* nicht die eines Buttons vom Typ 6 oder 7 ist, gibt MaxScrollLength% den Wert 0 zurück.

Siehe auch ButtonOpen (494), ButtonSetState (495).

## SUB WindowBox

User Interface/Window

*Anwendung* `WindowBox groesse`

*Nutzen* Zeichnet innerhalb des aktuellen Fensters einen Rahmen (einfache Umrandung).

*groesse* sind die vier Zahlen *y1%*, *x1%*, *y2%*, *x2%*, die - relativ zur oberen linken Ecke des aktuellen Fensters - die Größe des Kästchens angeben. Das Kästchen wird nicht um die angegebene Fläche,



sondern auf ihrem Rand gezeichnet; um eine einzelne Zeile zu umranden, muß also y2% um 2 größer sein als y1%.

*Bemerkung* » WindowBox ist fast identisch mit der Routine Box aus dem General-Teil der User Interface-Toolbox. Der Unterschied ist, daß WindowBox relativ zum aktuellen Fenster (und in dessen Farben) zeichnet, während Box absolute Koordinaten benutzt und etwas flexibler ist (Rahmenart etc. frei wählbar).

*Siehe auch* WindowLine (504), Box (513).

## SUB WindowClose

User Interface/Window

*Anwendung* WindowClose *nummer%*

*Nutzen* Schließt ein beliebiges Fenster. Alle Buttons und Eingabefelder, die sich in diesem Fenster befinden, werden ebenfalls geschlossen.  
*nummer%* ist die Nummer des zu schließenden Fensters; wenn Sie für *nummer%* 0 angeben, werden sämtliche Fenster geschlossen.

*Bemerkung* » Wenn Sie mit dieser Funktion das aktuelle Fenster schließen, ist das aktuelle Fenster nach dem Aufruf dieser Funktion das, welches zuletzt aktuelles Fenster war (also "oben" auf dem Fensterstapel sitzt).

*Siehe auch* WindowOpen (505), ButtonClose (493), EditFieldClose (497).

## SUB WindowCls

User Interface/Window

*Anwendung* WindowCls

*Nutzen* Leert das aktuelle Fenster. Die Hintergrundfarbe wird die, die als Fensterhintergrund bei WindowOpen angegeben wurde.

*Bemerkung* » Benutzen Sie WindowCls nicht, solange Buttons oder Eingabefelder im Fenster sichtbar sind. Diese Buttons und Eingabefelder sind nach WindowCls zwar vom Bildschirm verschwunden, aber immer noch unsichtbar aktiv, was zu unerwünschten Resultaten führen kann.

» Sollte es dennoch einmal nötig sein, kann die Prozedur ButtonShow (als Argument die Nummer des Buttons angeben) benutzt werden, um einen verschwundenen Button wieder ans Licht zu bringen. Diese Funktion ist hier jedoch nicht weiter dokumentiert, weil es sich bei ihr um eine interne Routine handelt.

*Siehe auch* WindowPrint (506), WindowOpen (506).

*Anwendung*    WindowColor *vordergrund%*, *hintergrund%*

*Nutzen*        Ändert die Textfarben eines Fensters, die bei WindowOpen angegeben wurden.

*vordergrund%* darf von 0 bis 15 reichen, *hintergrund%* von 0 bis 7.

*Bemerkung*    » Sie erzielen eine ungestörte Optik, wenn Sie den Texthintergrund immer gleich wie den Fensterhintergrund (siehe WindowOpen) wählen.

» Wenn Sie die theoretische Farbenvielfalt eines Fensters (Fenstervor- und -hintergrund, Textvor- und -hintergrund, hervorgehoben) voll ausnutzen, sieht das Fenster zumeist unangenehm grell aus.

*Siehe auch*    WindowOpen (506), WindowCls (501).

**FUNCTION WindowCols****User Interface/Window**

*Anwendung*    *x%* = WindowCols(*fensternummer%*)

*Nutzen*        WindowCols gibt die Breite des Fensters *fensternummer%* zurück (wie bei WindowOpen angegeben). Die Breite ist zugleich die Anzahl der beschreibbaren Spalten, da der Rahmen um das Fenster gezeichnet wird.

Die Verwendung von WindowCols ist nur bei Fenstern sinnvoll, deren Größe der Benutzer verändern kann, da sie bei allen anderen Fenstern ja immer so bleibt, wie sie beim WindowOpen-Befehl angegeben wurde.

*Bemerkung*    » Die Breite des aktuellen Fensters ermitteln Sie unter Zuhilfenahme der Funktion WindowCurrent mit WindowCols(WindowCurrent).

*Siehe auch*    WindowRows (507), WindowCurrent (503), WindowOpen (506).

*Anwendung*    `x% = WindowCurrent`

*Nutzen*        Gibt die Nummer des aktuellen Fensters zurück. Viele Funktionen beziehen sich immer nur auf das aktuelle Fenster, bei einigen muß jedoch auch die Fensternummer angegeben werden, so daß man dort diese Funktion benutzen muß, um sie auf das aktuelle Fenster anzuwenden.

Optisch unterscheidet sich das aktuelle Fenster von den anderen zum einen dadurch, daß es als oberstes auf dem Bildschirm "liegt", also das einzige Fenster ist, das in jedem Falle vollständig sichtbar ist, und zum anderen durch einen Schatten.

Wenn WindowCurrent den Wert 0 zurückgibt, bedeutet das, daß kein Fenster geöffnet ist.

*Bemerkung*    » Mit WindowSetCurrent, WindowOpen und WindowClose kann ein anderes Fenster zum aktuellen Fenster werden.

*Siehe auch*    WindowSetCurrent (508), WindowOpen (505), WindowClose (501).

**SUB WindowDo****User Interface/Window**

*Anwendung*    `WindowDo button%, eingabefeld%`

*Nutzen*        WindowDo ist die Routine, die die Kontrolle über das gesamte Fenstersystem ausübt. Wenn Sie WindowDo aufrufen, läßt sie den Benutzer so lange im aktuellen Fenster herumspielen, bis irgend-etwas Wichtiges passiert. Als "irgendetwas Wichtiges" versteht WindowDo zum Beispiel diverse (aber nicht alle) Tasten, die der Benutzer drückt, das Auswählen eines anderen Fensters oder eines Menüpunkts oder das Schließen eines Fensters.

Welches Ereignis dafür sorgte, daß WindowDo verlassen wurde, kann danach mit der Funktion Dialog abgefragt werden, und dort finden Sie auch eine komplette Liste aller Ereignisse, die unter die Definition "irgendetwas Wichtiges" fallen.

Die WindowDo-Prozedur hat zwei Argumente, *eingabefeld%* und *button%*. Ist *eingabefeld%* größer als 0, dann setzt WindowDo den Cursor auf das Eingabefeld mit der angegebenen Nummer und läßt den Benutzer den Text dort edieren. Ist *eingabefeld%* gleich 0, aber *button%* enthält eine Zahl, so wird der Cursor auf den Button Nr. *button%* im aktuellen Fenster gesetzt. Wenn beide Variablen 0 sind, gibt es keinen Cursor.

*Bemerkung* » Wenn der Benutzer ein Fenster verschiebt, kümmert sich WindowDo selbst darum, die Darstellung zu aktualisieren. Verändert der Benutzer die Größe, dann muß das Fenster außerhalb von WindowDo neu beschrieben werden. Wenn der Benutzer ein anderes Fenster wählt oder das aktuelle schließt, unternimmt WindowDo nichts - das Programm muß dann selbst mit Dialog abfragen, was passiert ist, und mit entsprechenden Prozeduraufrufen reagieren.

*Siehe auch* WindowOpen (505), WindowClose (501).

## **SUB WindowInit**

**User Interface/Window**

*Anwendung* WindowInit

*Nutzen* Initialisiert einige globale Arrays etc.; WindowInit muß aufgerufen werden, bevor irgendeine der Routinen aus dem Window-Bereich der User Interface-Toolbox benutzt wird.

Ein späterer Aufruf von WindowInit schließt alle Fenster, Buttons und Eingabefelder.

*Siehe auch* WindowClose (501).

## **SUB WindowLine**

**User Interface/Window**

*Anwendung* WindowLine zeile%

*Nutzen* Zeichnet eine durchgehende Linie durch das aktuelle Fenster auf Zeile zeile% (relativ zur oberen linken Fenster-Ecke).

*Bemerkung* » Wenn Sie in einem Fenster, in das mit WindowLine Linien gezeichnet wurden, ein WindowCls ausführen, bleibt von der Linie links und rechts ein häßlicher Rest. Um diesen loszuwerden, müssen Sie den Rahmen um das Fenster neu zeichnen oder das Fenster schließen und wieder öffnen. Wenn Sie die Linie über ein WindowCls hinweg behalten wollen, muß sie sofort nach WindowCls neu gezeichnet werden.

*Siehe auch* WindowBox (501), WindowCls (501).

## **SUB WindowLocate**

**User Interface/Window**

*Anwendung* WindowLocate y%, x%

*Nutzen* Setzt den Fenster-Cursor im aktuellen Fenster auf die Zeile y%, Spalte x% (relativ zu oberer linker Fensterecke). Dieser Cursor ist nicht sichtbar, sondern nur für den WindowPrint-Befehl relevant.

*Siehe auch* WindowPrint (506).

**Anwendung** *x%* = WindowNext

**Nutzen** Beim Öffnen eines Fensters mit WindowOpen müssen Sie eine Fensternummer angeben, die noch nicht von einem anderen Fenster benutzt wird. In manchen Situationen ist es jedoch schwer oder überhaupt nicht festzustellen, welche Nummern noch frei sind. Hier hilft die Funktion WindowNext, die (etwa vergleichbar mit FREEFILE für Dateinummern) die Nummer der kleinsten noch unbelegten Fensternummer zurückgibt. Wenn alle Fensternummern belegt sind, ist der Funktionswert von WindowNext 0. Dann kann kein weiteres Fenster geöffnet werden, bevor nicht eines geschlossen wurde.

**Bemerkung** » Falls die maximale Fensterzahl (im Originalzustand 10) für Ihr Programm nicht ausreicht, können Sie die Konstante MaxWindow in der Datei GENERAL.BI erhöhen.

**Siehe auch** WindowOpen (505), WindowClose (501).

## **SUB WindowOpen**

## **User Interface/Window**

**Anwendung** WindowOpen *nummer%*, *groesse*, *farben*, *typ*, *titel\$*

**Nutzen** Öffnet ein Fenster auf dem Bildschirm und macht es zum aktuellen Fenster. WindowOpen benutzt - auch wenn es in der Beschreibung oben auf den ersten Blick anders aussieht - 16 Parameter:

*nummer%* ist die Nummer für das zu öffnende Fenster; wenn unter dieser Nummer bereits ein Fenster existiert, wird es zuvor geschlossen.

*groesse* steht für die vier Zahlen *y1%*, *x1%*, *y2%* und *x2%*, die die obere linke und die untere rechte Ecke des Fensters in Zeilen und Spalten angeben. Der Rahmen wird hier um das Fenster herum gezeichnet, so daß die angegebene Fläche die wirklich beschreibbare ist. *y1%* muß größer als MinRow, *x1%* größer als MinCol, *y2%* keiner als MaxRow und *x2%* kleiner als MaxCol sein; diese Konstanten sind in GENERAL.BI als 2, 1, 25 und 80 definiert.

*farben* steht für fünf Farbangaben (Datentyp INTEGER) in dieser Reihenfolge:

- die Vordergrundfarbe für Text im Fenster;
- die Hintergrundfarbe für Text im Fenster;
- die Vordergrundfarbe für das Menü (Rahmen etc.);
- die Hintergrundfarbe für das Menü;
- die Vordergrundfarbe für hervorgehobene Buchstaben im Text.

Die Vordergrundfarben können im Bereich 0 bis 15, die Hintergrundfarben von 0 bis 7 gewählt werden. Mit WindowColor können die Textfarben nachträglich noch verändert werden; siehe auch die Bemerkung dort.

*typ* ist ebenfalls ein Sammelsurium von 5 INTEGER-Parametern. Die ersten vier können entweder TRUE (für ja) oder FALSE (für nein) sein und geben an:

» ob das Fenster verschiebbar ist (solche Fenster erhalten als obere Begrenzung keine Linie, sondern ein Muster, das es erlaubt, das Fenster mit der Maus zu verschieben);

» ob das Fenster vom Benutzer geschlossen werden kann (durch Klicken auf das "+"-Symbol in der oberen linken Ecke);

» ob das Fenster eine veränderbare Größe hat (Klicken auf rechte untere Ecke und Bewegen der Maus);

» ob das Fenster ein exklusives Fenster ist (Solange ein exklusives Fenster auf dem Schirm ist, kann der Benutzer nichts außerhalb dieses Fensters anwählen, also weder andere Fenster noch Menüs. Das Alert- und das ListBox-Fenster sind Fenster dieser Art.)

Als fünfter Bestandteil von *typ* wird eine Zahl angegeben, die die Rahmenart bestimmt: 0 für keinen Rahmen (Leerzeichen), 1 für eine einfache und 2 für eine doppelte Linie.

Schließlich folgt die Überschrift für das Fenster, die in der Mitte der oberen Begrenzungslinie eingeblendet wird: *titel\$*. *titel\$* kann leer sein, dann wird kein Titel angezeigt.

*Bemerkung* » Wenn Sie keine noch freie Zahl für *nummer%* wissen, können Sie die Funktion WindowNext benutzen.

*Siehe auch* WindowNext (505), WindowClose (501), Alert (492), ListBox (499), WindowColor (502), WindowSetCurrent (508).

## SUB WindowPrint

User Interface/Window

*Anwendung* WindowPrint *ausgabeart%*, *text\$*

*Nutzen* Gibt einen beliebigen Text (*text\$*) im aktuellen Fenster aus. In einem frisch geöffneten oder mit WindowCls gelöschten Fenster ist die Ausgabeposition Zeile 1, Spalte 1; ansonsten gilt die mit WindowLocate festgesetzte Position oder die, auf die die letzte WindowPrint-Anweisung den Cursor gesetzt hat.

Für *ausgabeart%* gibt es sieben Möglichkeiten:

<b><i>ausgabeart%</i> Verhalten von WindowPrint</b>	
1	Linksbündige Ausgabe. Der Text wird einfach abgeschnitten, wenn er zu lang für die Zeile ist. Der Cursor wird an den Anfang der nächsten Zeile gesetzt; notfalls wird der Fensterinhalt um eine Zeile aufwärts geschoben.
2	Linksbündige Ausgabe. Text, der über den rechten Fensterrand hinausragt, wird in die nächste Zeile umgebrochen. Der Cursor wird auf den Anfang der nächsten Zeile gesetzt; notfalls wird der Fensterinhalt verschoben.
3	Linksbündige Ausgabe. Wie 2, jedoch wird der Text hier nicht brutal umgebrochen, sondern nur an Leerzeichen.
4	Zentrierte Ausgabe auf der aktuellen Zeile. Sonst wie 1.
-1, -2, -3	Identisch mit ihren positiven Äquivalenten, jedoch wird hier der Cursor direkt hinter den ausgegebenen Text gesetzt, so daß die Ausgabe von weiteren Zeichen möglich ist.

*Bemerkung* » Wenn in Ihrem Fenster Buttons, Eingabefelder oder Linien vorkommen, müssen Sie vorsichtig sein: Wenn Sie zuviel Text ausgeben, wird der gesamte Fensterinhalt nach oben verschoben. Die Linien hinterlassen dabei links und rechts häßliche Ränder, und die Buttons werden nicht mehr dort angezeigt, wo mit der Maus geklickt werden muß, um sie zu wählen.

*Siehe auch* WindowCls (501), WindowLocate (504).

<b>FUNCTION WindowRows</b>	<b>User Interface/Window</b>
----------------------------	------------------------------

*Anwendung* *x%* = WindowRows(*fensternummer%*)

*Nutzen* WindowRows gibt die Höhe des Fensters *fensternummer%* zurück (wie bei WindowOpen angegeben). Die Höhe ist zugleich die Anzahl der beschreibbaren Zeilen, da der Rahmen um das Fenster gezeichnet wird.

Die Verwendung von WindowRows ist nur bei Fenstern sinnvoll, deren Größe der Benutzer verändern kann, da sie bei allen anderen Fenstern ja immer so bleibt, wie sie beim WindowOpen-Befehl angegeben wurde.

*Bemerkung* » Die Höhe des aktuellen Fensters ermitteln Sie mit Hilfe der Funktion WindowCurrent mit WindowRows (WindowCurrent).

*Siehe auch* WindowCols (502), WindowCurrent (503), WindowOpen (505).

*Anwendung* WindowScroll *zeilen%*

*Nutzen* Verschiebt den Inhalt des aktuellen Fensters um *zeilen%* Zeilen nach oben (bei positiven Werten) oder nach unten (bei negativen Werten). Die sich dadurch ergebenden leeren Zeilen werden mit der Fensterhintergrundfarbe (siehe WindowOpen) gefüllt.

Bei *zeilen%* = 0 verhält sich WindowScroll identisch mit WindowCls: Es löscht den gesamten Fensterinhalt des aktuellen Fensters.

*Bemerkung* » Die Prozedur Scroll aus dem General-Teil der User Interface-Toolbox erlaubt flexibleres Scrollen (Verschieben); wenn Sie es nur auf Fenster anwenden wollen, ist WindowScroll allerdings einfacher zu benutzen.

*Siehe auch* WindowCls (501), Scroll (517).

*Anwendung* WindowSetCurrent *fensternummer%*

*Nutzen* Macht das Fenster Nr. *fensternummer%* zum aktuellen Fenster. Es wird in der Bildschirmdarstellung dadurch über alle anderen Fenster gelegt, so daß es - unter Umständen als einziges - vollständig sichtbar wird. Das Fenster bekommt einen Schatten.

*Siehe auch* WindowCurrent (503), WindowOpen (505).



# Maus-Routinen

## SUB MouseBorder

User Interface/Mouse

*Anwendung* MouseBorder y1%, x1%, y2%, x2%

*Nutzen* Schränkt den Maus-Bewegungsbereich ein. Nach Ausführung dieses Befehls, dem die Koordinaten als Paare aus Zeilen- und Spaltennummern (wie bei LOCATE) übergeben werden, kann der Mauszeiger nur noch in dem angegebenen Bereich bewegt werden.

*Bemerkung* » Gehen Sie mit diesem Befehl sparsam um. Zwar ist es reizvoll, den Benutzer die Maus immer nur dorthin bewegen zu lassen, wo ein Klick auch sinnvoll ist - aber der Benutzer wird irritiert, wenn der Mauszeiger plötzlich an einer unsichtbaren Grenze auf dem Schirm "hängenbleibt".

» Dieser Befehl entspricht der Verwendung der Funktionen 7 und 8 des Maus-Interrupts (Interrupt 33h); siehe Anhang F.

*Siehe auch* MouseInit (501).

## SUB MouseDriver

User Interface/Mouse

*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 ist TRUE, 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.

» Der Anhang F gibt eine Übersicht über eine große Anzahl von Mausfunktionen, die Sie alle mittels der MouseDriver-Prozedur oder mit einem direkten INTERRUPT-Aufruf an den Mausinterrupt 33h aktivieren können.

*Siehe auch* INTERRUPT (205).

*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 in 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.

*Beispiel* (zum Mauszeiger-Zähler):

```
SUB DateiListe
  MouseHide 'Maus-Zähler geht auf -1
  PRINT "Folgende Dateien sind vorhanden:"
  AnzeigeDateiListe
  PRINT "Bitte drücken Sie eine Taste!"
  MouseShow 'Maus-Zähler auf 0
  SLEEP
END SUB

SUB AnzeigeDateiListe
  MouseHide 'Maus-Zähler auf -2
  ' jetzt wird die Liste angezeigt...
  MouseShow 'Maus-Zähler auf -1
END SUB
```

AnzeigeDateiListe ist ein unabhängiges SUB, das auch aus anderen Routinen aufgerufen werden kann. Es schaltet selbst den Maus-Cursor ab. Wenn es aber aus DateiListe aufgerufen wird, macht es nichts, daß ein MouseShow-Befehl ausgeführt wird, denn die Maus wird erst nach dem zweiten MouseShow-Befehl wieder angezeigt, wenn der Maus-Zähler wieder auf 0 steht.

*Siehe auch* MouseInit (511), MouseShow (511).

*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 (510), MouseShow (511).

**SUB MousePoll****User Interface/Mouse**

*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 (wie bei LOCATE); *links%* und *rechts%* - entweder TRUE oder FALSE - geben an, ob die entsprechenden Mausknöpfe gedrückt sind.

*Bemerkung* » Wenn Sie die Funktion 3 des Maustreiber-Interrupts, die auch von MousePoll benutzt wird, direkt aufrufen (entweder mit INTERRUPT oder mit MouseDriver), können Sie erstens auch den mittleren Mausknopf (falls vorhanden) abfragen und zweitens (für den Grafikmodus) die Mauskoordinaten pixel- und nicht zeilenbeziehungsweise spaltenweise erhalten. Mehr dazu im Anhang F.

*Siehe auch* MouseDriver (509), MouseInit (511).

**SUB MouseShow****User Interface/Mouse**

*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.

*Beispiel* Siehe Beispiel zu MouseHide.

*Siehe auch* MouseHide (510), MouseInit (511).

# General-Routinen

## FUNCTION AltToASCII\$

User Interface/General

*Anwendung*    `x$ = AltToASCII$(taste$)`

*Nutzen*        Gibt den reinen Tastencode einer ALT-Tastenkombination zurück. *taste\$* ist ein zwei Zeichen langer String, wie er von INKEY\$ zurückgegeben wird, wenn der Benutzer ALT und irgendeine andere Taste betätigt (siehe auch Zeichencode-Tabelle in Anhang D.2).

Falls es sich dabei um den Code für ALT und einen Buchstaben (keine Umlaute), eine Zahl, das Minus- oder das Gleichheitszeichen handelt, gibt AltToASCII\$ dieses ursprüngliche Zeichen (als Großbuchstaben) zurück. Falls *taste\$* keine der genannten Kombinationen ist, ist der Funktionswert von AltToASCII\$ ein Leerstring ("").

*Beispiel*       Dieses Programmfragment zeigt eine denkbare Tastenverarbeitungs-routine mit AltToASCII\$:

```
REPEAT
  a$ = INKEY$
UNTIL a$ <> ""
IF LEN(a$) = 2 THEN
  SELECT CASE AltToASCII$(a$)
    CASE ""
      ' Behandlung aller anderen Zwei-Zeichen-
      ' Codes, zum Beispiel Shift-Tab oder
      ' Pfeiltasten
    CASE "A"
      ' Behandlung von ALT A
    CASE "X"
      ' Behandlung von ALT X
    CASE "1"
      ' Behandlung von ALT 1
  END SELECT
ELSE
  ' Behandlung von normalen Zeichen
END IF
```

*Siehe auch*    INKEY\$ (322), MenuInkey\$ (484).

## SUB AttrBox

User Interface/General

*Anwendung* AttrBox y1%, x1%, y2%, x2%, farbattribut%

*Nutzen* Ändert die Farbe aller Zeichen innerhalb des durch die beiden Koordinatenpaare y1%, x1% und y2%, x2% (beide Zeilen- und Spaltenangaben wie bei LOCATE) spezifizierten Bereichs auf den angegebenen Farbcode.

*Bemerkung* » Der Farbcode enthält gleichzeitig Informationen über Vorder- und Hintergrundfarbe. Die folgenden Routinen berechnen aus zwei Farben den Farbcode und umgekehrt:

```
FUNCTION FarbCode% (VG AS INTEGER, HG AS INTEGER)
    FarbCode = (VG MOD 16) + 16 * HG - 128 * (VG > 15)
END FUNCTION
```

```
SUB FarbeAusCode (VG AS INTEGER, HG AS INTEGER,
FarbCode AS INTEGER)
    IF FarbCode > 127 THEN VG = 16 ELSE VG = 0
    VG = VG + FarbCode MOD 16
    HG = INT((FarbCode MOD 128) / 16)
END SUB
```

*Siehe auch* SCREEN (Funktion) (377).

## SUB Box

User Interface/General

*Anwendung* Box bereich, farbe, rand\$, fuellen%

*Nutzen* Zeichnet auf den Rand des angegebenen *bereichs*, der aus den vier Zahlen y1%, x1%, y2%, x2% besteht, einen Rahmen. Dabei wird die Farbe farbe (bestehend aus *vordergrund%*, *hintergrund%*) benutzt; wenn *fuellen%* TRUE ist, wird der Innenraum des Rahmens mit dem entsprechenden Zeichen aus *rand\$* in der angegebenen Farbe gefüllt, sonst wird er so gelassen, wie er ist.

*rand\$* beschreibt die Art des Rahmens, der zu zeichnen ist. *rand\$* muß neun Zeichen enthalten, und zwar in dieser Reihenfolge: Obere linke Ecke, horizontale Linie oben, obere rechte Ecke, vertikale Linie links, Füllzeichen für die Mitte (wird nur benutzt, wenn *fuellen%* = TRUE, muß aber in jedem Falle angegeben werden), vertikale Linie rechts, untere linke Ecke, horizontale Linie unten,

untere rechte Ecke. Wenn Sie *rand\$* leer lassen, wird eine Box mit einfachem Rahmen gezeichnet.

*Bemerkung* » Beachten Sie, daß Box nicht um den Bereich einen Rahmen zeichnet, sondern auf seinen Rand. Die angegebenen Koordinaten müssen im Bereich MinRow bis MaxRow beziehungsweise MinCol bis MaxCol liegen; diese Konstanten sind in GENERAL.BI als 2, 25, 1 und 80 definiert.

» Im Anhang D.3 finden Sie die ASCII-Zeichentabelle.

*Siehe auch* WindowBox (501), Scroll (517), WindowScroll (508).

## SUB GetBackground

User Interface/General

*Anwendung* GetBackground *y1%, x1%, y2%, x2%, speicher\$*

*Nutzen* GetBackground ist das Textmodus-Äquivalent zu GET im Grafikmodus. Mit GetBackground können Sie einen beliebigen Bildschirmbereich (vertreten durch die Koordinaten *y1%, x1%, y2%, x2%*) in einer Stringvariable abspeichern (*speicher\$*). GetBackground kümmert sich automatisch darum, *speicher\$* die benötigte Länge zuzuordnen.

*Bemerkung* » Diese Routine ist extrem wichtig für den Einsatz von Fenster-technik. Sie kann auch benutzt werden, um ganze Bildschirm-Darstellungen in einen String zu kopieren. Danach können beliebige andere Informationen angezeigt werden, und mit PutBackground ist in Nullkommanichts der alte Bildschirm wiederhergestellt. Das ist zwar ein bißchen langsamer als die Benutzung verschiedener Bildschirmseiten (siehe PCOPY; aktuelle und virtuelle Seite, siehe SCREEN), aber dafür hardware-unabhängig.

» Die angegebenen Koordinaten müssen im durch MinCol, MinRow, MaxCol und MaxRow (Konstanten aus GENERAL.BI) erlaubten Bereich sein. MinRow ist dort auf 2 gesetzt, um Kompatibilität mit den Menü-Routinen zu gewährleisten (Zeile 1 ist dort die Menüleiste); Sie werden das vielleicht ändern wollen, um GetBackground flexibler benutzen zu können.

» Wenn Sie größere Mengen Text vom Bildschirm holen wollen, sollten Sie mit Far Strings arbeiten (siehe Kapitel 12.3).

» Sie können GetBackground auch verwenden, um zu prüfen, ob ein bestimmter Text auf dem Bildschirm steht (wenn zum Beispiel mit SHELL ein Programm aufgerufen wurde und interessant ist, ob es eine bestimmte Fehlermeldung ausgegeben hat). GetBackground speichert in *speicher\$* zunächst die Breite, dann die Höhe des Bereiches ab (je 1 Byte); dann folgen abwechselnd immer ein Textbyte und ein Farbbyte (zur Dekodierung eines Farbattributs siehe Bemerkung zu AttrBox), und am Schluß stehen noch zwei Leerzeichen.

Um aus einem mit GetBackground gewonnenen String die reine Textinformation zu extrahieren, können Sie also diese Funktion verwenden:

```
FUNCTION NurText$ (BackGround AS STRING)

    DIM Laenge AS INTEGER, Extrakt AS STRING
    Laenge = LEN(BackGround) \ 2 - 2
    Extrakt = SPACE$(Laenge)

    FOR i% = 1 TO Laenge
        MID$(Extrakt, i%, 1) = MID$(BackGround, 1 + 2*i%, 1)
    NEXT

    NurText$ = Extrakt

END FUNCTION
```

*Siehe auch* PutBackground (516), AttrBox (513), PCOPY (356), SCREEN (Funktion) (377), GET (318).

## FUNCTION GetShiftState

User Interface/General

*Anwendung* x% = GetShiftState(taste%)

*Nutzen* Gibt den Status einer der Spezialtasten zurück, die mit dem gewöhnlichen INKEY\$-Befehl nicht abgefragt werden können.

Der Funktionswert von GetShiftState ist entweder TRUE (wenn die Taste im Augenblick gedrückt [für Shift, CTRL, ALT] oder an [für Caps, Scroll, Num Lock, Insert] ist) oder FALSE, wenn die Taste nicht gedrückt beziehungsweise aus ist.

Die Tasten Caps Lock, Num Lock ("Num"), Scroll Lock "(Roll)" und Insert ("Einfg") haben eine Schalterfunktion, das heißt, sie werden durch einmaliges Drücken ein- und durch nochmaliges Drücken wieder ausgeschaltet. Bei den ersten dreien ist das offensichtlich, da zumeist eine Anzeige auf der Tastatur den aktuellen Status kundtut. Aber auch die Insert-Taste, die kein Lämpchen hat, wird von DOS wie ein Schalter behandelt.

<i>taste%</i>	<b>...fragt den Status dieser Taste ab:</b>
0	Shift rechts
1	Shift links
2	CTRL ("Strg")
3	ALT
4	Scroll Lock ("Roll")
5	Num Lock ("Num")
6	Caps Lock (Großschreibung)
7	Insert ("Einfg")

*Bemerkung* » Diese Funktion benutzt den Interrupt 16h, um den Tastaturstatus abzufragen. Wenn Sie diesen Interrupt (Unterfunktion 2, siehe Anhang E) direkt aufrufen, können Sie auch Tastenkombinationen testen (zum Beispiel CTRL-ALT-Shift gleichzeitig).

» Es gibt keine Möglichkeit, die ALT GR-Taste zu prüfen, die auf vielen neuen Tastaturen vorhanden ist.

*Siehe auch* INTERRUPT (205).

<b>SUB PutBackground</b>	<b>User Interface/General</b>
--------------------------	-------------------------------

*Anwendung* PutBackground *y1%, x1%, speicher\$*

*Nutzen* Gibt einen mit GetBackground im Textmodus gesicherten Bildschirmbereich wieder aus. *y1%* und *x1%* sind Zeile und Spalte für die obere linke Ecke; *speicher\$* ist der String, in den mit GetBackground der Bildschirmbereich kopiert wurde.

*Bemerkung* » PutBackground prüft nicht, ob von den angegebenen Koordinaten bis zum Bildschirmrand genügend Platz vorhanden ist, um den Bereich abzubilden. Wenn Sie das Bildschirmlimit überschreiten, kann das unerwünschte Folgen haben (das Bild wird umgebrochen, es tritt ein Fehler auf, oder Speicherbereiche werden überschrieben).

» Wie auch GetBackground erlaubt PutBackground nur Koordinaten im Bereich der Konstanten aus GENERAL.BI.

*Siehe auch* GetBackground (514), PUT (367).



*Anwendung* Scroll *y1%, x1%, y2%, x2%, zeilen%, hintergrund%*

*Nutzen* Schiebt einen Bildschirmbereich auf- oder abwärts. Der durch *y1%, x1%, y2%, x2%* (Einheiten wie beim LOCATE-Befehl) begrenzte Ausschnitt wird...

» bei *zeilen% > 0*: um *zeilen%* nach oben verschoben. Die Zeilen *y2%-zeilen%+1* bis einschließlich *y2%* werden dadurch leer und mit Leerzeichen der angegebenen Farbe gefüllt.

» bei *zeilen% < 0*: um *zeilen%* nach unten verschoben. Die Zeilen *y1%* bis einschließlich *- zeilen%-1* werden dadurch leer und mit Leerzeichen der angegebenen Farbe gefüllt.

» bei *zeilen% = 0*: mit Leerzeichen der angegebenen Farbe überschrieben.

Beim Verschieben werden Zeilen, die dabei eigentlich über den angegebenen Bereich hinausgeschoben würden, gelöscht. Die Prozedur verändert also grundsätzlich die Bildschirmanzeige nur im angegebenen Bereich.

*hintergrund%* ist die Farbe, in der die neu erzeugten Leerzeilen beziehungsweise, bei *zeilen% = 0*, das ganze Fenster gefüllt werden soll. Der erlaubte Bereich ist 0 bis 7.

*Bemerkung* » Diese Prozedur benutzt den Interrupt 10h, um die Bereichsverschiebung durchzuführen.

» Wenn Sie die Koordinaten falsch herum angeben (das kleinere *x* beziehungsweise *y* zuerst), werden sie bleibend vertauscht, sind also auch in Ihrem Programm danach nicht mehr das, was sie waren.

» Es sind nur Zeilenangaben zwischen MinRow und MaxRow und Spaltenangaben zwischen MinCol und MaxCol erlaubt (Konstanten aus GENERAL.BI).

*Siehe auch* WindowScroll (508), AttrBox (513), VIEW PRINT (415).



---

## **Sektion    Anhänge VIII**

---

- **Alle Limits auf einen Blick**
  - **Alle Switches  
auf einen Blick**
  - **Alle Fehlermeldungen  
auf einen Blick**
  - **Tastatur- und  
Zeichencodes**
  - **Ausgewählte Interrupts**
  - **Mausfunktionen**
  - **Tabellen dieses Buches**
  - **Liste der Programm-  
beispiele**
-



# Anhang A:

# Alle Limits auf einen Blick

## 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 QBX 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 QBX 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.

## 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) und Ziffern (0-9). 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.

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 beziehungsweise beim Aufruf von QBX der Switch /Ah angegeben wird. Größer als 131.072 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 Dateinummern gleichzeitig in Gebrauch sein. Eine RANDOM-Datei kann maximal 2.147.483.647 Datensätze enthalten (eine BINARY-Datei ebenso viele Bytes\*). Die Satzlänge einer RANDOM-Datei darf 32.767 nicht überschreiten. Die Gesamtgröße einer Datei ist allenfalls durch das Betriebssystem begrenzt.

---

\* Sie kann länger sein, aber jenseits der genannten Grenze liegende Daten können nicht gelesen oder geschrieben werden.

# ISAM

Eine Datenbank darf maximal 300 Indexlisten haben; wenn diese Grenze nicht durch den Switch /I beim Kompilieren beziehungsweise bei PROISAM oder PROISAMD.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 ISASM-Buffer werden benutzt; das Minimum sind 6 (für PROISAM.EXE) beziehungsweise 9 (für PROISAMD.EXE). In den Typdefinitionen aller zu einem Zeitpunkt gleichzeitig geöffneten ISAM-Datenbanken dürfen zusammen nicht mehr als 60 nicht indexierbare Felder vorkommen. Nicht indexierbare 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 QBX lauffähig zu sein, nicht größer als 65.535 Bytes sein.

Dokumente und Include-Dateien, die in QBX geladen werden, dürfen nicht größer als 65.536 Bytes sein.

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 »-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

Es dürfen maximal 32 Libraries angegeben werden. Die maximale Anzahl von Overlays ist 63.

# Anhang B:

## Alle Switches auf einen Blick

### QBX

#### Aufruf:

QBX [*switches*] [/RUN] *programmname* [/CMD *command\$*]

#### Switches:

/Ah	Huge Arrays.
/B	QBX-Bildschirmanzeige in schwarz/weiß.
/C: <i>b</i>	Standard-Kommunikations-Buffer auf <i>b</i> Bytes setzen.
/CMD: <i>command\$</i>	COMMAND\$ setzen.
/Ea	Arrays in EMS auslagern.
/E: <i>ems</i>	Nur <i>ems</i> KB vom EMS benutzen.
/Es	Expanded Memory für QBX und Routinen anderer Sprachen.
/G	Beschleunigt den Bildschirmaufbau bei CGA.
/H	Maximale Bildschirmauflösung benutzen.
/K: <i>tastaturdatei</i>	Angegebene Tastaturdefinitionsdatei laden.
/L [ <i>quicklibrary</i> ]	Quick Library laden.
/MBF	Funktionen <i>MKS\$</i> , <i>MKD\$</i> , <i>CVS</i> und <i>CVD</i> durch <i>MBF</i> -Funktionen ersetzen.
/NOF	Menüs "Utility", "Help" und "Options" deaktivieren.
/NOHI	Keine Highlight-Zeichen.
/RUN	Programm sofort starten.



# BC

## Aufruf:

BC [*modulname* [, [*objectname*] [, *listname*]]] [*switches*] [;]

## Switches:

/A	Compiler erzeugt ein Listing des Assembler-Codes.
/Ah	Huge Arrays.
/C: <i>b</i>	Standard-Kommunikations-Buffer auf <i>b</i> 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.
/FBr	Erzeugt Browser-Informationen für PWB nur für globale Definitionen.
/FBx	Erzeugt vollständige Browser-Informationen für PWB.
/FPi	Benutzt die "Emulator"-Libraries.
/Fs	Modul benutzt Far Strings.
/G2	Produziert Code, der nur ab 80286 aufwärts funktioniert.
/Ib: <i>x</i>	Spezifiziert die Anzahl von ISAM-Puffern.
/Ie: <i>x</i>	Setzt die Menge an EMS (in KB), die für nicht-ISAM- Anwendungen freigelassen werden soll.
/Ii: <i>x</i>	Setzt die maximale Anzahl von ISAM-Indizes.
/LP	Erzeugt ein OS/2-Protected Mode-Object File.
/LR	Erzeugt ein Real Mode-Object File für DOS.
/MBF	Ersetzt die Funktionen <i>MKS\$</i> , <i>MKD\$</i> , <i>CVS</i> und <i>CVD</i> durch ihre Pendants mit dem Anhängsel <i>MBF</i> .
/O	Sorgt dafür, daß beim Linken ein Stand-Alone-EXE- Programm erzeugt wird.
/Ot	Optimiert die Geschwindigkeit für <i>SUB</i> -, <i>FUNCTION</i> - und <i>DEF FN</i> -Aufrufe.
/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.
/T	Unterdrückt Warnungen.
/V	Ermöglicht das Event Trapping
/W	Wie /V, prüft aber nur an jeder Zeilennummer bezie- hungsweise an jedem Zeilenlabel
/X	Erlaubt ON ERROR und RESUME NEXT
/Z	Listet die Fehler während desKompilierens für PWB
/Zd	Erzeugt Object-File für SYMDEB
/Zi	Erzeugt Object-File für CodeView

# PWB

## Aufruf:

PWB [*switches*] [*programmname*] [*programmname*]...

## Switches:

/D[A][S][T]	/DA verhindert, daß die PWB-Zusätze automatisch geladen werden. /DS verhindert, daß die Datei CURRENT.STS eingelesen wird. /DT verhindert, daß die PWB versucht, Befehle für sich aus der Datei TOOLS.INI zu entnehmen. /D alleine wirkt wie /DA, /DS und /DT zugleich.
/E <i>text</i>	Führt die in <i>text</i> genannten PWB-Befehle sofort aus
/M <i>cursor</i>	Positioniert den Cursor an der durch <i>cursor</i> spezifizierten Stelle in einer geladenen Datei.
/PF <i>datei</i>	Lädt die NMAKE-kompatible Programmliste <i>datei</i> .
/PL	Lädt die letzte in der PWB benutzte Programmliste.
/PP <i>datei</i>	Lädt die PWB-kompatible Programmliste <i>datei</i> .
/R	Verhindert, daß irgendeine geöffnete Datei in verändertem Zustand auf die Patte geschrieben werden kann.
/T	(nur unmittelbar vor einem Dateinamen erlaubt) Legt fest, daß die genannte Datei eine temporäre Datei ist.

# LINK

## Aufruf:

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

LINK @*steuerungsdatei*

## Switches:

/?	Alle LINK-Switches als Tabelle zeigen.
/BA	Batch-Modus.
/CO	Codeview-Informationen einbinden.
/E	EXE-File komprimieren.
/F	Far Calls, wenn möglich, in Near Calls umwandeln.
/HE	LINK-Hilfe aufrufen und anzeigen.
/INF	Detaillierte LINK-Informationen während des Ablaufs anzeigen.
/LI	In die Listendatei die Adressen von Zeilennummern einschließen.

(Fortsetzung nächste Seite)

/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.
/PM:typ	Presentation Manager-Typ setzen: NOVIO = nicht PM-kompatibel, VIO = PM-kompatibel.
/Q	Quick Library erstellen.
/SE:x	Maximale Segmentanzahl 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.

Befehl	Wirkung
+	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.
/Fs	RTM benutzt Far Strings.
/LP	RTM für OS/2 erzeugen.
/LR	RTM für Real Mode erzeugen.
/DEFAULT	Standard-RTM zu den angegebenen Switches erstellen (kein RTM-Name und keine Exportliste werden benötigt).

# NMAKE/NMK

## Aufruf:

NMAKE [*switches*] [*makrodefinitionen*] [*programmname*]...

NMAKE @steuerungsdatei

## Switches:

/A	Alle Programme neu erstellen.
/C	NMAKE-Copyright-Meldung und Warnungen unterdrücken.
/D	Datum der letzten Änderung für jede Datei anzeigen.
/E	Betriebssystemvariablen können nicht mehr per Makrodefinition in der Beschreibungsdatei undefiniert werden.
/F <i>name</i>	Angegebene Datei anstelle von MAKEFILE als Beschreibungsdatei benutzen.
/I	Fehler von Programmen, die NMAKE startet, ignorieren.
/N	Simulationsmodus. NMAKE zeigt alle Befehle an, die es ausführen <i>würde</i> , führt sie aber nicht wirklich aus.
/NOLOGO	NMAKE-Copyright-Meldung unterdrücken.
/P	Alle Makrodefinitionen und Abhängigkeitsbeschreibungen ausgeben.
/Q	Status-Code größer 0 zurückgeben, wenn das Programm überholt war.
/R	Vordefinierte und in TOOLS.INI enthaltene Ableitungsregeln ignorieren.
/S	Befehle, die NMAKE ausführt, nicht anzeigen.
/T	Verhindern, daß die überholten Dateien wirklich aktualisiert werden, sondern Datum jeder überholten Datei auf das heutige Datum setzen.
/X <i>name</i>	Alle Fehlermeldungen in die angegebene Datei <i>name</i> schreiben.

# Anhang C: Alle Fehlermeldungen auf einen Blick

## C.1 BASIC-Fehlermeldungen (nach Nummern)

Die Tabelle enthält die englischen Fehlerbezeichnungen, deren exakte Bedeutung beziehungsweise Übersetzung Sie in C.2 nachschlagen können.

3	RETURN without GOSUB
4	Out of DATA
5	Illegal function call
6	Overflow
7	Out of memory
9	Subscript out of range
10	Duplicate definition
11	Division by zero
13	Type mismatch
14	Out of string space
16	String formula too complex
19	No RESUME
20	RESUME without error
24	Device timeout
25	Device fault
27	Out of paper
40	Variable required
50	FIELD overflow
51	Internal error
52	Bad file name or number
53	File not found
54	Bad file mode
55	File already open
56	FIELD statement active
57	Device I/O error
58	File already exists
59	Bad record length
61	Disk full
62	Input past end of file
63	Bad record number
64	Bad file name

*(Fortsetzung nächste Seite)*

67	Too many files
68	Device unavailable
69	Communication-buffer overflow
70	Permission denied
71	Disk not ready
72	Disk-media error
73	Feature unavailable
74	Rename across disks
75	Path/File access error
76	Path not found
80	Feature removed
81	Invalid name
82	Table not found
83	Index not found
84	Invalid column
85	No current record
86	Duplicate value for unique index
87	Invalid operation on NULL index
88	Database needs repair
89	Insufficient ISAM buffers

## C.2 BASIC-Fehlermeldungen (alphabetisch)

Das Kürzel QB steht bei QBX-Fehlermeldungen, CT bei Fehlermeldungen, die während des Kompilierens, und RT bei Fehlern, die in einem laufenden Programm auftreten können.

### **Argument-count mismatch (CT)**

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.

### **Array already dimensioned (CT, RT)**

Statische Arrays dürfen nur ein einziges Mal mit DIM definiert werden; dynamische Arrays nur dann weitere Male, wenn sie inzwischen mit ERASE gelöscht wurden. OPTION BASE darf nur benutzt werden, wenn noch keine Arrays definiert wurden.

### **Array not defined (CT bei QBX)**

### **Array not dimensioned (CT bei BC)**

In QBX muß jedes benutzte Array mit einem DIM-Befehl vereinbart worden sein, sonst läuft das Programm nicht. 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.

### **Array too big (CT bei BC)**

Versuchen Sie, das Array zu verkleinern (vielleicht, indem Sie bei einem selbst-definierten Typ einige Bytes sparen). Benutzen Sie dynamische Arrays (\$DYNAMIC) und notfalls den Compiler-Switch /Ah. Lesen Sie nach bei Kapitel 12, Abschnitt 4, "Huge Arrays".

### **AS clause required (CT)**

#### **AS clause required on first declaration (CT)**

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.

### **AS missing (CT)**

Sie haben beim OPEN-Befehl das AS vergessen oder falsch angebracht.

### **Asterisk missing (CT)**

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.

### **Bad file mode (QB, RT)**

In QBX 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 Merge aus dem File-Menü in Ihr Programm einzubinden. Die entsprechende Datei muß im Text-Format vorliegen (laden Sie sie normal, und speichern Sie sie mit "Save As" als Textdatei).

Als Runtime-Fehler 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 Lesengeöffnete Datei schreiben oder aus einer zum Schreiben geöffneten Datei lesen wollen oder versuchen, mit gewöhnlichen Befehlen eine ISAM-Datei zu bearbeiten und umgekehrt.

### **Bad file name (QB, RT)**

Sowohl in QBX als auch als Runtime-Fehler taucht diese Meldung auf, wenn Sie eine Datei öffnen, laden, speichern oder löschen wollen oder sonst irgendetwas 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.



### **Bad file name or number (RT)**

Ein OPEN FOR ISAM-Befehl konnte nicht richtig ausgeführt werden, weil der Dateiname ungültig war (siehe *bad file name*) 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 Referenzteil).

### **Bad record length (RT)**

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.

### **Bad record number (RT)**

In einem GET-, PUT- oder SEEK-Befehl haben Sie eine Satznummer angegeben, die kleiner als 1 oder größer als  $2^{31} - 1$  ist.

### **BASE missing (CT)**

Keine Variable darf OPTION heißen, denn der Compiler erwartet danach das Wort BASE, weil er einen OPTION BASE-Befehl vermutet.

### **Binary source file (CT)**

Ein BASIC-Programm, das Sie kompilieren wollten, ist weder im ASCII- beziehungsweise Text-Format noch im komprimierten QBX-Format gespeichert (Das komprimierte Format von QuickBASIC 2 und 3 ist nicht mehr kompatibel!). Diese Meldung kann auch als bloße Warnung auftreten, wenn Sie das gültige QBX-Format benutzen, aber die Compiler-Optionen /Zi oder /Zd angegeben haben, weil CodeView mit den komprimierten Source-Code-Zeilen nichts anfangen kann.

### **Block IF without END IF (CT)**

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 Kapitel.

### **Breakpoints not allowed on CASE clauses or END SELECT (QB)**

Sie dürfen auf den ersten Befehl nach einer CASE-Anweisung keinen Breakpoint setzen. Setzen Sie ihn stattdessen weiter vorne, und benutzen Sie dann die Einzelschritt-Funktionen, um den Verlauf des Programms zu beobachten.

### **Buffer size expected after /C: (CT)**

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.

### **BYVAL only allowed with numeric elements (CT)**

BYVAL darf nur mit den Datentypen INTEGER, LONG, CURRENCY, SINGLE und DOUBLE benutzt werden.

### **/C: buffer size too large (CT)**

Die maximale Größe für den Kommunikationspuffer ist 32.767. Der Compiler bricht nicht ab, sondern ignoriert den Switch.

### **Cannot generate listing for BASIC binary source files (CT)**

Sie können kein Source-Listing erstellen (Compiler-Switch /A), wenn das Programm im komprimierten QBX-Format abgespeichert ist. Lassen Sie /A weg, oder laden Sie das Programm mit QBX, und speichern Sie es als Textfile.

### **Cannot start with 'FN' (CT)**

Variablen- und Subroutinen-Namen dürfen nicht mit FN beginnen, da sie sonst mit einer DEF FN-Funktion verwechselt werden könnten.

### **CASE without SELECT (CT)**

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 Kapitel).

### **Colon expected after /C (CT)**

Der Compiler-Switch /C erfordert die Angabe eines Doppelpunktes und der Kommunikationspufferlänge.

### **Comma missing (CT)**

Sie haben ein Komma und möglicherweise nachfolgende Parameter vergessen, zum Beispiel PCOPY (0). In QBX heißt diese Meldung "Expected: ,".

### **COMMON and DECLARE must precede executable statements (CT)**

COMMON und DECLARE müssen im Programm vor jedem ausführbaren Befehl erscheinen. Nicht ausführbare Befehle sind alle die, die nur Vereinbarungen darstellen: Alle Metabefehle, COMMON, CONST, DATA, DECLARE, DEFxxx, alle DIM-Befehle bis auf Vereinbarungen von dynamischen Arrays, EVENT ON/OFF, OPTION BASE, REM, SHARED, STATIC und TYPE.

### **COMMON in Quick library too small (CT in QBX)**

In der Quick Library sind weniger Variablen mit COMMON deklariert als im gerade aktiven Hauptprogramm. 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 illegal (CT)**

Sie haben einen ungültigen Namen benutzt, um einen COMMON-Block zu betiteln.

### **Communication-buffer overflow (RT)**

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.

### **CONST/DIM SHARED follows SUB/FUNCTION (CT)**

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.

### **Control structure in IF...THEN...ELSE incomplete (CT)**

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.

### **Currency type illegal in alternate math pack (CT)**

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.

### **Database needs repair (RT)**

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 7.8) kann benutzt werden, um die Datenbank wieder in Ordnung zu bringen.

### **Data-memory overflow (CT)**

Der zur Verfügung stehende Speicherplatz reicht nicht aus, um das Programm zu kompilieren. Es ist möglich, daß ein Programm in QBX einwandfrei läuft und beim Kompilieren dann diese Meldung erzeugt, wenn es zu groß ist (QBX hat nicht so strenge Speicher-Restriktionen wie der eigentliche Compiler).

» 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
- » verzichten Sie auf Compiler-Switches wie /D, die den erzeugten Code verlängern.

### **DECLARE required (CT)**

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 "Equal sign missing"-Fehler, und der zweite führt zu "Array not defined", beide eventuell in Kombination mit "Syntax Error".

### **DEF FN not allowed in control statements (CT)**

DEF FN darf nicht innerhalb von Kontrollstrukturen wie IF...END IF, FOR...NEXT oder SELECT...END SELECT benutzt werden. Auch das Umgehen einer DEF FN-Anweisung mit GOTO würde Ihnen nichts helfen, denn eine irgendwo mit DEF FN vereinbarte Funktion ist im ganzen Programm selbst dann gültig, wenn sie niemals wirklich aufgerufen wird.

### **DEF without END DEF (CT)**

Zu einem DEF FN fehlt das zugehörige END DEF. Siehe auch "Unerklärliche Strukturfehler" weiter unten in diesem Kapitel.

### **DEFTYPE character specification illegal (CT)**

Sie haben einen DEFxxx-Befehl (DEFLNG, DEFDBL, DEFINT, DEFSNG, DEFCUR, DEFSTR) falsch angewendet.

### **Device fault (RT und QB)**

Innerhalb QBX 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 "Device timeout"-Fehler auftreten, wenn eine der Drucker- oder Kommunikationsschnittstellen nicht oder nicht in der erwarteten Weise reagiert.

### **Device timeout (RT und QB)**

Innerhalb QBX bedeutet dieser Fehler dasselbe wie "Device Fault". Innerhalb eines Programms kommt ein "Device timeout" zustande, wenn eine Schnittstelle nicht in der vorgegebenen Zeit reagiert (beim Öffnen der Kommunikationsschnittstellen können verschiedene Timeout-Zeiten angegeben werden).

### **Device unavailable (RT und QB)**

Das Programm beziehungsweise QBX versucht, eine Datei auf einem Laufwerk zu öffnen, das nicht existiert.

**Disk full (RT und QB)**

Das Schreiben von Daten in eine Datei ist nicht möglich, weil die Diskette/Platte voll ist.

**Disk-media error (RT und QB)**

Auf der Diskette oder Festplatte ist ein Hardware-Fehler beim Lesen oder Schreiben von Daten aufgetreten.

**Disk not ready (RT und QB)**

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 den meisten Schreibzugriffen.

**Division by zero (CT und RT)**

Dieser Fehler tritt auf, wenn eine Zahl durch Null dividiert wird. 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 without LOOP (CT)**

Zu einem DO fehlt das LOOP. Siehe auch "Unerklärliche Strukturfehler" weiter unten in diesem Kapitel.

**Document too large (QB)**

Sie haben versucht, in QBX ein Dokument zu laden, das größer als 64 KB ist. Das ist nicht möglich.

**Duplicate definition (CT oder RT)**

- » 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.

**Duplicate label (CT)**

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.)

### **Duplicate value for unique index (RT)**

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.

### **Dynamic array element illegal (CT)**

VARPTR\$ darf nicht auf Elemente aus dynamischen Arrays angewandt werden. Sollte es wirklich notwendig sein, können Sie zunächst eine temporäre skalare Variable benutzen, ihr den betreffenden Arraywert zuweisen, und die Adresse dieser Variable dann mit VARPTR\$ ermitteln. Außer für PLAY und DRAW sollte man allerdings VARPTR\$ nicht mehr benutzen; VARPTR, VARSEG, SADD und SSEG übernehmen diese Funktion.

### **Dynamic array illegal (CT)**

Innerhalb der Definition eines selbstdefinierten Typs (TYPE...END TYPE) dürfen nur symbolische Konstanten oder Konstanten, nicht aber Variablen als Array-Dimensionsvereinbarung benutzt werden, weil einem selbstdefinierten Typen schon beim Kompilieren eine feste Länge zugeordnet werden muß.

### **Element not defined (CT)**

Dieser Fehler tritt auf, wenn Sie ein Element einer Variable selbstdefinierten Typs benutzen, das in der TYPE...END TYPE-Definition nicht vorkommt. Normalerweise können Sie beispielsweise 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 without IF (CT)**

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 corrupt (RT)**

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. Wenn Sie beim Linken des Programms das Verzicht-File NOEMS.OBJ angeben, wird das EMS nicht für Overlays benutzt. Lesen Sie in Kapitel 6.3 über den Compiler-Switch /Es nach. Siehe auch "Unerklärliche Systemfehler" weiter unten in diesem Kapitel.

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 without DEF (CT)**

Sie haben END DEF benutzt, ohne vorher eine Funktionsdefinition mit DEF FN eingeleitet zu haben. Siehe auch "Unerklärliche Strukturfehler" weiter unten in diesem Kapitel.

### **END IF without block IF (CT)**

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 Kapitel.

### **End of file unexpected in TYPE declaration**

Die .BAS-Datei ist mitten innerhalb einer TYPE...END TYPE-Definition plötzlich zu Ende.

### **END SELECT without SELECT**

Sie haben END SELECT benutzt, ohne daß zuvor ein zugehöriges SELECT aufgetreten wäre. Siehe auch "Unerklärliche Strukturfehler" weiter unten in diesem Kapitel.

### **END SUB or END FUNCTION must be last line in window (CT)**

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 beziehungsweise EXIT FUNCTION benutzen.

Dieser Fehler tritt manchmal unberechtigterweise und ohne ersichtlichen Grund auf. Dann hilft es zumeist, das Programm (mit "Save As" aus dem "File"-Menü) als Textdatei zu speichern, QBX zu verlassen und dann erneut zu starten.

### **END SUB/FUNCTION without SUB/FUNCTION (CT)**

Einer Prozedur oder Funktion fehlt die SUB- beziehungsweise FUNCTION-Zeile.

### **END TYPE without TYPE (CT)**

Sie haben END TYPE benutzt, ohne vorher die Typdefinition mit TYPE eingeleitet zu haben.

### **Equal sign missing (CT)**

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 QBX 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.

### **Error during QBX initialization (QB)**

QBX hat Probleme mit der Hardware (zuwenig Speicher?) oder anderen geladenen Programmen.

#### **Error loading file xxx - Cannot find file (QB)**

#### **Error loading file xxx - Disk I/O error (QB)**

#### **Error loading file xxx - DOS memory area error (QB)**

#### **Error loading file xxx - Invalid format (QB)**

#### **Error loading file xxx - Out of memory (QB)**

Diese Fehler treten auf, wenn eine Datei (zumeist eine Quick Library) von QBX aus den angegebenen Gründen nicht geladen werden kann. "DOS memory area error" kann bedeuten, daß irgendein Programm den Speicher unbefugt oder inkorrekt manipuliert hat - booten Sie neu; "Invalid Format" kann bedeuten, daß Sie eine Quick Library zu laden versuchten, die mit QuickBASIC 4.0 oder 4.5 erstellt wurde.

#### **Error loading run-time module xxx: Cannot find file in PATH (RT)**

#### **Error loading run-time module xxx: Disk I/O error (RT)**

#### **Error loading run-time module xxx: DOS memory-area error (RT)**

#### **Error loading run-time module xxx: Incompatible run-time module (RT)**

#### **Error loading run-time module xxx: Invalid format (RT)**

#### **Error loading run-time module xxx: Memory allocation error (RT)**

#### **Error loading run-time module xxx: Out of memory (RT)**

Diese Fehler treten bei der Ausführung eines Programmes auf, wenn ein Runtime-Modul nicht geladen werden kann. Die Meldungen sprechen weitgehend für sich; "Memory allocation error" und "DOS memory area error" können bedeuten, daß der Speicher von einem Programm unbefugt oder inkorrekt manipuliert wurde, und "Invalid format" deutet darauf hin, daß bei der Erstellung des Runtime-Moduls mit BUILDRTM etwas schiefgegangen ist. Über "Incompatible run-time module" gibt es weitere Informationen in Kapitel 20.7.

### **Error loading overlay (RT)**

Beim Nachladen eines Overlays aus dem EMS oder von der Diskette/Platte trat ein Fehler auf. Das kann an Speicher- oder Plattenproblemen liegen, die der Benutzer des Programms zu verantworten hat, aber auch daran, daß Ihr EXE-Programm einen Fehler enthält (vorsichtshalber neu kompilieren). Siehe auch "Unerklärliche Systemfehler" weiter unten in diesem Kapitel.

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.

### **/Es option incompatible with /Ea (QB)**

Sie können beim Aufruf von QBX nicht zugleich /Es und /Ea angeben.



### **EVENT ON without /v or /w on command line (CT)**

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.

### **EXIT DO not within DO...LOOP (CT)**

Sie können EXIT DO nur innerhalb einer DO...LOOP-Schleife benutzen. Siehe auch "Unerklärliche Strukturfehler" weiter unten in diesem Kapitel.

### **EXIT FOR not within FOR...NEXT (CT)**

EXIT FOR darf nur innerhalb einer FOR...NEXT-Schleife benutzt werden. Siehe auch "Unerklärliche Strukturfehler" weiter unten in diesem Kapitel.

### **Expression too complex (CT)**

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.

### **Extra file name ignored (CT bei BC)**

Ein überflüssiger Dateiname, den Sie beim Aufruf von BC angegeben haben, wird ignoriert.

### **Far heap corrupt (RT)**

Dieser Fehler tritt auf, wenn von externen Routinen (oder mit dem POKE-Befehl) Speicherinhalte verändert wurden, auf die BASIC angewiesen ist. Siehe auch "Unerklärliche Systemfehler" weiter unten in diesem Kapitel.

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.

### **Feature removed (RT)**

Das Programm benutzt einen Befehl, der durch ein Verzicht-File beim Linken ausgeschlossen wurde (zum Beispiel SCREEN 9, wenn Sie mit NOGRAPH, NOEGA oder TSCNIOxx gelinkt haben).

### **Feature unavailable (RT)**

Das Programm benutzt unter DOS einen Befehl, der nur für OS/2 erlaubt ist (zum Beispiel die Funktion SHELL) oder unter OS/2 einen Befehl, der nur unter DOS erlaubt ist (zum Beispiel PCOPY), oder es wird ein Befehl benutzt, der (zum Beispiel OPEN mit LOCK) erst ab einer höheren DOS-Version erlaubt ist.

### **FIELD buffer overflow (RT)**

#### **FIELD overflow (RT)**

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.

### **FIELD statement active (RT)**

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.

### **File already exists (RT)**

Es wurde versucht, mit NAME einer Datei einen neuen Namen zuzuordnen, unter dem bereits eine andere Datei existiert.

### **File already open (RT)**

Dieser Fehler tritt auf, wenn Sie mit dem OPEN-Befehl eine Datei öffnen, die bereits geöffnet ist, oder wenn Sie mit KILL eine Datei zu löschen versuchen, die gerade geöffnet ist.

Kein Fehler tritt allerdings auf, wenn Sie eine RANDOM-, BINARY- oder ISAM-Datei erneut öffnen.

Außerdem wird dieser Fehler erzeugt, wenn bei OPEN eine bereits besetzte Dateinummer verwendet wird (benutzen Sie FREEFILE).

### **File not found (RT, QB)**

Ein RUN-, CHAIN-, KILL-, NAME- oder OPEN FOR INPUT-Befehl konnte die genannte Datei nicht finden, oder es wurde versucht, eine Datei in QBX zu laden, die nicht vorhanden ist (auch mit \$INCLUDE).

### **File previously loaded (QB)**

Es kann in QBX keine Datei geladen werden, die sich schon im Speicher befindet. Wahrscheinlich haben Sie übersehen, daß QBX die Datei, die Sie laden wollten, durch ein .MAK-File bereits automatisch geladen hat.

### **Fixed-length string illegal (CT)**

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.

### **FOR index variable already in use (CT)**

In verschachtelten FOR...NEXT-Schleifen haben Sie mehrmals bei FOR dieselbe Zähler-Variable benutzt.

### **FOR index variable illegal (CT)**

Sie haben eine Variable selbstdefinierten Typs oder einen String als Zähler-Variable bei FOR...NEXT benutzt.

**FOR without NEXT (CT)**

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 Kapitel.

**Formal parameter specification illegal (CT)**

Sie haben einen Fehler in der Variablenaufzählung bei SUB, FUNCTION oder DECLARE gemacht.

**Formal parameters not unique (CT)**

In einer DECLARE-, SUB- oder FUNCTION-Zeile kommt derselbe Variablenname mehrfach vor.

**Function already defined (CT)**

Eine SUB- oder FUNCTION-Zeile bezieht sich auf eine Funktion oder Prozedur, die bereits mit SUB oder FUNCTION definiert wurde.

**Function name illegal (CT)**

Der Name einer mit DEF FN definierten Funktion beginnt nicht mit FN.

**Function not defined (CT oder RT bei QB)**

Eine DEF FN-Funktion wird benutzt, bevor sie definiert ist, oder Sie vergaßen das Laden einer Quick Library.

**GOSUB missing (CT)**

Sie haben ON *event* ohne GOSUB benutzt.

**GOTO missing (CT)**

Sie haben ON ERROR ohne RESUME oder GOTO benutzt.

**GOTO or GOSUB expected (CT)**

Nach einer Formulierung ON *variable* fehlt das GOTO oder GOSUB.

**Identifier cannot end with %, &, !, #, \$, or @ (CT)****Identifier cannot include period (CT)**

Prozedurnamen dürfen keinen Typenbezeichner besitzen; Variablen-, Prozedur- und Funktionsnamen dürfen nur dann einen Dezimalpunkt im Namen haben, wenn der Teil vor dem Dezimalpunkt nicht der Name einer Variablen selbstdefinierten Typs ist.

**Identifier expected (CT)**

Der Name einer Variablen, Prozedur oder Konstanten wird erwartet.

**Identifier too long (CT)**

Namen für Typen, Variablen, Konstanten und Prozeduren dürfen nicht länger als 40 Zeichen sein.

### **Illegal function call (RT)**

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 Compiler-Methoden etc. nicht anwendbar sind.

SCREEN 3 erzeugt einen Illegal function call, wenn MSHERC.COM nicht geladen ist.

Schlagen Sie in jedem Fall im Referenzteil nach, und prüfen Sie, welche Nutzungsbeschränkungen für die verwendete Funktion/den verwendeten Befehl gelten.

### **Illegal in direct mode (QB)**

Sie haben im Immediate-Fenster einen Befehl eingegeben, der nur in Programmen verwendet werden darf.

### **Illegal in SUB, FUNCTION or DEF FN (CT)**

Sie haben im Prozedurcode einen Befehl benutzt, der nur im Modulcode erlaubt ist (zum Beispiel SHARED, COMMON, CLEAR).

### **Illegal number (CT)**

Sie haben eine Zahl angegeben, die im Kontext nicht erlaubt ist (zum Beispiel eine Fließkommazahl als Integer-Konstante oder als Index bei DIM).

### **Illegal outside SUB, FUNCTION, or DEF FN (CT)**

#### **Illegal outside SUB or FUNCTION (CT)**

Sie haben einen Befehl im Modulcode benutzt, der nur im Prozedurcode erlaubt ist. Dazu zählen beispielsweise EXIT SUB oder STATIC.

### **Illegal outside of TYPE block (CT)**

Sie haben eine Zeile eingegeben, die QBX für einen Teil einer Typdefinition hält, aber die Zeile liegt nicht innerhalb TYPE...END TYPE. Vielleicht haben Sie ein COMMON, DIM, STATIC oder SHARED vergessen?

### **Illegal type character in numeric constant (CT)**

Eine Zahl enthält einen ungültigen Typbezeichner.

### **\$INCLUDE-file access error (CT bei BC)**

Eine mit REM \$INCLUDE angegebene Datei kann nicht geladen werden.

### **Include file too large (QB)**

Eine Include-Datei darf maximal 64 KB haben.

### **Incorrect DOS version. Link with OVLDOS21.OBJ (RT)**

Ein Programm, das mit Overlays arbeitet und unter DOS 2.1 gestartet wird, ist ohne OVLDOS21.OBJ gelinkt worden.

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.

### **Index not found (RT)**

Sie haben mit SETINDEX oder DELETEINDEX einen Index benannt, der nicht in der angegebenen ISAM-Datenbank existiert.

### **Input file not found (CT bei BC)**

Die angegebene .BAS-Datei ist nicht vorhanden.

### **Input past end of file (RT)**

Sie haben versucht, mit INPUT\$ oder INPUT aus einer sequentiellen Datei Daten zu lesen, obwohl aus dieser Datei bereits alles gelesen wurde.

### **Insufficient EMS to load overlays (RT)**

Das EMS bietet nicht genügend Platz, um die Overlays, die zu einem Programm gehören, hineinzuladen. Sorgen Sie für mehr EMS-Speicher, oder linksen Sie mit NOEMS.OBJ, damit das EMS nicht für Overlays benutzt wird.

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.

### **Insufficient ISAM buffers (RT)**

Für eine ISAM-Operation sind nicht genügend ISAM-Puffer vereinbart worden. Siehe Kapitel 7.5 und 7.6.

### **Integer between 1 and 32767 required (CT)**

Sie haben für einen Befehl, der eine positive INTEGER-Zahl benötigt, eine falsche Zahl angegeben.

### **Internal error (RT)**

#### **Internal error near xxxx (CT bei BC)**

Böse Sache. Sie sind auf einen der Fehler im BASIC 7.1 PDS gestoßen. Er kann beim Kompilieren oder beim Programmablauf auftreten; beim Programmablauf läßt er sich manchmal mit einer Trapping-Routine abfangen, manchmal nicht. Das nützt 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. Weil er, wenn Ihr Programm gewöhnlich genug ist, schon bei den Testläufen bei Microsoft aufgetreten wäre.

Ändern Sie auf gut Glück an Ihrem Programm herum. Das folgende Programm (es hat die Aufgabe, die Tab-Zeichen in einem String in Leerzeichen umzuwandeln), verursacht beim Kompilieren mit dem Befehl BC NOTAB.BAS /G2/O/Ot/S/V/X; einen Internal Error.

```

SUB NoTab (i$) STATIC
  DO WHILE INSTR(i$, CHR$(9))
    X% = INSTR(i$, CHR$(9))
    i$ = LEFT$(i$, X% - 1) + SPACE$(9 - (X% MOD 8 - »
      8 * (X% MOD 8 = 0))) + MID$(i$, X% + 1)
  LOOP
END SUB

```

In diesem speziellen Fall gelang es mir, den Fehler zu vermeiden, indem ich das `STATIC` vom `SUB` entfernte. Es klappte aber auch, wenn ich stattdessen entweder das `/S` oder das `/V` aus dem `BC`-Aufruf wegließ. Eine dritte Möglichkeit war, die lange Zeile in der Mitte in einige kürzere Zeilen aufzuteilen.

Ein Internal Error kann natürlich auch die "Spätfolge" eines nicht diagnostizierten Programmierfehlers Ihrerseits sein (siehe auch "Unerklärliche Systemfehler" weiter unten in diesem Kapitel). Ich wollte Sie mit dem kleinen Beispiel ermutigen, einfach ein wenig an den Compiler-Switches oder den Variablenvereinbarungen im Programm herumzuspielen; Sie dürfen hoffen, irgendwann keinen Internal Error mehr zu bekommen.

### **Invalid Character (CT)**

Ihr Programm enthält ein Steuerzeichen oder ein anderes nicht erlaubtes Zeichen (zum Beispiel einen Umlaut außerhalb von Stringkonstanten).

### **Invalid column (RT)**

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 indexierbaren Feldes in einer `CREATEINDEX`-Anweisung benutzen.

### **Invalid constant (CT)**

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)`).

### **Invalid identifier (CT)**

Sie haben einen ungültigen Namen für einen Typen, eine Variable, eine Konstante oder eine Prozedur benutzt. Erlaubt sind in solchen Namen nur Buchstaben A-Z, Zahlen und der Dezimalpunkt (mit Einschränkungen); sie müssen mit einem Buchstaben anfangen und dürfen maximal 40 Zeichen lang sein.

### **Invalid name (RT)**

Sie haben 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. Das ist bei ISAM nicht erlaubt.

**Invalid operation on NULL index (RT)**

SEEK kann nicht benutzt werden, wenn der Null-Index aktiv ist.

**Label not defined (CT)**

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.

**Left parenthesis missing (CT)**

Eine geöffnete runde Klammer fehlt. Vielleicht erwartet der Compiler an der Stelle ein Array, und Sie haben nur eine einfache Variable angegeben?

**Line invalid. Start again (BC)**

Sie haben auf eine Frage nach einem Dateinamen eine ungültige Antwort gegeben.

**Line number or label missing (CT)**

BASIC erwartet an der betreffenden Stelle eine Zeilennummer beziehungsweise ein -label.

**Line too long**

Die Zeilenlänge darf 255 Zeichen nicht überschreiten.

**LOOP without DO (CT)**

Zu einem DO fehlt das LOOP. Siehe auch "Unerklärliche Strukturfehler" weiter unten in diesem Kapitel.

**Lower bound exceeds upper bound (CT)**

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 *Illegal function call* erzeugt; die Meldung tritt schon beim Kompilieren auf, wenn Sie Konstanten beim Dimensionieren verwenden.

**Math overflow (CT)**

Eine Berechnung hat ein zu großes Ergebnis.

**\$Metacommand error (CT)**

Sie haben einen Fehler bei einem \$INCLUDE-Befehl gemacht. Die Syntax ist REM \$INCLUDE - Doppelpunkt - Apostroph - Dateiname - Apostroph - neue Zeile.

**Minus sign missing (CT)**

BASIC vermißt ein Minuszeichen.

### **Module level code too large (QB)**

Ihr Modulcode ist zu groß für QBX. Modulcode kann nicht ins EMS verschoben werden. Versuchen Sie, einen Teil davon zu Prozedurcode zu machen, indem Sie neue SUBs und FUNCTIONs erstellen.

### **Module not found. Unload module from program? (QB)**

QBX konnte eine der in einer .MAK-Datei genannten Dateien nicht finden und will wissen, ob die Datei trotzdem in der .MAK-Datei stehenbleiben darf.

### **Must be first statement on the line (CT)**

In einer Block-IF...THEN-Konstruktion (einer mehrzeiligen also) muß das IF, das ELSE, das ELSEIF beziehungsweise das END IF jeweils der erste Befehl auf einer neuen Zeile sein.

### **Name of subprogram illegal (CT)**

Sie haben einer Prozedur oder Funktion einen ungültigen oder schon benutzten Namen gegeben. Prozedurnamen dürfen keine Typbezeichner enthalten.

### **Nested function definition (CT)**

Innerhalb einer SUB...END SUB oder FUNCTION...END FUNCTION-Konstruktion darf kein zweiter solcher Block auftreten. Dasselbe gilt auch für DEF FN.

### **NEXT missing (CT)**

Zu einem FOR fehlt das zugehörige NEXT. Siehe auch "Unerklärliche Strukturfehler" weiter unten in diesem Kapitel.

### **No current record (RT)**

Sie haben den DELETE-, den RETRIEVE- oder den UPDATE-Befehl angewendet, obwohl der Dateizeiger bereits vor dem ersten beziehungsweise hinter dem letzten Datensatz steht.

### **No line number in xxx at address yyy (RT)**

Ihr Programm enthielt irgendeinen Fehler, aber BASIC konnte nicht mehr feststellen, wo der war, weil es eigentlich die Zeilennummer angeben wollte, in der beziehungsweise nach der der Fehler auftrat, die Zeilennummertabelle jedoch überschrieben oder zerstört wurde. Guter Tip: Siehe "Unerklärliche Systemfehler" weiter unten in diesem Kapitel.

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.

### **No main module (CT)**

Sie können das Programm in QBX nicht starten, solange Sie nicht irgendeins der geladenen Module zum Hauptmodul erklärt haben.



**No RESUME (RT)**

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.

**Not watchable (QB)**

Die Variable, die Sie in den Watch-Bereich aufgenommen haben, kann in der aktuellen Code-Einheit nicht angezeigt werden, weil sie nicht existiert.

**Numeric array illegal (CT)**

VARPTR\$ darf nicht mit Arrays benutzt werden.

**ON ERROR without /E on command line (CT)**

Das Programm enthält ON ERROR-Befehle, ohne daß Sie /X oder /E auf der BC-Befehlszeile angegeben haben.

**ON event without /V or /W on command line (CT)**

Ihr Programm benutzt den ON *event*-Befehl, ohne daß Sie beim Aufruf von BC /V oder /W angegeben haben.

**Only simple variables allowed (CT)**

Mit READ und INPUT können nur einfache Variablen, keine ganzen Arrays oder Variablen selbstdefinierten Typs, eingelesen werden.

**Option unknown (BC)**

Sie haben einen ungültigen Switch angegeben.

**Out of DATA (RT)**

Ein READ-Befehl wurde ausgeführt, obwohl die letzte DATA-Konstante bereits gelesen worden ist.

**Out of data space (RT oder CT)**

Einer der vielen "Speicherplatz zu knapp" Fehler, dem Sie zu entgehen versuchen können, indem Sie zum Beispiel dynamische Arrays benutzen, den Stack verkleinern oder kleinere Dateipuffer beim Öffnen von sequentiellen Dateien angeben.

**Out of memory (RT oder CT)**

Dieser Fehler kann immer auftreten - als Fehler beim Aufruf des Compilers oder von QBX, 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 QBX 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 QBX nicht möglich. Rufen Sie QBX mit der Option /NOF auf,

dann haben Sie etwas mehr Speicher zur Verfügung. Sparen Sie mit Arrays; benutzen Sie nur die kleinsten geeigneten Datentypen.

### **Overflow (RT)**

Das Ergebnis einer Berechnung ist zu groß für den angegebenen Datentyp. Der Befehl `a! = b% + c%` verursacht zum Beispiel einen Overflow, 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.

Außerdem kann ein Overflow auftreten, wenn Sie in einer `ISAM`-Datenbank, die schon 28 Indizes enthält, einen neuen Index erstellen, oder wenn Sie einen Index erstellen, bei dem die Summe der Längen der indizierten Felder 255 übersteigt.

### **Overflow in numeric constant (CT)**

Sie haben eine zu große Zahl im Programm benutzt.

### **Overlays incompatible with /PACKCODE (RT)**

Overlays können nicht benutzt werden, wenn mit dem Switch `/PACKC` gelinkt wird. Ein Fehler wird jedoch erst während des Programmablaufs, nicht schon beim Linken angezeigt.

### **Parameter type mismatch (CT)**

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 `QBX` den Aufruf einer Prozedur oder Funktion verändern, `QBX` aber bereits eine `DECLARE`-Zeile eingefügt hat. In `QBX` können Sie dann einfach die betreffende `DECLARE`-Zeile löschen, da sie ohnehin neu erstellt wird.

### **Path not found (RT)**

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.

### **Path/File access error (RT oder QB)**

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.

### **Permission denied (RT)**

Sie haben versucht, eine Read Only-Datei zu überschreiben, eine Datei zu benutzen, die vom Netzwerksystem gesperrt wurde, oder (unter `OS/2`) eine Speicherstelle anzusprechen, auf die Ihr Programm keinen Zugriff hat.

### **Procedure already defined in Quick library (CT)**

Sie können in QBX keine Prozedur eingeben, wenn unter gleichem Namen bereits eine Prozedur in einer geladenen Quick Library existiert.

### **Program-memory overflow (CT bei BC)**

Ein einzelnes Modul darf nicht größer als 64 KB sein (QBX 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.

### **Read error on standard input (CT)**

Wenn der Compiler keine Daten von der Tastatur beziehungsweise, wenn die Eingabe mit dem <-Zeichen umgeleitet ist, von der angegebenen Datei einlesen kann, tritt dieser Fehler auf. In Zusammenhang mit der PWB kann dieser Fehler auftreten, wenn die PWB eine Befehlszeile für den Compiler generiert, die länger als 124 Zeichen ist. Da die PWB stets die Browse-Dateinamen, den Objektdatei- und den BASIC-Dateinamen mit Pfadangabe ausschreibt, kann das schon der Fall sein, wenn Sie mit einem 15 Zeichen langen Directorynamen arbeiten. Sie müssen Ihre Programmliste neu erstellen und zuvor das Build-Directory ändern.

### **Record/string assignment required (CT)**

Sie haben in LSET eine Variable von numerischem Datentyp angegeben.

### **Record variable required (CT)**

Sie haben einen INSERT-, RETRIEVE- oder UPDATE-Befehl benutzt, ohne die dazugehörige Zugriffsvariable anzugeben.

### **Redo from start (RT)**

Auf einen INPUT-Befehl hin wurden eine falsche Anzahl oder falsche Typen von Variablenwerten eingegeben. Die Eingabe muß wiederholt werden.

### **Rename across disks (RT)**

Die beiden Dateinamen, die mit dem NAME-Befehl angegeben werden, müssen auf dem gleichen Laufwerk liegen.

### **Requires DOS 2.10 or later (RT)**

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.

### **RESUME without /X on command line (CT)**

Ein Programm, daß 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.

**Skipping forward to END TYPE statement (CT)**

Wegen eines Fehlers in TYPE...END TYPE wird alles von der angezeigten Zeile bis zum END TYPE ignoriert.

**Statement cannot occur within INCLUDE file (CT)**

Eine Prozedur- oder Funktionsdefinition mit SUB oder FUNCTION darf nicht in einem Include-File stehen. Sie muß direkt in das Programm eingefügt werden.

**Statement cannot precede SUB/FUNCTION definition (QB)**

In einem Fenster, das für eine Prozedur geöffnet wurde, dürfen vor dem SUB-beziehungsweise FUNCTION-Befehl in QBX nur REM- oder DEFxxx-Befehle stehen.

**Statement ignored (CT)**

TRON und TROFF werden ignoriert, wenn Sie nicht /D angeben.

**Statement illegal in TYPE block (CT)**

Außer REM dürfen innerhalb von TYPE...END TYPE keine Befehle benutzt werden.

**Statement unrecognizable (CT)**

Ein Wort in Ihrem Programm kann nicht als Befehl identifiziert werden, kann jedoch auch keine Variable sein.

**Statements/labels illegal between SELECT CASE and CASE (CT)**

Außer REM ist kein Befehl und auch keine Zeilennummer zwischen SELECT CASE und der ersten CASE-Zeile erlaubt.

**STOP in module xxxx at address yyyy (RT)**

Ein STOP-Befehl hat Ihr Programm beendet.

**String assignment required (CT)**

Sie haben einen RSET-Befehl falsch angewendet. Er kann nur mit Strings benutzt werden.

**String constant required for ALIAS (CT)**

In einer DECLARE-Anweisung haben Sie hinter ALIAS keinen Alias-Namen (in Anführungszeichen) angegeben.

**String expression required (CT)**

An der angegebenen Stelle wird ein String oder ein Stringausdruck erwartet.

**String formula too complex (RT)**

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.

### **String space corrupt (RT oder QB)**

Mein persönlicher Lieblingsfehler. 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, dessen Ursache meist schwer zu finden ist. Wenn er in QBX auftritt, verabschiedet sich QBX mitsamt allen geladenen Programmen ins DOS, und man sollte neu booten, bevor man QBX erneut aufruft. Tritt er in einem kompilierten Programm auf, stürzt der Rechner entweder postwendend ab, oder es erfolgt die Rückkehr ins Betriebssystem.

"String space corrupt" 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; ein POKE-Befehl kann ebenfalls daran schuld sein, wenn er wichtige Bytes im Stringspeicher getroffen hat. Häufige Ursache sind auch COMMON-Blöcke, die nicht genau übereinstimmen.

Siehe auch "Unerklärliche Systemfehler" weiter unten in diesem Kapitel.

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.

### **String variable required (CT)**

An der angegebenen Stelle wird ein String erwartet.

### **SUB/FUNCTION without END SUB/END FUNCTION (CT)**

Bei einer Prozedur oder Funktion fehlt der END SUB- beziehungsweise END FUNCTION-Befehl. In QBX tritt diese Meldung manchmal ohne ersichtlichen Grund auf (d.h., das Programm ist völlig in Ordnung, trotzdem beharrt QBX darauf, es fehle ein END SUB). Dann ist es ratsam, das Programm als Textdatei zu speichern, QBX zu verlassen und neu aufzurufen.

### **Subprogram error (CT)**

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.).

### **Subprograms not allowed in control statements (CT)**

Sie können eine Prozedur (SUB, FUNCTION, DEF FN) nicht innerhalb einer Kontrollstruktur wie DO...LOOP, IF...THEN...ELSE etc. benutzen.

**Subscript out of range (RT oder CT)**

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). QBX 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.

**Subscript syntax illegal (CT)**

Sie haben einen ungültigen Ausdruck als Array-Index angegeben.

**Syntax error (CT und RT)**

Als CT-Fehler wird diese Meldung von falsch eingetippten BASIC-Befehlen oder von Prozeduraufrufen, denen die zugehörige Prozedur abhanden gekommen ist, verursacht.

Als RT-Fehler wird er generiert, wenn Sie mit DATA in eine numerische Variable eine Stringkonstante einzulesen versuchen, oder wenn die Anzahl der Variablen bei einem ISAM-SEEK-Befehl die Anzahl der indizierten Felder übersteigt.

**Table not found (RT)**

Sie haben mittels DELETETABLE eine Datenbank löschen wollen, die in der genannten ISAM-Datei gar nicht vorhanden ist.

**THEN missing (CT)**

In einer IF-Konstruktion haben Sie das THEN vergessen.

**TO missing (CT)**

Sie haben in einer FOR...NEXT-Anweisung das TO vergessen (oder versehentlich einen Buchstaben angehängt o.ä., so daß BASIC Ihr TO für eine Variable hält).

**Too many arguments in function call (BC)**

Sie haben versucht, einer Prozedur oder Funktion mehr als 60 verschiedene Variablen als Parameter zu übergeben.

**Too many dimensions (CT)**

Ein Array darf maximal 60 Dimensionen haben.

**Too many files (RT oder CT)**

Als Fehler beim Kompilieren tritt diese Meldung nur auf, wenn Sie Include-Files auf mehr als fünf Ebenen verschachteln.

Als RT-Fehler wird "Too many files" 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 Referenzteil) oder wenn Sie zu viele ISAM-Datenbanken gleichzeitig öffnen wollen (siehe "Anzahl gleichzeitig geöffneter ISAM-Dateien und -Datenbanken" in Kapitel 7.7).

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.

### **Too many labels (CT)**

In einer ON...GOTO- oder ON...GOSUB-Konstruktion können nicht mehr als 60 Zeilennummern oder -labels angegeben werden.

### **Too many local string variables in procedure (CT)**

Eine Prozedur oder Funktion darf maximal 255 lokale Stringvariablen haben. Sie können das Problem umgehen, indem Sie die Prozedur globale Variablen benutzen lassen.

### **Too many named COMMON blocks (CT)**

Ein Programm darf nur maximal 126 verschiedene COMMON-Blocks, die mit Blocknamen versehen sind, enthalten.

### **Too many TYPE definitions (CT)**

Es dürfen maximal 240 TYPE-Definitionen in einem Programm enthalten sein.

### **Too many variables for INPUT (CT)**

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 *String formula too complex*-Fehler auf).

### **Type mismatch (CT oder RT)**

Als RT-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 CT-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.

### **Type more than 65535 bytes (CT)**

Eine TYPE-Definition darf nicht mehr als 65.535 Bytes umfassen.

### **Type not defined (CT)**

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.

### **Type not valid for ISAM open (CT)**

Der Datentyp, den Sie bei einer ISAM-Open-Anweisung benutzen, ist entweder nicht mit TYPE...END TYPE definiert (die eingebauten Typen INTEGER, LONG etc. sind nicht erlaubt), seine Name ist länger als 30 Zeichen, oder er enthält ein Feld vom Typ SINGLE. Dieser Typ ist in ISAM nicht erlaubt (siehe "Datentyp SINGLE" in Kapitel 7.7).

### **TYPE statement improperly nested (CT)**

In Prozeduren darf kein TYPE...END TYPE angewendet werden.

### **TYPE without END TYPE (CT)**

Zu einem TYPE fehlt das END TYPE, oder das END TYPE wurde nicht als solches erkannt. (Haben Sie es vielleicht als ein Wort geschrieben?)

### **Typed variable not allowed in expression (CT)**

In Ausdrücken sind keine Variablen von selbstdefiniertem Typ erlaubt. PRINT *z* funktioniert zum Beispiel nicht, wenn *z* eine Variable selbstdefinierten Typs ist.

### **Unprintable error (RT)**

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.

### **Variable-length string required (CT)**

Sie haben in einem FIELD-Befehl einen String fester Länge angegeben.

### **Variable name not unique (CT)**

Sie definieren einen Variablennamen, der schon benutzt wird oder der einen Punkt enthält und dessen Vor-Punkt-Anteil der Name einer Variable selbstdefinierten Typs ist.

### **Variable required (CT oder RT)**

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 RT-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.



## **WEND without WHILE (CT)**

## **WHILE without WEND (CT)**

Zu einem WHILE-Befehl fehlt das WEND oder umgekehrt. Siehe auch "Unerklärliche Strukturfehler" weiter unten in diesem Kapitel.

# **Unerklärliche Systemfehler**

Wenn Fehler auftreten, deren Ursache Sie nicht feststellen können, und wenn es sich dann auch noch um Fehler wie "String space corrupt" oder "No line number" handelt, ist es sehr wahrscheinlich, daß der Fehler schon verursacht wird, lange bevor er ans Licht tritt. Grundsätzlich kann ziemlich jeder RT-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 QBX 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".

# **Unerklärliche Strukturfehler**

Strukturfehler wie "NEXT without 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.

## C.3 LINK-Fehler (Auswahl)

Hier folgt eine Auswahl von LINK-Fehlermeldungen, die mit BASIC auftreten können und die *nicht* für sich selbst sprechen.

### **Cannot open module-definitions file**

Die angegebene Definitionsdatei konnte nicht geöffnet werden. Vielleicht haben Sie sich in einer Steuerungsdatei mit den Zeilen verzählt, so daß der Linker das, was Sie als EXE-Dateinamen gemeint haben, als Namen einer Definitionsdatei versteht?

### **Cannot open response file**

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.

### **Cannot create temporary file**

### **Cannot open temporary file**

Für die temporäre Datei, die LINK anlegen wollte, war nicht genügend Platz auf der Platte.

### **Cannot open run file**

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.

### **File not suitable for /EXEPACK, relink without**

Das mit /E komprimierte File ist länger, als es das nichtkomprimierte wäre.

### **Fixup overflow**

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).

### **Invalid object module**

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.

### **Name of output file is...**

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.

### **Out of space for run file**

Das EXE-file konnte nicht erzeugt werden, weil die Diskette/Platte voll ist.

### **Quick Library support module missing**

Wenn Sie Quick Libraries erzeugen wollen, müssen Sie die Library QBXQLB.LIB mit angeben.

### **Requested segment limit too high**

Es ist nicht genügend Speicherplatz vorhanden, um die mit /SE angegebene maximale Segmentzahl zu verwalten.

### **Response line too long**

Eine Zeile in der mit @ eingeleiteten Steuerungsdatei war länger als 255 Zeichen.

### **Symbol defined more than once**

#### **Symbol multiply defined**

In verschiedenen angegebenen OBJ-Files oder Libraries ist dieselbe Prozedur oder Funktion mehrfach definiert. Dieser Fehler kann auch auftreten, wenn Sie falsche Libraries benutzen, oder wenn Sie Verzicht-Files benutzen, ohne /NOE anzugeben. 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 verschiedenen Compiler-Switches kompiliert wurden.

### **Too many libraries**

Es dürfen nicht mehr als 32 Libraries angegeben werden. Notfalls können Sie aus mehreren Libraries mittels des Programms LIB.EXE eine einzige Library erstellen.

### **Too many overlays**

Ein Programm darf maximal 63 Overlays haben.

### **Too many segments**

Benutzen Sie den Switch /SE, um die erlaubte Segmentzahl höher als den Defaultwert 128 einzustellen.

### **Unexpected end-of-file**

Eine Library hat ein ungültiges Format. Erzeugen Sie sie neu, oder prüfen Sie sie zunächst mit LIB.

### **Unexpected end-of-file on scratch file**

Sie haben eine Diskette entfernt, auf die der Linker eine temporäre Datei geschrieben hatte.

**Unresolved external**

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 genannten Routinen enthält.

## C.4 LIB-Fehler (Auswahl)

### **Cannot open response file**

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.

### **Module not in library; ignored**

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.

### **Module redefinition ignored**

Wenn Sie eine OBJ-Datei in die Library aufnehmen wollen, die bereits enthalten ist, wird der Befehl ignoriert.

### **Page size too small**

Sie haben mit /P eine ungültige Seitengröße angegeben.

### **Symbol defined in module xxx, redefinition ignored**

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.

## C.5 Fehlercodes der Toolboxes

Die Presentation Graphics- 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.

### Presentation Graphics

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 ungültigen 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	BASIC-Fehler	Ein gewöhnlicher BASIC-Fehler (beispielsweise 7 = Out of memory) ist aufgetreten; zu der Fehlernummer wird 200 addiert, also wäre ChartErr bei "Out of memory" 207.

# Fonts

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 registrieren 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 (versuchen Sie's mit Far Strings, wenn Sie es noch nicht probiert haben).
>200	BASIC-Fehler	Ein gewöhnlicher BASIC-Fehler (beispielsweise 7 = Out of memory) ist aufgetreten; zu der Fehlernummer wird 200 addiert, also wäre FontErr bei "Out of memory" 207.

# Anhang D:

## Tastatur- und Zeichencodes

### D.1 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 Tastaturteiler ordnet der Taste verschiedene Zeichen zu, und die Hersteller schreiben einen anderen Buchstaben drauf.

Taste	Scancode	Taste	Scancode
ESC	1	? ß \	12
F1	59	' `	13
F2	60	Q q	16
F3	61	W w	17
F4	62	E e	18
F5	63	R r	19
F6	64	T t	20
F7	65	Z z	21
F8	66	U u	22
F9	67	I i	23
F10	68	O o	24
F11	87	P p	25
F12	88	Ü ü	26
		* + ~	27
◦ ^	41		
1 ! <sup>2</sup>	2	A a	30
2 " <sup>3</sup>	3	S s	31
3 §	4	D d	32
4 \$	5	F f	33
5 %	6	G g	34
6 &	7	H h	35
7 /	8	J j	36
8 (	9	K k	37
9 )	10	L l	38
0 =	11	Ö ö	39

(Fortsetzung nächste Seite)



(Fortsetzung)

<b>Taste</b>	<b>Scancode</b>	<b>Taste</b>	<b>Scancode</b>
Ä ä	40	Insert 0	82
# ' `	41	Delete ,	83
< >	43	Home 7	71
Y y	44	End 1	79
X x	45	PgUp 9	73
C c	46	PgDn 3	81
V v	47	Pfeil auf 8	72
B b	48	Pfeil ab 2	80
N n	49	Pfeil links 4	75
M m	50	Pfeil rechts 6	77
, ;	51	5 Num.block	76
. :	52	- Num.block	74
- _	53	+ Num.block	78
Leertaste	57	÷ Num.block	53
Tab	15	* Num.block	55
Caps Lock	58		
Shift links	42	* PrtSc	55
CTRL	29	Num Lock	69
ALT, ALT GR	56	Scroll Lock	70
Shift rechts	54		
Backspace	14		
Enter	28		

## D.2 ASCII-Codes der verschiedenen Tastenkombinationen

Angegeben sind zu allen Tasten die Codes, die zurückgegeben werden, wenn man die Taste "pur", mit Shift, mit CTRL oder mit ALT drückt. Dabei sind sogenannte *erweiterte* Codes kursiv angegeben. Erweiterten ASCII-Codes geht ein CHR\$(0)-Zeichen voran; wenn sie mit INKEY\$ eingelesen werden, befinden sich 2 Zeichen im betreffenden String.

Taste	pur	Shift	CTRL	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
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

(Fortsetzung nächste Seite)

(Fortsetzung)

<b>Taste</b>	<b>pur</b>	<b>Shift</b>	<b>CTRL</b>	<b>ALT</b>
ESC	27	27	27	-
Tab	9	15	148	165
Enter	13	13	10	28
Leertaste	32	32	32	32
Backspace	8	8	127	14

### **Weiße Tasten auf dem Nummernblock:**

(Wenn Num Lock an ist, sind die Spalten für "pur" und "Shift" vertauscht.)

<b>Taste</b>	<b>pur</b>	<b>Shift</b>	<b>CTRL</b>	<b>ALT</b>
Home	71	55	119	-
Pfeil auf	72	56	141	-
PgUp	73	57	132	-
Pfeil links	75	52	115	-
Mitte	-	53	143	-
Pfeil rechts	77	54	116	-
End	79	49	117	-
Pfeil ab	80	50	145	-
PgDn	81	51	118	-
Insert	82	48	146	-
Delete	83	44	147	-

### **Graue Tasten auf/neben dem Nummernblock:**

<b>Taste</b>	<b>pur</b>	<b>Shift</b>	<b>CTRL</b>	<b>ALT</b>
Home	71	71	119	151
Pfeil auf	72	72	141	152
PgUp	73	73	132	153
Pfeil links	75	75	115	155
Pfeil rechts	77	77	116	156
End	79	79	117	59
Pfeil ab	80	80	145	160
PgDn	81	81	118	161
Insert	82	82	146	162
Delete	83	83	147	163
Enter	13	13	10	166
Druck	-	-	114	-
+	43	43	144	78
-	45	45	142	74
x	42	42	150	55
÷	47	47	149	164

Die folgenden Tasten können auf verschiedenen Tastaturen unterschiedlich belegt sein; die Werte sind deshalb nur für den deutschen Tastaturtreiber gültig. Selbst hier gibt es Einschränkungen, weil insbesondere die Tasten "^ °" und "# " häufig vertauscht und/oder verschieden belegt werden.

<b>Taste</b>	<b>pur</b>	<b>Shift</b>	<b>CTRL</b>	<b>ALT</b>
^ °	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	-	-
< >	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

## Äquivalente Tastenbezeichnungen

Die Rückschritt-Taste heißt auch Backspace und ist zumeist grau und mit einem längeren Pfeil nach links beschriftet.

Die CTRL-Taste heißt auf deutschen Tastaturen "Strg". Delete (Del) wurde zu "Entf" oder "Löschen", Insert (Ins) zu "Einfg", Home zu "Pos 1", PgUp zu "Bild hoch", PgDn zu "Bild tief", Num Lock zu "Num" und "Scroll Lock" zu "Roll". Die PrtSc-Taste heißt häufig "Druck", und die Break-Taste ist mit "Pause/Untbr" beschriftet.

## D.3 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 Ä Ö Ü ä ö ü ß 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 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 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 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.

Bei Bildschirmen hängt die Anzeige der oberen 128 ASCII-Zeichen (Zeichen-codes ab 128) von der im Betriebssystem des jeweiligen Computers eingestellten sogenannten "Codeseite" ab.

Unter MS-DOS und Windows 3.1x ist in den westlichen Ländern meistens die Codeseite 437 ("Englisch") standardmäßig voreingestellt, bei Windows 95 und neuer meist die Codeseite 850 ("Mehrsprachig/Lateinisch"). Durch die Codepage 850 werden leider einige ASCII-Zeichen nicht mehr richtig angezeigt, z.B. die Kreuzungs-Symbole zwischen Doppel- und Einfach-Linien.

Im folgenden finden Sie die ASCII-Tabellen für die Codeseiten 437 und 850.

Reguläre ASCII-Zeichen <Zeichencodes 0 - 127>															
000	(nul)	016	<dle>	032	sp	048	0	064	@	080	P	096	`	112	p
001	(soh)	017	<dc1>	033	!	049	1	065	A	081	Q	097	a	113	q
002	(stx)	018	<dc2>	034	"	050	2	066	B	082	R	098	b	114	r
003	(etx)	019	!!	035	#	051	3	067	C	083	S	099	c	115	s
004	(eot)	020	¶	036	\$	052	4	068	D	084	T	100	d	116	t
005	(enq)	021	<nak>	037	%	053	5	069	E	085	U	101	e	117	u
006	(ack)	022	<syn>	038	&	054	6	070	F	086	V	102	f	118	v
007	(bel)	023	<etb>	039	'	055	7	071	G	087	W	103	g	119	w
008	(bs)	024	<can>	040	<	056	8	072	H	088	X	104	h	120	x
009	(tab)	025	<em>	041	>	057	9	073	I	089	Y	105	i	121	y
010	(lf)	026	(eof)	042	*	058	:	074	J	090	Z	106	j	122	z
011	(vt)	027	<esc>	043	+	059	;	075	K	091	[	107	k	123	<
012	(np)	028	(fs)	044	,	060	<	076	L	092	\	108	l	124	!
013	(cr)	029	<gs>	045	-	061	=	077	M	093	]	109	m	125	>
014	(so)	030	<rs>	046	.	062	>	078	N	094	^	110	n	126	~
015	(si)	031	<us>	047	/	063	?	079	O	095	_	111	o	127	Δ

Erweiterte ASCII-Zeichen <Zeichencodes 128 - 255>																	
128	Ç	143	Ñ	158	Ñ	172	¼	186	¶	200	ß	214	¶	228	Σ	242	λ
129	Ü	144	É	159	Ñ	173	½	187	¶	201	¶	215	¶	229	σ	243	≤
130	é	145	æ	160	á	174	«	188	¶	202	¶	216	¶	230	μ	244	∫
131	à	146	ð	161	í	175	»	189	¶	203	¶	217	¶	231	γ	245	∫
132	ä	147	ô	162	ó	176	¶	190	¶	204	¶	218	¶	232	ø	246	÷
133	å	148	ö	163	ú	177	¶	191	¶	205	¶	219	¶	233	ø	247	≈
134	ä	149	ò	164	ñ	178	¶	192	¶	206	¶	220	¶	234	Ω	248	°
135	ã	150	û	165	ñ	179	¶	193	¶	207	¶	221	¶	235	ö	249	·
136	ã	151	ü	166	ñ	180	¶	194	¶	208	¶	222	¶	236	∞	250	·
137	ë	152	ÿ	167	ñ	181	¶	195	¶	209	¶	223	¶	237	∞	251	√
138	è	153	0	168	ì	182	¶	196	¶	210	¶	224	α	238	€	252	√
139	ì	154	ü	169	í	183	¶	197	¶	211	¶	225	ø	239	π	253	√
140	î	155	ç	170	í	184	¶	198	¶	212	¶	226	Γ	240	∞	254	■
141	î	156	ç	171	½	185	¶	199	¶	213	F	227	¶	241	±	255	
142	ä	157	ÿ														

ASCII-Tabelle für die Codeseite 437 (MS-DOS und Windows 3.1)

Reguläre ASCII-Zeichen (Zeichencodes 0 - 127)															
000	<nul>	016	<dle>	032	sp	048	0	064	@	080	P	096	`	112	p
001	<soh>	017	<dc1>	033	!	049	1	065	A	081	Q	097	a	113	q
002	<stx>	018	<dc2>	034	"	050	2	066	B	082	R	098	b	114	r
003	<etx>	019	<dc3>	035	#	051	3	067	C	083	S	099	c	115	s
004	<eot>	020	<dc4>	036	\$	052	4	068	D	084	T	100	d	116	t
005	<eng>	021	<nak>	037	%	053	5	069	E	085	U	101	e	117	u
006	<ack>	022	<syn>	038	&	054	6	070	F	086	V	102	f	118	v
007	<bel>	023	<eth>	039	'	055	7	071	G	087	W	103	g	119	w
008	<bs>	024	<can>	040	<	056	8	072	H	088	X	104	h	120	x
009	<tab>	025	<em>	041	>	057	9	073	I	089	Y	105	i	121	y
010	<lf>	026	<eof>	042	*	058	:	074	J	090	Z	106	j	122	z
011	<vt>	027	<esc>	043	+	059	;	075	K	091	[	107	k	123	<
012	<np>	028	<fs>	044	,	060	<	076	L	092	\	108	l	124	!
013	<cr>	029	<gs>	045	-	061	=	077	M	093	]	109	m	125	>
014	<so>	030	<rs>	046	.	062	>	078	N	094	^	110	n	126	~
015	<si>	031	<us>	047	/	063	?	079	O	095	_	111	o	127	Δ

Erweiterte ASCII-Zeichen (Zeichencodes 128 - 255)															
128	Ç	143	Ë	158	×	172	¼	186		200	ˆ	214	í	228	õ
129	ü	144	É	159	ƒ	173	½	187	]]	201	ı	215	î	229	ö
130	é	145	æ	160	á	174	«	188	]]	202	ı	216	ï	230	µ
131	â	146	æ	161	í	175	»	189	ç	203	ı	217	ı	231	ı
132	à	147	ô	162	ó	176	ı	190	ı	204	ı	218	ı	232	ı
133	á	148	ü	163	ú	177	ı	191	ı	205	ı	219	ı	233	ı
134	ä	149	ò	164	ñ	178	ı	192	ı	206	ı	220	ı	234	ı
135	ç	150	ü	165	ñ	179	ı	193	ı	207	ı	221	ı	235	ı
136	è	151	ü	166	ı	180	ı	194	ı	208	ı	222	ı	236	ı
137	ë	152	ü	167	ı	181	ı	195	ı	209	ı	223	ı	237	ı
138	ê	153	ü	168	ı	182	ı	196	ı	210	ı	224	ı	238	ı
139	ı	154	ü	169	ı	183	ı	197	ı	211	ı	225	ı	239	ı
140	ı	155	ı	170	ı	184	ı	198	ı	212	ı	226	ı	240	ı
141	ı	156	ı	171	ı	185	ı	199	ı	213	ı	227	ı	241	ı
142	ı	157	ı												

ASCII-Tabelle für die Codeseite 850 (Windows 95 und höher)





## D.4 ISAM-Sortiertabellen

### Tabelle für Deutsch, Französisch, Englisch, Italienisch und Portugiesisch

A, Ä, Å, a, à, á, â, ä, å

Æ und æ werden wie ae einsortiert

B, b

C, Ç, c, ç

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

M, m

N, Ñ, n, 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, ŷ

Z, z

# Tabelle für Finnisch, Isländisch, Dänisch, Norwegisch und Schwedisch

A, a, à, á, â

B, b

C, Ç, c, ç

D, d

E, É, e, è, é, ê, ë

F, f

G, g

H, h

I, i, î, í, î

J, j

K, k

L, l

M, m

N, Ñ, n, ñ

O, o, ò, ó, ô

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, ŷ

Z, z

Æ, æ

Ý

Å, å

Ä, ä

Ö, ö

# Tabelle für Spanisch

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, î, 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, ŷ

Z, z

# Tabelle für Holländisch

Entspricht der Tabelle für Deutsch, Französisch, Englisch, Italienisch und Portugiesisch, mit dem einzigen Unterschied, daß ŷ nicht nach y, sondern wie ij einsortiert wird.

# Anhang E: Ausgewählte Interrupts

Noch einige Hinweise zum Aufruf der Interrupts und zur Auswertung der zurückgegebenen Daten:

In der folgenden Liste finden Sie zu jedem Interrupt die Ein- und Ausgabewerte. Ein Interrupt-Aufruf sieht im allgemeinen etwa so aus:

- \_ Registervariablen definieren
- \_ Eingaberegister setzen
- \_ Interrupt aufrufen
- \_ Ausgaberegister auslesen

Für die Definition der Registervariablen können Sie die Typen `RegType` oder `RegTypeX` verwenden; `RegTypeX` unterstützt zusätzlich die Register `DS` und `ES`, die für manche Interrupts gebraucht werden.

Die Registervariablen haben die 16-Bit-Register `AX`, `BX`, `CX` und `DX`; trotzdem wird in vielen Beschreibungen zwischen dem H- und dem L-Register unterschieden. Wenn unter "Eingabe" zum Beispiel steht "`AL = 07h`, `AH = 22h`", müssen Sie daraus erst ein `AX`-Register basteln. Eine Möglichkeit dazu ist im Abschnitt 2 des Kapitels 16, "DOS-Interrupts und ihre Nutzung", geschildert; mit Hexadezimalzahlen geht das aber noch einfacher: Sie müssen nur die Werte für `AH` und `AL` hintereinanderschreiben (zuerst `AH`, dann `AL`).

Im genannten Beispiel wäre die eine Möglichkeit:

```
Reg.AX = 256 * &h22 + &h07
```

und die andere:

```
Reg.AX = &h2207
```

Wie Sie sehen, ist dieses Hintereinanderschreiben zumindest für die Eingabewerte extrem praktisch. Deshalb sind im folgenden auch fast alle Zahlen als Hexadezimalzahlen angegeben. Ich verwende dafür das Postfix `h`; `20h` bedeutet bei mir also dezimal 32. In BASIC müssen Sie, wie schon die Beispiele zeigen, stattdessen für Hexadezimalzahlen das Präfix `&h` benutzen, also `&h20` für dezimal 32.

Der Aufruf eines Interrupts funktioniert mit der Prozedur `INTERRUPT` oder `INTERRUPTX`:

```
INTERRUPT interruptnummer, eingaberegister, ausgaberegister
```

In den meisten Fällen können Sie für *eingaberegister* die gleiche Variable wie für *ausgaberegister* benutzen (wie auch in allen Beispielen hier im Buch), da die

Eingaberegister nicht mehr interessant sind, wenn der Interrupt aufgerufen wurde. Die Ausgabe des Interrupts wird dann in dieselbe Variable geschrieben, in die Sie Ihre Eingabe plazierte haben.

In einigen Interruptbeschreibungen ist die Rede vom "Zero-Flag" oder vom "Carry-Flag". Dabei handelt es sich um Bits aus dem Flags-Register. Dem Carry-Flag entspricht der Term `Regs.Flags AND 1` (er ist das erste Bit), während der Zero-Flag durch `Regs.Flags AND 64` repräsentiert werden kann (beide Male vorausgesetzt, daß Ihre Registervariable "Regs" heißt). Mit dem Zero-Flag müssen Sie aufpassen: Wenn im folgenden steht: "Zero-Flag = 1 bedeutet ...", müßten Sie daraus in einem Programm `IF Regs.Flags AND 64 = 64 THEN ...` und nicht `IF Regs.Flags AND 64 = 1 THEN ...` machen.

## Schreiben eines Zeichens mit Farbe

Bildschirm

### Interrupt 10h, Funktion 09h

*Eingabe* AH = 09h; BH = Bildschirmseite; CX = Anzahl der (gleichen) Zeichen, die geschrieben werden sollen; AL = ASCII-Code für das Zeichen; BL = Farbattribut

*Kommentar* Das Zeichen wird an der Cursorposition auf den Bildschirm geschrieben. Das Farbattribut errechnet sich nach dem üblichen Schlüssel (siehe Eintrag zu `AttrBox` im Referenzteil der User Interface/General-Toolbox). Im Grafikmodus müssen, wenn CX größer als 1 ist, alle Zeichen in die aktuelle Zeile passen. Der Cursor wird nicht verschoben.

Dieser Interrupt bringt Geschwindigkeitsvorteile gegenüber `PRINT`, wenn ein Zeichen mehrmals hintereinander geschrieben werden soll (Rahmen etc).

## Schreiben eines Zeichens ohne Farbe

Bildschirm

### Interrupt 10h, Funktion 0Ah

*Eingabe* AH = 0Ah; sonst wie 10h.09h bis auf BL (fällt weg)

*Kommentar* Diese Funktion entspricht 10h.09h; hier wird das Zeichen allerdings in der Farbe dargestellt, die der Bildschirm an der Cursorposition schon hat. Die Farb-Bytes werden hierbei nicht geändert, so daß das Schreiben in einen andersfarbigen Bereich auch verschiedene Farben hervorbringt.

## Interrupt 10h, Funktion 0Fh

*Eingabe* AH = 0Fh

*Ausgabe* AL = Videomodus; AH = Anzahl der Zeichen pro Zeile; BH = Nummer der aktuellen Bildschirmseite

*Kommentar* Vor allem die Anzahl der Zeichen pro Zeile und Bildschirmseite sind von Interesse. Die Aufschlüsselung des Videomodus ist folgende:

AL-Nummer	entsprechender Bildschirm-Modus
0	SCREEN 0, WIDTH 40 auf Monochromkarte
1	SCREEN 0, WIDTH 40 auf Farbkarte
2	SCREEN 0, WIDTH 80 auf Monochromkarte
3	SCREEN 0, WIDTH 80 auf Farbkarte
4	SCREEN 1
5	wie 4, jedoch schwarz/weiß-Darstellung der Farben (mit SCREEN nicht einstellbar)
6	SCREEN 2
7	wie 6, jedoch schwarz/weiß-Darstellung der Farben (mit SCREEN nicht einstellbar)
8	SCREEN 3
13	SCREEN 7
14	SCREEN 8
15	SCREEN 9
16	SCREEN 10
17	SCREEN 11
18	SCREEN 12
19	SCREEN 13
64	SCREEN 4

Weitere Modi, beispielsweise hochauflösende VGA-Modi, sind bei speziellen Grafikkarten möglich. Mit Interrupts können Sie den Grafikmodus zwar auch *setzen*, aber es hat keinen Sinn, zum Beispiel einen Spezialmodus einer VGA-Karte zu setzen, der mit SCREEN nicht einstellbar ist, weil dann die Grafikbefehle höchstwahrscheinlich nicht funktionieren.

## Interrupt 11h

*Ausgabe* AX = Konfigurationsbyte

*Kommentar* Das Konfigurationsbyte wird wie folgt entschlüsselt (Sie können die Funktion TeilZahl aus dem Listing INHALT.BAS in Kapitel 14 benutzen, um AX korrekt zu zerlegen).

### -für ATs/386er/486er

Bit 1	ist 1, wenn das System Diskettenlaufwerke hat
Bit 2	ist 1, wenn ein Coprozessor vorhanden ist
Bits 5-6	Bildschirmmodus beim Hochfahren des Systems (0 = unbekannt, 1 = 40*25 mit Color-Karte; 2 = 80*25 mit Color-Karte; 3 = 80*25 mit Mono-Karte)
Bits 7-8	Anzahl der Diskettenlaufwerke (nur, wenn Bit 1 = 1 ist): 0 = 1 Laufwerk, 1 = 2 Laufwerke usw.
Bits 10-12	Anzahl der RS232-Karten (COM-Schnittstellen)
Bits 15-16	Anzahl der Druckerschnittstellen (LPTx)

### -für PCs und XTs

Bit 1	ist 1, wenn das System Diskettenlaufwerke hat
Bits 5-6	Bildschirmmodus beim Hochfahren des Systems (0 = unbekannt, 1 = 40*25 mit Color-Karte; 2 = 80*25 mit Color-Karte; 3 = 80*25 mit Mono-Karte)
Bits 7-8	Anzahl der Diskettenlaufwerke (nur, wenn Bit 1 = 1 ist): 0 = 1 Laufwerk, 1 = 2 Laufwerke usw.
Bit 9	ist 0, wenn der Rechner keinen DMA-Chip hat
Bits 10-12	Anzahl der RS232-Karten (COM-Schnittstellen)
Bit 13	ist 1, wenn ein Spieleadapter (Joystick) vorhanden ist
Bits 15-16	Anzahl der Druckerschnittstellen (LPTx)

# Feststellen der Speichergröße vor 1 MB

## Interrupt 12h

*Ausgabe* AX = Speicher vor der 1-MB-Grenze in KB

*Kommentar* Um Speicher über 1 MB ermitteln zu können, wird 15h.88h benutzt; diese Funktion existiert aber nur bei ATs, da PCs ohnehin auf 640 KB beschränkt sind.

## Feststellen der Speichergröße über 1 MB

System

### Interrupt 15h, Funktion 88h

*Eingabe* AH = 88h

*Ausgabe* AX = Größe des Speichers in KB

## Rechner neu booten

System

### Interrupt 19h

*Kommentar* Dieser Interrupt tut genau das, was üblicherweise die Tastenkombination CTRL-ALT-Del auslöst.

## Zeichen holen, ohne es aus dem Puffer zu löschen

Tastatur

### Interrupt 16h, Funktion 01h

*Eingabe* AH = 01h

*Ausgabe* Zero-Flag = 1: Es ist kein Zeichen vorhanden; bei Zero-Flag = 0 gilt: AL ist der ASCII-Code der Taste, AH der Scancode. Ist AL = 0, handelt es sich um einen erweiterten Tastaturcode, der dann in AH anstelle des Scancodes steht.

*Kommentar* Dieser Interrupt kann verwendet werden, wenn man zum Beispiel feststellen möchte, ob ESC gedrückt wurde, ohne aber das Zeichen (wie es mit INKEY\$ geschähe) wirklich aus dem Tastaturpuffer zu entfernen.

## Tastaturstatus erfragen

Tastatur

### Interrupt 16h, Funktion 02h

*Eingabe* AH = 02h

*Ausgabe* AL = Statusbyte

*Kommentar* Das Tastaturstatusbyte wird wie folgt aufgeschlüsselt:

Bit 8	ist 1, wenn Insert-Mode aktiv
Bit 7	ist 1, wenn Caps-Lock an ist
Bit 6	ist 1, wenn Num-Lock an ist
Bit 5	ist 1, wenn Scroll-Lock an ist
Bit 4	ist 1, wenn ALT gedrückt ist
Bit 3	ist 1, wenn CTRL (Strg) gedrückt wird
Bit 2	ist 1, wenn die linke Shift-Taste gedrückt wird
Bit 1	ist 1, wenn die rechte Shift-Taste gedrückt wird



## Druckerstatus erfragen

Drucker

### Interrupt 17h, Funktion 02h

*Eingabe* AH = 02h; DX = Nummer des Druckers (LPT1: ist 0 usw.)

*Ausgabe* AH = Statusbyte

*Kommentar* Das Druckerstatusbyte wird wie folgt aufgeschlüsselt:

AH AND 128	ist 0, wenn der Drucker arbeitet, sonst 128
AH AND 32	ist 32, wenn der Drucker kein Papier hat
AH AND 16	ist 16, wenn der Drucker Online ist
AH AND 1	ist 1, wenn es einen Time-Out-Fehler gab (kein Drucker vorhanden)

## aktuelles Laufwerk, Abfrage Laufwerksanzahl

DOS

### Interrupt 21h, Funktion 0Eh

*Eingabe* AH = 0Eh; DL = Code des Laufwerks (0 = A:, 1 = B: etc.)

*Ausgabe* AL = Anzahl der vorhandenen (logischen) Laufwerke

*Kommentar* Diese Funktion entspricht dem DOS-Befehl <Laufwerk>:. Außerdem wird die Anzahl der logischen Laufwerke zurückgegeben, d.h. eine Festplatte, die in mehrere Laufwerke unterteilt ist, zählt auch mehrfach.

## Informationen über ein Laufwerk einholen

DOS

### Interrupt 21h, Funktion 1Ch

*Eingabe* AH = 1Ch; DL = Laufwerksnummer (0 für das aktuelle Laufwerk, 1 für A:, 2 für B: etc.)

*Ausgabe* AL = Anzahl der Sektoren pro Cluster; DX = Anzahl der Cluster; DS = Segment-, BX = Offset-Adresse des Media-Deskriptors

*Kommentar* Der Media-Deskriptor umfaßt nur ein Byte. Wenn Sie den Aufruf INTERRUPTX 21h, Regs, Regs benutzt haben, können Sie den Media-Deskriptor mit DEF SEG = Regs.DS: Deskriptor% = PEEK(BX) abfragen. Seine Aufschlüsselung:

F8h	Festplatte
1F0h	Diskette 3½ Zoll, 1,44 MB
F9h	Diskette 3½ Zoll, 720 KB oder 5¼ Zoll, 1,2 MB
FDh	Diskette 5¼ Zoll, 360 KB
FCh	Diskette 5¼ Zoll, 180 KB
FEh	Diskette 5¼ Zoll, 160 KB
FFh	Diskette 5¼ Zoll, 320 KB
FAh	Diskette mit 1 Seite, 80 Tracks, 8 Sektoren
FBh	Diskette mit 2 Seiten, 80 Tracks, 8 Sektoren

## Datum erfragen

DOS

### Interrupt 21h, Funktion 2Ah

*Eingabe* AH = 2Ah

*Ausgabe* CX = Jahr, DH = Monat, DL = Tag, AL = Wochentag (0-6, 0 = Sonntag)

*Kommentar* Das Datum an sich ist uninteressant, denn es steht ja über die DATE\$-Funktion zur Verfügung; die Ermittlung des Wochentags geht jedoch über diesen Interrupt weniger umständlich als mit der Date-Toolbox oder eigenen Routinen.

## DOS-Versionsnummer feststellen

DOS

### Interrupt 21h, Funktion 30h

*Eingabe* AH = 30h

*Ausgabe* AL = obere Versionsnummer, AH = zweistellige untere Versionsnummer (bei DOS 4.01 wäre AL = 4, AH = 1; bei DOS 3.1 wäre AL = 3, AH = 10)

## Freie Disketten-/Plattenkapazität feststellen

DOS

### Interrupt 21h, Funktion 36h

*Eingabe* AH = 36h; DL = Laufwerksnummer (0 = aktuelles Laufwerk, 1 = A, 2 = B usw.)

*Ausgabe* AX = -1 (beziehungsweise 65.536, bei Benutzung der Funktion UnsignedInt) wenn Gerät nicht vorhanden, sonst AX \* BX \* CX = freier Speicherplatz in Bytes

*Beispiel* Siehe Listing FREI.BAS in Kapitel 14, "DOS-Interrupts und ihre Nutzung", Abschnitt 4.

## Attribut einer Datei erfragen

DOS

### Interrupt 21h, Funktion 43h, Unterfunktion 0

*Eingabe* AH = 43h; AL = 0; DS = Segmentadresse des Dateinamens; DX = Offsetadresse des Dateinamens

*Ausgabe* CX = Dateiattribut (wenn Carry-Flag = 1: Fehler aufgetreten)

*Kommentar* Die Aufschlüsselung des Dateiattributs lautet wie folgt:

Bit 1	ist 1, wenn Read-Only-Attribut gesetzt
Bit 2	ist 1, wenn Hidden-Attribut gesetzt
Bit 3	ist 1, wenn System-Attribut gesetzt
Bit 4	ist 1, wenn Volume-Label-Attribut gesetzt (Volume Label = Datenträgerbezeichnung)
Bit 5	ist 1, wenn es sich um ein Directory und nicht um eine Datei handelt
Bit 6	ist 1, wenn Archive-Attribut gesetzt

## Attribut einer Datei setzen

DOS

### Interrupt 21h, Funktion 43h, Unterfunktion 1

*Eingabe* AH = 43h; AL = 1; DS = Segmentadresse des Dateinamens; DX = Offsetadresse des Dateinamens; CX = Dateiattribut

*Ausgabe* wenn Carry-Flag = 1: Fehler aufgetreten

*Kommentar* Die Aufschlüsselung des Dateiattributs ist bei 21h.43h.0 beschrieben; diese Funktion darf nicht benutzt werden, um Volume Label oder Directory-Attribut zu verändern

## Setzen der DTA-Adresse

DOS

### Interrupt 21h, Funktion 1Ah

*Eingabe* AH = 1Ah; DS = Segmentadresse der neuen DTA; DX = Offsetadresse der neuen DTA (die DTA, die Disk Transfer Area, wird für andere Interrupts gebraucht).

*Kommentar* Wenn Sie Interrupts benutzen, die Daten in der DTA zurückgeben (beispielsweise die beiden folgenden), müssen Sie zuvor einen String definieren, der lang genug ist, um die Daten aufnehmen zu können, und dann mit Hilfe dieses Interrupts die DTA-Adresse auf die Adresse dieses Strings setzen. Betrachten Sie dazu das Beispiel.

*Beispiel* Siehe Listing INHALT.BAS in Kapitel 14, Abschnitt 4.

**Interrupt 21h, Funktion 4Eh**

*Eingabe* AH = 4Eh; CX = Dateiattribut (siehe 21h.43h.0); DS = Segment-,  
DX = Offsetadresse des Dateinamens

*Ausgabe* Carry-Flag = 1: Keine Datei gefunden

*Kommentar* Mit DX = 0 werden nur alle Dateien gesucht, bei denen kein Attribut gesetzt ist. Mit DX = 1 würden zusätzlich alle mit Read-Only-attribut gesucht; bei DX = 16 zusätzlich alle Directories etc., so daß man mit DX = 39 alle Dateien findet und mit DX = 55 alle Dateien und Subdirectories.

Wenn eine Datei gefunden wird, schreibt dieser Interrupt Informationen über sie in den DTA-Bereich. Die Zeichen 23-24 der DTA enthalten die Uhrzeit, 25-26 das Datum, 27-30 die Dateigröße, und 31-43 den Dateinamen, der durch ein Null-Zeichen (ASCII 0) abgeschlossen wird.

Die Entschlüsselung von Dateigröße, -datum und -uhrzeit können Sie den Routinen GetSizeFile, GetDateFile und GetTimeFile des Listings INHALT.BAS entnehmen.

*Beispiel* Siehe Listing INHALT.BAS in Kapitel 14, Abschnitt 4.

**Weiteren Directory-Eintrag suchen****DOS****Interrupt 21h, Funktion 4Fh**

*Eingabe* AH = 4Fh

*Ausgabe* Carry-Flag = 1: Keine weitere Datei gefunden

*Kommentar* Wenn eine weitere Datei gefunden wurde, der Carry-Flag also Null ist, stehen deren Daten in der DTA, die bei 21h.4Eh beschrieben wird.

*Beispiel* Siehe Listing INHALT.BAS in Kapitel 14, Abschnitt 4.

# Anhang F:

## Mausfunktionen

Die hier aufgezählten Mausfunktionen gehören im Prinzip auch in Anhang E, denn sie sind allesamt Funktionen des Interrupts 33h. Zwei Gründe sprechen jedoch dafür, die Mausroutinen gesondert zu erwähnen: Erstens sind sie nur verfügbar, wenn ein Maustreiber geladen ist, und zweitens kann man sie recht komfortabel über die Funktion `MouseDriver` aus der User Interface-Toolbox aufrufen, weil alle Mausfunktionen maximal 3 Parameter haben.

Sie können die folgenden Funktionen entweder mit

`MouseDriver Funktionsnummer, BX, CX, DX`

oder mit einem gewöhnlichen Interrupt-Aufruf an den Interrupt 33h aufrufen, bei dem in das Eingaberegister AX die Funktionsnummer eingetragen wird.

### Maustreiber-Reset

Maus

#### Funktion 00h

*Ausgabe*      AX = -1, wenn ein Maustreiber installiert ist; BX = Anzahl der Mausknöpfe

### Mauscursor anzeigen

Maus

#### Funktion 01h

Siehe Bemerkung zu `MouseShow` im Referenzteil der User Interface/Mouse-Toolbox.

### Mauscursor verstecken

Maus

#### Funktion 02h

Siehe Bemerkung zu `MouseHide` im Referenzteil der User Interface/Mouse-Toolbox.

**Funktion 03h**

*Ausgabe* BX = Status der Mausknöpfe. Bit 1 steht für den linken, Bit 2 für den mittleren und Bit 3 für den rechten Mausknopf. Die Bits sind 1, wenn der Knopf gerade gedrückt ist. Als CX wird die horizontale, als DX die vertikale Mausposition zurückgegeben. Wie auch bei allen anderen Mausfunktionen, handelt es sich bei den Koordinaten weder um Pixel- noch um Zeilen- und Spaltenkoordinaten, sondern um Punktkoordinaten des virtuellen Mausbildschirms. Dessen Auflösung hängt vom Bildschirmmodus ab, in dem sich das System befand, als der Maustreiber initialisiert wurde, und nicht von dem, in dem sich das System im Augenblick befindet. Bis auf die SCREEN-Modi 0, 1 und 7, die zu einer Mausbildschirm-Auflösung von 640 x 200 führen, entspricht die Auflösung des virtuellen Mausbildschirms derjenigen des SCREEN-Modus, in dem der Treiber initialisiert wird.

Zwar ist von der Maustreiber-Software her vorgesehen, daß beim Umschalten den SCREEN-Modus auch automatisch der virtuelle Bildschirm angepaßt wird; da BASIC aber nicht unbedingt Maustreiberkonforme Routinen zum Umschalten der Modi benutzt, klappt das nicht so, wie es eigentlich sollte.

Allen Problemen können Sie entgehen, indem Sie nach jedem Umschalten des SCREEN-Modus den Maustreiber neu initialisieren.

**Mauszeiger an bestimmte Position setzen****Funktion 04h**

*Eingabe* CX = horizontale, DX = vertikale Mausposition.

**Wie oft wurde ein Mausknopf gedrückt?****Funktion 05h**

*Eingabe* BX = 0 für linken, 1 für mittleren, 2 für rechten Mausknopf.

*Ausgabe* AX = Status aller Mausknöpfe (wie Funktion 03h); BX = Anzahl der Betätigungen des Mausknopfes, der angefordert wurde, seit dem letzten Aufruf dieser Funktion für diesen Mausknopf; CX = horizontale, DX = vertikale Mausposition zum Zeitpunkt des letzten Drückens auf den angeforderten Mausknopf.

## Wie oft wurde ein Mausknopf losgelassen?

Maus

### Funktion 06h

Identisch mit Funktion 05h, nur wird hier gezählt, wie oft ein Knopf losgelassen, nicht, wie oft er betätigt wurde.

## Horizontale Mausgrenzen festlegen

Maus

### Funktion 07h

*Eingabe* CX = minimale, DX = maximale horizontale Mausposition für zukünftige Bewegungen des Mauszeigers

## Vertikale Mausgrenzen festlegen

Maus

### Funktion 08h

*Eingabe* CX = minimale, DX = maximale vertikale Mausposition für zukünftige Bewegungen des Mauszeigers

## Relative Mausbewegung feststellen

Maus

### Funktion 0Bh

*Ausgabe* CX = horizontale, DX = vertikale Entfernung des Mauszeigers von der Position, an der er sich befand, als diese Funktion zum letzten Mal aufgerufen wurde

## Lichtgriffel-Emulation anschalten

Maus

### Funktion 0Dh

Von nun an kann die Maus wie ein Lichtgriffel behandelt werden. Das Drücken eines Mausknopfes simuliert ein Drücken des Lichtgriffel-Knopfes. Der so entstehende "Lichtgriffel" kann mit den Befehlen PEN und ON PEN GOSUB abgefragt werden. Der Haken dabei ist leider, daß auch die Lichtgriffel-Emulation ein eigenes Koordinatensystem kennt, das sich von Maustreiber zu Maustreiber unterscheiden kann. Es sollte - per definitionem - mit dem wirklichen Koordinatensystem übereinstimmen, aber viele Maustreiber geben zum Beispiel für den Bildschirmmodus 0 eine Auflösung von 80 x 200 Punkten an. Wenig falsch machen können Sie, wenn Sie die Zeilen- und Spaltenkoordinaten benutzen.

## Lichtgriffel-Emulation abschalten

Maus

### Funktion 0Eh

## Einstellung der Maus-Sensitivität

Maus

### Funktion 0Fh

*Eingabe* CX = horizontales, DX = vertikales Verhältnis von Mickeys zu Punkten. Standard ist 8 für CX und 16 für DX. Mickeys sind Maus-Bewegungseinheiten; je kleiner die Werte, desto schneller die Mausbewegung.

## Schwelle für Geschwindigkeits-Verdopplung setzen

Maus

### Funktion 13h

*Eingabe* DX = Verdopplungsschwelle in Mickeys pro Sekunde. Ab einer bestimmten Bewegungsgeschwindigkeit halbiert der Maustreiber das Verhältnis von Mickeys zu Punkten, so daß sich die Maus dann doppelt so schnell bewegt. Die Schwelle, bei der diese Verdopplung eintritt, ist standardmäßig 64 und kann mit dieser Funktion beliebig eingestellt werden.

## Maus-Sensitivität ermitteln

Maus

### Funktion 1Bh

*Ausgabe* BX = horizontales, CX = vertikales Mickey/Punkte-Verhältnis (siehe Funktion 0Fh); DX = Verdopplungsschwelle in Mickeys / Sekunde (siehe Funktion 13h).

## Bildschirmseite für Maus-Cursor einstellen

Maus

### Funktion 1Dh

*Eingabe* BX = Bildschirmseite, auf der der Maus-Cursor angezeigt werden soll



## **Bildschirmseite des Maus-Cursors ermitteln**

**Maus**

### **Funktion 1Eh**

*Ausgabe*      BX = Bildschirmseite, auf der der Maus-Cursor im Augenblick angezeigt wird

## **Maustreiber abschalten**

**Maus**

### **Funktion 1Fh**

## **Maustreiber wieder einschalten**

**Maus**

### **Funktion 20h**

# Anhang G: Tabellen dieses Buches

Seite	Inhalt
9	Options-Auswahl bei Installation, Bedeutung
27	QBX-Switches
30 <i>ff</i>	Tastaturbelegung im QBX-Editor
33, 34	Tastaturbelegung unter QBX
45	Tastaturbelegung zum Debuggen
51	PWB-Switches
53	Tasten im PWB-Editor
57	Operationen für View Relationship (Browser)
65 <i>ff</i>	BC-Switches
71 <i>ff</i>	LINK-Switches
118	Schriftarten-Gruppe für Screen-Modi
122	Schrifttabelle
182	Array-Typ statisch oder dynamisch
222	Einschränkungen unter OS/2
223	Include-Dateien für OS/2
233	Auswirkungen bestimmter Compiler-/Linker-Schalter (Optimierung)
234, 235	Verzicht-Files
241	Standard-Runtime-Module
248	LIB-Befehle und -Switches
256	NMAKE-Switches
260	NMAKE-Spezialmakros
262	NMAKE-Befehle, ausgewählt
267	Tastaturdefinitionsdateien für MKKEY
268, 269	Funktionen für Tastenbelegung
279, 280	CALL-Funktionen
303	DRAW-Zeichenbefehle
316	FRE-Argumente
330, 331	KEY-Argumente
347	ON <i>event</i> GOSUB-Argumente
349, 351, 352	OPEN-Argumente
358	PEN-Argumente
360	PLAY-Argumente
361	PMAP-Argumente
362	POINT-Argumente
364	PRINT-Argumente (Formatierung)
379 <i>ff</i>	SCREEN-Möglichkeiten
382	Standardfarben
399	STICK-Argumente
401	STRIG-Argumente
418	Anzahl Zeilen x Zeilenbreite bei unterschiedlichen Konfigurationen
440 <i>ff</i>	FormatXS-Argumente

(Fortsetzung nächste Seite)

Seite	Inhalt
462, 463	LoadFont-Argumente
469	SetGTextDir-Argumente
482	MenuCheck-Argumente
492	Alert-Argumente
496, 497	Dialog-Argumente
507	WindowPrint-Argumente
517	GetShiftState-Argumente
521	Variablen-Wertebereiche
524	QBX-Switches
525	BC-Switches
526	PWB-Switches
526, 527	LINK-Switches
527	LIB-Switches
528	BUILDRTM-Switches
529	NMAKE/NMK-Switches
530, 531	BASIC-Fehlermeldungen nach Nummern
531 ff	BASIC-Fehlermeldungen alphabetisch sortiert
558 ff	LINK-Fehler
561	LIB-Fehler
562	Fehlercodes der Presentation-Graphics-Toolbox
563	Fehlercodes der Fonts-Toolbox
564, 565	Scancodes (Tastatur)
566 ff	ASCII-Codes der Tastenkombinationen
569 ff	ASCII-Codes
573 ff	ISAM-Sortierttabellen
578	Werte zum Feststellen aktueller Videomodus
579	Werte zum Feststellen aktuelle Konfiguration
580	Bitbelegung für Tastaturabfrage
581	Abfragen Druckerstatus
582	Kennzeichnung Speichermedium
583	Attribute einer Datei

# Anhang H:

## Liste der Programmbeispiele

### Listings im Textteil

Seite	Listing	Name	Erläuterung
85	7-1	ISAMDEMO.BAS	ISAM-Zugriffe
111	9-1	LINGL.BAS	Lösen linearer Gleichungen (Matrizenmathematik)
120	9-2	FONTDEMO.BAS	Fonts benutzen
127	9-3	GRDEMO1.BAS	Balkendiagramm erstellen
133	9-4	GRDEMO2.BAS	Diagramme erstellen
140	9-5	WINDINC.BAS	Statements zum Benutzen der Window- Routinen
145	10-1	QSORT.BAS	Quick-Sort
148	10-2	ISORT.BAS	Insert-Sort
149	10-3	BUCKET.BAS	Bucket-Sort
150	10-4	MSORT.BAS	Merge-Sort
154	10-5	SUCHBIN.BAS	Binäres Suchen
155	10-6	SUCHBIND.BAS	Binäres Suchen
158	10-7	FAKULT.BAS	Fakultät berechnen rekursiv
168	11-1	GETPUTS.BAS	Dateien lesen und schreiben
176	12-1	BITMAN.BAS	Bits manipulieren
180	12-2	SEGFSTR.BAS	Segmentzuteilung zu Far Strings
197	15-1	ERROR1.BAS	Fehlerroutinen
199	15-2	ERROR2.BAS	Fehlerroutinen
201	15-3	ERROR3.BAS	Fehlerroutinen
208	16-1	INT21.BAS	Interrupt 21 aufrufen
210	16-2	INHALT.BAS	diverse DOS-Interrupt-Aufrufe

Weitere Listings im Referenzteil bei den einzelnen Befehlen.

# Anhang I : Befehlsreferenz – funktionsorientiert gegliedert

*Anhang I wurde von Thomas Antoni verfasst und der PDF-Version hinzugefügt; er war in der Papierausgabe von Frederik Ramm nicht enthalten.*

Diese Befehlsliste ist praxisgerecht nach Funktionen sortiert. Das hat schon schon fast jeder Programmierer einmal vermisst: Er will ein bestimmtes Programmierproblem lösen und hätte gerne eine Auflistung derjenigen Befehle, die ihm eventuell dabei helfen können.

## **Übersicht**

1. Datentypen
2. Farbcodes
3. Die wichtigsten Bildschirm-Modi
4. Variablen, Felder und Konstanten
5. Mathematische Operationen
6. Zahlen konvertieren und runden
7. Zufallszahlen
8. Zeichenketten (Strings) manipulieren
9. Text anzeigen, Cursor setzen
10. Tastatureingaben
11. Grafik anzeigen
12. Sound ausgeben aus dem PC-Speaker
13. Wartezeiten, Uhrzeit, Datum
14. Schleifen und Verzweigungen
15. Vergleichsoperationen
16. Logische Operationen
17. Prozeduren (SUBs und FUNCTIONS) und Module
18. Dateien bearbeiten
19. DOS-Dateisystembefehle
20. DOS- und Kommandozeilen-Befehle ausführen
21. Fehlerbehandlung
22. Benutzerdefinierte Ereignisverfolgung
23. Joystick
24. Serielle Schnittstelle
25. Druckerausgabe
26. Lichtgriffel
27. Direkte Speicher-, I/O- u. Treiberzugriffe
28. Trace Ein-/Ausschalten (für Debugging)
29. ISAM-Datenbanken
30. Meta-Befehle (Compiler-Direktiven)

Die Details zu jedem Befehl kann man in Sektion VII nachlesen. Weitere Informationen sind in der vorzüglichen Online-Hilfe von BASIC PDS verfügbar: Wenn Sie über einen bestimmten Befehl mehr erfahren wollen, dann tippen Sie in der Entwicklungsumgebung einfach das Befehlsschlüsselwort ein und betätigen Sie dann sofort die Funktionstaste F1.

## 1. Datentypen

Suffix	Datentyp	Speicherbedarf	Wertebereich, Bemerkungen
%	INTEGER (Ganzzahl mit Vorzeichen)	2 Bytes	-32 768...32 767
&	LONG (Lange Ganzzahl mit Vorzeichen)	4 Bytes	-2 147 483 648...2 147 483 647
!	SINGLE (einfach lange Gleitpunktzahl)	4 Bytes	-3.402823 E38...-1.40129 E-45 für negative Werte 1.40129 E-45 ... 3.402823 E38 für positive Werte <b>Bei Verwendung der "Alternate-Math Lib" mit /FPa:</b> -3.402823 E38...-1.175494 E-38 für negat. Werte 1.175494 E-38...3.402823 E38 für positive Werte
*#	DOUBLE (doppelt lange Gleitpunktzahl)	8 Bytes	-1.79769313486231 D308... -4.94065 D-324 für negative Werte 4.94065 D-324 ... 31.79769313486231 D308 für positive Werte <b>Bei Verwendung der "Alternate-Math Lib" mit /FPa:</b> -1.79769313486231 D308 ... -2.2250738585072 D-308 für negative Werte 2.2250738585072 D-308...1.79769313486231 D308 für positive Werte
@	CURRENCY (Währungstyp)	8 Bytes	-922337203685477.5808 ... 922337203685477.5807
\$	STRING	n+4 Bytes	<i>Speicherbedarf: 1 Byte je Zeichen, 4 Bytes für den Deskriptor; max. Länge 32 767 Bytes</i>
\$	STRING fester Länge (STRING * anzahl)	n Bytes	<i>Speicherbedarf: 1 Byte je Zeichen max. Länge 32 767 Bytes</i>
--	<b>Record</b>		<i>(Datensatztyp; mit TYPE ... UNTYPE deklariertes Anwenderdefiniertes Feld )</i>

## 2. Farbcodes

In der folgenden Tabelle sind die Standard-Farbattribute für Farbbildschirme aufgeführt; siehe auch Seite 382. In SCREEN 0 sind als Hintergrundfarben nur die ersten 8 Farben (Codes 0...7) erwendbar.

0 = Schwarz	4 = Rot	8 = Grau	12 = Hellrot
1 = Blau	5 = Violett	9 = Hellblau	13 = Rosa
2 = Grün	6 = Braun	10 = Hellgrün	14 = Gelb
3 = Türkis	7 = Hellgrau	11 = Helltürkis	15 = Weiß

## 3. Die wichtigsten Bildschirm-Modi

In der folgenden Tabelle sind die wichtigsten Bildschirm-Modi aufgeführt. Eine detaillierte Beschreibung finden Sie ab Seite 378. Die Modi 3, 4, 8 und 10 haben heute kaum noch eine Bedeutung, weil Sie nur für "exotische" oder für schwarz /weiß-Bildschirme gelten.

SCREEN	Beschreibung, Auflösung, Farben, [Speicherbedarf], Bildschirmseiten
0	Textmodus, für alle Grafikkarten, 80*25 43 50 und 40*25 43 50 Spalten * Zeilen mit WIDTH einstellbar. Läuft als einziger Bildschirmmodus auch problemlos im DOS-Teilfenster von Windows, 16 Farben, 8 Bildschirmseiten (0-7)
1	CGA/EGA/VGA-Karte, 320*200 Grafik, 30*25 Text, 4 aus 16 Farben, [2 Bits pro Pixel in 1 Ebene für GET/PUT], 1 Bildschirmseite (0)
2	CGA/MCGA/EGA/VGA-Karte, 640*200 Grafik, 80*25 Text, 2 aus 16 Farben, [1 Bit pro Pixel in 1 Ebene für GET/PUT], 1 Bildschirmseite (0)
7	EGA/VGA-Karte, 320*200 Grafik, 40*25 Text, 16 Farben, 8 Bildschirmseiten (0-7). Ruckelfreie Animationen auch auf langsamen Rechnern möglich, [4 Bits pro Pixel in 4 Ebenen für GET/PUT]
9	EGA/VGA-Karte, 640*350 Grafik, 80*15 Text, bis 16 Farben, [4 Bits pro Pixel (bei 16 Farben) in 4 Ebenen für GET/PUT], 2 Bildschirmseiten (0-1)
11	VGA-Karte, 640*480 Grafik, 80*25 30 50 60 Text (Voreinstellung: 80*30), 2 aus 256 Farben, gut geeignet für s/w-Grafiken, [1 Bit pro Pixel in 1 Ebenen für GET/PUT], 1 Bildschirmseite
12	VGA-Karte, 640 x 480 Grafik, 80*30 50 60 Text (Voreinstellung: 80*30), 6 aus 256 Farben, [4 Bits pro Pixel in 4 Ebenen für GET/PUT], 1 Bildschirmseite
13	VGA-oder MCGA-Karte, 320 x 200 Grafik, 40*25 Text, 256 Farben, [8 Bits pro Pixel in 1 Ebene für GET/PUT], 1 Bildschirmseite

## 4. Variablen, Felder und Konstanten

Befehl	Beschreibung	Seite
DIM	Variable oder Feld deklarieren und Speicherplatz dafür reservieren	20 299
OPTION BASE	Untere Grenze für Feld-Indices festlegen	353
CONST	Konstante deklarieren mit Wertzuweisung	289
DATA	Konstante(n) festlegen zur Verwendung mit READ	292
READ	Konstanten von DATA-Bereich einlesen	369
RESTORE	DATA-Bereich am Anfang oder ab der angegebenen Zeile erneut beginnen	372
DEFtype	Datentyp für Variablen/Funktionen mit bestimmten Anfangsbuchstaben definieren (type = INT   LNG   SNG   DBL   STR)	297
TYPE...END TYPE	Anwenderdefiniertes Feld definieren (z.B. für Datenbank)	410
LBOUND	Niedrigsten Feldindex ermitteln	333
UBOUND	Höchsten Feldindex ermitteln	411
\$DYNAMIC	Dynamische (mit REDIM und ERASE änderbare) Felder erzeugen	304 182
\$STATIC	Statische Felder erzeugen	397 182
ERASE	Statisches Feld löschen / dynamisches Feld aus dem Speicher entfernen	307
REDIM	Größe eines dynamischen Feldes ändern	370
COMMON	Globale Variable modulübergreifend deklarieren	20 288
SWAP	Inhalte zweier Variablen gleichen Typs miteinander vertauschen	406
LET	Wertzuweisung an eine Variable (veraltet LET a = b statt a=b)	334
LSET	Anwenderdefinierte Felder einander zuweisen	340

## 5. Mathematische Operationen

Befehl	Beschreibung	Seite
$x + y$	Addition	13
$x - y$	Subtraktion	13
$x * y$	Multiplikation	13
$x / y$	Division	13
$x \setminus y$	Ganzzahl-Division, liefert den ganzzahligen Anteil von $x/y$ ; z.B. $13 \setminus 5 = 2$	13
$x \text{ MOD } y$	Modulo-Operation (liefert den Rest der Division $x/y$ ; z.B. $13 \text{ MOD } 5 = 3$ )	13
$x^y$	Exponentialfunktion $x^y$	13
SQR (x)	Quadratwurzel aus x	390
$y = x^{(1/n)}$	n-te Wurzel aus x	-
SGN(x)	Vorzeichen von x (Signum-Funktion)	393
ABS (x)	Absolutbetrag von x	275
SIN (x)	Sinus von x (x im Bogenmaß; $360^\circ$ entspr. $2 * \text{Pi}$ ); z.B. $\sin 30^\circ = \text{SIN}(30 * 3.1416/180)$	393



COS (x)	Cosinus von x (x im Bogenmaß)	290
TAN (x)	Tangens von x (x im Bogenmaß)	407
ATN (x)	Arcustangens von x (ergibt den Winkel im Bogenmaß)	275
EXP (x)	x-te Potenz zur Basis e ( $e^x$ )	311
LOG (x)	Natürlicher Logarithmus ln(x) (Logarithmus zur Basis e)	338

## 6. Zahlen konvertieren und runden

Befehl	Beschreibung	Seite
INT (x!)	Gleitpunkt-Integer-Wandlung mit Abrundung nach unten	327
FIX (x!)	Nachkommastellen abtrennen ohne Rundung	313
PRINT USING	Zahlenwerte formatiert und "kaufmännisch" gerundet anzeigen (n.5 immer aufrunden, 4.5 -> 5   5.5 -> 6) und Gleitpunkt- in Festpunktzahl umwandeln	363
LPRINT USING	Zahlenwerte formatiert und "kaufmännisch" gerundet ausdrucken (n.5 immer aufrunden, 4.5 -> 5   5.5 -> 6) und Gleitpunkt- in Festpunktzahl umwandeln	363
CINT (x!)	SINGLE Gleitpunktzahl zu Ganzzahl "wissenschaftlich" zur nächsten geraden Zahl runden (4.5 -> 4   5.5 -> 6)	283
CLNG(x#)	DOUBLE Gleitpunktzahl zu Ganzzahl "wissenschaftlich" zur nächsten geraden Zahl runden (4.5 -> 4   5.5 -> 6)	284
CSNG	Numerischen Wert in SINGLE-Gleitpunktzahl umwandeln	290
CDBL	Numerischen Wert in DOUBLE-Gleitpunktzahl umwandeln	281
CCUR	Numerischen Wert in CURRENCY-Wert umwandeln	281
STR\$ (x)	Zahl x in Zeichenkette umwandeln (ASCII-String)	400
VAL (x\$)	String x\$ in Zahl umwandeln	412
HEX\$ (x)	Zahl x in Hexadezimalzahl-String umwandeln	320
OCT\$ (x)	Zahl x in Oktalzahl-String umwandeln	345

## 7. Zufallszahlen

Befehl	Beschreibung	Seite
RANDOMIZE z	Zufallsgenerator abhängig von der Zahl z initialisieren	368
RANDOMIZE TIMER	Zufallsgenerator abhängig vom System-Timer initialisieren (Normalfall)	369
RND	Zufallszahl erzeugen (ergibt einen SINGLE-Wert zwischen 0.000000 und 0.999999)	374

## 8. Zeichenketten (Strings) manipulieren

Befehl	Beschreibung	Seite
LEFT\$	Liefert die angegebene Anzahl Zeichen links aus einem String zurück	333
RIGHT\$	Liefert die angegebene Anzahl Zeichen rechts aus einem String zurück	374
MID\$	Liefert die angegebene Anzahl Zeichen mitten aus einem String zurück	342
LTRIM\$	Schneidet führende Leerzeichen aus einem String heraus	342

RTRIM\$	Schneidet am Ende eines Strings stehende Leerzeichen heraus	375
INSTR	Sucht eine Zeichenkette in einem String	327
LCASE\$	Groß- in Kleinbuchstaben umwandeln (nicht für Umlaute)	333
UCASE\$	Klein- in Großbuchstaben umwandeln (nicht für Umlaute)	411
MID\$	Teil eines Strings durch einen anderen String ersetzen	342
LSET	String fester Länge links ausrichten	340
RSET	String fester Länge rechts ausrichten	375
STR\$	Numerischen Wert in String umwandeln	400
VAL	String in numerischen Wert umwandeln	412
SPACE\$	Liefert einen String mit der angegebenen Anzahl von Leerzeichen	394
STRING\$	Liefert einen String mit identischen Zeichen wählbarer Länge	402
LEN	Liefert die Länge des angegebenen Strings in Anzahl Bytes	334
ASC	Liefert den ASCII-Code des angegebenen Zeichens	275
CHR\$	Liefert das Textzeichen mit dem angegebenen ASCII-Code	282
t\$ = t1\$ + t2\$	Strings zusammenfügen (z.B. "Es" + "sel" = "Esel")	-

## 9. Text anzeigen, Cursor setzen

Befehl	Beschreibung	Seite
CLS	Bildschirm löschen / in der aktuellen Hintergrundfarbe einfärben	285
COLOR	Vordergrundfarbe (Text) und Hintergrundfarbe festlegen	286
PRINT	Numerische Werte oder Text anzeigen	363
PRINT USING	Numerische Werte und Text formatiert oder gerundet anzeigen	363
PRINT TAB	Text ab angegebener Spalte anzeigen	406
LOCATE	Cursor auf eine Bildschirmposition setzen für Anzeigen und Eingaben	337
SPC	Eine angegebene Anzahl von Leerzeichen ausgeben (nur nach PRINT und LPRINT)	394
WIDTH	Anzahl der Spalten und Zeilen der Bildschirmanzeige ändern	417
CSRLIN	Die momentane Zeilenposition des Cursors ermitteln	291
POS(n)	Die momentane Spaltenposition des Cursors ermitteln	362
VIEW PRINT	Anzeigefenster („Text-Viewport“) im angegebenen Zeilenbereich festlegen	414
WRITE	Datensatz auf dem Bildschirm anzeigen	420

## 10. Tastatureingaben

Befehl	Beschreibung	Seite
INPUT	Tastatureingabe anfordern und Eingabewert in Variable(n) ablegen	325f
INKEY\$	Ein Zeichen von der Tastatur einlesen und in Tastaturpuffer löschen	322
SLEEP	Warten auf beliebige Tastenbetätigung (leert nicht den Tastaturpuffer)	393
LINE INPUT	Text-Zeile (inkl. Kommas) von der Tastatur einlesen	336
ON KEY	Ereignisgesteuertes Aufrufen einer SUBroutine bei Tastenbetätigung	346f
KEY	Funktionstasten mit Textstring belegen	329

KEY	Ereignisverfolgung für Tastenbetätigungen aktivieren / deaktivieren	330
EVENT OFF	Ereignisverfolgung deaktivieren (für ALLE Ereignisse)	309
EVENT ON	Ereignisverfolgung aktivieren (für ALLE Ereignisse)	309

## 11. Grafik anzeigen

Befehl	Beschreibung	Seite
SCREEN	Einen Grafikmodus oder eine Bildschirmauflösung wählen	23 377 378
CLS	Bildschirm löschen	285
COLOR	Setzt die aktuelle Vorder- und Hintergrundfarbe	286
LINE	Zeichnet eine Linie oder ein Viereck	334
CIRCLE	Zeichnet einen Kreis, eine Ellipse oder einen Kreis-/Ellipsenbogen	283
DRAW	Zeichnet vielfältige Grafikelemente mit einer "Grafik-Makro-sprache"	302
PSET	Zeichnet einen Punkt in der angegebenen Farbe oder der Vordergrundfarbe	366
PRESET	Zeichnet einen Punkt in der angegebenen Farbe oder löscht ihn	363
PAINT	Füllt einen Bildschirmbereich oder ein Objekt mit einem Farbmuster	354
VIEW	Definiert ein Bildschirmfenster („Grafik-Viewport“) für nachfolgende Grafikausgaben	414
WINDOW	Skaliert die Fensterkoordinaten für nachfolgende Grafikausgaben	419
PMAP	Rechnet Fensterkoordinaten in physikalische Koordinaten um und umgekehrt	361
POINT	Liefert die Cursorposition oder die Farbe eines Pixels	361
PALETTE	Ändert eine oder mehrere Farben in der Palette	355
GET	Bildschirminhalt in numerischem Feld im Speicher ablegen	317
PUT	Anzeigeinformationen aus einem numerischen Feld zur Anzeige bringen	366
PCOPY	Bildschirmseiten kopieren ("Page Flipping" für schnellen Bildwechsel)	356
BSAVE	Kopiert den Inhalt eines Bildschirmspeicherbereichs in eine Datei	276
BLOAD	Lädt den Inhalt einer mit BSAVE erstellten Datei in den Bildschirmspeicher	276

## 12. Sound ausgeben aus dem PC-Speaker

Befehl	Beschreibung	Seite
BEEP	Piepston ausgeben	275
SOUND	Ton mit der angegebenen Dauer und Frequenz ausgeben	394
PLAY	Eine oder mehrer Noten abspielen mit einer "Musik-Makrosprache"	359
ON PLAY	Ereignisgesteuerte Bearbeitung des Notenpuffers definieren	346
PLAY OFF	Ereignisverfolgung für Notenpuffer deaktivieren	359
PLAY ON	Ereignisverfolgung für Notenpuffer aktivieren	359
PLAY STOP	Ereignisverfolgung für Notenpuffer unterbrechen und speichern	359

### 13. Wartezeiten, Uhrzeit, Datum

Befehl	Beschreibung	Seite
SLEEP	Die angegebene Anzahl von ganzen Sekunden warten	393
TIMES	Liefert die aktuelle Uhrzeit	408
DATES	Liefert das aktuelle Datum	293
TIMER	Liefert die seit Mitternacht vergangene Anzahl von Sekunden als SINGLE-Wert (für Wartezeitenbildung)	408
ON TIMER	Ereignisgesteuerte Bearbeitung des Timers definieren	346
TIMER OFF	Ereignisverfolgung für Timer deaktivieren	409
TIMER ON	Ereignisverfolgung für Timer aktivieren	409
TIMER STOP	Ereignisverfolgung für Timer anhalten und speichern	409

### 14. Schleifen und Verzweigungen

Befehl	Beschreibung	Seite
FOR...NEXT	Schleife mit Zählvariable	314
EXIT FOR	FOR...NEXT-Schleife vorzeitig verlassen	311
DO...LOOP UNTIL	Schleife durchlaufen bis Ende-Bedingung erfüllt	301
DO...LOOP WHILE	Schleife durchlaufen solange Wiederholbedingung erfüllt	301
EXIT DO	DO...LOOP-Schleife vorzeitig verlassen	311
WHILE...WEND	Schleife durchlaufen solange Wiederholbedingung erfüllt (veraltet)	417
IF...THEN...ELSE	Verzweigung abhängig von einer Bedingung	320
SELECT CASE	Mehrfachverzweigung	387
END	BASIC Programm oder SUB/FUNCTION beenden (im Interpreter-Modus Rückkehr zur Entwicklungsumgebung)	305
SYSTEM	Rückkehr zum Betriebssystem (auch im Interpreter-Modus)	406
GOTO	Unbedingter Sprung zur angegebenen Sprungmarke	320
ON...GOTO	Mehrfachverzweigung (zu einer der angegebenen Sprungmarken)	348
ON...GOSUB	Mehrfachverzweigung (Aufruf einer der angegebenen lokalen SUBroutinen)	348

### 15. Vergleichsoperationen

Befehl	Beschreibung	Seite
x = y	Abfrage auf Gleichheit	14
x <> y	Abfrage auf Ungleichheit	14
x < y	Abfrage auf kleiner als	14
x > y	Abfrage auf größer als	14
x <= y	Abfrage auf kleiner oder gleich	14
x >= y	Abfrage auf größer oder gleich	14

## 16. Logische Operationen

Befehl	Beschreibung	Seite
NOT	Logische Invertierung (Komplement)	-
AND	UND-Verknüpfung (Konjunktion)	-
OR	ODER-Verknüpfung (Disjunktion)	-
XOR	Exklusives ODER	-
EQV	Äquivalenz	-
IMP	Implikation	-

## 17. Prozeduren (SUBs und FUNCTIONs) und Module

Befehl	Beschreibung	Seite
SUB...END SUB	SUB-Prozedur mit Übergabe-Parametern deklarieren	19 402
FUNCTION...END FUNCTION	FUNKTION-Prozedur mit Übergabe-Parametern und Typ deklarieren	19 402
CALL	SUB-Prozedur aufrufen	19 277
EXIT SUB	SUB-Prozedur vorzeitig verlassen	311
EXIT FUNCTION	FUNCTION-Prozedur vorzeitig verlassen	311
DECLARE	Prozedur im Hauptprogramm deklarieren	294
COMMON	Variablen modulübergreifend deklarieren, auch für CHAIN-Module	20 288
SHARED	Variablen in Prozedur deklarieren und dem Hauptprogramm bekanntmachen	20 390
STATIC	Wert von Variablen und Felder zwischen den Prozedur-Aufrufen konservieren	397
CHAIN	Die Kontrolle an eine andere Programmdatei übergeben (gemeinsamer Zugriff auf COMMON-Variablen möglich)	187 281
RUN	Aktuelles Programm oder eine andere Programmdatei neu starten	188 376
CALL ABSOLUTE	Externes Maschinenspracheprogramm aufrufen	279
CALL INTERRUPT	DOS-System-Interrupt-Routine aufrufen	279 576f
CALL INTERRUPTX	DOS-System-Interrupt-Routine aufrufen	279 576f
GOSUB	Lokale SUB-Prozedur aufrufen	19 319
RETURN	Lokale SUB-Prozedur beenden	373
FN...	Lokale FUNCTION-Prozedur aufrufen (veraltet)	295
DEF FN...END DEF	Lokale FUNCTION-Prozedur deklarieren (veraltet)	295
EXIT DEF	Lokale FUNCTION-Prozedur vorzeitig verlassen (veraltet)	311
ALIAS	Verweist auf den Namen einer Nicht-BASIC-Prozedur	-
BYVAL	Parameterübergabe an Nicht-BASIC-Prozedur mit Call by Value	-
CDECL	Prozedur-Parameterübergabe gemäß C-Konventionen	294
CALLS	Aufruf einer Nicht-BASIC-SUB-Prozedur	-

COMMAND\$	Liefert die Befehlszeile, mit der ein BASIC-EXE-Programm aufgerufen wurde, inklusive Übergabeparametern	287
-----------	---	-----

## 18. Dateien bearbeiten

Befehl	Beschreibung	Seite
\$INCLUDE	Externe Quellsprachdatei ins Programm einfügen	322
OPEN	Datei zum Lesen oder Schreiben öffnen	348
CLOSE	Datei schließen	285
RESET	Alle offenen Dateien schließen	372
WRITE #	Datensatz in eine Sequentielle Datei schreiben	420
PRINT #	Daten in eine Sequentielle Datei schreiben	363
PRINT USING #	Daten formatiert in eine Sequentielle Datei schreiben	363
INPUT #	Daten aus einer Sequentiellen Datei lesen	326
INPUT\$	Zeichenkette aus einer Datei lesen	325
LINE INPUT #	Textzeile aus einer Sequentiellen Datei lesen	336
TYPE...END TYPE	Datensatz für Random-Datei deklarieren (Anwenderdefiniertes Feld)	410
LSET	Daten zwischen 2 Anwenderdefinierten Feldern (Datensätzen) kopieren	340
PUT	Variable in eine Random-Datei schreiben	366
GET	Daten aus einer Random-Datei lesen	317f
FREEFILE	Die nächste unbenutzte Dateinummer ermitteln	317
FILEATTR	Gibt Informationen über den Modus einer geöffneten Datei zurück	312
EOF	Ermittelt, ob beim Lesen das Dateiende erreicht ist	307
LOC	Ermittelt die Schreib-/Leseposition in einer geöffneten Datei	336
LOF	Liefert die Länge einer Datei in Bytes zurück	338
SEEK	Position des Schreib-/Lesezeigers ermitteln/ändern	385f
LOCK...UNLOCK	Zugriff auf geöffnete Datei oder Datensatz im Netzwerk sperren	337
WIDTH	Datenbreite je Zeile festlegen (Zeilenvorschübe werden automatisch eingefügt)	417
FIELD	FIELD-Puffer für Datensatz deklarieren (veraltet, für RANDOM-Dateien)	311
LSET	Daten in den FIELD-Puffer schreiben (Vorbereitung für PUT)	340
RSET	Daten rechtsbündig in den FIELD-Puffer schreiben (Vorbereitung für PUT)	375
MKIS	INTEGER-Wert in Pseudostring umwandeln (für FIELD-Puffer)	343
MKL\$	LONG-Wert in Pseudostring umwandeln (für FIELD-Puffer)	343
MKS\$	SINGLE-Wert in Pseudostring umwandeln (für FIELD-Puffer)	343
MKD\$	DOUBLE-Wert in Pseudostring umwandeln (für FIELD-Puffer)	343

MKC\$	CURRENCY-Wert in Pseudostring umwandeln (für FIELD-Puffer)	343
MKSMBF\$	SINGLE-Wert in Pseudostring umwandeln (MBF-Gleitpunktformat)	344
MKDMBF\$	DOUBLE-Wert in Pseudostring umwandeln (MBF-Gleitpunktformat)	344
CVI	Pseudostring in INTEGER-Wert umwandeln (für FIELD-Puffer)	291
CVL	Pseudostring in LONG-Wert umwandeln (für FIELD-Puffer)	291
CVS	Pseudostring in SINGLE-Wert umwandeln (für FIELD-Puffer)	291
CVD	Pseudostring in DOUBLE-Wert umwandeln (für FIELD-Puffer)	291
CVC	Pseudostring in CURRENCY-Wert umwandeln (für FIELD-Puffer)	291
CVSMBF	Pseudostring in SINGLE-Wert umwandeln (MBF-Gleitpunktformat)	292
CVDMBF	Pseudostring in DOUBLE-Wert umwandeln (MBF-Gleitpunktformat)	292

## ***19. DOS-Dateisystembefehle***

<b>Befehl</b>	<b>Beschreibung</b>	<b>Seite</b>
FILES	Liste der Dateien im angegebenen Verzeichnis anzeigen	313
KILL	Datei löschen	332
NAME	Den Namen einer Datei oder eines Verzeichnisses ändern, Dateien verschieben	344
MKDIR	Neues Verzeichnis erstellen	343
CHDIR	Verzeichnis wechseln	282
CHDRIVE	Laufwerk wechseln	282
RMDIR	Verzeichnis löschen	374
CURDIR\$	Aktuellen Verzeichnispfad ermitteln	291
DIR\$	Datei ermitteln, deren Namen dem angegebenen Suchbegriff entspricht	300

## ***20. DOS- und Kommandozeilen-Befehle ausführen***

<b>Befehl</b>	<b>Beschreibung</b>	<b>Seite</b>
SHELL	Übergibt einen Befehls-String an den Kommandointerpeter des Betriebssystems	391f
ENVIRON	Umgebungsvariable lesen, setzen oder löschen	305f
COMMAND\$	Liefert die Befehlszeile, mit der ein BASIC-EXE-Programm aufgerufen wurde	287

## 21. Fehler bearbeiten

Befehl	Beschreibung	Seite
ON ERROR	Fehlerbearbeitungsroutine festlegen und freigeben/sperren	345
RESUME	Programmkontrolle zurückgeben nach Abarbeiten einer Fehlerbearbeitungsroutine	372
RETURN	Bearbeitung an der Fehlerstelle fortsetzen	373
ERR	Fehlercode eines zur Laufzeit aufgetretenen Fehlers ermitteln	308
ERL	Zeilennummer im Quellspracheprogramm, bei der ein Fehler aufgetreten ist	308
ERDEV	Liefert den Fehlercode von dem letzten fehlerverursachenden Gerät	308
ERDEV\$	Liefert den Namen des letzten fehlerverursachenden Geräts	308
ERROR	Simuliert das Auftreten eines Fehlers (z.B. für Testzwecke)	309

## 22. Benutzerdefinierte Ereignisverfolgung

Befehl	Beschreibung	Seite
ON UEVENT	Benutzerdefinierte Ereignisverfolgung definieren	347
CALL SetUEvent	Startpunkt des Ereignisses definieren	279 411
UEVENT ON	Benutzerdefinierte Ereignisverfolgung aktivieren	411
UEVENT OFF	Benutzerdefinierte Ereignisverfolgung deaktivieren	411
UEVENT STOP	Benutzerdefinierte Ereignisverfolgung unterbrechen mit Speicherung	411
EVENT OFF	Ereignisverfolgung deaktivieren (für ALLE Ereignisse)	309
EVENT ON	Ereignisverfolgung aktivieren (für ALLE Ereignisse)	309

## 23. Joystick

Befehl	Beschreibung	Seite
STICK	Liefert die x/y/Schubregler-Koordinaten des Joysticks A oder B zurück	399
STRIG	Liefert den Status der Feuerknöpfe zurück	401
ON STRIG	Ereignisgesteuert auf Betätigung von Joystick-Tasten reagieren	346
STRIG OFF	Ereignisverfolgung für Joystick-Tasten deaktivieren	401
STRIG ON	Ereignisverfolgung für Joystick-Tasten aktivieren	401
STRIG STOP	Ereignisverfolgung für Joystick-Tasten unterbrechen und speichern	401



## 24. Serielle Schnittstelle

Befehl	Beschreibung	Seite
OPEN "COM..."	Serielle Schnittstelle aktivieren und konfigurieren	348
LOF	Liefert die Anzahl der unbelegten Bytes im Sendepuffer	338
LOC	Liefert die Anzahl der im Empfangspuffer stehenden zu lesenden Zeichen	336
ON COM	Ereignisgesteuerte Bearbeitung der seriellen Schnittstelle definieren	346f
COM OFF	Ereignisverfolgung für serielle Schnittstelle deaktivieren	287
COM ON	Ereignisverfolgung für serielle Schnittstelle aktivieren	287
COM STOP	Ereignisverfolgung für serielle Schnittstelle anhalten und speichern	287
WIDTH "COMn:"	Legt die Länge von Text-Ausgabezeilen fest (Zeilenvorschübe werden automatisch eingefügt)	417
ERDEV	Fehler von Kommunikationsschnittstelle melden	308

## 25. Druckerausgabe

Befehl	Beschreibung	Seite
OPEN "LPTn:"	Drucker für die Datenausgabe öffnen	348
LPRINT	Daten an den Ducker ausgeben	339
LPRINT USING	Zahlenwerte formatiert oder kaufmännisch gerundet ausdrucken	363f
WIDTH "LPTn:"	Anzahl der zu druckenden Spalten festlegen (Zeilenvorschübe werden automatisch eingefügt)	417
LPRINT TAB	Text ab angegebener Spalte drucken	406
LPOS	Liefert die Anzahl der Zeichen, die nach dem letzten CR (Zeilenvorschub) ausgegeben wurden	339
LPRINT SPC	Eine angegebene Anzahl von Leerzeichen ausdrucken	394
ERDEV	Fehler von Kommunikationsschnittstelle melden	308

## 26. Lichtgriffel

Befehl	Beschreibung	Seite
PEN	Liefert die x/y-Koordinaten des Lichtgriffels	358
ON PEN	Ereignisgesteuerte Bearbeitung für den Lichtgriffel definieren	346f
PEN OFF	Ereignisverfolgung für Lichtgriffel deaktivieren	358
PEN ON	Ereignisverfolgung für Lichtgriffel aktivieren	358
PEN STOP	Ereignisverfolgung für Lichtgriffel unterbrechen und speichern	358

## 27. Direkte Speicher-, I/O- u. Treiberzugriffe

Befehl	Beschreibung	Seite
DEF SEG	Setzt die aktuelle Segmentadresse	297
PEEK	Liest ein Speicherbyte von der angegebenen Offsetadresse	357
POKE	Schreibt ein Byte zur angegebenen Offsetadresse	362
BSAVE	Kopiert den Inhalt eines Speicherbereichs ab der angegebenen Offsetadresse in eine Datei	276
BLOAD	Lädt eine mit BSAVE erstellte Datei in den Hauptspeicher	276
VARPTR	Liefert die Offset-Adresse der angegebenen numerischen Variablen	412
VARSEG	Liefert die Segment-Adresse der angegebenen numerischen Variablen	413
VARPTR\$	Liefert die Adresse der angegebenen Variablen als String (für DRAW u.PLAY)	413
STACK	Setzt die Stackgröße zurück bzw. liefert die maximal mögliche Stackgröße	396
CLEAR	Setzt alle Variablen zurück oder legt die Stackgröße fest	284
SETMEM	Legt den vom Far Heap beanspruchten Speicher fest oder meldet ihn	389
FRE	Meldet den freien Platz im Near und Far Speicher sowie im EMS und Stack	316
SADD	Liefert die Offset-Adresse der angegebenen Stringvariablen	377
SSEG	Liefert die Segment-Adresse der angegebenen Stringvariablen	395
SSEGADD	Liefert Offset- und Segment-Adresse der angegebenen Stringvariablen	396
INP	Byte von I/O-Port lesen	324
OUT	Byte zum I/O-Port senden	354
WAIT	Wartet bis das angegebene Bitmuster an einem I/O-Port erscheint	416
IOCTL	Steuer-/Statusdaten von Gerätetreiber empfangen/sendern	328

## 28. Trace Ein-/Ausschalten (für Debugging)

Befehl	Beschreibung	Seite
TRON	Trace einschalten (Anzeige der durchlaufenen Zeilennummern)	409
TROFF	Trace ausschalten	409
STOP	Programm beenden und Modulnamen/Codeadresse anzeigen	400

## 29. ISAM-Datenbanken

Befehl	Beschreibung	Seite
TYPE...END TYPE	Datentyp für ISAM-Datensatz deklarieren (Anwenderdefiniertes Feld)	410
LSET	Daten zwischen 2 Anwenderdefinierten Feldern kopieren	340
OPEN..FOR ISAM...	Eine ISAM-Datenbankdatei öffnen	428
CLOSE	Eine ISAM-Datenbankdatei schließen	423
DELETETABLE	Löscht eine Datenbank komplett mit allen Datensätzen und Indices	425
CREATEINDEX	Datenbank-Index erstellen	423

GETINDEX\$	Liefert den Namen des aktuellen Index zurück	427
SETINDEX	Macht den angegebenen Index zum aktiven Datenbankindex	433
DELETEINDEX	Löscht einen Index aus einer ISAM-Datenbank	425
TEXTCOMP	Zwei Strings bezüglich der ISAM-Sortierreihenfolge vergleichen	433
MOVEFIRST	Setzt den Dateizeiger auf den ersten Datensatz	428
MOVELAST	Setzt den Dateizeiger auf den letzten Datensatz	428
MOVENEXT	Setzt den Dateizeiger auf den nächsten Datensatz	428
MOVEPREVIOUS	Setzt den Dateizeiger auf den vorhergehenden Datensatz	428
SEEKGT	Datensatz suchen, dessen Inhalt größer als der angegebene Wert ist	432
SEEKGE	Datensatz suchen, dessen Inhalt größer oder gleich dem angegebenen Wert ist	432
SEEKEQ	Datensatz suchen, dessen Inhalt gleich dem angegebenen Wert ist	432
BOF	Stellt fest, ob der Dateizeiger vor dem Beginn einer ISAM-Datei steht	421
EOF	Stellt fest, ob der letzte Datensatz einer Datenbank erreicht ist	426
LOF	Liefert die Anzahl der Datensätze in einer ISAM-Datei zurück	427
FILEATTR	Liefert Modus und Typ einer geöffneten ISAM-Datei zurück	426
INSERT	Fügt einen neuen Datensatz in eine ISAM-Datenbank ein	427
RETRIEVE	Aktuellen Datensatz lesen und in Variable abspeichern	430
UPDATE	Aktuellen Datensatz mit dem Inhalt einer Variablen überschreiben	434
DELETE	Löscht den aktuellen Datensatz aus einer ISAM-Datenbank	425
BEGINTRANS	Beginn einer Transaktionsaufzeichnung	421
COMMITTRANS	Beendet die gerade laufende Transaktion	423
CHECKPOINT	Speichert alle durchgeführten Datenbankänderungen auf die Platte	422
ROLLBACK	Datenbankänderungen rückgängig machen (bezogen auf den Save-Punkt)	431
SAVEPOINT	Save-Punkt markieren, ab dem Datenbankänderungen rücknehmbar sind	431

### 30. Meta-Befehle (Compiler-Direktiven)

Befehl	Beschreibung	Seite
' (Hochkomma)	Kommentar einleiten, den der Compiler bis Zeilenende überliest	371
REM	Kommentarzeile einleiten, die der Compiler bis Zeilenende überliest	371
\$DYNAMIC	Dynamische (mit REDIM und ERASE änderbare) Felder erzeugen	304 182

\$STATIC	Statische Felder erzeugen	397 182
\$INCLUDE	Externe Quellsprachedatei einfügen	322

# Stichwortverzeichnis

## Sonderzeichen, Ziffern

%, &, @, ., !, # (Datentyp) 13  
 +, -, \*, /, ^, \ (Operator) 13  
 =, <, >, <=, >=, <> (Operator) 14

+, -, -+, \*, -\* (LIB) 248

/A

(BC) 65

(NMAKE) 256

/AH (QBX) 27

/Ah (BC) 65

/B (QBX) 27

/BA (LINK) 71

/C (NMAKE) 256

/C:

(BC) 65

(QBX) 28

/CMD (QBX) 28

/CO (LINK) 71

/D

(BC) 65

(NMAKE) 256

(PWB) 51

/E

(BC) 65

(LINK) 71

(NMAKE) 256

(PWB) 51

/E: (QBX) 28

/Ea (QBX) 28

/Es

(BC) 65

(QBX) 28

/F (NMAKE) 256

/F/PACKC (LINK) 71

/FBr (BC) 66

/FBx (BC) 66

/FPa (BC) 66

/FPi (BC) 66

/Fs (BC) 66

/G (QBX) 28

/G2 (BC) 66

/H (QBX) 28

/HE (LINK) 71

/I (NMAKE) 256

/Ib (BC) 66

/Ie: (BC) 66

/Ii: (BC) 66

/INF (LINK) 71

/K: (QBX) 28

/L (QBX) 28

/LI (LINK) 71

/LP (BC) 66

/LR (BC) 66

/M

(LINK) 72

(PWB) 51

/MBF

(BC) 66

(QBX) 28

/N (NMAKE) 256

/NOD (LINK) 72

/NOE

(LIB) 248

(LINK) 72

/NOF (QBX) 28

/NOF/NOP (LINK) 72

/NOHI (QBX) 29

/NOL (LINK) 72

/NOLOGO

(LIB) 248

(NMAKE) 256

/NON (LINK) 72

/O (BC) 66

/O: (LINK) 72

/Ot (BC) 67

/P (NMAKE) 256

/PA: (LIB) 248

/PAU (LINK) 72

/PF (PWB) 51

/PL (PWB) 51

/PM: (LINK) 72

/PP (PWB) 51

/Q

(LINK) 73

(NMAKE) 256

## /R

(BC)	67
(NMAKE)	256
(PWB)	51
/RUN (QBX)	29
/S	
(BC)	67
(NMAKE)	256
/SE: (LINK)	73
/T	
(BC)	67
(NMAKE)	256
(PWB)	51
/V (BC)	67
/W (BC)	67
/X	
(BC)	67
(NMAKE)	256
/Z (BC)	67
/Zd (BC)	67
/Zi (BC)	67

## A

ABS	275
Add-On-Libraries	4, 103
Alert	492
AltToASCII\$	512
Analyze-Routinen	135
AnalyzeChart	470
AnalyzeChartMS	471
AnalyzePie	471
AnalyzeScatter	472
AnalyzeScatterMS	472
Array	16 ff
ASC	275
ASCII-Code	14
ASCII-Tabelle	569 ff
ATN	275
AttrBox	513
Ausführungsgeschwindigkeit	227
Axistype	130

## B

Backtracking	159
.BAS (Extension)	61
BASIC	
Compiler / Interpreter	v
Neuerungen bei PDS 7.1	3
BC.EXE	64
Bedienung des Editors	29 ff
BEEP	275
BEGINTRANS	421
Binäres Suchen	153
BITMAN.BAS (Prog.)	176
BLOAD	276
BOF	421
Booten	580
Box	513
BSAVE	276
BUCKET.BAS (Prog.)	149
BucketSort	148
BUILDRTM	239
ButtonClose	493
ButtonInquire	493
ButtonOpen	494
ButtonSetState	495
ButtonToggle	495

## C

CALL	19, 277
CCUR	281
CDBL	281
CHAIN	187, 281
Chart	472
CHARTB.BI	128
ChartEnvironment	128 ff, 132
ChartMS	473
ChartPie	474
ChartScatter	474
ChartScatterMS	475
ChartScreen	475
CHDIR	282
CHDRIVE	282
CHECKPOINT	422

CHRS	282
CINT	283
CIRCLE	283
CLEAR	284
CLNG	284
CLOSE	285, 423
CLS	285
COLOR	286
COM	287
COMMAND\$	287
COMMITTRANS	423
COMMON	20, 288
CONST	289
Coprozessor	195
COS	290
CREATEINDEX	423
CSNG	290
CSRLIN	291
CURDIR\$	291
CURRENCY (Datentyp)	4, 13
CVx	291
CVxMBF	292

## D

DATA	292
DATES	293
Date-Library	106 ff
Referenz	435
Datei	21, 34
Attribute	583
Binär	21
Random Access	21
sequentiell	21
Dateiverwaltung	165 ff
Datenkonvertierung (ISAM)	98
Datentyp	156
selbstdefiniert	13
DateSerial	435
DateValue	435
Datum	582
Day	436
DDB	443
Debugger	40 ff, 55
DECLARE	294

DEF FN	295
DEF SEG	297
DefaultChart	476
DefaultFont	458
DEFxxx	297
DELETE	425
DELETEINDEX	425
DELETETABLE	425
Dialog	496
Dialogboxen	32
DIM	20, 299
DIR\$	300
Directory	584
Dokument	35
DO...LOOP	301
DOS	
Interrupt	205 ff
Version	582
DOUBLE (Datentyp)	13
DRAW	302
Druckerstatus	581
Druckertreiber	169
DTA-Adresse	583
SDYNAMIC	304

## E

EditFieldClose	497
EditFieldInquire\$	498
EditFieldOpen	498
Editor	29 ff, 52
EMS	28, 189
END	305
ENVIRON	306
ENVIRONS	305
EOF	307, 426
ERASE	307
ERDEV, ERDEV\$	308
ERR, ERL	308
ERROR	309
ERROR1.BAS (Prog.)	197
ERROR2.BAS (Prog.)	199
ERROR3.BAS (Prog.)	201
EVENT ON, OFF	309
.EXE (Extension)	62

EXIT	311
EXP	311
Extended / Expanded Memory	217 ff

## F

FAKULT.BAS (Prog.)	158
Far Strings	4, 8
Farbpalette	135
Fehlerbehandlung	197
Aufruf	207
Datenübergabe	206
Fehlermeldungen	530 ff
Fehlersuche	40 ff
FIELD	311
FILEATTR	312, 426
FILES	313
Finance-Library	108 ff
Referenz	443
FIX	313
FONTDEMO.BAS (Prog.)	120
Font-Toolbox	115 ff
Referenz	458
FOR...NEXT	314
Formatierung	105 ff
Format-Library	104 ff
Referenz	440
FormatX\$	440
Formular	
-beschreibung	170
-beschriftung	177
Fortsetzung von Zeilen	vii
FRE	316
FREEFILE	317
Funktionen	18
FV	443

## G

Gemischtsprachliches Programmieren	50
General-Toolbox	136
Referenz	512
GET	317, 318
GetBackground	514

GetFontInfo	458
GetGTextLen	460
GETINDEX\$	427
GetMaxFonts	460
GetPaletteDef	477
GetPattern\$	477
GETPUTS.BAS (Prog.)	168
GetRFontInfo	460
GetShiftState	515
GetTotalFonts	461
GOSUB	19, 319
GOTO	320
Grafikmodus	23
GRDEMO1.BAS (Prog.)	127
GRDEMO2.BAS (Prog.)	133
GTextWindow	461

## H

HEX\$	320
Hour	436

## I

IF...THEN...ELSE	320
INCLUDE (Umgebungsvariable)	7
\$INCLUDE	322
Include-Datei	35
Index (ISAM)	81
INHALT.BAS (Prog.)	210
INKEY\$	322
INP	324
INPUT	326
INPUT\$	325
INSERT	427
Insertsort	147
Installation	5 ff
INSTR	327
INT	327
INT21.BAS (Prog.)	208
INTEGER (Datentyp)	13
Interrupts	576 ff
IOCTL	328
IOCTL\$	328



IPmt	444
IRR	445
ISAM	4, 9, 77 ff, 245
Details	95 ff
Referenz	421
Sortierung	83, 573 ff
Transaktion	83
ISAMCVT	97
ISAMDEMO.BAS (Prog.)	85
ISAMIO	99 ff
ISAMPACK	102
ISAMRPR	102
ISORT.BAS (Prog.)	148

## K

KEY	329
KEY (Trapping)	330
KILL	332
Klammern in Formeln	13
Kompilieren, separates	61 ff
Konfiguration ermitteln	579

## L

LabelChartH	479
LabelChartV	480
Laufwerk	581, 582
LBOUND	333
LCASE\$	333
LEFT\$	333
LegendType	131
LEN	334
LET	334
.LIB (Extension)	62
LIB	
(Umgebungsvariable)	7
LIB.EXE	247 ff
Library	7, 45 ff, 250
Limits	521 ff
LINE	334
LINE INPUT	336
LINGL.BAS (Prog.)	111
LINK.EXE	68 ff

Link-Steuerungsdatei	73
ListBox	498
LoadFont	462
LOC	336
LOCATE	337
LOCK, UNLOCK	337
LOF	338, 427
LOG	338
Logische Verknüpfung	15
LONG (Datentyp)	13
LPOS	339
LPRINT	339
LSET	340
LTRIM\$	342

## M

MakeChartPattern\$	480
MatAddr	453
Mathematik-Bibliothek	195
MatDetx	455
MatInvx	455
MatMultx	454
Matrizenmathematik-Toolbox	110 ff
Referenz	453
MatSEqnx	456
MatSubx	453
Maus-Funktionen	585 ff
Maus-Routinen Referenz	509
MaxScrollLength	500
MenuCheck	482
MenuColor	483
MenuEvent	483
MenuInit	484
MenuInkey\$	484
MenuItemToggle	485
MenuOff	486
MenuOn	486
MenuPreProcess	487
MenuSet	487
MenuSetState	489
MenuShow	489
Menü	35 ff, 40 ff
Debug	41
Edit	37

File	35
Help	39
Options	39
Run	40
Utility	38
Menü-Routinen Referenz	482
Mergesort	149
MID\$	342
Minute	436
MIRR	446
MKDIR	343
MKKEY	267 ff
MKx\$	343
MKxMBF\$	344
MOD (Operator)	13
Modul	34
Modulcode	34
Month	436
MouseBorder	509
MouseDriver	509
MouseHide	510
MouseInit	511
MousePoll	511
MouseShow	511
MOVExxx	428
MSORT.BAS (Prog.)	150
Musterpalette	135

## N

NAME	344
Neue Funktionen	273
NEW-VARS.BAT / CMD 7	
NMAKE	54 f, 253 ff
Makros	260 ff
NMK.COM	264
Now	437
NPer	447
NPV	447
numerische Formatierung	105

## O

.OBJ (Extension)	61
------------------	----

OCT\$	345
ON ERROR	345
ON <i>event</i> GOSUB	346
ON...GOSUB,GOTO	348
OPEN	348, 428
Operator	13, 14
Optimierung	4, 227 ff
Ausführungsgeschwindigkeit	227
Programmgröße	231
Verzicht-Files	234
OPTION BASE	353
OS/2	50, 221 ff
OUT	354
OutGText	465
Overlay	4, 189

## P

PAINT	354
PALETTE	355
Parameter	18
Parameterdatei	165
PATH	6
PCOPY	356
PEEK	357
PEN	358
Pfad-Angabe	6
PLAY	359
PMAP	361
Pmt	448
POINT	361
POKE	362
POS	362
PPmt	449
Presentation Graphics Toolbox	124 ff
Referenz	470
PRESET	363
PRINT	363
Professional Development System	
vi, 4, 5 ff.	
Programm	35
Programmer's Workbench	11, 49 ff
Aufruf	51
Programmgröße	231
Programmieren, rekursiv	158, 163

Programmliste	54
PROISAM.EXE	94
Prozedurcode	34
Prozeduren	18
PSET	366
PUT	366, 367
PutBackground	516
PV	449

## Q

QBX	27, 49
.QLB ( <i>Extension</i> )	62
QSORT.BAS ( <i>Prog.</i> )	145
Quick Library	45 ff
QuickSort	145

## R

RANDOMIZE	368
Rate	450
READ	369
Rechner booten	580
Records ( <i>Datentyp</i> )	13, 16 ff
REDIM	370
RegionType	129
RegisterFonts	465
RegisterMemFont	466
Rekursives Programmieren	158, 163
REM	371
RESET	372
ResetPaletteDef	481
RESTORE	372
RESUME	372
RETRIEVE	430
RETURN	373
RIGHT\$	374
RMDIR	374
RND	374
ROLLBACK	431
RSET	375
RTRIM\$	375
RUN	376

## S

SADD	377
SAVEPOINT	431
SCREEN	23, 377, 378
Scroll	517
Schreiben	
mit Farbe	577
ohne Farbe	577
Schriften	115 ff
Second	437
SEEK	385, 386
SEEKxx	432
SEGFSR.BAS ( <i>Prog.</i> )	180
SELECT CASE	387
SelectFont	466
Separates Kompilieren	61 ff
SetFormatCC	442
SetGCharSet	467
SetGTextColor	468
SetGTextDir	469
SETINDEX	433
SetMaxFonts	469
SETMEM	389
SetPaletteDef	481
SGN	390
SHARED	20, 390
SHELL ( <i>Funktion</i> )	391
SHELL ( <i>Befehl</i> )	392
ShortCutKeyDelete	490
ShortCutKeyEvent	490
ShortCutKeySet	490
SIGNAL	393
SIN	393
SINGLE ( <i>Datentyp</i> )	13, 96
SLEEP	393
SLN	451
Sortieren	145 ff
SOUND	394
Source Browser	50, 56 ff
SPACE\$	394
SPC	394
Speichergröße ermitteln	579, 580
SQR	395
SSEG	395

SSEGADD	396
STACK	396
Standard-BASIC Referenz	273
Standard-Runtime-Module	241
STATIC	397
\$STATIC	397
STICK	399
STOP	400
STR\$	400
STRIG	401
STRING (Datentyp)	13
STRING\$	402
SUB/FUNCTION	19, 402
SUCHBIN.BAS (Prog.)	154
SUCHBIND.BAS (Prog.)	155
Suchen	153 ff
SWAP	406
Switches	524 ff
SYD	451
SYSTEM	406

## T

TAB	406
TAN	407
Tastaturcodes	564
Tastaturstatus	580
Tastatur umbelegen	267 ff
TEXTCOMP	433
Text-Datenbank	173
Textmodus	23
TIMES	408
TIMER (Funktion)	408
TIMER (Befehle)	409
TimeSerial	437
TimeValue	438
TitleType	129
Toolboxen	4, 109 ff
TOOLS.INI	52
TRON/TROFF	409
TYPE	410

## U

UBOUND	411
UCASE\$	411
UEVENT	411
Umfangreiche Programme	187
UnRegisterFonts	469
UPDATE	434
User-Interface-Toolbox	137
Referenz	482

## V

VAL	412
Variablen	13 ff, 18 ff, 175
Boolesche V.	175
Far Strings	178
Near / Far Strings	179
Statische / automatische V.	184
Statische / dynamische Felder	182
Strings	177
VARPTR	412
VARPTR\$	413
VARSEG	413
Verzichtfile	8, 234
Videomodus ermitteln	578
VIEW	414
VIEW PRINT	415

## W

WAIT	416
Weekday	439
WHILE...WEND	417
WIDTH	417
WINDINC.BAS (Prog.)	140
WINDOW	419
WindowBox	500
WindowClose	501
WindowCls	501
WindowColor	502
WindowCols	502
WindowCurrent	503
WindowDo	503

<b>WindowInit</b>	<b>504</b>
<b>WindowLine</b>	<b>504</b>
<b>WindowLocate</b>	<b>504</b>
<b>WindowNext</b>	<b>505</b>
<b>WindowOpen</b>	<b>505</b>
<b>WindowPrint</b>	<b>506</b>
<b>Window-Routinen Referenz</b>	<b>492</b>
<b>WindowRows</b>	<b>507</b>
<b>WindowScroll</b>	<b>508</b>
<b>WindowSetCurrent</b>	<b>508</b>
<b>WRITE</b>	<b>420</b>

## **Y**

<b>Year</b>	<b>439</b>
-------------	------------

## **Z**

<b>Zeichencodes</b>	<b>564</b>
<b>Zeichen ermitteln</b>	<b>580</b>
<b>Zeilen fortsetzen</b>	<b>vii</b>
<b>Zeitcode-Formatierung</b>	<b>105</b>

# BASIC PDS 7.1

Programmieren mit dem Microsoft Professional Development System

## Was das Buch bietet . . .

eine kompetente Anleitung für den Umgang mit der professionellen Entwicklungsumgebung Microsoft BASIC PDS 7.1

## Worum es geht . . .

- ▶ Installation
- ▶ Die Entwicklungsumgebung QBX
- ▶ Die Entwicklungsumgebung PWB
- ▶ Das ISAM-Datenbanksystem
- ▶ Add-On Libraries
- ▶ Toolboxes

## Und außerdem . . .

- ▶ Praktische Algorithmik
- ▶ Optimierungsverfahren
- ▶ Zusatzprogramme zum Compiler
- ▶ Referenzteil

## Was der Leser benötigt . . .

- ▶ HARDWARE: IBM AT/80386/80486 und Kompatible
- ▶ SOFTWARE: MS/PC-DOS ab Version 3.0 und MS-Basic PDS 7.1

## Besondere Kennzeichen . . .

- ▶ Das Buch stellt die umfangreichen Möglichkeiten der Basic PDS-Entwicklungsumgebung dar. Zudem gibt es dem Benutzer zahlreiche praxisorientierte Tips zur effizienten Programmierung und zur Optimierung seiner Programme.

## Der Autor . . .

- ▶ ist ein hochkarätiger Kenner von im professionellen Entwicklungsbereich einsetzbaren BASIC-Dialekten.

	Hardware	Tools	Programmierung	Standard-Software
Einsteiger				
Fortgeschrittene			<b>BASIC PDS 7.1</b>	
Profis				



09800



9 783528 151522

ISBN N 3-528-15152-8 DM +098.00

**Buchregal-Hinweis:**  
Programmiersprache

**inkl. Diskette**

