ATARI ST

Maier

# DAS GROSSE ST_BASIC BUCH

*OMIKRON.basic interpreter + compiler*

DATA BECKER

Michael Maier

# Das große ST-BASIC-Buch

DATA BECKER

1. Auflage 1988

ISBN 3-89011-283-8



DATA BECKER GmbH
Merowingerstr. 30
4000 Düsseldorf

Text verarbeitet mit Word 4.0, Microsoft
Ausgedruckt mit Hewlett Packard LaserJet II
Druck und Verarbeitung Graf und Pflügge, Düsseldorf

Wichtiger Hinweis:
Die in diesem Buch wiedergegebenen Schaltungen, Verfahren und Programme werden ohne Rücksicht auf die Patentlage mitgeteilt. Sie sind ausschließlich für Amateur- und Lehrzwecke bestimmt und dürfen nicht gewerblich genutzt werden.

Alle Schaltungen, technischen Angaben und Programme in diesem Buch wurden von dem Autoren mit größter Sorgfalt erarbeitet bzw. zusammengestellt und unter Einschaltung wirksamer Kontrollmaßnahmen reproduziert. Trotzdem sind Fehler nicht ganz auszuschließen. DATA BECKER sieht sich deshalb gezwungen, darauf hinzuweisen, daß weder eine Garantie noch die juristische Verantwortung oder irgendeine Haftung für Folgen, die auf fehlerhafte Angaben zurückgehen, übernommen werden kann. Für die Mitteilung eventueller Fehler ist der Autor jederzeit dankbar.

Für Birgit, Michaela und Eva

## Auf ein Wort

Ein Buch über BASIC zu schreiben, das wäre mir, einem eingefleischten C-Programmierer, der sich auch - wenn es unbedingt sein muß - in die tieferen Sphären der Assemblerprogrammierung vorwagt, vor gar nicht allzu langer Zeit nicht einmal im Traum eingefallen.

Zu tief mir saß noch der Schreck meiner ersten Begegnung mit dem ST-BASIC in den Knochen, denn schon die einfachsten Befehle bescherten mir den Genuß zahlreicher Bömbchen auf dem Monitor. Um ehrlich zu sein, ich habe meinen Computer nie wieder mit diesem BASIC belästigt.

Abhilfe von diesem Mißstand erhoffte sich die ST-Fan-Gemeinde (ich nicht mehr ...) vom englischen Softwarehaus Metacomco, das einen neuen BASIC-Interpreter entwickeln sollte. Als dieser dann mit einiger Verzögerung erschien, hatte man zwar auf Fehlerfreiheit geachtet, doch Geschwindigkeit schien auch für die Programmierer von Metacomco ein Fremdwort zu sein. Nein, auch dieses BASIC war - trotz seines konkurrenzlosen Preises - nicht das Gelbe vom Ei.

Jetzt, drei Jahre nachdem der ST der staunenden Fachwelt vorgestellt wurde, unternimmt Atari einen dritten Anlauf in Sachen ST-BASIC. Gerüchte darüber kursierten schon länger in ST-Kreisen, unklar war jedoch, für welchen Interpreter man sich bei Atari entscheiden würde. Die Gerüchteküche glaubte zu wissen, daß es sich dabei nur um Omikron.- oder GfA-BASIC handeln konnte, zwei Interpreter deutscher Produktion, die beweisen, daß BASIC keine lahme, unstrukturierte Programmiersprache aus grauer Vorzeit sein muß.

Die Würfel fielen zugunsten des Omikron.BASIC, das sich in seiner neuen Version 3.00 jetzt ST-BASIC nennen darf. Mit diesem Schritt hat Atari ein dunkles Kapitel in der Geschichte des ST zugeschlagen. Vorbei sind die Zeiten, da BASIC-Pro-

grammierer auf die teueren Interpreter anderer Firmen zurückgreifen mußten, nur weil das mitgelieferte BASIC einfach nicht zu gebrauchen war.

Doch mit den wachsenden Möglichkeiten dieser mächtigen Programmiersprache ist es bestimmt nicht leichter geworden, gute Programme zu schreiben. Man muß dazu nur nicht mehr auf eine Compilersprache zurückgreifen, sondern kann dies ab sofort auch in BASIC erledigen. Geblieben ist als unabdingbare Voraussetzung neben dem Beherrschen der verwendeten Programmiersprache das nötige Wissen um die Interna des Computers.

Und genau dies ist der springende Punkt. BASIC, verschrien als kinderleicht zu erlernende Programmiersprache, die höchstens blutige Anfänger hinter dem Ofen hervorzulocken vermag, verschrien, ermöglicht plötzlich die Erstellung professioneller Software. Der Preis, den man für diese Errungenschaft zahlen muß, ist der Verlust jener Einfachheit, die BASIC einst zum Siegeszug verholfen hat. Trotz alledem ist es aber immer noch leichter, ein professionelles Programm in BASIC zu entwickeln als in einer Compilersprache oder gar in Assembler.

Dieses Buch richtet sich deshalb an alle, die etwas tiefer in die Materie einsteigen möchten, um die fantastischen Möglichkeiten, mit denen das neue ST-BASIC aufwarten kann, voll auszukosten. Ich wünsche Ihnen viel Spaß bei der Lektüre und der Umsetzung des neu erworbenen Wissens in eigene Projekte!

# Inhaltsverzeichnis


**1. Der Editor ..... 15**

1.1 ST-BASIC laden ..... 15
1.2 Der Bildschirm-Editor ..... 16
1.3 Der Full-Screen-Editor ..... 20
1.4 Der Fullscreen-Editor des Omikron.BASIC ..... 32
1.5 ST-Omikron.BASIC verlassen ..... 38

**2. ST-BASIC Grundkurs ..... 39**

2.1 Ausgabe auf dem Monitor mit PRINT ..... 39
2.2 Variablentypen in ST-BASIC ..... 42
2.3 Das erste Programm ..... 49
2.4 Programme speichern, laden und löschen ..... 55
2.5 Wie sag ich's dem Computer? ..... 58
2.6 Mathematische Funktionen ..... 61
2.7 Strings und Stringmanipulation ..... 67
2.8 Variablenfelder ..... 90
2.9 Programmierhilfen ..... 92
2.10 Strukturierte Programmierung ..... 99
2.11 Alles eine Frage der Routine ..... 121
2.12 READ ..... 140
2.13 BASIC-Allerlei ..... 143

**3. Dateiverwaltung ..... 181**

3.1 Dateien auf Diskette ..... 181
3.2 Ohne Kanäle geht gar nichts ..... 184
3.3 Noch ein Print, aber mit Write geht's auch ..... 186
3.4 Sequentielle Dateien einlesen ..... 188
3.5 Files kopieren ..... 191
3.6 Die File-Selector-Box ..... 195
3.7 Fehler abfangen ..... 204
3.8 Backup-Dateien ..... 207

3.9 Wir sorgen für Ordnung auf der Diskette ..... 211
3.10 Relative Dateien ..... 213
3.11 Minidatei - diesmal relativ ..... 217
3.12 Warum das Rad noch einmal erfinden? ..... 230
3.13 Schwarz auf Weiß - Druckerausgabe ..... 236
3.14 Mehrzeilige Suchvorgaben mit OR mit AND ..... 240

**4. Das Betriebssystem des Atari ST ..... 241**

4.1 Systemvariablen ..... 241
4.2 TOS, GEMDOS, BIOS und XBIOS ..... 246
4.3 GEMDOS ..... 248
4.4 Das BIOS ..... 257
4.5 Das XBIOS ..... 260
4.6 Betriebssystemprogrammierung ..... 270

**5. Grafikprogrammierung ..... 279**

5.1 Einfache Grafikbefehle ..... 279
5.2 BITBLT ..... 283
5.3 Bildschirm laden ..... 285
5.4 Objekte verschieben und drehen ..... 287

**6. GEM ..... 291**

6.1 Arbeiten mit dem RCS ..... 292
6.2 GEM-Programmierung unter ST-BASIC ..... 301
6.3 Schieberegler ..... 332
6.4 Pull-Down-Menüs ..... 333
6.5 Eine eigene File-Selector-Box ..... 344
6.6 Fenstertechnik ..... 357

**7. Multitasking ..... 367**

**8. Der Compiler ..... 371**

8.1 Die Bedienung des Compilers ..... 372
8.2 Compiler-Steuerworte ..... 373
8.3 BASIC-Programme auf dem Compiler ..... 375
8.4 Programme optimieren ..... 377
8.5 Fehlermeldungen des Compilers ..... 378
8.6 Verarbeitungstypen der Funktionen ..... 379
8.7 Hilfsprogramme auf der Compilerdiskette ..... 380

**Anhang ..... 383**

Anhang A: ASCII-Tabelle ..... 383
Anhang B: Scancodes des Atari ST ..... 385
Anhang C: Verzeichnis aller VT-52-Sequenzen ..... 386
Anhang D: Fehlermeldungen ..... 387
Anhang E: TOS Fehlermeldungen ..... 393
Anhang F: GfA-BASIC-Programme umschreiben ..... 394

**Index ..... 395**

# 1. Der Editor

Es ist nicht möglich, ein Programm zu schreiben, ohne das Handwerkszeug zu beherrschen, das man dafür benötigt. Im Falle des ST- und Omikron.BASIC sind dieses Handwerkszeug zwei Editoren, mit deren Hilfe die Befehle eingegeben oder gar ganze Programme erstellt, abgespeichert, verändert und wieder geladen werden können. Deshalb ist ihnen - ehe wir zu größeren Taten schreiten - das erste Kapitel in diesem Buch gewidmet.

Leider ist es nicht möglich eine komplette Beschreibung aller Funktionen zu liefern, ohne dabei auf Fachausdrücke zurückzugreifen. Lesen Sie sich das folgende Kapitel zunächst einmal in Ruhe durch. Alle Funktionen, die für den Anfang wichtig sind, wie z.B. das Laden des ST-BASIC, sind so einfach wie nur irgendwie möglich erklärt. Und die übrigen Dinge, die kompliziert erscheinen, werden an späterer Stelle in diesem Buch unter Garantie noch einmal aufgegriffen.

Im übrigen empfehle ich Ihnen, dieses Kapiel noch einmal nach der Lektüre des Buches zu lesen. Spätestens dann werden Sie alle Funktionen verstehen!

## 1.1 ST-BASIC laden

Ehe mit dem Editor gearbeitet werden kann, muß zunächst einmal das BASIC von Diskette geladen werden. Legen Sie dazu die ST- bzw. Omikron.BASIC-Diskette in das Laufwerk, und klicken Sie zweimal kurz hintereinander mit der linken Maustaste auf das Symbol Laufwerk A. Jetzt öffnet sich ein Fenster, in dem alle auf der Diskette enthaltenen Dateien angezeigt werden.

Setzen Sie nun den Mauspfeil auf OM_BASIC, und starten den Ladevorgang durch einen erneuten Doppelklick mit der linken Maustaste. Sobald sich das Programm im Speicher befindet, meldet sich der BASIC-Interpreter mit dem Bildschirm-Editor.

Die folgenden Erklärungen beziehen sich auf den Editor des neuen ST-BASIC (Omikron.BASIC 3.00). Damit aber auch alle anderen, die mit einer älteren Version (kleiner als 3.00) arbeiten müssen, weiterlesen können, ist dem Full-Screen-Editor der älteren Versionen ein eigenes Kapitel gewidmet. Dieses Kapitel ist übrigens auch für alle ST-BASIC-Benutzer relevant, da hier die zusätzlichen Möglichkeiten der Editor-Steuerung über die Funktionstasten beschrieben werden.

## 1.2 Der Bildschirm-Editor

Nachdem Omikron.BASIC geladen wurde, erscheint in der linken oberen Ecke die Startmeldung mit der Versionsnummer. Ein paar Zeilen darunter finden Sie einen blinkenden Punkt, den Cursor. Mit Hilfe der Pfeiltasten, die sich zwischen dem Hauptblock der Tastatur und dem seperaten Zehnerblock befinden, kann der Cursor in alle vier Richtungen bewegt werden.

```
***     OMIKRON.BASIC V3.0     © OMIKRON.Software     ***
- Press [Help] to enter editor -
1788142 bytes free.
OK
```

*Abb. 1.1: Der Bildschirm-Editor nach dem Laden*

Eine Betätigung der Taste <Home> setzt den Cursor in die linke obere Ecke des Bildschirms, während ein gleichzeitiges Drücken der Tasten <Control> und <Home> den Bildschirm löscht, und den Cursor in die linke obere Bildschirmecke setzt.

### Zeichen löschen

<Backspace> löscht das Zeichen, das sich links des Cursors befindet, indem der Cursor um eine Position nach links gefahren wird. Der Rest der Zeile bleibt in seiner Position unverändert. Mit <Delete> kann ebenfalls ein Zeichen vom Bildschirm getilgt werden. <Delete> sorgt jedoch dafür, daß der gesamte Zeilenrest hinter dem Cursor um eine Position nach links mitwandert. Ein Beispiel verdeutlicht dies wohl am schnellsten:

```
Das große ST-BASIC*Buch
```

Die Cursorposition wird durch ein Sternchen (*) markiert. <Backspace> führt nun zu:

```
Das große ST-Basi* Buch
```

<Delete> dagegen zieht das Wort 'Buch' um eine Position nach links, und verhindert somit das Entstehen von Leerzeichen:

```
Das große ST-Basi*Buch
```

### Zeichen einfügen

Möchten Sie jedoch ein Zeichen ergänzen, können Sie mit <Insert> an der aktuellen Cursorposition ein Leerzeichen einfügen. Der Cursor behält dabei seine Position bei, der Zeilenrest rechts des Cursors wandert um eine Position weiter:

```
Das gro ße ST-BASIC Buch
```

Befindet sich der Cursor über dem 'e', während <Insert> gedrückt wird, erscheint an dieser Position ein Leerzeichen:

```
Das gro e ST-BASIC Buch
```

Jetzt kann der fehlende Buchstabe eingesetzt werden.

### Zeile einfügen und löschen

Möchten Sie nicht nur einen Buchstaben einsetzen oder löschen, sondern gleich ganze Zeilen, müssen Sie folgende Tasten drüken:

*<Control> <Cursor hoch>*
Löscht die Zeile, in der sich der Cursor gerade befindet.

*<Control> <Cursor runter>*
Fügt eine Zeile ein.

*<Control> <Delete>* Löscht den Zeilenrest hinter dem Cursor.

### Der Insert-Modus

Normalerweise werden Zeichen, die sich unter der aktuellen Cursorposition befinden, einfach überschrieben. Dies ist jedoch nicht immer wünschenswert. Möchten Sie ein Zeichen an der Cursorposition eingefügen, ohne daß das darunterliegende Zeichen überschrieben wird, müssen Sie mit

*<Control> <Insert>* den Insert-Modus einschalten. Eine nochmalige Betätigung dieser Tastenkombination schaltet den Insert-Modus wieder aus. Dieser Modus zieht auch eine Veränderung der Funktionsweise von <Backspace> und <Delete> nach sich:

*<Backspace>* übernimmt die Funktion, die <Delete> im Normalmodus besitzt.

*<Delete>* löscht das Zeichen unter dem Cursor, und läßt den Zeilenrest nachrücken, der Cursor behält seine Position jedoch bei (rückt also nicht nach links!).

### Der VT52-Emulator

Der Monitor des Atari ST ist in gewisser Weise intelligent. Er versteht nämlich bestimmte Befehle, wie das Löschen des Bildschirms, das Merken der aktuellen Cursorposition, usw. Diese Befehle werden allesamt mit der Taste <Esc> (Escape) eingeleitet, gefolgt von einem (oder mehreren) Buchstaben. Im Anhang dieses Buches finden Sie eine Auflistung sämtlicher Sequenzen des VT52-Standards. Diese Funktionen können Sie auch im Bildschirm-Editor nutzen. Drücken Sie dazu einfach zuerst die Taste <Esc> und anschließend den dazugehörigen Buchstaben, und sofort wird der Befehl ausgeführt.

### Link-Zeilen

Auf dem Monitor des Atari können maximal 80 Zeichen in einer Zeile dargestellt werden. Werden Zeilen eingegeben, die länger als 80 Zeichen sind, schreibt der Editor einfach in der folgenden Zeile weiter. Damit auch optisch erkennbar ist, daß der Inhalt dieser Zeile nur die Fortsetzung der vorhergehenden Zeile darstellt, erscheint am linken Rand ein senkrechter Strich (|), der im Fachjargon Pipe genannt wird. Auch bei einer anderweitigen Ausgabe, die den rechten Rand des Monitors überschreiten würden, wird in der darauffolgenden Zeile weitergeschrieben, die dann wieder mit einem Pipe-Zeichen beginnt.

### Sonstige Tastenkombinationen

| | |
|---|---|
| *<Return>* | Schreibt die Zeile, in der sich der Cursor befindet, in den Programmspeicher, wenn sie mit einer Zahl beginnt, ansonsten faßt der Interpreter den Zeileninhalt als Befehle auf, die sofort ausgeführt werden müssen. Deshalb spricht man in diesem Fall auch von Direktmodus. |
| *<Control><Cursor links>* | Setzt die linke obere Ecke des Rahmens. |

*<Control><Cursor rechts>* Setzt die rechte untere Ecke des Rahmens.

Der Rahmen dient dazu, Ausgaben (z.B. über PRINT) nicht mehr auf dem gesamten Bildschirm darstellen zu lassen, sondern nur in einem bestimmten Bereich, der durch diesen Rahmen begrenzt wird. Auch der Cursor kann dann nur noch innerhalb dieses Rahmens bewegt werden.

*<Home><Home>* Eine zweimalige Betätigung der Taste <Home> löscht den Rahmen wieder. Ab sofort kann der Cursor in den gesamten Bereich des Bildschirms gesetzt werden.

*<Alternate><Control>* Löscht den Tastaturpuffer. In diesem Puffer speichert ST-BASIC alle gedrückten Tasten, die nicht sofort zur Ausführung gelangen können (z.B. bei einem Tastendruck, während das Programm arbeitet).

*<Shift><Shift>* Werden beide Shift-Tasten des ST gleichzeitig gedrückt, so wird auf den Screen (Bilschirm) umgeschaltet, der vor der Rückkehr in den Direktmodus vom Programm benutzt wurde.

## 1.3 Der Full-Screen-Editor des ST-BASIC (Version 3.00)

Der Full-Screen-Editor dient zur Eingabe Ihres Programmtextes, während der Bildschirm-Editor üblicherweise nur zur Ausführung von Kommandos im Direktmodus benutzt wird. Im Gegensatz zum Bildschirm-Editor ist er mit einer ganzen Reihe von Funktionen ausgestattet, die die komfortable Eingabe eines Pro-

grammtextes ermöglichen. So besitzt dieser Editor unter anderem eine Menüleiste, über die sämtliche Funktionen mit der Maus angewählt werden können.

Diesen Editor erreichen Sie aus dem Bildschirm-Editor heraus über eine Betätigung der Taste <Help>. Mit <Control> <c> können Sie den Editor wieder verlassen. Auch ein Anklicken des Menüpunktes Quit Edit, den Sie in der Menüleiste unter File finden, bewirkt ein Verlassen des Full-Screen-Editors.

### Die Menüzeile

In der ersten Zeile finden Sie die Menüleiste, die folgende Menüpunkte beinhaltet: File, Find, Block, Mode, Go und Run. Um einen Menü-Aufruf abzuwickeln, fährt man mit dem Maus-pfeil auf einen dieser Menünamen. Automatisch rollt dabei das Menü herunter und gibt seine Funktionen preis, die durch eine Berührung mit dem Mauszeiger invertiert werden. Durch Drücken der linken Maustaste wird ein invertierter Eintrag angewählt.

Die Menüleiste beinhaltet jedoch noch eine weitere Funktion: Es lassen sich nämlich auch die einzelnen Punkte der Menüleiste anklicken. In diesem Fall wird dann die Funktion ausgelöst, die an erster Stelle innerhalb des entsprechenden Menüpunktes zu finden ist.

### Der Menüpunkt File

Unter diesem Menüpunkt sind sämtliche Funktionen zusammengefaßt, die zum Laden und Speichern von Programmen benötigt werden:

*Save *.**
Speichert das Programm, das sich im Computer befindet, unter dem Namen, der in die File-Selector-Box eingetragen wird, ab.

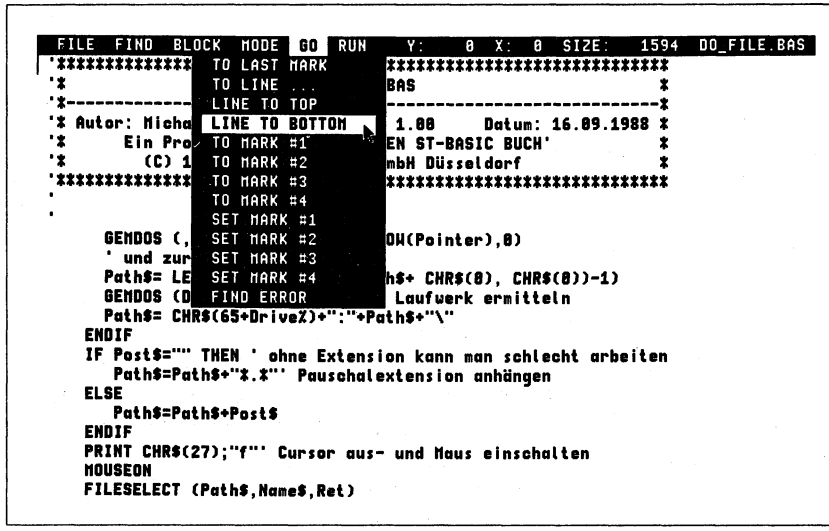



*Abb. 1.2: Der Full-Screen-Editor*

*Load *.**
Lädt ein Programm in den Speicher. Der Name muß wieder über eine File-Selector-Box eingegeben werden.

*Save Block*
Speichert einen markierten Block ab.

*Load Block*
Lädt einen mit Save Block auf Diskette gespeicherten Block (bzw. einen ASCII-Text) von Diskette und fügt ihn gemäß den angegebenen Zeilennummern in ein evtl. schon existierendes Programm ein.

*Directory*
Zeigt das Inhaltsverzeichnis der Diskette an.

*New*
Löscht ein Programm im Speicher.

*Quit Edit*
Verlässt den Full-Screen-Editor.

### Der Menüpunkt Find

Dieser Menüpunkt enthält Funktionen, mit deren Hilfe Wörter (Buchstaben) bzw. Tokens innerhalb des Programms gesucht werden können. Bei Find (Suchen) wird zwischen allgemeinen Suchbegriffen oder Befehlswörtern (Tokens) unterschieden. Während der Interpreter bei normalen Suchbegriffen die Zeichenkette immer dann als gefunden meldet, wenn er sie irgendwo im Programm antrifft - dies kann auch innerhalb von Worten sein, funktioniert die Suche nach einem Token etwas anders:

Um Speicherplatz zu sparen, legen BASIC-Interpreter ihre Befehle nicht im Klartext (Print), sondern als kurze Befehlscodes ab. Diese Befehlcodes bezeichnet man als Tokens. Die Suche nach einem Token klappt folglich nur dann, wenn die gesuchte Buchstabenkombination in einen Token umgewandelt wurde. Neben den eigentlichen Befehlen werden auch die Variablennamen als Tokens im Programmtext abgelegt, und erst kurz vor der Ausgabe auf dem Monitor in Klartext gewandelt.

Verwendet man Find Token bei der Suche nach einem Variablennamen, kann man sicher sein, daß nur die gewünschte Variable als gefunden gemeldet wird, nicht aber Variablen, die diese Buchstabenkombination irgendwo in ihrem Namen tragen. Möchten Sie mit Find Token eine Feldvariable ansprechen, können Sie folgendes als Suchbegriff eingeben:

*Array( )*
Es wird nach einer Feldvariable mit einem Index gesucht, Array(5,9) würde folglich nicht gefunden.

*Array(,,)*
Sucht nach der Feldvariable, die drei Indizes trägt.

Für Prozedur- und Funktionsdefinitionen sowie für Marken gelten bei der Eingabe des Such-String eigene Regeln:

- Eine Prozedur muß bei der Eingabe des Suchbegriffes vor dem Namen ein P erhalten: P Zeile() sucht die Definition der Procedure Zeile mit einem Übergabeparamter. Tauchen mehrere gleichnamige Prozeduren auf, die sich nur durch die Anzahl ihrer Parameter unterscheiden, muß ab dem 2. Parameter je ein Komma pro Übergabeparameter angegeben werden: P Zeile(,) sucht nach einer Definition mit 2 Parametern.

- Funktionsdefinitionen muß ein FN vorangestellt werden, damit sie via Find Token ermittelt werden können.

- Marken müssen also solche durch ein dem Namen vorangestelltes Minuszeichen kenntlich gemacht werden.

### Die Einträge in diesem Menüpunkt

*Find Next*
Setzt die Suche nach einem bereits eingegebenen Suchbegriff fort. Der Suchbegriff muß mit dem nächsten Eintrag eingegeben werden:

*Find...*
Gestattet die Eingabe eines Suchbegriffes. Findet der Computer die entsprechende Buchstabenkombination im Programm, so positioniert er den Cursor an diese Stelle. Ansonsten ertönt ein Glockenton.

*List...*
Durchsucht den gesamten Text und listet alle Stellen auf, an denen der Suchbegriff auftaucht.

*Replace all...*
Durchsucht das gesamte Programm nach der eingegebenen Buchstabenkombination. Stößt der Interpreter auf eine solche, wird sie ohne Rückfrage durch den neuen Text ersetzt.

*Query Replace*
Sucht einen bestimmten Text, und ersetzt ihn nach einer entsprechenden Rückmeldung durch einen anderen.

*Find Token*
Sucht nach Befehlswörtern (Variablen). Variablen, in denen diese Buchstabenkombination ebenfalls vorkommt, werden bei der Suche ignoriert.

*List Token*
Sucht nach Befehlswörtern (Variablen) und invertiert alle Stellen, an denen das Wort vorkommt.

*Rename Token*
Ersetzt Befehlswörter

*List to Printer*
Stellt auf Druckerausgabe um. Alle künftigen LIST-Befehle werden auf dem Drucker ausgegeben.

*Find Error*
Der Cursor wandert an den Anfang oder das Ende des Wortes, in dem ein Syntax- oder Type Mismatch Error erkannt wurde.

### Der Menüpunkt Block

Unter diesem Menüpunkt sind die Blockoperationen zusammengefaßt, die unter dem Full-Screen-Editor zur Verfügung stehen. Ehe mit einem Block gearbeitet werden kann, muß dieser erst einmal definiert werden. Dazu stehen Ihnen zwei Möglichkeiten offen:

1. Sie klicken mit der Maus an die Stelle, die die linke obere Ecke des Blockes bilden soll. Während Sie die Maustaste festhalten, fahren Sie den Mauszeiger an die Position, an der der Block enden soll. Lassen Sie nun die Taste los, der Block ist als solcher definiert.

2. Setzen Sie den Cursor auf das erste Zeichen, das der Block enthalten soll, und wählen den Menüpunkt Mark Block Start. Somit wäre der Blockanfang schon einmal markiert. Im nächsten Schritt setzen Sie dann den Cursor in die Zeile, mit der der Block enden soll, und klicken danach den Menüpunkt Mark Block End in der Menüleiste an. Damit ist der Block fertig definiert.

Bei der Definition sind zwei Formen von Blöcken zu unterscheiden:

- einzeilige Blöcke, die auch Ausschnitte einer Zeile enthalten können.
- mehrzeilige Blöcke, die immer mit dem Anfang der ersten Zeile beginnen, und dem Ende der letzten Zeile enden.

Die Einträge dieses Menüpunktes im einzelnen:

*Insert*
Kopiert einen definierten Block an die aktuelle Cursorposition

*Move*
Verschiebt einen Block an die neue Cursorposition. Der Block wird an seiner alten Position gelöscht.

*Kill*
Löscht einen Block unwiderruflich.

*Mark Bl. Start*
Die aktuelle Cursorzeile markiert den neuen Blockanfang. Mit dieser Funktion kann entweder ein neuer Block definiert (umständlich) oder die Blockobergrenze verschoben werden.

*Mark Bl. End*
Die aktuelle Cursorzeile wird zum neuen Blockende ernannt.

*Save Block *.**
Speichert einen definierten Block im ASCII-Format auf Diskette.

*Load Block *.**
Lädt einen Block und fügt ihn an der Cursorposition ein.

*Print Block*
Gibt einen Block auf dem Drucker aus.

*Hide*
Löscht die Blockmarkierungen, ohne den Block selbst zu löschen. Seine Grenzen bleiben weiterhin erhalten.

### Der Menüpunkt Mode

Dieser Menüpunkt gestattet die Einstellung allgemeiner Punkte, wie das Einschalten des EinfügemodusEinfüge- oder Überschreibmodus, das Umschalten zwischen verschiedenen Bildschirmen, oder das Aufspalten in zwei Teilbildschirme:

*Insert*
Schaltet den Einfügemodus (Häkchen wird dem Eintrag vorangestellt) ein und wieder aus.

*Switch Screen*
St-BASIC verwaltet zwei Teilbildschirme, wenn der Menüpunkt Split Screen aktiv ist. Mit Switch Screen gelangt man von einem in den anderen Teilbildschirm. Auch ein Mausklick in den entsprechenden Bildschirmteil bewirkt diesen Effekt.

*Split Screen*
Teilt den Bildschirm in zwei Teile auf, wobei ein dicker Balken in der Mitte beide Teile voneinander trennt. Wird dieser Balken mit der Maus angeklickt, kann die Aufteilung der beiden Bildschirmhälften verschoben werden. Mit Switch Screen gelangt man von einem zum anderen Bildschirmfenster. In beiden Teilbereichen wird jedoch das gleiche Programm angezeigt. Eine Änderung des Programms in einem Bildschirmteil führt dadurch logischerweise auch zu einer Änderung des Programms im anderen Teilfenster.

*Change Size*
Verändert die Buchstabengröße. Mit dieser Funktion ist es möglich größere Programmteile auf den Monitor zu bekommen. Eine nochmaliges Aktivieren dieses Eintrages führt zur Wiederherstellung des alten Modus.

*Line Numbers*
ST-BASIC gestattet die Programmierung mit und ohne Zeilennummern. Dieser Menüpunkt schaltet den gewünschten Eingabemodus ein. *Vorsicht!* Wird der Editor bei abgeschalteter Zeilennumerierung verlassen, führt der Interpreter (Fehler??!!) eine Neunumerierung in Einerschritten durch. Vorhandene Sprungziele bleiben jedoch unverändert!

*Show Errors*
Sucht den Programmtext nach Fehlern ab

*Save Settings*
Speichert folgende Informationen in einer Datei OMIKRON.INF, die dann bei einem Neu start des Programms sofort eingestellt werden:

- Belegung der Funktionstasten
- aktuelle Auflösung

- angewählter Modus (Insert- Überschreibmodus)
- Zeilennumern ein bzw. aus
- Variablenart-Einstellungen über DEFXXX.

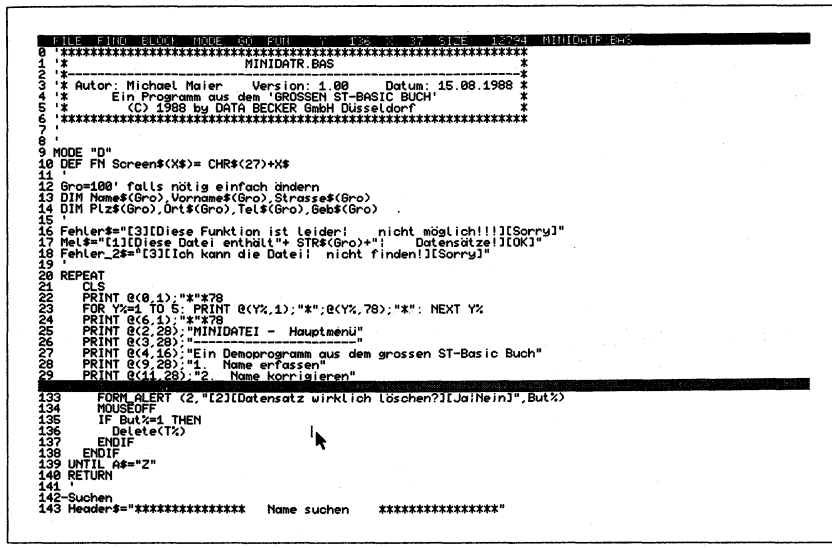



*Abb. 1.3: Der Full-Screen-Editor im Split-Modus*

## Der Menüpunkt GO

Der Menüpunkt GO beinhaltet unter anderem auch das Setzen und Anspringen von Marken. Eine Marke ist eine Stelle im Programm, die später auf Wunsch angesprungen werden kann.

*To last Mark*
Setzt den Cursor an die letze Marke

*To Line...*
Setzt den Cursor in die betreffende Zeile

*Line To Top*
Scrollt den Bildschirm weiter, bis die aktuelle Cursorzeile in der ersten Eingabezeile erscheint.

*Line to Bottom*
Scrollt den Bildschirm zurück, bis die aktuelle Cursorzeile am unteren Bildschirmrand angekommen ist.

*To Mark #X*
Diese vier Befehle dienen dazu, den Cursor an eine bestimmte Marke zu setzen.

*Set Mark #X*
Mit diesen vier Befehlen können die Marken 1 bis 4 definiert werden.

*Find Error*
Durchsucht das Programm nach Fehlern.

### Der Menüpunkt RUN

Dieser Menüpunkt gestattet das Starten und Aufrufen von Programmen:

*RUN*
Verläßt den Full-Screen-Editor und startet das Programm, das sich im Speicher befindet. Der gleiche Effekt kann auch mit <Control> <R> erzielt werden.

*Save & Run*
Speichert das Programm auf Diskette ab, ehe es ausgeführt wird.

*Tron & Run*
Schaltet TRON (Trace) ein, ehe das Programm gestartet wird.

*Compile*
Ruft den Omikron.Compiler Version 3.00 und größer auf. Besitzen Sie einen Compiler mit einer niedrigeren Versionsnummer, so muß das BASIC-Programm zuerst abgespeichert werden!

*Save & Compile*
Speichert das Programm vor dem Compileraufruf ab. (für Version kleiner 3.00). Der Editor erwartet den Compiler in Laufwerk C , falls ein solches angemeldet ist, ansonsten in Laufwerk A.

*Run *.Bas*
Lädt ein (BASIC-)Programm von Diskette und startet es sofort. Ein Programm das sich bis dato im Speicher befand, wird überschrieben.

*Exec *.Prg*
Dieser Menüpunkt lädt ein beliebiges Programm ein und führt es aus. Nach dem Programmende befinden Sie sich wieder im ST-BASIC. Sicherheitshalber sollte ein BASIC-Programm, das sich noch im Speicher befindet, abgespeichert werden, ehe ein anderes Programm aufgerufen wird. (Damit es Ihnen nicht so wie mir ergeht, und das neue Programm, weil es kein Blitter-TOS verträgt, nicht mehr verlassen werden kann. Tja, dann ist natürlich das BASIC-Programm im Speicher verloren! Nicht ganz: Versuchen Sie es in einem solchen Fall mit einem Reset, vielleicht haben Sie ja Glück und ST-BASIC meldet sich wieder).

*Accessory*
Schaltet auf ein GEM-Menü um, und gestattet somit das Aufrufen von gebooteten Accessories.

### BASIC-Befehle abkürzen

Folgende Befehle können abgekürzt werden, indem gleichzeitig die <Alternate>-Taste und der angegebene Buchstabe gedrückt wird:

| Abkürzung | führt zu |
|---|---|
| <Alternate><A> | ASC( |
| <Alternate><B> | BLOAD |
| <Alternate><C> | CONT |
| <Alternate><D> | DATA |
| <Alternate><E> | ELSE |
| <Alternate><F> | FOR |
| <Alternate><G> | GOTO |
| <Alternate><H> | HARDCOPY |
| <Alternate><I> | INPUT |
| <Alternate><K> | KEY |
| <Alternate><L> | LPRINT |
| <Alternate><M> | MID$( |
| <Alternate><N> | NEXT |
| <Alternate><O> | OPEN |
| <Alternate><P> | PRINT |
| <Alternate><R> | RETURN |
| <Alternate><S> | SYSTEM |
| <Alternate><T> | THEN |
| <Alternate><U> | USING |
| <Alternate><V> | VARPTR |
| <Alternate><W> | WHILE |
| <Alternate><X> | MOUSEX |
| <Alternate><Y> | MOUSEY |

Soviel zu den zusätzlichen Möglichkeiten des neuen ST-BASIC. Die im folgenden Kapitel stehenden Funktionen sind - soweit nicht anders angegeben - auch für Besitzer des neuen ST-BASIC relevant.

## 1.4 Der Fullscreen-Editor des Omikron.BASIC

Der Full-Screen-Editor wird bei allen BASIC-Versionen kleiner 3.00 mit dem Befehl EDIT (Return) betreten. Verlassen werden kann er mit der Tastenkombination <Control> <C>.

### Überschreib- und Einfügemodus

Auch der Full-Screen-Editor kennt die beiden Modi Einfügen und Überschreiben. Anhand der Cursorform kann man leicht erkennen, welcher Modus gerade aktiv ist: Besitzt der Cursor

nämlich das Aussehen eines senkrechtes Striches, der sich in Intervallen zu einem inversen Quadrat ausdeht und anschließend wieder auf einen Strich zusammenschrumpft, ist der Einfüge-Modus (Insert-Mode) gerade aktiv.

Andernfalls ist der Überschreib-Modus angewählt. Dann besitzt der Cursor die Form eines blinkenden Quadrates. Zwischen beiden Modi kann über die Tastenkombination <Control><Insert> hin- und hergeschaltet werden. Bevorzugen Sie das Umschalten über die Menüleiste, setzen Sie den Mauszeiger auf den Menüpunkt Mode, und klicken in dem darauf herunterrollenden Menü den Eintrag Insert an. Ein Häkchen, das ab sofort vor diesen Punkt gesetzt wird, zeigt an, daß der Einfügemodus aktiv ist.

### Zeilen einfügen, löschen, trennen und verbinden

Es stehen zwei Möglichkeiten zur Verfügung, um eine Zeile einzufügen: <Return> fügt unter der aktuellen Zeile eine Leerzeile ein <F9> fügt an der aktuellen Zeile eine Leerzeile ein. Eine Zeile löschen, geht natürlich ebenfalls: <Shift> <F9> entfernt die aktuelle Cursorzeile! Um eine Zeile aufzutrennen, muß zuerst <Shift> <F6> und anschließend <F9> gedrückt werden. Zwei Zeilen werden miteinander verbunden, wenn <Shift><F6> und anschließend <Shift><F9> gedrückt wird.

### Weitere Steuerfunktionen

| | |
|---|---|
| *<Tab>* | Setzt den Cursor an die nächste 8er Position. |
| *<Undo>* | Restauriert den Inhalt einer Zeile, solange der Cursor die Zeile nocht nicht verlassen hat. |
| *<Esc>* | Erzielt den gleichen Effekt wie <Undo> kann jedoch zusätzlich Funktionen vorzeitig beenden (Programm laden,...). |
| *<Home>* | Setzt den Cursor an den Programmanfang, bzw. wenn er sich bereits dort befindet, an |

das Programmende. Ein zweimaliges Betätigen von <Home> setzt den Cursor stets an den Programmschluß, wenn er sich inmitten des Programmtextes befindet.

*<Ctrl><Delete>* Löscht den Zeilenrest hinter dem Cursor.

*<Alternate>* Beendet die Funktion Suchen u. Ersetzen bzw. bricht die Wiederholfunktion ab.

### Blockoperationen

Um den Blockanfang festzulegen, setzen Sie den Cursor in die gewünschte Zeile und drücken zweimal hintereinander die Funktionstaste <F7>. Der Blockanfang ist definiert. Das Blockende wird festgelegt, indem Sie zuerst <F7> und anschließend <Shift><F7> drücken.

*<F7><F8>* Speichert einen definierten Block als ASCII-File auf der Diskette ab.

*<F7>,<Shift><F8>* Lädt einen als ASCII-File auf Diskette vorliegenden Block ein.

*<F7><F9>* Kopiert einen Block an die aktuelle Cursorposition.

*<F7>,<Shift><F9>* Entfernt den Blockinhalt aus dem Speicher.

### Suchen und Ersetzen

*<F2><F2> "Text" <Return>*
Sucht ab der aktuellen Cursorposition nach der Buchstabenkombination Text. Wird kein Text angegeben, wiederholt Omikron.BASIC die Suche mit dem zuletzt verwendeten Begriff.

*<F2><F3> "Text" <Return>*
Listet alle Stellen auf, an denen Text im Programm vorkommt.

*<F3><F2>* Sucht einen bestimmten Text, und ersetzt ihn nach Quittierung durch einen anderen. Anschließend wird die Suche fortgesetzt.

<F3><F3> Sucht und Ersetzt, diesmal jedoch ohne eine Nachfrage.

### GO-Funktion

Mit der Funktionstaste <F1> kann eine beliebige Zeile ange sprungen werden. Folgende Funktionen stehen zur Auswahl:

*<F1> <Zeilennr.>* Springt zur angegebenen Zeile.

*<F1> <+> <Offset>* Springt um <Offset> Zeilen nach unten (z.B. <F1> + 20).

*<F1> <-> <Offset>* Springt um <Offset> nach oben.

*<F1> <Cursor hoch>* Setzt die aktuelle Cursorzeile als oberste Bildschirmzeile.

*<F1> <Cursor runter>* Setzt die aktuelle Zeile als unterste Bildschirmzeile.

*<F10>* Blättert im Programm eine Seite vor.

*<Shift><F10>* Blättert um eine Seite zurück.

### Funktionstasten definieren

Nach einer Betätigung von <Shift><F7> können die Funktionstasten <F4> und <F5> (auch in Kombination mit <Shift>) mit einem beliebigen Text belegen, indem Sie die gewünschte Taste drücken. Nach Eingabe des gewünschten Textes, beenden Sie die Definition durch erneutes Drücken von <Shift><F7>.

### Repeat

Drücken Sie nach der Taste, die wiederholt werden soll <F6>, und geben anschließend die Anzahl der Wiederholungen ein. Ein Betätigen von <Return> bewirkt, daß das Zeichen in der gewünschten Anzahl auf dem Bildschirm erscheint.

### Programm laden und speichern

Nach einer Betätigung der Funktionstaste <F8> werden Sie nach dem Dateinamen gefragt, unter dem das Programm abgespeichert werden soll. Das Programm wird als ASCII-Datei auf Diskette abgelegt. Um ein Programm von Diskette zu laden, drücken Sie die Tastenkombination <Shift><F8>. Befinden sich im ASCII-File Zeilen, die Omikron-BASIC nicht versteht, werden diese Zeilen durch inverse Darstellung gekennzeichnet.

### Zeilennummern ein- und ausschalten

Die Tastenkombination <Control><Clr> sorgt dafür, daß Programm ohne vorangestellte Zeilennummern eingegeben werden können. Ein erneutes Betätigen dieser Tasten hebt den Modus wieder auf.

*Vorsicht:* Wird der Full-Screen-Editor mit abgeschalteter Zeilennumerierung verlassen, so führt Omikron.BASIC eine Umnumerierung sämtlicher Zeilen in Einerabständen durch! Sprungziele bleiben jedoch unverändert. Auf diese Weise können leicht unbrauchbare Programme entstehen, wenn als Sprungadressen Zeilennummern verwendet werden.

### Bildschirmaufteilung

Ähnlich wie der Editor des neuen ST-BASIC kann auch der alte Editor-Bildschirm in unabhängige Teilbereiche aufgeteilt werden. Diese Split-Funktion wird mit <Shift><F1> aktiviert, der Cursor kann dann mit <Shift><F2> in die jeweils andere Bildschirmhälfte (Split wird hier in vertikaler Richtung durchgeführt) gesetzt werden. Sie können mit <Shift><F3> auch alternative Bildschirmdarstellungen wählen. (z.B. 44*108 Zeichen, 57*128 Zeichen in Hires).

## 1.5 ST-Omikron.BASIC verlassen

Um den Interpreter wieder zu verlassen, gehen Sie in den Bildschirm-Editor, tippen

```
SYSTEM
```

ein, und betätigen die Return-Taste. Die anschließende Rückfrage ist mit Y zu quittieren. Tippen Sie dagegen die Taste N, wird der Interpreter nicht verlassen.

# 2. ST-BASIC Grundkurs

Bisher haben Sie eine Menge Theorie über die Handhabung des Editors gehört. Aus diesem Grund möchte ich Sie nicht länger auf die Folter spannen und so schnell wie möglich zur Praxis kommen.

## 2.1 Ausgabe auf dem Monitor mit PRINT

Laden Sie das ST-BASIC (Omikron.BASIC) - falls noch nicht geschehen -, und geben Sie folgende Zeile ein:

```
print "Hallo"
```

Sobald Sie die <Return>-Taste betätigen, erscheint in der Zeile darunter "Hallo". Damit haben wir bereits den ersten Befehl kennengelernt, mit dem wir einen Text (in unserem Beispiel das Wort Hallo) auf dem Bildschirm ausgeben können. Der Text, der auf dem Bildschirm dargestellt werden soll, muß in Anführungszeichen hinter dem Befehl PRINT stehen. Was geschieht nun, wenn wir die Anführungszeichen einfach weglassen?

```
print Hallo
```

Auf dem Monitor erscheint nach Betätigung der <Return>-Taste diesmal nicht das Wort Hallo, wie wir es vielleicht erwartet hätten, sondern eine Null (0). Wie läßt sich dies erklären?

Der Computer unterscheidet zwischen Buchstaben und Variablen. Buchstaben, hat man ihm beigebracht, müssen in Anführungszeichen stehen. In unserem zweiten Beispiel hat er vergeblich nach Anführungszeichen gesucht. Deshalb wußte er, daß es sich bei diesem Hallo nicht um einen Text (Aneinanderreihung von Buchstaben) handeln konnte. Folglich muß Hallo eine Variable sein. Mit Sicherheit wurden Sie während Ihrer Schulzeit im Mathematikunterricht schon einmal mit Variablen konfrontiert. Dort dienen Sie in (Un-) Gleichungen als Platzhalter, die eine

bestimmte Zahl verkörpern. In der Gleichung x + 2 = 10 muß man x durch die Zahl 8 ersetzen, damit die Gleichung stimmt. Eine ähnliche Funktion besitzen Variablen in einer Programmiersprache: Sie verkörpern einen Wert (Zahl).

Stellen Sie sich eine solche Variable einmal als eine kleine Schachtel vor, die sich im Computer befindet. Jede Schachtel trägt einen Namen, in unserem obigen Beispiel Hallo. Kommt der Computer zu der Ansicht, daß es sich um keinen Text, sondern um eine Variable handelt, beginnt er eifrig in seinem Speicher zu suchen: Stößt er dabei auf eine Schachtel, die den angegebenen Namen trägt, so ließt er ihren Inhalt aus, und schreibt das auf diese Weise ermittelte Ergebnis direkt auf den Monitor. Findet er dagegen keine Schachtel mit passendem Namen, erscheint dort eine Null. In der Informatik spricht man allerdings nicht von Schachteln, sondern von Variablen, und das werden wir ab sofort auch so machen.

Bliebe noch zu klären, wie wir es schaffen, einer Variablen einen Wert zuzuweisen. Dazu wird in BASIC das Gleichheitszeichen (=) benutzt, das als Zuweisungsoperator fungiert:

```
let Hallo = 3
```

Tippen Sie diese Zeile ab, und vergessen Sie nicht, anschließend die <Return>-Taste zu drücken. Dies ist deshalb wichtig, da der Computer erst mit seiner Arbeit beginnt, wenn <Return> gedrückt wurde. Seine erste Aufgabe besteht nun darin, nachzusehen, ob in seinem Speicher schon eine Variable mit derartigem Namen vorhanden ist. Wird er nicht fündig, so legt er eine neue Variable mit diesem Namen an. Der Zuweisungsoperator sagt ihm, daß er dieser Variablen den Wert 3 zuweisen soll. Jetzt müßte die Zahl 3 auf dem Monitor erscheinen, wenn wir den Befehl

```
print Hallo
```

eingeben. Und tatsächlich erscheit eine 3 auf unserem Monitor. Der vorangestellte Befehl LET kann problemlos weggelassen werden, am Ergebnis dieser Operation ändert sich nichts. Dies liegt daran, daß der BASIC-Interpreter intellegent genug ist, eine Zuweisung zu erkennen.

Nach diesem kurzen Exkurs über Variablen möchte ich ganz gerne noch einmal auf den Befehl PRINT zurückkommen, der ja Gegenstand dieses Kapitels ist. Ich habe Ihnen bisher verraten, daß ein Text, der auf dem Computerbildschirm ausgegeben werden soll, hinter PRINT in Anführungszeichen stehen muß. Fehlen die Anführungszeichen, betrachtet der Computer das folgende Wort als eine Variable und gibt deren Wert auf dem Bildschirm aus. Möchten Sie Text und Variablen gleichzeitig ausgeben, können Sie das mit ; bewerkstelligen:

```
print "Heute ist der";Hallo;". September"
```

Auf dem Monitor erscheint: "Heute ist der 3. September". Der Computer schreibt zuerst "Heute ist der" auf den Monitor, da es sich um einen Text in Anführungszeichen handelt. Das Semikolon bewirkt, daß der Inhalt der nachfolgenden Variablen Hallo nicht in eine neue Zeile gedruckt, sondern gleich hinter "Heute ist der" weitergeschrieben wird. Hier folgt nun der Inhalt der Variablen Hallo, der in unserem kleinen Beispiel 3 beträgt. Wiederum stößt der Computer auf ein Semikolon, er muß den folgenden Text also hinter die 3 schreiben.

Vielleicht ist Ihnen aufgefallen, daß zwischen "Heute ist der" und der 3 ein Leerzeichen ausgegeben, während hinter der 3 und dem darauffolgenden Text kein Leerzeichen eingefügt wurde. Die Erklärung dafür ist einfach: wie wir bereits festgestellt haben, handelt es sich bei unserem Hallo um eine Variable, die einen bestimmten Wert (hier die Zahl 3) verkörpert. Eine Zahl kann entweder positive oder auch negative Werte annehmen. Da ein Vorzeichen nicht unbedingt die Lesbarkeit erhöht, wird nur bei negativen Werten ein Vorzeichen (-) ausgegeben, bei positiven Werten wird statt dessen der Zahl ein Leerzeichen vorangestellt. Diesen Effekt habe ich ausgenutzt und so das benötigte Leerzeichen erhalten.

Fassen wir noch einmal zusammen:

- Mit dem Befehl PRINT können Texte und Variablen auf dem Monitor ausgegeben werden.
- Texte, die ausgegeben werden sollen, müssen in Anführungszeichen stehen, da ansonsten der Text als Variablenname aufgefasst wird.
- Das Semikolon (;) innerhalb eines PRINT-Befehls bewirkt, daß darauffolgender Text direkt angefügt wird.
- Wird eine Variable mit PRINT ausgedruckt, so stellt der Computer bei positiven Werten ein Leerzeichen, bei negativen Werten der Zahl ein Minuszeichen voran.

## 2.2 Variablentypen in ST-BASIC

Bisher haben wir nur eine Variable, die wir Hallo nannten, benutzt. Dieser Variablen haben wir eine positive Zahl (3) zugewiesen. Doch damit ist das Thema Variablen noch lange nicht ausgereizt. Was geschieht, wenn wir dieser Variablen eine Kommazahl zuordnen?

```
Hallo = 234.18
```

Der PRINT-Befehl bringt es an den Tag: Die Ziffern hinter dem Dezimalpunkt werden vom Computer einfach abgeschnitten. Dagegen werden die Nachkommastellen bei folgender Zuweisung berücksichtigt:

```
Hallo! = 234.18
```

Vergleichen wir beide Variablennamen, fällt nur ein Unterschied auf: Das Ausrufezeichen am Ende der zweiten Variablen muß dafür verantwortlich sein, daß Nachkommastellen berücksichtigt werden. Und so ist es auch! Neben dem Namen, den die Variable tragen soll, muß dem Computer noch mitgeteilt werden, was in dieser Variablen gespeichert werden soll. Dies geschieht über sogenannte Postfixe, also Zeichen, die an den Namen der Variablen angehängt werden.

### Integervariablen

Als Integer werden ganze Zahlen ohne Nachkommastellen bezeichnet. Die erste Variable, die wir Hallo nannten, war eine solche Integervariable. Solange man ihr ganze Zahlen zuweist, gibt es keine Probleme. Diese treten erst dann auf, wenn man versucht, einer Integer eine Kommazahl zuzuordnen. Nicht. daß dies nicht ginge, der Computer akzeptiert die Zuweisung. Die Nachkommastellen werden allerdings abgeschnitten, da er für sie keinen Platz in der Variablen hat. Um dies zu verstehen, muß ich etwas weiter ausholen:

Schauen wir dazu einmal die (Integer-) Zahl 3548 an. Zerlegt man diese Zahl nach der Wertigkeit ihrer Stellen, erhält man folgendes Ergebnis:

```
3 * 1000 + 5 * 100 + 4 * 10 + 8 * 1
```

Auf diese Weise lassen sich alle Zahlen des Dezimalsystems, das auf den Ziffern 0 bis 9 beruht, darstellen. Von rechts her betrachtet, hat die erste Stelle stets die Wertigkeit 1, wobei sich die Wertigkeit von Stelle zu Stelle verzehnfacht. Der Computer tut sich bei der Zahlendarstellung etwas schwerer, da er keine zehn, sondern nur zwei Ziffern kennt, die 0 und die 1. Reiht man mehrere Ziffern aneinander, so besitzt auch hier die erste Stelle (von rechts aus betrachtet) die Wertigkeit 1. Während sich beim Dezimalsystem die Wertgkeit von Stelle zu Stelle verzehnfacht, verdoppelt sie sich bei dem vom Computer verwendeten Dualsystem. Die Binärzahl 111111 besitzt folglich den Wert:

```
1 * 32 + 1 * 16 + 1 * 8 + 1 * 4 + 1 * 2 + 1 * 1
```

oder einfacher ausgedrückt: 63. Man hat sich nun darauf geeinigt, stets 8 solcher Binärstellen (Bit) zu einer Binärzahl (Byte) zusammenzufassen. Mit 8 Bit bzw. 1 Byte lassen sich dann 256 verschiedene Werte darstellen. Im Dezimalsystem ausgedrückt, sind dies die Zahlen von 0 bis 255. Mit 256 Werten kann man aber noch keine großen Sprünge machen. Aus diesem Grund

verwendet man für alle Zahlen, die größer als 255 sind, zwei Bytes. Dieses zweite Byte hat nun die Wertigkeit 256. Die Dezimalzahl 3548 benötigt im Computer zwei Bytes:

```
13 (2. Byte)
220 (1. Byte)
```

Im Inneren des Rechnes sieht dies dann so aus:

```
00001101 (1.Byte)
11011100 (2. Byte)
```

Zwei dergestalt zusammengefasste Bytes werden häufig auch als Word bezeichnet. Neben Byte und Word gibt es noch Long, wobei dann vier Bytes zusammengefaßt werden. Mit einer solchen Longinteger können Werte von

```
-2147483658 bis +2147483657
```

dargestellt werden.

Im Gegensatz zu anderen BASIC-Interpretern betrachtet ST-BASIC alle Variablen als Longinteger, falls diese nicht anderweitig durch Postfixe gekennzeichnet sind. Verstehen Sie jetzt, warum Hallo keine Kommazahl speichern konnte? Für den Interpreter ist Hallo eine Integervariable vom Typ long. Jeder Versuch ihr eine Fließkommazahl zuzuweisen, wird mit dem Abschneiden der Nachkommastellen bestraft.

Jetzt aber endlich zu den Postfixen, mit denen die verschiedenen Variablentypen gekennzeichnet werden:

| Variablentyp | darstellbarer Zahlenbereich | Postfix |
|---|---|---|
| Integer-Byte | von 0 bis 255 | %B |
| Integer-Word | von -32768 bis 32767 | %W oder % |
| Integer-Long | von -2147483658 bis 2147483657 | %L bzw.nichts |

*Anmerkung:* Integer-Byte kann nur in Variablenfeldern (Arrays) verwendet werden.

Beispiele für die Verwendung der Postfixe:

| Variable | Typ |
|---|---|
| Hallo%B(X) | Integer-Byte |
| Hallo%W | Integer-Word |
| Hallo% | Integer-Word |
| Hallo%L | Integer-Long |
| Hallo | Integer-Long |

## Fließkommavariablen

Während sich mit Integervariablen lediglich ganze Zahlen darstellen lassen, können mit Fließkommavariablen (Float) auch Nachkommastellen bzw. extrem große Zahlen, die den Bereich einer Longinteger überschreiten, dargestellt werden. ST-BASIC unterscheidet zwischen Fließkommazahlen mit einfacher Genauigkeit (Single Precision) und doppelter Genauigkeit (Double Precision). Für eine Fließkommavariable mit doppelter Genauigkeit verleibt sich der Computer 10 Bytes ein, für einfache Genauigkeit begnügt er sich mit nur 6 Bytes. Bei Single Precision wird mit "etwa" 9 Nachkommastellen (wovon 7 ausgegeben werden), bei Double Precision etwa 19 Stellen hinter dem Komma (17 werden ausgegeben) gerechnet. Der darstellbare Zahlenbereich liegt bei beiden Fließkommaformaten bei 10 hoch 4931.

| Variablentyp | darstellbarer Zahlenbereich | Postfix |
|---|---|---|
| Float-Single | +- 10 hoch 4931 | ! |
| Float-Double | +- 10 hoch 4931 | # |

| Variable | Typ |
|---|---|
| Hallo! | Fließkommazahl mit einfacher Genauigkeit |
| Hallo# | Fließkommazahl mit doppelter Genauigkeit |

Möchten Sie einer Fließkommazahl mit doppelter Genauigkeit eine Zahl (Konstante) zuweisen, wie etwa

```
Hallo# = 1.66#
```

sollte diese durch ein Postfix (#) als Konstante mit doppelter Genauigkeit gekennzeichnet werden. Andernfalls wird der Fließkommavariablen Hallo# nur eine mit einfacher Genauigkeit gespeicherte Konstante (1.66) zugewiesen, und dies ist ja nicht im Sinne des Erfinders.

### Stringvariablen (Zeichenketten)

Sie haben bisher mit Integer- und Fließkommavariablen Bekanntschaft geschlossen. Beide Arten sind zur Speicherung von Zahlen gedacht. Neben Zahlen kennt der Computer noch Buchstaben oder genauer ausgedrückt: Zeichen. Zeichen deshalb, da Buchstaben nur eine Teilmenge aus den Zeichen sind. Der Buchstabe G ist ein Zeichen, die Klammer (, um nur ein Beispiel herauszugreifen, ist ein Zeichen, aber kein Buchstabe.

Zeichen wird im Computerjargon auch character (abgekürzt char) genannt. Mehrere aneinander gehängte Zeichen bilden eine Zeichenkette oder einen String. Stringvariablen wird zur Kennzeichnung ihres Standes ein Dollarzeichen $ nachgestellt: Hallo$ (gesprochen: Hallostring). Weisen wir einer solchen Variablen einmal einen String zu:

```
Test$ = "Hallo"
```

Mit print Test$ erscheint auf dem Monitor Hallo. Ein String muß in Anführungszeichen stehen, damit ihn der Computer von einer Variablen unterscheiden kann. Zeilen wie

```
Test$ = Hallo
Test% = "Hallo"
Test# = "Hallo"
```

werden vom Computer mit der Fehlermeldung TYPE MISMATCH ERROR (einer Variabel wurde ein Wert zugewiesen, den diese aufgrund ihrer Deklaration (das Postfix!) gar nicht aufnehmen kann) geahndet.

### Variablendeklaration einmal anders

Ein Postfix am Ende des Namens ist eine Möglichkeit, eine Variable zu deklarieren. In ST-BASIC kann dies auch per Befehl am Anfang eines Programms geschehen. DEFINT "A,B,C" weist den Computer beispielsweise an, alle Variablen im Programm, die mit den Anfangsbuchstaben A, B, oder C beginnen, als Integer-Word (zwei Bytes lang!!) aufzufassen, solange die betreffende Variable nicht durch ein entsprechendes Postfix anderweitig deklariert ist.

Stößt der Computer während des Programmablaufs auf eine Variable A$, so wird er - ungeachtet der Deklaration am Programmanfang durch den Befehl DEFINT "A,B,C" - diese als Stringvariable betrachten, der eine Zeichenkette zugewiesen werden muß. Eine Variable August wird dagegen als Integervariable vom Typ Word behandelt, da sie nicht anderweitig (z.B. als Doublefloat) deklariert ist.

Die Deklarationsbefehle im einzelnen:

| Befehl | Deklaration für Typ |
|---|---|
| DEFINT | Integer-Word |
| DEFINTL | Integer-Long |
| DEFSNG | Float - single precision |
| DEFDBL | Float - double precision |
| DEFSTR | Stringvariable |

Beispiele für die Anwendung:

DEFSNG "A-Z" Alle Variablen mit den Anfangsbuchstaben von A bis Z werden als Fließkommavariablen mit einfacher Genauigkeit (single precison) deklariert.

DEFSTR "A,C,F-M" Alle Variablen mit den Anfangsbuchstaben A, C und F bis M werden als Stringvariablen behandelt.

Diese Befehle müssen stets in der ersten Programmzeile stehen und als erste Befehle in ein Programm eingegeben werden, ansonsten werden sie vom Computer einfach nicht beachtet. Haben Sie dennoch einmal einen Deklarationsbefehl vergessen, können Sie sich mit einem "Klimmzug" aus der Affäre retten: Fügen Sie den Befehl nächträglich (in die erste Programmzeile) ein und speichern Sie das Programm als ASCII ab. Wenn Sie das Programm dann erneut laden, werden die Variablen gemäß der Deklaration in der ersten Zeile angelegt - der Schaden ist behoben.

Und noch etwas: Andere BASIC-Interpreter betrachten Variablen, die nicht durch ein Postfix besonders gekennzeichnet sind, als Fließkommavariablen mit einfacher Genauigkeit. Möchten Sie Programme oder Routinen von anderen BASIC-Interpretern übernehmen, sollten Sie alle Variablen ohne Postfix mit DEFSNG "A-Z" als Singe-Float deklarieren, da andernfalls ST-BASIC diese für Long-Integer hält und Nachkommastellen einfach ignoriert.

### Flags

Flags werden in Programmen benutzt, um Wahrheitswerte zu speichern. Ein Wahrheitswert kann entweder wahr oder falsch sein. Im ersteren Fall speichert man eine -1 (wahr), im letzteren eine 0. Wozu aber dieser Aufwand?

Nehmen wir einmal an, Sie haben eine kleine Textverarbeitung geschrieben. Beim Einlesen eines Textes von Diskette setzen Sie pauschal ein Flag auf 0. Wird nun im Text eine Änderung durchgeführt (z.B. ein Tippfehler ausgebessert), weisen Sie diesem Flag auf den Wahrheitswert wahr (-1) zu. Vor dem Verlassen des Programms wird dieses Flag abgefragt. Für den Fall, daß sein Wert -1 beträgt, kann der Text mit den durchgeführten Änderungen entweder automatisch abgespeichert, oder aber eine Warnmeldung ausgegeben werden, damit der Benutzer den Text auf Diskette bringen kann, ehe seine Korrekturen verloren sind.

Für ein Flag könnten Sie eine Fließkommavariable mit doppelter Genauigkeit verwenden. Auf diese Art und Weise hätten Sie dann 10 Bytes ihres (möglicherweise) kostbaren Speicherplatzes zum Fenster hinausgeworfen. Aus diesem Grund kamen schlaue Menschen auf die Idee, eine Variable einzuführen, die nur die Werte 0 und -1 zu speichern vermag. 0 verkörpert in diesem Fall den Wahrheitswert falsch, während -1 den Wahrheitswert wahr repräsentiert.

Flags werden in ST-BASIC als solche durch das Postfix %F gekennzeichnet. Eine Einschränkung ist dabei zu beachten. Dieser Variablentyp darf - ebenso wie die Integer-Bytes - nur in Variablenfeldern (Arrays) verwendet werden. Was Arrays sind, erfahren Sie allerdings erst etwas später. Der Vollständikeit halber wollte ich die Flags aber schon hier einmal vorstellen.

Nachdem wir nun sämtliche Variablentypen, die in ST-BASIC existieren, erschöpfend behandelt haben, sollten Sie sich von Ihrer Erschöpfung ein wenig erholen, ehe wir unser erstes Programm schreiben.

## 2.3 Das erste Programm

Bisher wurden Sie mit einer ganzen Menge Theorie bombadiert. Aber dies hatte seinen Grund: Wenn wir schon ein Programm schreiben, dann bitte ein etwas sinnvolles:

Die Pension "Schlumpf" im bayerischen Wald vermietet Fremdenzimmer in verschiedenen Größen an Touristen. Der Inhaber dieser Pension, Herr Gargamel, hat sich einen Computer angeschafft, der ihn bei den wichtigsten Arbeiten, die in dieser Pension anfallen, entlasten soll. Eine immer wiederkehrende Arbeit besteht darin, die in DM angegebenen Zimmerpreise für österreichische Gäste in Schilling umzurechnen, damit diese den Rechnungsbetrag in ihrer Landeswährung begleichen können.

Sie ahnen es schon: Unser erstes Programmprojekt soll Beträge, die in DM ausgewiesen werden, direkt in österreichische Schilling umrechnen. Was brauchen wir dazu?

1 DM entspricht etwa 7 ÖS (Österreichische Schilling), der Wechselkurs beträgt dann folglich 1/7 oder ausgerechnet und (auf vier Stellen hinter dem Komma) gerundet: 0,1429. Dividiert man den Rechnungsbetrag durch diesen Kurswert (0,1429), erhält man die Summe in Schilling.

Gehen wir (vorläufig) davon aus, daß nur volle DM-Beträge in Schilling umzurechnen sind, können wir die Rechnungssumme in einer Integervariablen ablegen. Für den Kurswert werden wir uns einer Fließkommavariablen mit einfacher Genauigkeit (für unsere Bedürfnisse vollkommen ausreichend) bedienen. Das Ergebnis unserer Division müssen wir auch noch in einer Variablen festhalten. Logischerweise wird dies ebenfalls eine Fließkommavariable sein, da bei der Umrechnung durch die Division Nachkommastellen entstehen können.

```
10 '****************************************************************
20 '*                        WAEHRUNG.BAS                          *
30 '*--------------------------------------------------------------*
40 '* Autor: Michael Maier    Version: 1.00     Datum: 22.08.1988 *
50 '*      Ein Programm aus dem 'GROSSEN ST-BASIC BUCH'             *
60 '*        (C) 1988 by DATA BECKER GmbH Düsseldorf                *
70 '****************************************************************
80 '
90 '
100 Kurs!=.1429: REM Wechselkurs DM/ÖS
110 Betrag%=300: REM Rechnungssumme
120 Waehrung!=Betrag%/Kurs!'  DM in ÖS umrechnen
130 CLS : PRINT Betrag%;" DM";" entsprechen";Waehrung!;" Schilling"
140 END
```

Dies ist also das erste Programm, mit dessen Hilfe DM-Beträge in Schilling umgerechnet werden können. Am Anfang einer jeden Zeile steht eine Nummer, die Zeilennummer. In BASIC-Interpretern früherer Jahre waren diese Zeilennummern Pflicht, jede Zeile mußte mit einer Nummer versehen sein. Da keine Nummer zweimal innerhalb eines Programms vorkommen durfte, wurden üblicherweise 10er Abstände in der Zeilennummerierung gewählt. Wollte man Programmtext nachträglich einfügen, konnten die dazwischenliegenden Zahlenbereiche ausgenutzt werden, und man ersparte sich auf diese Art die zeitraubende Umnumerierung der folgenden Zeilen.

ST-BASIC ist hier wesentlich flexibler: Die Zeilennummern können wahlweise eingegeben werden. Möchten Sie keine Zeilennummern mit eingeben, so schalten Sie diese im Full-Screen-Editor einfach über die Tastenkombination <Control> <Clr/Homme> aus, bzw. wieder ein.

Ich selbst bevorzuge die Programmierung ohne Zeilennummern (die Macht der Gewohnheit!). Aus diesem Grund werde ich weitgehend auf sie verzichten. Da ST-BASIC intern - auch bei abgeschalteter Zeilennumerierung - selbstständig Zeilennummern anlegt, wird in den Listings jede Zeile mit einer Nummer beginnen. Sie brauchen diese nicht mit eingeben, falls Sie das eine oder andere Programm selbst abtippen möchten. Achten Sie jedoch darauf, daß die Zeilennumerierung auch wirklich abgeschaltet wurde, damit Sie keine böse Überraschung erleben, wenn Sie das Programm laufen lassen möchten. Doch jetzt zurück zum Programm:

Der Kurswert wird in der Fließkommavariablen (single precision), die den sinnigen Namen Kurs trägt, abgelegt. Das Suffix am Ende des Variablennamens sagt dem Computer, daß in der Variablen Kurs eine Fließkommazahl mit einfacher Genauigkeit zugewiesen werden soll. Der Kurswert selbst steht hinter dem Zuweisungsoperator =. Die Nullstelle vor dem Komma kann getrost weggelassen werden, sie spielt keine Rolle. Noch etwas: Im Gegensatz zur herkömmlichen Schreibweise erwartet der Computer kein Dezimalkomma sondern statt dessen einen Dezimalpunkt, wie Sie dem Listing unschwer entnehmen können.

In einer Programmzeile können mehrere Anweisungen stehen, die allerdings durch einen Doppelpunkt (:) getrennt werden müssen. In Zeile 100 ist dies der Fall: Hinter der Zuweisung des Kurswertes steht ein Doppelpunkt, der den nächsten Befehl (REM) abtrennt.

### REM

Der BASIC-Befehl REM (vom englischen Remark: Anmerkung) gestattet das Einstreuen von Kommentaren in ein Programm. Dies ist gerade bei längeren Programmen wichtig. Ist ein Pro-

gramm nämlich einmal fertig entwickelt und man möchte es ein paar Monate später verbessern, kann es leicht passieren, daß man sich in seinem eigenen Programmcode nicht mehr zurechtfindet. Moral von der Geschicht': Nur nicht mit Kommentaren geizen!

Der Rest der Zeile hinter REM wird vom Computer einfach überlesen. Es ist deshalb nicht möglich, einen neuen Befehl hinter einer REM-Anweisung zu plazieren. Der Computer würde ihn einfach nicht beachten. Statt REM kann in ST-BASIC auch das Hochkomma (') benutzt werden, um einen Kommentar in das Programm einzufügen. Im obigen Beispiel wurde von dieser Möglichkeit im sogenannten Header (Programmvorspann in den Zeilen 10 bis 90) Gebrauch gemacht.

Die Rechnungssumme wird - da nur volle DM-Beträge in Betracht kommen - einer Integervariaben zugewiesen. Auf den Befehl LET habe ich auch hier verzichtet, da es ohne ihn ebenso geht. In der Variablen Kurs ist jetzt also der Kurswert, in Betrag% (übrigens eine Integer vom Typ Word) die Rechnungssumme gespeichert.

Der nächste Schritt wird in der Programmzeile 120 durchgeführt: Die Rechnungssumme (Betrag%) wird durch den Kurswert (Kurs!) geteilt (der Operator für die Division lautet /) und das auf diese Weise ermittelte Ergebnis der (Fließkommavariablen, aber warum sage ich das?) Waehrung! zugewiesen.

Dort ist jetzt der Betrag Schilling enthalten und wartet direkt darauf, per PRINT-Befehl auf dem Bildschirm ausgegeben zu werden. Zuvor wird jedoch der Optik wegen der Bildschirm mit CLS (CLear Screen: lösche Bildschirm) gelöscht. Dann erst ist es soweit: Durch einen Doppelpunkt von CLS getrennt steht der PRINT-Befehl. Und jetzt geht es gleich rund: Der Rechnungsbetrag in DM wird zuerst ausgegeben. Dann sorgt der Strichpunkt hinter Betrag% dafür, daß der Computer in derselben Zeile weiterschreibt. Es folgt ein Text (DM), der als solcher durch die Anführungszeichen gekennzeichnet ist.

Das nächste Semikolon hätte ich mir sparen können, doch dann hätten Sie nicht so schön gesehen, daß auch zwei Texte miteinander verbunden werden können. Da der Computer direkt weiterschreibt, beginnt der Text mit einem Leerzeichen (Space). Der letzte Befehl END, sagt dem Computer, daß er an dieser Stelle fertig ist.

Puh! Jetzt müßte die Funktionsweise des Programms eigentlich geklärt sein. Also probieren wir es gleich einmal aus! Benutzen Sie dazu den Full-Screen-Editor (beim ST-BASIC mit <Help>, bei Omikron.BASIC über die Eingabe des Befehls EDIT und Drücken von <Return>). Tippen Sie nun die Zeilen 100 bis 140 ab und starten das Programm. Auch hier haben es Besitzer des ST-BASIC wieder einfacher: Sie brauchen bloß die Tastenkombination <Control> <R> zu drücken und schon sehen sie das Ergebnis auf dem Monitor. Ansonsten muß der Editor erst über <Control> <C> verlassen und danach RUN (<Return> nicht vergessen!) eingetippt werden. Etwas umständlicher, aber das Ergebnis ist in beiden Fällen identisch:

```
300 DM entsprechen 2099.3702 Schilling
```

Um andere DM-Beträge in Schilling umrechnen zu lassen, müssen Sie wieder in den Full-Screen-Editor, dort den Wert hinter der Variablen Betrag% ausbessern und das Programm erneut starten.

### Arithmetikoperatoren

Neben der Division, die in Programmzeile 130 durchgeführt wurde, sind noch weitere arithmetische Funktionen möglich: Addition, Subtraktion, Multiplikation und Potenzieren. Für jede dieser Funktionen existiert ein Operator, der aus folgender Tabelle ersichtlich ist:

| Operator | Funktion | Beispiel |
|---|---|---|
| + | Addition | A% = 3 + 4 |
| - | Subtraktion | A% = 5 - 2 |
| * | Multiplikation | A% = 2 * 3 |

| Operator | Funktion | Beispiel |
|---|---|---|
| / | Division | A# = 1 / 3 |
| \ | Integer-Division | A# = 6 \ 4 |
| ^ | Potenzieren ("hoch") | A% = 2^3 |

**Anmerkungen:**

- Diese Operationen können nicht nur mit Konstanten (Zahlen), sondern auch Variablen (sofern sinnvoll) durchgeführt werden. Die Anweisung A% = H% + 3 weist der Variablen A% den Inhalt der Variablen H% zu und adddiert dazu den Wert 3. Aber auch das ist möglich (auch wenn es ungewohnt aussieht!): A% = A% + 1. Der Inhalt der Variablen A% wird um den Wert 1 erhöht. Das = fungiert in diesem Fall ja als Zuweisungsoperator und nicht als Gleichheitszeichen!

- Der Unterschied zwischen der normalen Division und der Integer-Division liegt darin, daß das Ergebnis einer Integer-Division stets ganzzahlig ist, während bei der normalen Division Kommastellen entstehen können. A! = 7 \ 2 weist der Variable A! (obwohl Fließkommavariable!) den Wert 3 zu. Der Rest der Division kann mit der Modulo-Funktion ermittelt werden: A% = 7 MOD 2 ergibt den Wert 1 in A%. (Das Ergebnis der Integerdivision aus 7 \ 2 ist bekanntlich 3. Folglich kann der Rest der Division mit 7 - (3*2) = 1 berechnet werden.). Da der Computer auf die umständliche Berechnung der Nachkommastellen verzichten kann, resultiert daraus ein gewaltiger Geschwindigkeitsvorteil bei der Integer-Division gegenüber der normalen Division.

  Soweit arbeitet das Programm ja zufriedenstellend, ganz glücklich wird Herr Gargamel mit unserer Version allerdings noch nicht sein:

- Die Rechnungssumme muß jedesmal direkt im Programm ausgebessert werden. Besser wäre es, wenn das Programm nach dem Betrag fragen und diesen entsprechend umrechnen würde.

- Der in Schilling umgerechnete Wert besitzt 4 Stellen hinter dem Komma, obwohl nur 2 Stellen nötig sind. Deshalb sollte das Programm auch nur 2 Nachkommastellen ausgeben, die dann entsprechend auf- bzw. abgerundet werden müssen.

Diese beiden Verbesserungen sollen Gegenstand der zweiten Version unseres kleinen Programms sein. Zuvor muß aber noch ein anderes Problem gelöst werden:

## 2.4 Programme speichern, laden und löschen

Das Programm befindet sich im Speicher des Computers. Falls Sie jetzt den Computer ausschalten (Halt! Tun Sie's nicht!), oder den Resetknopf betätigen, wird das Programm gelöscht. Bei obigem Programm bedeutet dies noch keine Tragödie, da die 5 Programmzeilen wieder eingegeben sind. Stellen Sie sich aber vor, ein solches Programm bestände aus mehreren Seiten und die müßten jedesmal von neuem eingegeben werden.

### SAVE

Der Befehl SAVE speichert ein Programm auf Diskette ab. Die Syntax lautet:

```
SAVE ["<Programmname>"]
```

bzw.

```
SAVE ["<Programmname>"],A
```

In beiden Fällen wird das Programm auf der Diskette mit dem Namen <Programmname.BAS> abgelegt. An der Endung (Extension) .BAS die der Interpreter an den Namen hängt, können Sie erkennen, daß es sich um ein BASIC-Programm handelt.

Der Programmname selbst darf nicht länger als 8 Zeichen sein. Möchten Sie das Programm mit einer anderen Extension speichern, so muß diese explizit angegeben werden (z.B. Programm.BAK). Folgende Extensions sollten Sie jedoch nach Möglichkeit vermeiden, da sie feste Bedeutungen besitzen:

| Extension | Bedeutung |
|---|---|
| .TOS | Ausführbares Programm, das unter TOS, dem Betriebssystem des Atari ST läuft. |
| .PRG | Ausführbares Programm, das unter GEM läuft. |
| .TTP | Ausführbares Programm, wobei Parameter beim Programmstart angegeben werden können (Tos Takes Parameter). |
| .DOC | Dokumenten-Datei für Anleitungen, ... |
| .BAS | BASIC-Programm. |
| .BAK | BAcKup-Datei (vgl. Kapitel 3). |

Ohne die Angabe eines Programmnamens benutzt ST-BASIC den letzten Programmnamen. Dies kann entweder der bei LOAD oder bei NEW angegebene Name sein. Deshalb auch die eckigen Klammern, die andeuten, daß der Programmname evtl. auch weggelassen werden kann.

Wird nach dem Programmnamen noch ein ,A angehängt, so speichert der Computer das Programm ebenfalls auf Diskette, allerdings in einer unkodierten Form. Auf diese Weise gespeicherte Programme können dann z.B. von Textverarbeitungsprogrammen eingelesen werden. Dies ist bei mit dem einfachen SAVE abgespeicherten Programmen nicht möglich, da diese in kodierter Form abgespeichert werden (spart Speicherplatz und ermöglicht schnellere Programmausführungszeiten). Auch wenn mehrere Programme zusammengehängt ("gemerged") werden sollen, müssen sie in unkodierter Form auf Diskette vorliegen.

### NEW

Der Befehl NEW entfernt ein Programm und den Inhalt sämtlicher Variablen aus dem Speicher des Computers. Gehen Sie deshalb schon im eigenen Interesse sorgsam mit diesem Befehl um. Er ist aber immer dann vonnöten, wenn Sie ein neues Programm

erstellen möchten und sich noch ein altes im Speicher befindet. Alternativ kann mit diesem Befehl gleich ein Name für das neue Programm angegeben werden. Die Syntax dafür lautet:

```
NEW "<Programmname>"
```

In Zukunft kann dann bei einem SAVE auf die Angabe des Dateinamens verzichtet werden, da er bereits mit diesem Befehl angegeben wurde.

### LOAD

Last not least der Befehl zum Laden eines auf Diskette abgespeicherten Programms:

```
LOAD ["<Programmname"]
```

Die Datei mit dem in Anführungszeichen stehenden Namen wird in den Speicher geladen. Alternativ dazu kann man sich die Angabe eines Namens ersparen und nur LOAD eingeben. In diesem Fall verwendet ST-BASIC den zuletzt bei SAVE, NEW bzw. LOAD benutzten Dateinamen. Es ist bei diesem Befehl unwichtig, ob die Programme auf Diskette in kodierter oder unkodierter Form vorliegen. ST-BASIC erkennt dies selbständig, und kodiert den Programmtext bei Bedarf.

### RUN

Den Befehl RUN habe ich schon einmal benutzt. Erinnern Sie sich? Er diente dazu, ein Programm im Speicher zu starten. Wird zusätzlich ein in Anführungszeichen stehender Programmname angegeben, lädt ST-BASIC dieses Programm in den Speicher und beginnt sofort mit dessen Ausführung.

```
RUN "WAEHRUNG"
```

lädt die Programmdatei WAEHRUNG.BAS in den Speicher und startet sofort mit der Ausführung. Der gleiche Effekt wird auch über die beiden Befehle

```
LOAD "WAEHRUNG"
RUN
```

erzielt.

## 2.5 Wie sag ich's dem Computer?

Nach diesem kurzen Intermezzo wagen wir uns an die verbesserte Version unseres Programms.

Das erste Problem: Die Eingabe des Rechnungsbetrages soll ab sofort gleich direkt, also ohne das umständliche Überschreiben des Programmtextes, möglich sein. Interaktive Programmierung wird dieser Vorgang vom Fachmann genannt, wenn der Computer innerhalb eines Programmes auf eine Eingabe wartet, ehe er im Programm fortfährt. ST-BASIC bietet eine Fülle derartiger Befehle: vom einfachen Standard-Eingabebefehl bis hin zur formatierten Eingabe, die selbst höchsten Ansprüchen in der professionellen Programmierung gerecht wird.

Für unsere Bedürfnisse reicht (vorläufig) der Standardeingabebefehl völlig aus. Er lautet:

```
INPUT
```

INPUT sagt dem Computer also, daß er dem Benutzer eine Möglichkeit bieten muß, eine Eingabe - in welcher Form auch immer - zu tätigen. Doch wohin mit dieser Eingabe? Um mit ihr weiterarbeiten zu können muß diese einer Variablen zugewiesen werden. Was würde es uns auch helfen, wenn der Computer die Eingabe irgendwo in seinem Speicher ablegt, ohne uns - den Programmierern - den Inhalt zu verraten? Folglich benötigen wir noch eine Variable, der die Eingabe zugewiesen werden soll. Diese muß hinter dem PRINT stehen.

```
Input A%
```

weist der Integervariablen A% den eingegebene Wert zu, während die Eingabe bei

```
Input A!
```

einer Fließkommavariablen, die den Namen A! trägt, zugewiesen wird. Möchten Sie einen String vom Benutzer eingeben lassen, muß dem Input-Befehl selbstverständlich eine passende Variable nachgestellt werden:

```
Input A$
```

Andernfalls wird der Text nicht akzeptiert, und der Variablen (z.B. A%, A!, A#, usw.) einfach der Wert 0 zugewiesen. In anderen BASICdialekten würde sich das Programm sogar mit einer Fehlermeldung verabschieden. Der Computer gibt - stößt er in einem Programm auf ein INPUT - ein Fragezeichen aus und wartet geduldig bis der Benutzer seine Eingabe durchgeführt hat. Für den Computer ist die Eingabe beendet, sobald die <Return>-Taste gedrückt wird. Dann holt er sich den eingegebenen Wert und weist ihn der angegebenen Variablen zu.

Soweit ist der INPUT-Befehl ja ganz recht, doch woher soll ein Mensch, der das Programm nicht geschrieben hat, wissen, was er eingeben soll? Seinen Namen, das Datum, die Uhrzeit? Um ein Programm benutzerfreundlich zu machen, sollte noch ein Text mit ausgegeben werden, der dem Benutzer verrät, welche Eingabe gewünscht wird (so ein Datum in Schilling umgerechnet ist sicher ganz amüsant, aber ...). Eine Möglichkeit dazu kennen Sie bereits: PTINT. Somit könnte die Programmzeile lauten:

```
PRINT " Rechnungssumme in DM: "; : INPUT Betrag%
```

Das Semikolon dient wieder dazu, das Fragezeichen direkt hinter dem Text Rechnungssumme in DM: auszugeben. Der Doppelpunkt trennt den INPUT- vom PRINT-Befehl. Als Ergebnis erhalten wir auf dem Monitor:

```
Rechnungssumme in DM: ?
```

Soweit, so gut! Aber es geht (noch!) einfacher! Der INPUT-Befehl ermöglicht nämlich auch die Angabe eines Textes, der mit ausgedruckt werden soll:

```
INPUT " Rechnungssumme in DM: ";Betrag%
```

bewirkt das gleiche Ergebnis auf dem Monitor, nur erscheint hierbei kein Fragezeichen:

```
Rechnungssumme in DM:
```

Möchten Sie mehrere Eingaben mit einem INPUT erledigen, müssen die einzelnen Variablen durch Kommata getrennt werden. Auch die Eingaben werden durch Komma getrennt:

```
10 INPUT " Bitte zwei Zahlen eingeben: ";A%,B%
20 PRINT " Die Summe der beiden Zahlen lautet: ";A%+B%
30 END

Bitte zwei Zahlen eingeben: 12,14 (Eingabe)
Die Summe der beiden Zahlen lautet: 26
```

Diese Art der Eingabe ist vor allem für Koordinatenpaare (x,y) sinnvoll:

```
10 REM Berechnung des Abstandes zwischen zwei Punkten
20 INPUT " Koordinaten für Punkt A (x1,y1): ";x1,x2
30 INPUT " Koordinaten für Punkt B (x2,y2): ";x2,y2
40 Abstand# = SQR((x1-x2)^2 + (y1-y2)^2)
50 PRINT " Der Abstand der beiden Punkte beträgt: ";Abstand#
60 END
```

In den Zeilen 20 und 30 werden die Koordinaten (durch Komma getrennt) für die Punkte A und B eingegeben. Soweit dürften noch keine Schwierigkeiten aufgetreten sein. Zeile 40 berechnet dann den Abstand dieser Punkte nach der Formel:

$$\text{Abstand} = \text{Wurzel aus } (x1 - x2)^2 + (y1 - y2)^2$$

Die Funktion SQR( <Wert> ) ergibt die Quadratwurzel von <Wert>, die der Variablen Abstand# zugewiesen wird. Der Operator ^ ist bereits bekannt und erzeugt eine Potenz, in unserem Fall das Quadrat der Differenzen x1 - x2 bzw. y1 - y2.

Zeile 50 gibt das so berechnete Ergebnis via Print auf dem Monitor aus. Da Kommata beim INPUT-Befehl dazu dienen, Eingaben, die verschiedenen Variablen zugewiesen werden sollen, zu trennen, ist es nicht möglich ein Komma einem String zuzuweisen. Ein anderer INPUT-Befehl erlaubt dies allerdings:

```
LINE INPUT
```

entspricht in seiner Synatx dem INPUT, Kommata können jedoch mit eingegeben werden. Stehen mehrere Variablen hinter dem Befehl, so ist für jede Variable eine eigene Eingabezeile nötig. Also:

```
10 LINE INPUT " Bitte zwei Zahlen eingeben: ";A%,B%
20 PRINT " Die Summe der beiden Zahlen lautet: "; A%+B%
30 END
```

Auf dem Monitor erscheint:

```
Bitte eine Zahl eingeben: 12 (Eingabe, mit Return beendet)
                          14 (Eingabe, mit Return beendet)
Die Summe der beiden Zahlen lautet: 26
```

Soviel zu INPUT bzw. LINE INPUT. In einem späteren Kapitel werde ich Ihnen die übrigen Eingabebefehle vorstellen.

## 2.6 Mathematische Funktionen

Ehe Sie jetzt völlig frustriert dieses Buch in die Ecke werfen, weil Sie seit Ihrer Schulzeit mit der Mathematik auf Kriegsfuß stehen, sollten Sie wissen, daß es gar nicht so schlimm wird, wie die Überschrift vielleicht befürchten läßt. Genaugenommen haben Sie sogar schon mit einer mathematischen Funktion Bekanntschaft geschlossen: Die Wurzelfunktion SQR(). Und zudem möchte ich keine Nachhilfestunde in Sachen Mathematik ertei-

len, sondern ein paar Funktionen vorstellen, die wir zur zweiten Verbesserung im Programm - dem Runden des umgerechneten Betrages auf zwei Nachkommastellen - benötigen.

Zur Wiederholung: Bei einem Rechnungsbetrag von 300 DM errechnete das Programm eine Summe von 2099,3702 Schilling. Die beiden letzten Kommastellen sind nicht relevant, also weg damit! Aber wie? Schon einmal wurden - damals freilich ungewollt - die Nachkommastellen einer Fließkommazahl abgeschnitten. Erinnern Sie sich? Genau, bei der Zuweisung einer Fließkommazahl an eine Integervariable fielen die Nachkommastellen weg.

An dieser Stelle muß ich etwas beichten: Die Nachkommastellen werden nicht einfach nur abgeschnitten. Bei Bedarf wird die resultierende Integerzahl nämlich noch aufgerundet. Der Computer sieht sich immer dann genötigt aufzurunden, falls der Nachkommanteil den Wert 0,5 übersteigt. Weist man also der Integerzahl A% den Wert 1,75 zu, erhält man in A% die Integerzahl 2."Nachtigall, ich hör' Dich trapsen!" mag der ein oder andere jetzt denken. Weist man einer Integervariable nämlich die Fließkommazahl Waehrung! aus unserem Programm zu, so erhält man als Ergebnis eine Integerzahl - bei Bedarf sogar noch gerundet! Und genau das wollten wir ja! Die beiden unnötigen (!?) Kommastellen sollten gestrichen werden. Doch bei unserer Zuweisung werden nicht nur die beiden letzten, sondern gänzlich alle Stellen hinter dem Komma entfernt. Ganz so einfach geht dies also nicht! Versuchen wir's anders:

Wenn wir - sozusagen mit einem Taschenspielertrick - die beiden ersten Nachkommastellen retten, dann die restlichen Nachkommastellen abschneiden und zu guter Letzt die beiden ersten Nachkommastellen wieder anhängen könnten, wäre unser Problem gelöst. Ein Versuch ist diese Idee mit Sicherheit wert. Zuerst also die beiden ersten Nachkommastellen retten:

Im Dezimalsystem besitzt jede Ziffer von rechts aus betrachtet den zehnfachen Wert ihrer Vorgängerin. Damit habe ich Sie schon in einem frühren Kapitel mächtig gelangweilt! Multipliziert man jetzt eine Zahl mit 10, wandern alle Ziffern um eine

Stelle nach rechts, bei einer Multiplikation mit 100 um 2 Stellen.

```
2345 * 10 = 23450
2345 * 100 = 234500
```

Ähnlich ergeht es Nachkommastellen bei einer Multiplikation mit 10 bzw. 100:

```
2099,3702 * 10 = 20993,702
2099,3702 * 100 = 209937,02
```

Jetzt kann man die beiden noch vorhandenen Nachkommastellen abschneiden, da die beiden ersten Stellen hinter dem Komma durch die Multiplikation mit 100 vor das Komma gewandert sind.

Diese beiden Stellen müssen nach dem Abschneiden der Nachkommastellen wieder hinter das Komma geschoben werden. Aber das stellt an dieser Stelle kein Problem mehr für uns dar. Wenn nämlich durch Multiplikation mit dem Wert 10 die Stellen um eine Position nach links weiterwandern, dann müssen Sie durch eine Division mit 10 wieder um eine Stelle nach rechts geschoben werden können:

```
209937 / 10 = 20993,7
209937 / 100 = 2099,37
```

Die einzige Fehlerquelle, die wir jetzt noch vermeiden müssen, ist die Integer-Division, die keine Nachkommastellen liefert. Ansonsten kann nichts mehr schiefgehen:

```
....
....
120 Waehrung=(Betrag%/Kurs!)*100' zwei Stellen nach links
125 Waehrung!=Waehrung/100' danach wieder nach rechts
....
....
```

Fügt man diese beiden Zeilen in unser kleines Programm ein, erhält man wie gewünscht zwei Stellen hinter dem Komma, und das sogar auf- bzw. abgerundet! Dies ist eine Lösung für das Problem, aber darauf wollte ich eigentlich gar nicht hinaus! Es geht nämlich noch kürzer (in einer Zeile!).

In ST-BASIC existieren drei mathematische Funktionen, die sich mit Vor- und Nachkommateilen einer Zahl befassen:

- INT
- FIX
- FRAC

### INT

Die Funktion INT schneidet den Nachkommateil einer Zahl ab, d.h. genauer gesagt, sie bildet die größte ganze Zahl:

```
A = INT(B)
```

Der Nachkommateil der Zahl B wird abgeschnitten, die auf diese Weise erhaltene Integerzahl der Variablen A zugewiesen. Bei negativen Zahlen erhalten Sie die größte ganze Zahl, die kleiner als B ist.

| B | Int(B) |
|---|---|
| +3.1 | 3 |
| +3.8 | 3 |
| +1.9 | 1 |
| -2.1 | -3 |
| -2.9 | -3 |

### FIX

Die Funktionsweise von FIX entspricht im wesentlichen der von INT. Bei negativen Zahlen wird der Nachkommateil jedoch ebenfalls abgeschnitten.

| B | Int(B) | Fix(B) |
|---|---|---|
| +3.1 | 3 | 3 |
| +3.8 | 3 | 3 |
| +1.9 | 1 | 1 |
| -2.1 | -3 | -2 |
| -2.9 | -3 | -2 |

## FRAC

FRAC schneidet bei positiven Zahlen den Vorkommateil einer Zahl ab, bei negativen Werten gilt: FRAC(X) = X - FIX(X).

| B | Frac(B) |
|---|---|
| 3.1 | .1 |
| 3.61 | .61 |
| 3 | 0 |
| -2.5 | -.5 |
| -6.9 | -.9 |

Wir möchten den Nachkommateil einer Zahl streichen, dazu benutzen wir entweder die Funktion INT oder FIX. Negative Werte werden in dem kleinen Beispiel kaum vorkommen, deshalb liefern beide Funktionen das gleiche Ergebnis. Benutzen wir also INT.

INT schneidet den Nachkommateil einer Zahl ab. Deshalb müssen wir auch in diesem Fall die ersten zwei Nachkommastellen durch eine Multiplikation mit 100 retten. Danach wird die Funktion auf den soeben erhaltenen Wert angewendet und schließlich gleich wieder durch 100 dividiert:

```
120 Waehrung!= INT(Betrag%/Kurs!*100)/100
```

Eine Kleinigkeit fehlt noch. Die Funktion INT schneidet nämlich die Nachkommastellen, ohne eine Rundung durchzuführen, einfach ab. Es bleibt uns also nichts anderes übrig, als die Rundung selbst in die Hand zu nehmen. Sehen wir uns dazu die Funktion INT anhand einer Zahlentabelle näher an:

| B | INT(B) |
|---|---|
| 2.0 | 2 |
| 2.1 | 2 |
| 2.2 | 2 |
| 2.3 | 2 |
| 2.4 | 2 |
| 2.5 | 2 |
| 2.6 | 2 |
| 2.7 | 2 |
| 2.8 | 2 |
| 2.9 | 2 |
| 3.0 | 3 |
| 3.1 | 3 |
| 3.2 | 3 |

Sie sehen deutlich, daß stets nur der Vorkommateil einer Zahl in das Ergebnis einfließt. Aber wieder können wir uns mit einem kleinen Trick behelfen. Da ab einem Nachkomma von ,5 gerundet werden soll, addieren wir zu unserer Zahl einfach den Wert 0,5. Ist der Nachkommawert der Zahl kleiner als 0,5, ändernt sich die Zahl vor dem Komma nicht, und die INT-Funktion schneidet weiterhin die Nachkommastellen ab. Wozu dann der ganze Aufwand?

Ab einem Nachkommateil von ,5 ändert sich durch die Addition auch der Wert der ersten Stelle vor dem Komma (z.B. 2,6 + .5 = 3,1). Wendet man auf diesen neuen Wert die INT-Funktion an, erhält man - ein Blick in obige Tabelle verrät's - den Wert 3! Genau das wollten wir ja erreichen!

Die zweite Version unseres Programms finden Sie anschließend abgedruckt. Die Funktionsweise müßte durch die bisher gemachten Ausführungen und Erklärungen verständlich sein. Eine Kleinigkeit wurde zusätzlich noch geändert. Die Variable Betrag! ist nun eine Fließkommazahl mit einfacher Genauigkeit, es ist also auch möglich, Kommastellen bei der Rechnungssumme mit einzugeben. Ferner wird der Bildschirm (logischerweise) vor dem INPUT-Befehl gelöscht.

```
10 '************************************************************
20 '*                        WAEH_V02.BAS                      *
30 '*----------------------------------------------------------*
40 '* Autor: Michael Maier    Version: 1.00     Datum: 22.08.1988 *
50 '*      Ein Programm aus dem 'GROSSEN ST-BASIC-BUCH'         *
60 '*        (C) 1988 by DATA BECKER GmbH Düsseldorf            *
70 '************************************************************
80 '
90 '
100 Kurs!=.1429: REM Wechselkurs DM/ÖS
110 CLS : INPUT " Rechnungssumme in DM: ";Betrag!
120 Waehrung!= INT((Betrag!/Kurs!*100)+.5)/100
130 PRINT : PRINT Betrag!;" DM";" entsprechen";Waehrung!;" Schilling"
140 END
```

## 2.7 Strings und Stringmanipulation

Über Strings habe ich mich schon einmal an anderer Stelle ausgelassen. Ehe ich Ihnen auch noch verrate, was man mit Zeichenketten so alles anstellen kann, lade ich Sie zu einem kleinen Ausflug in das Innere Ihres Computers ein.

### Buchstaben sind auch nur Zahlen

Was soll denn das schon wieder? Buchstaben sind Buchstaben und Zahlen sind Zahlen, da beißt die Maus keinen Faden ab, oder etwa doch? Sie kennen natürlich den Unterschied zwischen einer Zahl und einem Buchstaben, bzw. den daraus resultierenden Zeichenketten. Der Computer ist in diesem Punkt leider nicht so flexibel wie Sie. Ehe ein Buchstabe auf dem Monitor dargestellt werden kann, ist eine Menge Aufwand nötig:

Die Tastatur des Atari ST wird von einem eigenen Prozessor überwacht, der sinnigerweise nach seiner Aufgabe Tastaturprozessor genannt wird. Drückt der Benutzer eine Taste, meldet dies der Tastaturprozessor dem Betriebssystem. Dazu schickt er ein Datenpaket auf die Reise, anhand dessen die Taste genau identifiziert werden kann. Es ist aber falsch anzunehmen, ein Drücken der Taste <A> bewirkt, daß der Buchstabe a an das Be-

triebssystem weitergeleitet wird. Statt dessen erhält es die Information, welche Taste gedrückt wurde in Form einer Zahl. Mit Zahlen kann der Computer ja bestens umgehen.

Das Betriebssystem entscheidet nun, was zu geschehen hat. Muß der Buchstabe beispielsweise auf dem Monitor ausgegeben werden, so wird eine Tabelle zu Rate gezogen: alle Zeichen sind dort zusammengefasst und eindeutig anhand einer Nummer (Zahl) identifizierbar. In dieser Tabelle ist festgelegt, wie jedes Zeichen bei einer Ausgabe auf dem Monitor auszusehen hat. Diese Daten holt sich der Computer ab, und kann jetzt den Inhalt der entsprechenden Taste auf dem Monitor ausgeben.

Also ist für den Computer ein Buchstabe eigentlich auch nur eine Zahl, nämlich die Nummer dieses Buchstabens in einer Tabelle. Da auch bei einer Kommunikation zwischen Computer und Drucker bzw. einem anderen Computer keine Buchstaben, sondern lediglich die entsprechenden Zahlenwerte übertragen werden, wurde diese Tabelle für bestimmte Werte standarisiert. Es muß nämlich eindeutig feststehen, welche Zahl den Buchstaben a in dieser Tabelle repräsentieren soll. Schickt z.B. der Computer die entsprechenden Codes für das Wort "BASIC" an den Drucker und diese Werte wären nicht genormt, wäre es mehr als Zufall, wenn "BASIC" auch auf dem Drucker ausgegeben würde.

Standardisiert sind jedoch - wie gesagt - nur bestimmte Zeichen, darunter die Buchstaben und Ziffern. Sonderzeichen wie ä,ö,ü,ß gibt es in keiner anderen Sprache, sie wiedersetzen sich somit jedem Standard. Deshalb ist es jedem Computerhersteller freigestellt, bestimmte Sonderzeichen in den Zeichensatz des Computers zu integrieren, die von der Norm abweichen. Dies ist beispielsweise bei den Umlauten im Atari ST der Fall: Schließen Sie einen Epson-kompatiblen Drucker an den Computer an, so erhalten Sie beim Ausdruck statt geschweifter Klammern Umlaute. Umlaute werden dagegen vom Drucker einfach verschluckt. Die übrigen Buchstaben werden jedoch korrekt zu Papier gebracht, da ihr Code normiert ist.

Diese Tabelle, in der sämtliche Zeichen über einen standarisierten Code zu erreichen sind, heißt ASCII-Tabelle (American Standard Code for Information Interchange). Sie umfaßt 256 Zeichen, von denen 128 Werte standarisiert sind. Darunter fallen Buchstaben, Ziffern, Interpunktionszeichen, und bestimmte Steuerzeichen.

Soweit, so gut, aber wie kommt man nun an den Code eines Zeichens heran? Dazu dient in BASIC die Function

```
ASC
```

Das Zeichen, dessen Wert Sie wissen möchten, muß in Klammern hinter dem ASC stehen: ASC("a"). Selbstverständlich muß der auf diese Weise resultierende Wert einer Variablen zugeordnet werden, damit er nicht irgendwo im Speicher verloren geht:

```
A% = ASC("a")
```

weist der Variablen A% den ASCII-Wert des Zeichens a zu. Aber auch die Umkehrfunktion ist möglich. Dabei wird das zu einem ASCII-Wert gehörende Zeichen ermittelt:

```
PRINT CHR$(97)
```

gibt den Buchstaben a auf dem Monitor aus.

Auf diese Weise werden die Zahlen in Buchstaben umgewandelt und umgekehrt. Noch ein paar Beispiele:

```
10 A$ = "A"
20 PRINT ASC(A$)
30 END
```

Der Code des Zeichens A wird ermittelt und auf dem Monitor ausgegeben. Er beträgt stets 65.

```
10 A$ = "Aller Anfang ist leicht"
20 PRINT ASC(A$)
30 END
```

Wie Sie aus den drei Programmzeilen leicht ersehen können, wird auch hier die Zahl 65 ausgegeben, obwohl der Funktion ASC in diesem Fall nicht nur ein Zeichen, sondern ein ganzer String als Argument (der Wert in Klammern) übergeben wurde. Bei Übergabe eines Strings wird nur der ASCII-Wert des ersten Zeichens ermittelt, die übrigen Zeichen werden vom Computer einfach ignoriert.

```
10 A% = 65
20 PRINT CHR$(A%)
30 END
```

Auf dem Monitor erscheint der Buchstabe A. Zu beachten ist hierbei, daß der ASCII-Code nur für Werte von 0 bis 255 definiert ist. Übergibt man größere Werte oder negative Zahlen als Argument, wird dies vom Computer mit einer Fehlermeldung quittiert, die zu einem Programmabbruch führt. Aber auch Sonderzeichen können mit der CHR$-Funktion auf den Bildschirm gebracht werden:

```
PRINT CHR$(14);CHR$(15)
```

Diese beiden Werte erzeugen auf dem Bildschirm das Atari-Symbol. Steuerzeichen finden sich ebenfalls in der Tabelle:

```
PRINT CHR$(7)
```

läßt einen Glockenton (auch als Piepsen bezeichnet) erklingen. Die Umkehrfunktion, mit der der ASCII-Wert des soeben erzeugten Tones ermittelt werden kann, ist jedoch nicht möglich. Falls Sie es dennoch schaffen sollten, so geben Sie mir bitte Bescheid! Ein weiters, häufig gebrauchtes Steuerzeichen besitzt in der ASCII-Tabelle den Wert 13 und wird als Carriage Return (Wagenrücklauf) bezeichnet. Dies ist der ASCII-Code, den auch die <Return>-Taste beim Betätigen auslöst.

Die ASCII-Tabelle des Atari ST finden Sie im Anhang dieses Buches. Wir werden in späteren Kapiteln noch einige Male auf sie stoßen.

Also stimmt es doch! Der Computer unterscheidet intern nicht zwischen Buchstaben und Zahlen. Was auf den ersten Blick als störend und ungewohnt empfunden wird, erweist sich aber schon bald als Vorteil. Möchten Sie z.B. in einem Programm Namen alphabetisch sortieren, kommt der ASCII-Code sehr gelegen. Der Buchstabe B besitzt den Wert 66, A dagegen nur 65, Z schließlich wird als Schlußlicht durch den Wert 90 dargestellt. Deshalb ist für den Computer der Buchstabe B größer als der Buchstabe A. Werden nun Namen sortiert, greift der Computer auf die entsprechenden ASCII-Codes der Buchstaben zurück. Also auf Zahlen, die er eindeutig nach ihrer Größe ordnen kann. Bei Buchstaben könnte er dies nicht.

Aber auch in anderen Fällen ist der ASCII-Code für den Programmierer von Nutzen. Möchten Sie beispielsweise bestimmte Daten im Programm verschlüsseln, um sie vor neugierigen Blicken anderer zu schützen, kann dies über den ASCII-Code geschehen. Sie ermitteln einfach die ASCII-Werte der Buchstaben, die nach einem bestimmten System codiert werden. Ehe die Daten später wieder auf dem Monitor ausgegeben werden, lassen sie diese vom Computer dekodieren und basteln sich so einen String über die CHR$-Funktion zusammen, wobei die soeben dekodierten Zahlen als Argumente benutzt werden. So einfach geht das.

### String-Addition

Zahlen können Sie schon addieren. Aber auch eine Addition von Strings ist in BASIC problemlos möglich. Als Operator dient auch hier das +-Zeichen. Es bewirkt, daß zwei Strings hintereinander gehängt werden. Der auf diese Weise neu entstehende String darf jedoch nicht mehr als 32766 Zeichen enthalten.

```
10 A$ = "ST-BASIC"
20 B$ = "Buch"
30 C$ = A$+" "+B$
40 PRINT " Der Inhalt von A$: ";A$
50 PRINT " Der Inhalt von B$: ";B$
60 PRINT " Der Inhalt von C$: ";C$
70 END
```

Lassen Sie dieses kleine Programm laufen, so erscheint auf dem Monitor:

```
Der Inhalt von A$: ST-BASIC
Der Inhalt von B$: Buch
Der Inhalt von C$: ST-BASIC Buch
```

In Programmzeile 10 wird der Variablen A$ die Zeichenkette ST-BASIC, in Zeile 20 der Variablen B$ der String Buch zugewiesen. In Zeile 30 wird zuerst ein Leerzeichen (Space) an den Inhalt von A$ angehängt, danach folgt der Inhalt von B$. Die soeben durch String-Addition erhaltene Zeichenkette wird der Variablen C$ zugewiesen. Jetzt wird der Inhalt der einzelnen Variablen auf den Monitor geschrieben. Sie sehen, daß die beiden Stringvariablen durch das +-Zeichen verkettet wurden. Der Zwischenraum wurde mit einem Leerzeichen eingefügt, damit beide Wörter nicht direkt aneinandergehängt werden. Als ASCII-Code besitzt das Lerrzeichen den Wert 32. Folglich hätte ich die Zeile 30 auch so formulieren können:

```
C$=A$+CHR$(32)+B$
```

### String-Multiplikation

Ein Kuriosum, das ich bisher nur in ST-BASIC kenne: Strings können "multipliziert" werden. So liefert die Anweisung

```
A$ = "+" * 5
```

in der Variablen A$ den String +++++, also genau 5 mal das Zeichen +.

```
A$ = "-"*10+CHR$(32)*2+"Headline"+CHR$(32)*2+"-"*10
```

weist der Variablen A$ die Zeichenkette

```
---------- Headline ----------
```

zu. Es wird also die Rechenregel "Punkt vor Strich" bei der Verknüpfung beachtet, sogar Klammern dürfen gesetzt werden:

```
PRINT "+"+"-"*5
PRINT ("+"+"-")*5
```

Im ersten Fall erhalten Sie auf dem Monitor +-----, während im zweiten Fall das Ergebnis +-+-+-+-+- lautet. Der Multiplikationsoperator (*) muß hinter dem String stehen, den Sie vervielfachen möchten. Andernfalls meldet sich der Computer mit "Type mismatch" zu Wort.

Möchten Sie ihre Programme portabel gestalten, also auf andere Computer leicht übertragbar machen, so sollten Sie statt der String-Multiplikation des ST-BASIC möglichst die STRING$-Funktion bemühen, die das gleiche Ergebnis erzielt, jedoch im Gegensatz zur Stringmultiplikation auch in anderen Interpretern implementiert ist. Der einzige Unterschied zwischen beiden Varianten besteht darin, daß man bei STRING$ nur jeweils ein Zeichen vervielfachen kann, während der Operator (*) dies auch für Strings zuläßt. Abhilfe können hier folgende Zeilen schaffen, die einen String (A$) n-mal aneinanderhängen. Das Ergebnis der Multiplikation erhalten Sie in der Variablen Multi$:

```
...
...
Multi$="" FOR T% = 1 TO N'  N durch entsprechende Zahl ersetzen
 Multi$=Multi$+A$
NEXT T%
...
...
```

Keine Angst, obige Zeilen habe ich an dieser Stelle nur der Vollständigkeit halber eingefügt, damit Sie sehen, wie die String-Multiplikation in anderen BASIC-Dialekten realisiert werden kann, nämlich durch eine n-malige Addition der Strings.

### Stringkonvertierung mit Val und Str$

Auch wenn man von String-Addition oder gar String-Multi-plikation spricht, so bedeutet dies nicht, daß man mit Strings rechnen kann. Vielmehr handelt es sich bei diesen Operationen um eine Verknüfung von Zeichenketten. Mit Zahlen ist die Rechnerei dagegen unproblematisch. Versucht man zu der Zeichenkette 134 die Zahl 3 zu addieren, führt dies zu einer Fehlermeldung. Äpfel und Birnen lassen sich einfach nicht addieren. Wandelt man die Zeichenkette 134 dagegen vor der Addition auch in eine Zahl um, gibt es keine Probleme. Zur Umwandlung eines Strings in eine Zahl (numerischen Ausdruck) wird in BASIC der Befehl

```
VAL(Zeichenkette)
```

verwendet. Dabei wird die Zeichenkette von links nach rechts systematisch abgeklappert, und solange umgewandelt, bis der Interpreter auf ein Zeichen stößt, das nicht umgewandelt werden kann, oder das Ende der Zeichenkette erreicht ist. Damit ist die Arbeit für den Computer erledigt. Selbst wenn später noch einmal ein Zahlenausdruck folgen sollte, wird dieser nicht konvertiert:

| Zeichenkette | Val (Zeichenkette) |
|---|---|
| 354 | 354 |
| 21.September | 21 |
| Basic | 0 |
| 32.5_blabla | 32,5 |
| 43 tausend 432 | 43 |

Links sehen Sie die Zeichenkette, die als Argument bei Val angewendet wird. Kann der String nicht konvertiert werden (z.B. bei BASIC), liefert die Funktion den Wert 0. Auch VAL("DM 42.17") führt zu dem Ergebnis 0, da bereits das erste Zeichen des Strings nicht konvertiert werden kann. VAL("42.17 DM") dagegen gibt als Ergebnis der Operation die Zahl 42,17 zurück.

Der umgekehrte Vorgang. Ein numerischer Ausdruck (Zahl) soll in eine Zeichenkette umgewandelt werden. Auch das geht, und zwar mit der Funktion

```
STR$(<numerischer Ausdruck>)
```

Als Ergebnis liefert die Funktion einen Stringausdruck, der einer Stringvariablen zugewiesen werden kann:

| numerischer Wert | resultierende Zeichenkette |
|---|---|
| 12 | "12" |
| 48.53 | "48.53" |
| -14 | "-14" |
| STR$(43+7) | "50" |
| STR$(A%) | je nach Inhalt der Variable A% |

An die erste Stelle des entstehenden Strings wird bei positiven Werten ein Leerzeichen gestellt, das als Platzhalter für ein evtl. auftretendes negatives Vorzeichen dient.

### Stringmanipulation

Zeichenketten werden mit dem Operator (+) verknüpft. Sie können jedoch auch wieder "auseinandergenommen" werden:

### LEFT$

Der Befehl Left$ liefert aus einem STRING <String> genau die ersten n Zeichen von links:

```
A$=LEFT$(<String>,n)
```

Die Variable B$ enthalte die Zeichenkette "Donaudampfschiffahrtsgesellschaft". Dann weist LEFT$ der Variablen A$ folgende Zeichenketten zu: (A$=LEFT$(B$,n))

| n | resultierende Zeichenkette |
|---|---|
| 5 | Donau |
| 10 | Donaudampf |
| 15 | Donaudampfschiff |
| 20 | Donaudampfschiffahrt |

## RIGHT$

Während LEFT$ die ersten n Zeichen des Strings liefert, erreicht man die letzten n Zeichen der Zeichenkette mit der Funktion RIGHT$. Die Syntax beider Befehle ist wieder identisch:

```
A$ = RIGHT$(<String>,n)
```

| n | resultierende Zeichenkette bei RIGHT$ |
|---|---|
| 12 | gesellschaft |
| 23 | schiffahrtsgesellschaft |
| 28 | dampfschiffahrtsgesellschaft |

## MID$

Häufig benötigt man einen bestimmten Teilstring aus einer Zeichenkette. Dieser Teilstring kann über MID$ erreicht werden. Zwei Variationen von MID$ sind möglich:

```
A$ = MID$(<String>,n)
```

liefert alle Zeichen ab der n-ten Position bis zum String-Ende von <String>:

| n | resultierende Zeichenkette |
|---|---|
| 6 | dampfschiffahrtsgesellschaft |
| 11 | schiffahrtsgesellschaft |
| 22 | gesellschaft |

Bei der zweiten Variante kann zusätzlich mit angegeben werden, wie viele Zeichen (<Anzahl>) ab der n-ten Position zurückgegeben werden sollen:

```
A$ = MID$(<String>,Anzahl,n)
```

| n | Anzahl | resultierende Zeichenkette |
|---|---|---|
| 6 | 5 | dampf |
| 6 | 11 | dampfschiff |
| 11 | 6 | schiff |
| 1 | 5 | Donau |
| 16 | 5 | fahrt |

Mit diesen Befehlen dürfte es kein Problem mehr bereiten, eine Zeichenkette in ihre Bestandteile zu zerlegen. Möchten Sie bestimmte Teilstrings in einer Zeichenkette abändern, so haben Sie hierfür mehrere Möglichkeiten:

Beispiel:

```
A$ = "Das kleine ST-BASIC Buch"
```

Möchten Sie "kleine" durch "grosse" ersetzten, wird dies über LEFT$ und MID$ etwa so bewerkstelligt:

```
A$ = LEFT$(A$,4)+"grosse"+MID$(A$,11)
```

Sie holen sich die ersten vier Zeichen (Das) mit Hilfe von LEFT$ und hängen das gewünschte "grosse" an. Zum Schluß werden die restlichen Zeichen von A$ ab der 11. Position angefügt, und der neu gewonnene Stringausdruck der Variablen zugewiesen. In A$ ist jetzt die Zeichenkette "Das grosse ST-BASIC-Buch" enthalten. Das gleiche Ergebnis kann auch mit LEFT$ und RIGHT$ erzielt werden:

```
A$ = LEFT$(A$,4)+"grosse"+RIGHT$(A$,14)
```

Ich kann Ihnen sogar noch eine dritte Alternative anbieten:

```
MID$(A$,5,6) = "grosse"
```

Im letzteren Fall wurde eine spezielle Version von MID$ benutzt. MID$(<String>,n,Anzahl) = X$ ersetzt <Anzahl> Zeichen ab der n-ten Position in <String> durch X$. Beliebte Fehlerquellen bei LEFT$, MID$ und RIGHT$:

- Es wird versucht, mehr Zeichen aus dem String abzuholen, als überhaupt enthalten sind: A$="Hallo" B$=LEFT$(A$,10) führt zu einer Fehlermeldung, da der String lediglich aus 5 Buchstaben besteht. Ebenso falsch sind folgende Ausdrücke: B$=MID$(A$,6), B$=RIGHT$(A$,10), B$=MID$(A$,2,8), usw.
- Der resultierende Wert ist bei allen drei Funktionen wieder eine Zeichenkette, die einer Stringvariablen zugewiesen werden muß: B$=LEFT$(A$,2). Flasch ist: B%=LEFT$(A$,2)
- LEFT$, MID$, und RIGHT$ dürfen nur auf Zeichenketten angewendet werden. Falsch wäre: A$=MID$(B, 2,3), da B eine Integer- und keine Stringvariable ist.
- Das erste Zeichen eines Strings hat die Position 1: A$=MID$(B$,1,5) entspricht A$=LEFT$(B$,5). Falsch ist dagegen: A$=MID$(A$,0,5).

## LEN

Die Funktion LEN, auf einen Stringausdruck angewendet, liefert als Ergebnis die Anzahl der in dieser Zeichenkette enthaltenen Buchstaben:

| String | LEN(<String>) |
|---|---|
| Donau | 5 |
| ST-Basic | 8 |
| Atari ST | 8 |
| g | 1 |

```
10 CLS : INPUT " Bitte geben Sie ein Wort ein: ";A$ 20 Laenge%=
LEN(A$)
30 PRINT " Das Wort ";A$;" enthält";Laenge%;" Buchstaben"
40 END
```

Obiges Programm verdeutlicht noch einmal die Funktionsweise von LEN. In Zeile 10 wird zuerst der Bildschirm gelöscht, anschließend erhält der Benutzer die Aufforderung ein beliebiges Wort einzugeben, das der Variablen A$ zugewiesen wird. Zeile 30 ermittelt über LEN(A$) die Anzahl der in A$ enthaltenen

Buchstaben und weist das Ergebnis der Integervariablen Laenge% zu. In Zeile 40 wird dann das Wort sowie seine Länge auf dem Bildschirm angezeigt.

### SPC bzw. SPACE$

Beide Funktionen erzeugen einen String, der aus <Anzahl> Leerzeichen (Spaces) besteht:

```
A$=SPC(10)
A$=SPACE$(10)
```

weisen der Variablen A$ einen Leerzeichenstring der Länge 10 zu. Diese Funktionen erfahren ihre Anwendung immer dann, wenn Daten formatiert auf dem Bildschirm oder dem Drucker ausgegeben werden sollen und unterschiedliche Längen durch Leerzeichen ausgeglichen werden müssen, damit ein übersichtliches Formular entsteht.

**Anmerkung:** In anderen BASIC-Dialekten ist die Funktion SPC auf den PRINT-Befehl beschränkt. Ein Leerzeichenstring kann dort nur über die Funktion Space$ erzeugt werden. In ST-BASIC sind beide Alternativen gleichwertig. Möchten Sie jedoch Programme auf andere BASIC-Interpreter übertragen, kann es unter Umständen zu Problemen kommen, wenn Sie SPC außerhalb des Print-Befehls verwenden.

### STRING$

STRING$ entspricht der String-Multiplikation mit *, ist jedoch auf das Vervielfachen eines Zeichens beschränkt:

```
A$=STRING$(<Anzahl>,<Zeichen>)
```

weist der Variablen A$ <Anzahl>-mal das Zeichen <Zeichen> zu. Ein paar Beispiele:

| Anzahl | Zeichen | STRING$(Anzahl, Zeichen) |
|---|---|---|
| 10 | "+" | ++++++++++ |
| 5 | "A" | AAAAA |
| 5 | CHR$(65) | AAAAA |
| 5 | CHR$(32) | " " vgl. Space$(5) |

### MIRROR$

Die Funktion A$=MIRROR$(<String>) weist der Variablen A$ den Inhalt von <String> zu, jedoch von hinten nach vorne gelesen:

| <String> | MIRROR$(<String>) |
|---|---|
| hallo | ollah |
| birgit | tigrib |
| roma | amor (so ein Zufall!) |
| basic | cisab |

### UPPER$

Die Funktion A$=UPPER$(<String>) weist der Variablen A$ den Inhalt von <String> zu, sämtliche Kleinbuchstaben werden in Großbuchstaben umgewandelt, Sonderzeichen und Zahlen bleiben unberührt:

| <String> | Upper$(<String>) |
|---|---|
| langmeier | LANGMEIER |
| bader | BADER |
| Peter | PETER |
| SaBINe | Sabine |

**Tip:** Da Sonderzeichen von dieser Funktion nicht betroffen sind, die Umlaute ä,ö,ü jedoch unter diesen in der ASCII-Tabelle abgelegt sind, sollte am Programmanfang mit dem Befehl MODE"D" der Modus "deutsch" eingestellt werden. In diesem Fall werden bei einem UPPER$ auch die Umlaute in Großbuchstaben gewandelt, das ß wird jedoch immer durch Doppel-S ersetzt.

### LOWER$

Die Umkehrfunktion zu UPPER$ lautet LOWER$ und wandelt sämtliche in <String> enthaltenen Großbuchstaben in Kleinbuchstaben um. Kleinbuchstaben, Sonderzeichen und Zahlen bleiben unverändert erhalten. Ist MODE"D" aktiv, werden auch die Umlaute Ä, Ö, Ü in Kleinbuchstaben gewandelt.

| <String> | Lower$(<String>) |
|---|---|
| LANGMEIER | langmeier |
| BaDer | bader |
| GSG9 | gsg9 |
| SaBINe | sabine |

### MODE

Eigentlich hat dieser Befehl ja nichts mit String-Manipulation zu tun, und dennoch hat er auf Befehle zur String-Manipulation Einfluß. Deshalb möchte ich ihn an dieser Stelle einbauen. Ein Beispiel für seine Anwendung haben Sie bei UPPER$ und LOWER$ kennengelernt.

MODE "D" schaltet auf den Modus "deutsch", MODE "GB" auf den Modus "Großbritannien" und MODE "USA" schließlich auf "amerikanisch". Sofern die verschiedenen Modi Einfluß auf einen Befehl ausüben, werde ich dies bei der Erklärung des entsprechenden Befehls nachreichen.

### String-Matching

Als String-Matching bezeichnet der Informatiker das Aufsuchen eines Teil-Strings in einem (Mutter-)String. Dies wäre z.B. der Fall, wenn Sie in diesem Buch das Wort String-Matching suchen würden. Der Teilstring bestünde dann aus String-Matching, während der Mutter-String der Inhalt dieses Buches wäre. Wir wollen an dieser Stelle jedoch nicht gleich nach den Sternen greifen, und uns mit dem Auffinden eines Teilstrings in einer Zeichenkette normaler Größe befassen. Glücklicherweise sind wir mit einer Funtion gesegnet, die uns diese Aufgabe abnimmt

(C-Programmierer haben da weniger Glück: sie müssen sich ihre eigene Funktion schreiben und das ist um einiges komplizierter...):

```
Pos=INSTR(<Mutterstring>,<Suchstring>)
```

Die Funktion INSTR untersucht, ob die Zeichenkette <Such-String> im <Mutter-String> enthalten ist. Ist dies der Fall, so wird der Variablen Pos die Position des Suchstrings innerhalb des Mutterstrings zugewiesen, andernfalls gibt die Funktion den Wert 0 zurück. Klingt kompliziert? Dann schnell ein Beispiel:

| <Mutterstring> | <Suchstring> | Position |
|---|---|---|
| Dies ist ein kleiner Test | ist | 6 |
| Dies ist ein kleiner Test | war | 0 |
| Dies ist ein kleiner Test | ie | 2 |

Zusätzlich kann noch angegeben werden, ab welcher Position mit der Suche im <Such-String> begonnen werden soll:

```
Pos=INSTR(<Beginn>,<Mutterstring>,<Suchstring>)
```

Der Befehl hat die gleiche Wirkung, mit der Suche nach dem <Suchstring> wird jedoch erst ab der Position <Beginn> begonnen. Wird der Such-String aufgefunden, so gibt die Funktion die Position des <Such-Strings> innerhalb des <Mutter-Strings> zurück, andernfalls wird der Wert 0 zugewiesen:

| <Mutterstring> | <Beginn> | <Suchstring> | Position |
|---|---|---|---|
| Bei diesem Beispiel ... | 1 | Bei | 1 |
| Bei diesem Beispiel ... | 5 | Bei | 12 |
| Bei diesem Beispiel ... | 13 | Bei | 0 |

Soweit zum String-Matching. Ein kleines Beispielprogramm soll die Anwendung noch einmal in einem größeren Zusammenhang verdeutlichen. Dabei kommt es mir hier weniger auf die Eleganz an (so etwas würde ich nie in einem normalen Programm fabrizieren), sondern vielmehr auf eine kurze Wiederholung des bisher Gelernten. Auf den letzten Seiten wurden Sie mit Theorie direkt erschlagen, aber derart knüppeldick kommt es so schnell nicht wieder. Doch nun zu unserem kleinen Programm.

Es soll ein Name in eine String-Variable eingegeben werden, und zwar zuerst der Vorname, anschließend durch ein Leerzeichen getrennt der Familienname. Der erste Buchstabe des Vornamens soll in Großschrift, die restlichen Zeichen bis zum Space in Kleinbuchstaben dargestellt werden. Der Familienname soll völlig in Großbuchstaben erscheinen. Um die Sache noch etwas zu verkomplizieren, soll der Name, sowie die Länge des Vor- und Zunahmens formatiert ausgegeben werden. Hier nun das Programm:

```
0  '*****************************************************************
1  '*                          NAME.BAS                             *
2  '*---------------------------------------------------------------*
3  '* Autor: Michael Maier    Version: 1.00     Datum: 30.08.1988 *
4  '*      Ein Programm aus dem 'GROSSEN ST-BASIC-BUCH'             *
5  '*        (C) 1988 by DATA BECKER GmbH Düsseldorf                *
6  '*****************************************************************
7  '
8  '
9  CLS : PRINT '  Bildschirm löschen und Leerzeile ausgeben
10 INPUT " Bitte geben Sie Ihren Vor- und Nachnamen ein: ";Name$
11 ' Nach einem Leerzeichen suchen ...
12 Pos%= INSTR(Name$, CHR$(32))
13 '
14 ' Hier müßte eigentlich überprüft werden, ob ein Leerzeichen
15 ' eingegeben wurde, da andernfalls der Variablen Pos% der Wert
16 ' 0 zugewiesen wird, und das Programm aussteigt ...
17 '
18 Name$= UPPER$( LEFT$(Name$,1)+ MID$(Name$,2))
19 MID$ (Name$,2,Pos%-2)= LOWER$( MID$(Name$,2,Pos%-2))
20 Name$= LEFT$(Name$,Pos%)+ UPPER$( RIGHT$(Name$, LEN(Name$)-Pos%))
21 '
22 Vorname$= LEFT$(Name$,Pos%-1)
23 Zuname$= MID$(Name$,Pos%+1)
24 '
25 PRINT
26 PRINT STRING$(60,"*")
27 PRINT SPC(2);"Name"; SPACE$(10);"Vorname"; SPACE$(10);
28 PRINT "Zeichen Nachname    Vorname"
29 PRINT "***"*29+"***"
30 PRINT
31 PRINT "  ";Zuname$; SPACE$(14- LEN(Zuname$));
32 PRINT Vorname$; SPACE$(17- LEN(Vorname$));
33 PRINT SPC( LEN("Zeichen"));
34 PRINT LEN(Zuname$); SPACE$(12- LEN( STR$( LEN(Zuname$)-1)));
```

```
35 PRINT LEN(Vorname$)
36 '
37 END
```

In Zeile 9 wird der Bildschirm gelöscht. Das nachfolgende PRINT sorgt dafür, daß die erste Bildschirmzeile leer bleibt. In Zeile 10 erhält der Benutzer die Aufforderung seinen Vor- und Zunahmen einzugeben. Dieser wird der Variablen Name$ zugewiesen.

Zeile 12 untersucht die Variable Name$ auf ein Leerzeichen (CHR$(32)) und weist die Position des Leerzeichens der Variablen Pos% zu. Da diese für die weiteren Operationen benutzt wird, müßte in einem anwenderfreundlichen Programm eine Abfrage eingebaut werden, ob ein Leerzeichen in Name$ zu finden ist. Andernfalls enthält die Variable Pos% den Wert 0 zugewiesen und scheidet für die Benutzung in der Stringmanipulation aus. Doch da wir dies noch nicht können, habe ich darauf verzichtet.

Jetzt geht's rund! In Zeile 18 wird zuerst der erste Buchstabe des Strings mit dem Befehl LEFT$(Name$,1) abgeschnitten. Da LEFT$ ein Argument von UPPER$ ist, wird der erste Buchstabe (falls nötig) in einen Großbuchstaben gewandelt. Die restlichen Zeichen bis zum Stringende werden mit String-Addition wieder angehängt und das Ergebnis der Variablen Name$ zugewiesen. Dort ist nun wieder die ursprüngliche Zeichenkette zu finden, der erste Buchstabe allerdings auf alle Fälle in Großschrift.

Im nächsten Schritt werden die restlichen Buchstaben des Vornamens in Kleinschrift umgewandelt. Dazu besitzt ST-BASIC bekanntlich den Befehl LOWER$. In Pos% ist die Position des Leerzeichens gespeichert. Wir müssen also alle Zeichen ab der 2. Position von Name$ (den ersten Buchstaben haben wir ja schon in der letzten Zeile in Großschrift umgewandelt), bis zu Pos%-1 in Kleinbuchstaben verwandeln. Pos%-1 deshalb, da Pos% die Position des Leerzeichens angibt, das nicht mehr mitgewandelt werden muß. Berücksichtigt man ferner, daß der erste Buchstabe im String nicht wieder in Kleinschreibweise dargestellt werden soll, müssen die Parameter in MID$ 2 und Pos%-2 lauten. Auf diese Weise resultiert ein Teilstring, der mit dem 2. Buchstaben

beginnt und vor dem Leerzeichen endet. Gleichzeitig kommen die beiden Spielarten von MID$ zum Einsatz: rechts des Zuweisungsoperators (=) wird der Teil-String herausgeschnitten und darauf die Funktion LOWER$ angewendet. Diese Zeichenkette wird wieder in Name$ eingebaut. Dies erfolgt über das MID$ links des Zuweisungsoperators.

Der Vorname ist fertig umgewandelt, jetzt geht es dem Zunamen an den Kragen. Dieser beginnt nach dem Leerzeichen und endet mit dem String-Ende. Auch hier könnte man wieder MID$ bemühen, aber es geht auch - obgleich etwas komplizierter - über RIGHT$. Als Parameter müssen wir in RIGHT$ angeben, wieviele Zeichen wir vom String-Ende abschneiden möchten. Dies können wir ausrechnen, indem wir von der Stringlänge (LEN) die Position des Leerzeichens (Pos%) subtrahieren. Übrig bleibt die Anzahl der Buchstaben des Zunamens. Diesen Parameter benutzen wir in RIGHT$, um den Zunamen zu isolieren. Mit UPPER$ erfolgt die Umwandlung des Zunahmens in Großbuchstaben. RIGHT$ hat noch einen weiteren Haken: Da wir den String auseinandergenommen haben, müssen wir ihn auch wieder selbst zusammensetzen. Aber das bereitet jetzt keine Schwierigkeiten mehr: LEFT$ liefert die Zeichenkette bis zum Zunamen, daran hängen wir den umgewandelten Zunamen. Fertig!

In den Zeilen 22 und 23 zerlegen wir dann den soeben unter vielen Mühen zusammgengesetzten Namen erneut in seine beiden Bestandteile: Vor- und Zunamen.

Zeile 25 sorgt wieder für eine Leerzeile, ehe in Zeile 26 ein aus sechzig Sternen (*) bestehender String auf den Bildschirm gebracht wird.

Zeile 27 sorgt ebenso wie die Zeile 28 für die Tabellenüberschrift. Dabei dürfte es keinerlei Verständnisprobleme geben. Achten Sie dabei allerings auf die Funktion des ; am Ende der Zeile 27. Es bewirkt, daß ein folgender PRINT-Befehl (in Zeile 28) nicht mit einer neuen Zeile beginnt, sondern seine Daten in die gleiche Zeile schreibt.

In Zeile 29 kommt die String-Multiplikation und String-Addition zum Zug. Auch hier werden wieder 60 Sternchen erzeugt, diesmal jedoch nicht mit STRING$. Es wäre übrigens logischer gewesen, 30-mal den String ** zu multiplizieren als 29 mal und daran dann ** zu hängen. Aber dann hätte ich Ihnen nicht so schön demonstrieren können, daß ST-BASIC "Punkt vor Strich" verknüpft.

Zeile 30 sorgt wieder für die Ausgabe einer Leerzeile, danach werden die einzelnen Daten formatiert ausgegeben. Formatiert bedeutet in diesem Zusammenhang, daß die einzelnen Daten in die Tablelle spaltenweise unter die jeweilige Überschrift eingetragen werden. Ab Spalte 3 soll der Zuname in der Tabelle erscheinen, ab Spalte 17 der Vorname usw.

Um den Zunamen ab der Spalte 3 ausgeben zu können, werden die ersten beiden Spalten mit " " übersprungen, danach folgt der Zuname. Je nach Eingabe ist der Zuname unterschiedlich lang. Ab Spalte 17 soll der Vorname ausgegeben werden. Deshalb müssen die nicht benötigten Spalten zwischen dem Ende des Zunamens und dem Beginn des Vornamens überbrückt werden. Die Funktion Space$ ist dafür geradezu prädestiniert: Es werden 14 (Spalte 17 - Spalte 3) Leerzeichen, abzüglich der Länge des schon ausgedruckten Zunamens, ausgegeben. Die nächste Ausgabeposition ist somit unabhängig von der Länge des Zunamens die Spalte 17. Dieses Spielchen wiederholt sich für die Ausgabe des Vornamens.

Zeile 33 gibt 7 Leerzeichen aus und setzt die nächste Schreibposition direkt unter Zuname. Dort wird die Anzahl der im Zunamen enthaltenen Buchstaben ausgegeben. Diese kann u.U. durchaus zweistellig werden. Deshalb müssen wir die nicht benötigten Spalten wieder überbrücken. Doch diesmal wird es nochmals etwas komplizierter. Wir haben eine Zahl (LEN liefert ja bekanntlich als Ergebnis eine (Integer-)Zahl) auf dem Monitor ausgegeben. Von einer Zahl wiederum kann die Länge nicht mit LEN ermittelt werden, da diese Funktion einen String als Argument erwartet. Soll sie doch einen bekommen!

Wir wandeln einfach die resultierende Länge mit der Funktion STR$ in einen String um. Von diesem String können wir wiederum die Länge problemlos über LEN ermitteln. STR$ hat aber eine Eigenheit: bei positiven Werten beginnt der String mit einem Leerzeichen (Platzhalter für Vorzeichen), ist also um einen Wert größer als es die umzuwandelnde Zahl war. Doch das macht gar nichts! Wir ziehen einfach eine 1 ab, schon stimmt das Ergebnis wieder! Diese Zeile sieht komplizierter aus, als sie eigentlich ist: wir ermitteln die Länge des Zunamens, verwandeln diese Zahl in einen String und erkunden von diesem wieder die Länge, abzüglich des Wertes 1 für das vorangestellte Leerzeichen. Für alle, denen das ganze Spielchen ein wenig kompliziert erscheint, es gibt noch eine wesentlich einfachere Möglichkeit die Stellen einer Zahl zu erfahren: Über die Logarithmusfunktion. An dieser Stelle möchte ich jedoch nicht näher darauf eingehen.

### Zahlensysteme

Es gibt ungezählte Lehrbücher über Programmiersprachen, die erst einmal die verschiedenen Zahlensysteme behandeln, ehe der geplagte Leser den ersten Befehl zu Augen bekommt. Da werden verschiedene Zahlen durch die Systeme jongliert, eine Akrobatik die ich Ihnen ersparen möchte, weil ST-BASIC diese Arbeit für Sie ebensogut erledigen kann.

Das Dezimalsystem müßte Ihnen ebenso wie das Dual- oder auch Binärsystem vertraut sein. Zwei weitere Zahlensysteme, die in der Computerei eine wichtige Rolle spielen, möchte ich hier ganz kurz ansprechen:

### Das Hexadezimalsystem

Während der Mensch im Dezimalsystem rechnet, kann der Computer damit herzlich wenig anfangen. Er bevorzugt das Dualsystem, das wiederum für den Menschen unverständlich ist. Aus diesem Grund hat man das Hexadezimalsystem eingeführt, das keiner von beiden versteht.

Aber Spaß beiseite, das Hexadezimalsystem arbeitet mit 16 verschiedenen Ziffer. Für die Zahlen 0 bis 9 kann man die Ziffern des Dezimalsystems benutzen, zur Darstellung der übrigen Zahlen borgt man sich die ersten sechs Buchstaben des Alphabets (A, B, C, D, E, F). Die Zahl A des Hexadezimalsystems besitzt im Dezimalsystem den Wert 10, F den Wert 15. Damit man erkennt, daß F in diesem Fall kein Buchstabe, sondern eine Zahl im Hexadezimalsystem ist, stellt man noch das Dollarzeichen $ voran: $A.

### Das Oktalsystem

Das Oktalsystem ist das System der C-Programmierer. Es besteht aus den Ziffern 0-8 und erhält als Kennzeichen ein & vorangestellt.

### Das Dualsystem

Das Dualsystem basiert auf den beiden Ziffern 0 und 1. Aber wozu sage ich das? Das einzig Neue daran ist, daß es durch ein vorangestelltes % gekennzeichnet wird. Es existiert noch eine weitere Möglichkeit das System anzugeben:

| Präfix | für Zahlensystem |
|---|---|
| &d | Dezimalsystem |
| &h | Hexadezimalsystem |
| &b | Dual(Binär-)system |
| &o | Oktalsystem |

Nun aber in medias res, der Umwandlung von Zahlen in ein anderes System:

### HEX$

Die Funktion HEX$ wandelt einen beliebigen numerischen Wert, der als Argument angegeben werden muß, in eine Hexadezimalzahl (genauer gesagt in eine Zeichenkette) um:

```
A$= HEX$(174)
```

liefert in A$ die Zeichenkette " AE". An der ersten Stelle steht stets ein Leerzeichen.

### BIN$

Die Funktion BIN$ wandelt einen numerischen Wert in eine Binärzahl (Zeichenkette) um:

```
A$= BIN$(160)
```

ergibt die Zeichenkette " 10100000" in A$.

### OCT$

Jetzt dürfen Sie dreimal raten, welche Funktion OCT$ auslöst! Richtig, die Umwandlung eines numerischen Ausdrucks in eine Oktalzahl (Zeichenkette):

```
A$= OCT$(63)
```

weist der Variablen A$ " 77" zu.

Möchten Sie die auf diese Weise erhaltenen Zeichenketten wieder in eine Dezimalzahl wandeln, erledigt das die Funktion VAL für Sie. Als Argument muß die Zeichenkette (mit vorangestelltem Kennzeichen für das Zahlensystem!) übergeben werden:

| Ausdruck | Inhalt von A |
|---|---|
| A= VAL("&77") | 63 |
| A= VAL("$AE") | 174 |
| A= VAL("%10100000") | 160 |

Liegt in A$ beispielsweise ein Binär-String vor, kann das Präfix einfach durch String-Addition vorne angehängt werden:

```
Zahl= VAL("%"+A$)
```

Das Gleiche gilt natürlich auch für das Hexadezimal- bzw. Oktalsystem. Das Präfix muß jedoch entsprechend abgeändert werden. Beim Hexadezimalsystem in $ (Zahl= VAL("$"+A$)) bzw. in & beim Oktalsystem: Zahl= VAL("&"+B$)

Soviel zu den unterschiedlichen Zahlensystemen, die - wie Sie soeben gesehen haben - in ST-BASIC keinerlei Probleme bei der Umwandlung bereiten. In diesem Buch werde ich bei der Zahlendarstellung so wenig wie nur irgendwie möglich vom Dezimalsystem abweichen, um die Programme übersichtlicher und einfacher zu gestalten. Trotzdem der Vollständigkeit halber dieser kleine Einschub.

## 2.8 Variablenfelder

Eine weitere Spezies der Gattung Variable stellen die Variablenfelder oder Arrays dar. Dabei werden mehrere Variablen eines Typs (z.B. Integer, Float oder String) unter einem Namen zusammengefasst. Ein in Klammern stehender Index am Ende des Variablennamens gibt darüber Aufschluß, welches Element aus dem Array angesprochen werden soll.

```
A$(3)= "Ein Beispiel für einen Array"
```

Anschaulicher kann man sich Variablenfelder als eine Kommode mit mehreren Schubladen (die übereinander angeordnet sind) vorstellen. Die Schubladen sind durchnumeriert, und eine einzelne Schublade kann über eine Nummer angesprochen werden. Im obigen Beispiel würde der String "Ein Beispiel für einen Array" in die Schublade mit der Nummer drei abgelegt. Möchten Sie sich den Inhalt dieser Schublade anzeigen lassen, so erreichen Sie dies mit dem Befehl

```
PRINT A$(3)
```

Damit man in einem Programm mit Feldvariablen arbeiten kann, müssen sie zuvor dimensioniert werden. Dabei wird dem Computer mitgeteilt, wieviel Platz er für dieses Array reservieren muß.

```
DIM A$(5)
```

richtet ein Feld A$ mit 6 Dimensionen ein, da der Index von 0 bis 5 läuft, also von A$(0) bis A$(5).

Wird in ST-BASIC (und nur hier!) der DIM-Befehl vergessen, dimensioniert der Interpreter selbständig mit einem maximalen Index von 10. Das heißt im Klartext, daß alle Zuweisungs- und Ausgabe-Operationen bei undimensionierten Feldern solange einwandfrei durchgeführt werden, solange der Index den Wert 10 nicht überschreitet. Den Versuch, in einem undimensionierten Array einen höheren Index als 10 anzusprechen, ahndet der Computer mit einer Fehlermeldung.

Dennoch sollten Sie sich erst gar nicht angewöhnen, den DIM-Befehl zu vernachlässigen. Andere BASIC-Versionen sind wesentlich penibler, und dies führt dann bei einem undimensionierten Feld unweigerlich zu einem Programmabbruch mit Fehlermeldung. Aber auch mehrdimensionale Arrays sind möglich:

```
DIM A$(5,5)
```

dimensioniert ein Array mit 36 Feldern. Zur Veranschaulichung des Ganzen muß wieder die Kommode herhalten. Diesmal besitzt sie 36 Schubladen. Möchten Sie die Schublade A$(3,2) ansprechen, so gehen Sie in die dritte Spalte und zweite Reihe. A$(4,1) finden Sie dann logischerweise in der vierten Spalte und ersten Reihe.

Nach diesem Schema können Sie Arrays mit drei, vier und mehr Dimensionen bilden. Bei mehrdimensionalen Arrays darf die Größe der ersten Dimension den Wert 65535 und die aller übrigen Dimensionen zusammen den Wert 65535 nicht übersteigen. Die Gesamtgröße des Feldes erhalten Sie durch Multiplikation der einzelnen Teildimensionen. Sie brauchen sich deshalb aber keine grauen Haare wachsen zu lassen, da eine Überschreitung der erlaubten Gesamtgröße in der Praxis so gut wie nie vorkommt, wenn ein Feld vernünftig dimensioniert wird.

| Variablen-Art | Platzbedarf in Byte je Element |
|---|---|
| Integer-Flag | 1 Byte für jeweils 8 Elemente |
| Integer-Byte | 1 |
| Integer-Word | 2 |
| Integer-Long | 4 |
| Single-Float | 6 |
| Double-Float | 10 |
| String | 6 (+ Länge + 10, sobald angelegt) |

## 2.9 Programmierhilfen

In diesem Kapitel möchte ich Ihnen gerne ein paar Befehle vorstellen, die dem Programmierer die tägliche Arbeit im Umgang mit St-BASIC vereinfachen.

### Funktionstastenbelegung

ST-BASIC sieht die Möglichkeit vor, die 10 vorhandenen Funktionstasten mit Zeichenfolgen zu belegen. Die Länge einer Zeichenfolge darf dabei den Wert 32 nicht überschreiten.

Wozu dieser Aufwand? Häufig wird während der Programmierung eine bestimmte Zeichenkette immer wieder benötigt. Sie können diese natürlich jedesmal über die Tastatur eintippen, aber irgendwann wird Ihnen dies zu blöd (Verzeihung!). Schließlich haben Sie die Möglichkeit, diese Zeichenkette auf eine beliebige der zehn Funktionstasten zu legen. Ein Druck auf die entsprechende Taste genügt und wie von Geisterhand erscheint die entsprechende Zeichenfolge auf dem Bildschirm.

Um einer Funktionstaste einen bestimmten Text zuzuweisen, wird in ST-BASIC der Befehl

```
KEY
```

verwendet.

```
KEY 2="ST-BASIC Buch"
```

weist der Funktionstaste <F2> den String "ST-BASIC-Buch" zu.

Sobald eine Funktionstaste mit einem Text belegt ist, genügt ein Druck auf die betreffende Funktionstaste (hier: <F2>), um die Zeichenfolge ("ST-BASIC-Buch") auf dem Bildschirm erscheinen zu lassen. Möchten Sie auf die Funktionstaste <F1> den Befehl RUN legen, und diesen bei Tastendruck auch gleich ausführen lassen, so müssen Sie den ASCII-Code bemühen. Da ein Betätigen der <Return>-Taste stets eine Zuweisung abschließt, ist es nicht möglich, <RETURN> durch Tastendruck einer Zeichenkette zuzuweisen. Der ASCII-Code für <RETURN> ist genormt und besitzt den Wert 13:

```
KEY 1="run"+chr$(13)
```

weist der der Funktionstaste <F1> den Befehl RUN zu, und führt diesen bei einem Druck auf die Funktionstaste gleich aus: das Programm im Speicher wird gestartet. Änlich ergeht es Ihnen mit den Anführungszeichen, auch sie müssen über den ASCII-Code (34) eingegeben werden:

```
KEY3="FILES"+CHR$(34)+"A:\ST_BASIC\*.BAS"+CHR$(34)+CHR$(13)
```

Ein Betätigen der Taste <F3> listet ab sofort sämtliche auf Disk A im Ordner ST_BASIC enthaltenen BASIC-Programme auf dem Monitor auf. Weitere 10 Befehlsfolgen können auf den Funktionstasten abgelegt werden, indem Sie eine Funktionstaste in Verbindung mit <Shift> drücken:

```
KEY 13= "CLS"+CHR$(13)
```

Drücken Sie gleichzeitig <Shift> <F3>, so wird der Bildschirm gelöscht (Funtionstaste 13), während bei <F3> alleine alle BASIC-Programme auf dem Monitor aufgelistet werden. Möchten Sie sich die Belegung aller Funktionstasten einmal ansehen, kann dies über den Befehl

```
KEY LIST
```

geschehen.

### Programm listen

Der Befehl LIST dient dazu, ein Programm ganz oder teilweise auf dem Bildschirm auszugeben. Alternativ kann das Listing auch an einen Drucker geleitet werden.

| | |
|---|---|
| *LIST* | Listet das gesamte Programm. |
| *LIST -200* | Listet alle Zeilen bis zur Zeile 200. |
| *LIST 220* | Listet die Zeile 220. |
| *LIST 100-200* | Listet alle Zeilen von 100 bis 200. |
| *LIST 200-* | Listet alle Zeilen ab Zeile 200. |

Der Bindestrich kann auch durch ein Komma ersetzt werden:

| | |
|---|---|
| *LIST 100,200* | Listet alle Zeilen von 100 bis 200. |

Daneben können die Zahlen auch durch Variablen ersetzt werden:

| | |
|---|---|
| *LIST (ERL),(ERL+50)* | Listet ab der Zeile, deren Nummer in der Variablen ERL steht, bis zur Zeile mit der Nummer ERL+50. Anmerkung: Tritt ein Fehler im Programm auf, so enthält die Variable ERL (ERorr Line) die Nummer der Zeile, in der der Fehler auftrat. In Abhängigkeit dieses Wertes kann dann in eine bestimmte Fehlerroutine verzweigt werden. Aber dazu später mehr. |

Statt einer Zeilennummer können auch Marken (Labels) angegeben werden:

| | |
|---|---|
| *LIST -Marke* | Listet das Programm bis zur Zeile, indem das Label Marke definiert ist. |

In einem Listing können Sie mit den Cursortasten auch nach oben und nach unten scrollen (blättern):

Laden Sie dazu das entsprechende Programm in den Speicher des Computers und lassen Sie sich mit dem Befehl LIST-100 (der Wert hinter LIST ist beliebig) alle Zeilen bis zur Zeilennummer 100 ausgeben. Mit der Taste <Cursor nach unten> können Sie den Cursor bis in die letzte Bildschirmzeile fahren. Dort angekommen rollt (scrollt) das Listing weiter, solange Sie die Cursortaste gedrückt gehalten oder erneut betätigen. Entsprechend können Sie mit <Cursor nach oben> in die entgegengesetzte Richtung scrollen, sobald sich der Cursor in der obersten Bildschirmzeile befindet. Bevorzugen Sie die Ausgabe des Listings auf einem Drucker (besonders bei längeren Programmen oft unerläßlich), ist dies mit dem Befehl

```
LLIST
```

möglich. Die Syntax entspricht dabei voll der des LIST-Befehls. Eine Umleitung des Listings in eine Datei auf Diskette erreichen Sie mit der folgenden Zeile.

```
OPEN "O",1,"<Dateiname>": CMD 1: LLIST: CLOSE 1
```

schreibt das Listing in eine Diskettendatei mit dem Namen <Dateiname>. Möchten Sie das Programm über Modem (RS232-Schnittstelle) Ihrem Freund vermachen, geht dies folgendermaßen:

```
OPEN "V",1: CMD 1: LLIST: CLOSE1
```

### Variableninhalte anzeigen

Der Befehl DUMP dient dazu, alle Variablen, die in einem Programm vorkommen und deren Inhalte auf dem Bildschirm auszugeben. Bei Arrays wird nur die Dimensionierung, nicht deren Inhalt auf dem Monitor aufgelistet:

```
10 A$="Hallo"
20 DIM Fliess#(10,2,5),Schrott(25)
30 Dummy!= 2.18
40 END

DUMP
```

```
A$="Hallo"
DIM FLIESS#(10,2,5)
DIM SCHROTT(25)
DUMMY!= 2.18

OK
```

Möchten Sie die Variablenliste lieber als Druckerlisting betrachten, müssen Sie

```
LDUMP
```

eingeben. Auch hier kann die Ausgabe wieder auf die Diskette umgeleitet werden:

```
OPEN "O",1,"<Dateiname>": CMD 1: LDUMP: CLOSE 1
```

Der Inhalt sämtlicher Variablen wird übrigens mit dem Befehl

```
CLEAR
```

gelöscht.

### Trace (Programmüberwachung)

Möchten Sie wissen, in welcher Reihenfolge der Computer das Programm abarbeitet, müssen Sie TRACE einschalten. Dies geht mit dem Befehl

```
TRON (Trace on)
```

Wird das Programm gestartet, so druckt der Computer vor Ausführung eines Befehls die Zeilennummer und den gerade bearbeiteten Befehl in Klammern auf dem Bildschirm aus. Auf diese Weise kann der Programmablauf einfach überwacht werden. Dazu ein Beispiel:

```
10 A$="Was weiß ich"
20 B%=30
30 X%=X%+1
40 PRINT A$
50 END

TRON: RUN

[10 LET] [20 LET] [30 LET] [40 PRINT] Was weiß ich
[50 END]
```

Möchten Sie TRACE wieder abschalten, so brauchen Sie nur

```
TROFF (TRACE OFF)
```

eingeben. Werden TRON und TROFF nicht im Direktmodus eingegeben, sondern innerhalb des Programms ein- und wieder ausgeschaltet, so verwaltet ST-BASIC die beiden Befehle in Ebenen: bei einem viermaligen Einschalten von TRACE sind vier TROFF-Befehle nötig, um den Trace-Modus wieder zu deaktivieren. Im Direktmodus werden dagegen mit TRON sämtliche Ebenen gelöscht und der Trace-Modus auf alle Fälle eingeschaltet. Möchten Sie noch mehr Kontrolle über Ihr Programm gewinnen, so kann

```
ON TRON GOSUB <Ziel>
```

weiterhelfen. Dabei wird nach jedem ausgeführten Befehl zu der in Ziel angegebenen Zeilennummer gesprungen und die dort stehenden Anweisungen ausgeführt, bis der Interpreter durch RETURN wieder zum Rücksprung aufgefordert wird. Der nächste Befehl wird abgearbeitet und wieder zu <Ziel> gesprungen. Somit können bestimmte Variablen im Programm überwacht werden. Je mehr Sie in der Unterroutine überprüfen, desto langsamen wird die Ausführungsgeschwindigkeit Ihres Programmes sein. Versuchen Sie sich deshalb kurz zu fassen. In der Variablen ERL steht die Zeilennummer, von der aus der Interpreter in die Unterroutine gesprungen ist, in ERR$ der zuletzt ausgeführte Befehl.

Sollten Sie obige Ausführungen nicht ganz verstanden haben, muß ich Sie wieder vertrösten. Zum Verständnis von ON TRON GOSUB werden Kenntnisse in der Programmierung von Unterroutinen benötigt, die Sie (noch) nicht besitzen. Trotzdem hoffe ich, daß die Funktionsweise von ON TRON GOSUB wenigstens einigermaßen klar geworden ist.

### RENUMBER

Wenn Sie mit Zeilennummern programmieren, kommt es immer wieder vor, daß die eine oder andere Zeile nachträglich eingefügt werden muß, andere Zeilen erübrigen sich und werden gelöscht. Ebenso kann es passieren, daß Sie zwischen zwei Zeilen eine neue Zeile einfügen möchten, aber keine Zeilennummer mehr dazwischenpaßt, da beide Zeilen bereits in Einerzeilenabständen aufeinander folgen.

Für diese Fälle hält ST-BASIC eine Funktion parat, mit deren Hilfe die Zeilen neu durchnumeriert werden können: RENUM. Als Parameter hinter RENUM kann die Zeilennummer angegeben werden, mit der das neue Programm, sowie die erste Zeile, bei der die Umnummerierung beginnen soll. Dritter und letztmöglicher Parameter ist dann die Schrittweite zwischen zwei Zeilennummern, die üblicherweise 10 beträgt.

Beispiele:

| | |
|---|---|
| *RENUM 100,50,10* | Numeriert das Programm ab Zeile 50 um. Zeile 50 wird zur neuen Zeilennummer 100, die darauffolgenden Zeilen werden in 10er-Schritten durchnumeriert, also 110, 120 usw. |
| *RENUM 10* | Numeriert das Programm in Zehnerschritten durch. Die erste Zeile im Programm erhält die Zeilennummer 10. |
| *RENUM* | Ohne Parameter numeriert das Programm in Zehnerschritten durch, die erste Zeile erhält die Nummer 100. RENUM ohne Parameter ist identisch mit RENUM 100. |

## 2.10 Strukturierte Programmierung

Um es gleich vorwegzunehmen, hinter dem Zauberwort Strukturierte Programmierung verbirgt sich nichts anderes, als der sauber gegliederte Aufbau (Struktur) eines Programms. Dem Anwender, der mit der Software arbeiten muß, wird es in der Regel herzlich egal sein, in wieweit das Programm strukturiert aufgebaut ist. Für ihn ist in erster Linie wichtig, daß die Software einwandfrei arbeitet.

Der engagierte Programmierer denkt darüber allerdings etwas anders: Er möchte sich auch noch zu einem späteren Zeitpunkt in seinem Programm zurechtfinden. Der berühmt-berüchtigte "Spaghetti-Code" früherer BASIC-Interpreter erschwerte dies in nicht unerheblicher Weise. Kreuz und quer wurde durch ein Programm gesprungen, das sich ob seiner mit möglichst vielen Befehlen vollgepackten Programmzeilen sowieso schon relativ schwer entschlüsseln ließ. Irgendwann bleibt dann bei einer derartigen Programmierweise die Übersicht auf der Strecke, das Programm wird zu einem Buch mit sieben Siegeln.

Um dies zu verhindern, bieten moderne Programmiersprachen die Möglichkeit zur Strukturierung eines Programms. Auch ST-BASIC bildet hier - für einen BASIC-Interpreter früherer Jahre ein Unding - keine Ausnahme. Das Konzept der strukturierten Programmierung ist so einfach wie genial: Eine gegebene Problemstellung wird in viele kleine Einzelprobleme zerlegt, Schritt für Schritt verfeinert und dabei auf elementare Befehle zurückgeführt. Ein Beispiel verdeutlicht dies wohl am besten:

Jeder von Ihnen kennt zweifelslos das Spiel "Mastermind", bei dem eine bestimmte Farb- oder Zahlenkombination erraten werden muß. Ein Spieler denkt sich eine solche Kombination aus, der andere muß sie durch geschicktes Kombinieren herausfinden. Dazu steckt er die vermutete Kombination auf das Spielbrett, der Partner überprüft seine Kombination mit der des Mitspielers. Stimmen zwei Farben (Zahlen) an einer Position überein, muß er dies anzeigen. Auch eine Farbe, die zwar in der Kombination seines Mitspielers auftaucht, jedoch an der falschen Stelle, muß angegeben werden. Auf dieser Basis wagt

der Spieler dann seinen zweiten Versuch. Dies wiederholt sich solange, bis entweder das Spielfeld voll (Kombination wurde nicht erraten) ist oder die Kombination herausgefunden wurde. Eine erste Zerlegung in Teilprobleme könnte etwa so aussehen:

1. Farbkombination ausdenken,
2. vermutete Kombination auf das Spielbrett stecken,
3. vermutete mit richtiger Kombination vergleichen,
4. wenn Kombination noch nicht korrekt erraten, dann mit Schritt 2 fortfahren.

Schritt 3 läßt sich noch weiter aufspalten:

3.1 Erste Position beider Kombinationen vergleichen,
3.2 wenn beide in der Farbe übereinstimmen,
3.3 dann anzeigen, daß Farben in Position und Farbe übereinstimmen,
3.4 sonst prüfen, ob diese Farbe wenigstens an einer anderen Position vorkommt.
3.5 Wenn dies der Fall ist,
3.6 dann anzeigen, daß Farbe korrekt erraten, jedoch an die falsche Position gesteckt wurde,
3.5 wenn schon alle Positionen verglichen,
3.6 dann mit Schritt 4 weiter,
3.7 sonst nächste Position vergleichen, weiter mit 3.2.

Das Gesamtproblem Mastermind wurde in einzelne Teilprobleme aufgespalten. Natürlich können diese Teilschritte noch verfeinert werden, aber dies soll uns an dieser Stelle nicht weiter interessieren.

### Wenn..dann..sonst...

Bei der Verfeinerung des obigen Problems sind wir bereits auf einen elementaren Befehl zur strukturierten Programmierung gestoßen: Die "Wenn..dann..sonst.."-Bedingung.

```
wenn beide in der Farbe übereinstimmen
      dann anzeigen, daß Farben in Position und Farbe
         übereinstimmen.
      sonst prüfen, ob ....
```

Diese "Wenn..dann..sonst.."-Bedingung wird in BASIC mit den Befehlen

```
if (wenn)
..... (Bedingung erfüllt)
then (dann)
..... (Anweisung(en) ausführen)
else (sonst)
..... (Anweisung(en) ausführen)
```

ausgedrückt.

Zur Konstruktion: Hinter dem Befehl IF folgt eine Bedingung (beide Farben stimmen überein). Ist diese Bedingung erfüllt (beide Farben stimmen überein), werden die Anweisungen hinter THEN (anzeigen, daß Farben ...) ausgeführt. Ist die Bedingung dagegen nicht erfüllt (die Farben stimmen nicht überein), gelangen die Anweisungen hinter ELSE zur Ausführung (prüfen, ob ...).

Wie aber kann der Computer prüfen, ob eine Bedingung erfüllt ist? Ganz einfach, nämlich durch einen simplen Vergleich. In unserem obigen Beispiel müßte man die Bedingung wie folgt formulieren:

```
if <Farbe_1 gleich Farbe_2>
   then ...
```

bzw. in syntaktisch richtiger Form:

```
if Farbe_1 = Farbe_2
   then ...
```

Der Operator = ist uns in seiner Funktion als Zuweisungsoperator (A$= "Hallo") schon bekannt, neu ist dagegen sein zweites Einsatzgebiet als Vergleichsoperator. Stellen Sie sich Farbe_1 und Farbe_2 als zwei Variablen vor, in denen die Farben ge-

speichert werden. Stimmen die Inhalte der beiden Variablen überein (sind also beide Farben identisch), so wird als Ergebnis der Operation der Wahrheitswert wahr geliefert, andernfalls falsch.

Diesen Wahrheitswert macht sich nun die IF-Konstruktion zu nutze, und führt bei wahr die Bedingung hinter THEN, ansonsten die Bedingung hinter ELSE aus. Der Vergleichsoperator = liefert also einen Wahrheitswert. Wahrheitswert deshalb, weil die beiden Möglichkeiten der Vergleichsoperation, im Computer wieder (wie sollte es auch anders sein) durch Zahlen dargestellt werden.

Die Zahl -1 steht dabei für den Wahrheitswert wahr, während 0 den Status falsch repräsentiert. Und hier wären wir wieder bei den Flags, die derartige Wahrheitswerte speichern können (erinnern Sie sich noch dunkel an Kapitel 2?). Einem Flag können nämlich lediglich die beiden Werte 0 (falsch) und -1 (wahr) zugewiesen werden. So ist es jederzeit möglich einem Flag den Wahrheitswert einer Vergleichsopertion zuzuweisen und diesen Wert später einer IF-Abfrage unterzujubeln! Das Ergebnis bleibt in beiden Fällen gleich:

```
100 DIM Flag%F(7)' nur keinen Speicherplatz vergeuden
110 Zahl_1%= 10
120 Zahl_2%= 20
130 Zahl_3%= 10
140 Flag%F(0)= Zahl_1%= Zahl_2%
150 Flag%F(1)= Zahl_3%= Zahl_1%
160 '
170 IF Flag%F(0) THEN PRINT "Zahl 1 gleich Zahl 2"
180 IF Flag%F(1) THEN PRINT "Zahl 1 gleich Zahl 3"
190 '
200 END
```

Sowohl in Zeile 140 als auch in Zeile 150 wird der Inhalt zweier Variablen miteinander verglichen. Das Ergebnis dieser Operationen wird jeweils einer als Flag deklarierten Variable (Flag%F(), diesen billigen Kalauer konnte ich mir einfach nicht verkneifen) zugewiesen. Die beiden IF der Zeilen 170 und 180 benutzen nun den Inhalt von Flag%F() als Wahrheitswert, um in Abhängigkeit

davon die hinter den beiden THEN stehenden Anweisungen auszuführen. Ebenso hätte man den Vergleich hinter der IF-Abfrage durchführen können:

```
...
170 IF Zahl_1%= Zahl_2% THEN PRINT "Zahl 1 gleich Zahl 2"
180 IF Zahl_3%= Zahl_1% THEN PRINT "Zahl 1 gleich Zahl 3"
190 ...
```

Da in Flags nur die beiden Wahrheitswerte 0 und -1 gespeichert werden können, kann dies vom Cumputer mit einem einzigen Bit erledigt werden. Ein Bit wiederum befindet sich nicht irgendwo im Computer, sondern ein aus 8 Bits bestehendes Bündel wird sauber zu einem Byte zusammengefaßt. Mit 1 Byte können folglich 8 solcher Flags realisiert werden. Dies ist der Grund dafür, warum Flags in ST-BASIC lediglich in Array-Notation auftauchen und 8 Flags jeweils genau ein Byte beanspruchen!

Doch zurück zur IF-Abfrage. Die in der gleichen Zeile hinter THEN stehende(n) Anweisung(en) werden nur dann ausgeführt, wenn das Ergebnis der Vergleichsoperation zur Aussage wahr geführt hat. ST-BASIC erlaubt noch eine andere Variante der IF..THEN..ELSE..-Struktur, wobei die einzelnen Befehle übere mehrere Zeilen verteilt werden können.

Dies bringt eine gewaltige Steigerung der Übersichtlichkeit mit sich, besonders bei längeren Programmen wirkt sich erfahrungsgemäß eine Verteilung der Struktur auf mehrere Zeilen für die Lesbarkeit positiv aus. Da die einzelnen hinter THEN stehenden Anweisungen auf mehrere Zeilen verteilt sind, muß dem Computer dann zusätzlich mitgeteilt werden, wann das Ende der Struktur erreicht ist. Dies geschieht mit dem Befehl ENDIF:

```
IF <Bedingung 1>
   THEN <Anweisung 1>
        <Anweisung 2>
        <Anweisung 3>
   ELSE
        IF <Bedingung 2>
           THEN <Anweisung 4>
```

```
                <Anweisung 5>
        ENDIF
ENDIF
```

Zwei IF-Strukturen sind ineinander verschachtelt. Schon optisch läßt sich die Abhängigkeit einer Anweisung von dem zugehörigen IF auf den ersten Blick erkennen.

Ist <Bedingung 1> erfüllt, gelangen die Anweisungen 1 bis 3 zur Ausführung, andernfalls die hinter dem ELSE stehenden Anweisungen. Doch dort findet sich eine zweite IF-Struktur. Diese kommt nur dann zur Geltung, wenn die Bedingung 1 nicht erfüllt ist. Ist die Bedingung 1 nicht, die Bedingung 2 jedoch erfüllt, kommen die Anweisungen 4 und 5 zum Zug.

Die zweite IF-Struktur stellt keine Alternativanweisungen in Form eines ELSE zur Verfügung. Auch das ist möglich, da nicht in jedem Fall ein ELSE benötigt wird. Interessant ist noch die Stellung des ENDIF. Die zweite IF-Struktur muß nämlich vor der ersten wieder geschlossen werden. Sonst klappt die ganze Verschachtelei nicht.

Der Übersichtlichkeit halber steht das ENDIF stets in der gleichen Spalte unter dem IF, das es abschließt. Späterstens jetzt müßte der Begriff der strukturierten Programmierung einleuchten. Mit dem IF..THEN..ELSE..ENDIF kann einem Programm nämlich eine richtige Struktur (Form) verpasstwerden. Und damit Sie den Vorteil der Strukturierung auch gleich sehen, hier einmal die unstrukturierte Form des obigen Beispiels (Bedingung mit B, Anweisung mit A abgekürzt):

```
IF <B1> THEN <A1>:<A2>:<A3> ELSE IF <B2> THEN <A4>:<A5>
```

Am Ergebnis ändert sich nichts, wohl aber ist die Übersicht irgendwo hinter dem ersten THEN auf der Strecke geblieben. Stellen Sie sich einmal vor, die entsprechenden Bedingungen bzw. Anweisungen stehen in Form von Befehlen in dieser Zeile. Unübersichtlicher geht's kaum noch und daher hat BASIC auch seinen Ruf als Programmiersprache mit "Spaghetti-Code".

Haben Sie besonders viele IF-Abfragen ineinander verschachtelt, rücken die hinter jedem neuen IF stehenden Anweisungen immer weiter nach rechts herein. Als platzsparende Alternative können Sie deshalb noch etwas anders strukturieren, indem Sie das THEN einfach in die gleiche Zeile wie das IF setzen:

```
IF <Bedingung 1> THEN
    <Anweisung 1>
    <Anweisung 2>
    <Anweisung 3>
ELSE
    IF <Bedingung 2> THEN
        <Anweisung 4>
        <Anweisung 5>
    ENDIF
ENDIF
```

Obwohl das Programm nichts an seiner Übersichtlichkeit verloren hat, wird mit jedem neuen IF nicht so weit nach rechts eingerückt, wie dies bei dem ersten Beispiel der Fall gewesen ist. Gerade bei komplexeren Strukturen birgt diese Methode einen Vorteil in sich, da der Bildschirmbereich besser ausgenutzt wird (die ganze Struktur rutscht nicht so schnell nach rechts und kann eher komplett auf dem Monitor dargestellt werden).

Als Vergleichsoperator mußte bisher das Gleichheitszeichen herhalten. Dies ist jedoch nicht für alle auftretenden Eventualitäten ausreichend. Oft muß geprüft werden, ob eine Zahl größer oder kleiner als eine andere ist, bzw. ob sich zwei Zahlen voneinander unterscheiden. Deshalb existieren noch ein paar weitere Vergleichsoperatoren, die diese Fälle abfangen. Allen gemeinsam ist die Tatsache, daß als Ergebnis ihrer Ermittlungen stets der Wert 0 für falsch und -1 für wahr geliefert wird:

| Vergleichsoperator | Bedeutung des Operators |
|---|---|
| = | gleich |
| < | kleiner |
| > | größer |
| <= | kleiner (oder) gleich |
| >= | größer (oder) gleich |
| <> | ungleich |

Die Operatoren <=, >=, und <> können auch in umgekehrter Reihenfolge angegeben werden:

| Vergleichsoperator | Bedeutung des Operators |
|---|---|
| => | größer (oder) gleich |
| =< | kleiner (oder) gleich |
| >< | ungleich |

Der leichteren Lesbarkeit eines Programms zuliebe sollten die Operatoren in der Reihenfolge benutzt werden, in der sie auch gesprochen werden (größer gleich).

Ein paar Beispiele:

```
IF A%  > B% THEN        ...wenn Inhalt von A% größer als B%
IF A% <> B% THEN        ...wenn Inhalt von A% ungleich B%
IF A$ <= B$ THEN        ...wenn A$ kleiner gleich B$
```

Selbstverständlich können auch Zeichenketten miteinander verglichen werden, da der Computer bei einem Vergleich auf die ASCII-Tabelle zurückgreift, in der den Buchstaben unterschiedliche Zahlen zugeordnet sind. (Sie erinnern sich ...)

Demzufolge ist der String "Hallo" kleiner als die Zeichenkette "Vanillepudding". Im Normalfall werden solche Vergleiche (abgesehen von dem Gleichheitsoperator =) aber nur für Sortierroutinen verwendet, und auch hier glänzt das ST-BASIC durch eine schon in den Interpreter integrierte Routine. Wir als Programmierer müssen uns folglich nicht mehr mit dem Sortieren herumschlagen, der Computer erledigt dies für uns mit nur einem Befehl sozusagen im Handumdrehen.

## GOTO

Der GOTO-Befehl ist der ärgste Feind der strukturierten Programmierung, da er den vorgegebenen Programmfluß von oben nach unten vollkommen durcheinanderbringt und den Computer

zwingt, an einer anderen Programmstelle fortzufahren. Trotzdem - oder vielleicht auch gerade deshalb - soll er in diesem Kapitel seinen Platz finden.

Die Syntax des GOTO ist einfach:

```
GOTO <Ziel>
```

<Ziel> ist im einfachsten Fall eine Zeilennummer:

```
10 PRINT "Hallo"
20 GOTO 10
30 END
```

Wenn Sie das Programm abtippen (machen Sie es lieber nicht), so druckt der Computer ständig das Wort Hallo auf den Bildschirm. Er wird nie damit aufhören! Warum das?

In Zeile 10 findet er die Aufforderung, das Wort Hallo auf dem Monitor darzustellen. Folgsam wie ein Computer nun einmal ist kommt er diesem Befehl auch prompt nach. In der nächsten Zeile findet er sich mit dem GOTO konfrontiert. Er muß sozusagen einen Schritt zurück, zur Zeile 10. Dort darf er wieder ein freundliches Hallo auf den Bildschirm bringen. Ehe er sich recht versieht, gelangt er über Zeile 20 schon wieder zur Zeile 10, und was macht er jetzt? Er stellt zum dritten Mal das Wörtchen Hallo auf dem Bildschirm dar. Dieses Spielchen wiederholt sich solange, bis Sie ihm entweder mit <Comtrol> <C> ein Ende bereiten (herzlichen Dank vom Computer) oder mittels des Netzschalters den Saft abdrehen.

ST-BASIC wäre jedoch nicht ST-BASIC, könnte es nicht noch ein paar zusätzliche Trümpfe ausspielen (die Zeilennummer kann nämlich auch berechnet werden):

```
GOTO 100+10*Zeile
```

springt je nach Inhalt der Variablen Zeile in eine bestimmte Programmzeile. Möchten Sie nur eine einzelne Variable angeben, muß diese nach einem GOTO in Klammern stehen (Begründung kommt gleich!):

```
GOTO (Zeile)
```

setzt die Programmausführung in der Zeile Zeile fort. Zeilennummern sind ganz ganz lieb und recht, aber was tun, wenn in einem Programm keine Zeilennummern vorkommen? Dafür können dann bestimmte Sprungstellen (Marken) vereinbart werden. Damit der Computer nicht versehentlich eine solche Marke (Label) mit einer Variablen verwechselt (und dabei in die ewigen Jagdgründe verschwindet, weil der Variablentyp den Sprung nicht zuließ) muß - gleichsam als Zeichen ihres Standes - ein Minuszeichen vor eine Marke gesetzt werden. Es versteht sich von selbst, daß nicht zwei gleiche Labels in einem Programm auftauchen dürfen, damit der Computer auch immer schön den Überblick behält (selbst dann noch, wenn Sie ihn schon längst verloren haben).

```
-Marke
```

definiert eine solche Sprungstelle (Marke) in einem Programm. Ein

```
GOTO Marke
```

bewirkt nun, daß die Programmausführung direkt hinter dem Label Marke fortgesetzt wird. Es hat sich als sehr sinnvoll herausgestellt, Marken statt Zeilennummern für Sprünge (wenn sie denn schon unbedingt sein müssen) zu verwenden. Durch sie erhöht sich die Lesbarkeit eines Programmes erheblich und eine Programmänderung (auch nach längerer Zeit) kann leichter durchgeführt werden. Wenn Sie die Marken auch noch sinnvoll taufen, kann fast nichts mehr schiefgehen.

Marken können übrigens auch mitten in einer Zeile definiert werden, müssen dann allerdings durch Doppelpunkte von den übrigen Befehlen abgetrennt werden:

```
100 PRINT"Grüß Gott": -Schleife: INPUT"Ihr Codewort: ";A$
110 IF A$ <> "ST-BASIC"
120    THEN GOTO Schleife
130 ENDIF
140 END
```

Nach der freundlichen Begrüßung durch den Computer (Grüß Gott), wird der Benutzer aufgefordert, sein Codewort einzugeben. Dieses Codewort wird der Variablen A$ zugewiesen. In Zeile 110 folgt ein Vergleich des eingegebenen Codewortes mit dem gespeicherten Code. Stimmen beide nicht überein, wird zur Marke Schleife gesprungen. Eine erneute und diesmal hoffentlich korrekte Eingabe des Geheimcodes ist von Nöten. Erst wenn der Code richtig eingegeben wurde, gibt der Computer das System zur weiteren Nutzung frei. Die Marke hinter dem GOTO kann auch in einer Stringvariablen abgelegt werden:

```
A$="Schleife": GOTO A$
```

Auch wenn diese Variante nicht unbindingt durch rasende Geschwindigkeit glänzt (sie ist vielmehr die langsamste Möglichkeit einen Sprungbefehl auszuführen) kann sie bisweilen von Nutzen sein.

## ON..GOTO..

Eng mit dem GOTO-befehl verwandt ist ON..GOTO... Die Syntax dafür lautet:

```
ON <Wert> GOTO <Ziel 1>,<Ziel 2>,<Ziel 3>, ...
```

<Wert> wird durch eine Variable repräsentiert. Ist der Wert 1, so erfolgt ein Sprung zu <Ziel 1>, bei 2 zu <Ziel 2>. Bei einem <Wert> von kleiner als Null wird kein Sprung durchgeführt. Auch Werte, deren Größe die Anzahl der hinter dem GOTO stehenden Ziele übersteigt, lassen den Computer kalt. Es wird einfach kein Sprung ausgeführt. Sie ahnen es sicher schon, daß nicht nur Zeilennummern als Ziele angegeben werden können: Labels, Berechnungen, ja sogar Stringvariablen, die Labelnamen beinhalten, können bunt gemischt werden:

```
ON Irgendwas GOTO 100,Ende,Weiß_der_Teufel_wohin$
```

Dieser ON..GOTO..- Befehl schafft gerade in Verbingung mit Labels Übersicht. Dies ist auch der Grund, warum er von Programmieren bei der Menü-Auswahl bevorzugt wird.

### Schleifen in allen Variationen

Einer der größten Vorteile, mit denen ein Computer aufwarten kann, ist das ständige Wiederholen bestimmter Anweisungen. Diese Wiederholungen werden in der Informatik über Schleifen realisiert. Eine Primitivschleife kann bereits mit einem GOTO-Befehl erzeugt werden:

```
100 PRINT"Grüß GOTT"
110 -Schleife: INPUT"Bitte geben Sie Ihr Passwort ein: ";Pa$
120 IF Pa$ <> "ST-BASIC" THEN
130    GOTO Schleife
140 ELSE
150    CLS
160    PRINT"Sie haben sich korrekt identifiziert!"
170 ENDIF
180 '
190 ' Ab hier folgt dann das eigentliche Programm ...
200 '
```

Wird das Passwort nicht korrekt eingegeben, so springt der Computer von Zeile 130 zu Zeile 110. In diesem Bereich liegt eine Schleife vor, auch wenn in der Computerei Schleifen normalerweise über eigene Strukturen realisiert werden.

### FOR..NEXT

Wie könnte es auch anders sein. Die in allen BASIC-Interpretern (selbst in denen der Computersteinzeit) auftauchende FOR..NEXT-Schleife kommt zuerst an die Reihe. Diesmal jedoch nicht mit grauer Theorie, sondern gleich direkt an einem praktischen Beispiel:

```
0 '*****************************************************************
1 '*                          MITTEL.BAS                           *
2 '*---------------------------------------------------------------*
3 '* Autor: Michael Maier    Version: 1.00     Datum: 29.08.1988 *
4 '*      Ein Programm aus dem 'GROSSEN ST-BASIC-BUCH'             *
5 '*        (C) 1988 by DATA BECKER GmbH Düsseldorf                *
6 '*****************************************************************
7 '
8 'Zuerst Anzahl der Werte für die Dimensionierung erfragen
9 '
```

```
10 CLS
11 INPUT " Anzahl der Werte: ";Zahl%
12 '
13 'Eingabe korrekt?
14 '
15 IF Zahl%<=0 THEN
16        PRINT " Was soll der Quatsch?"
17        END
18 ELSE
19        DIM Xi!(Zahl%-1)'  Dimensionierung von 0 bis ...
20 ENDIF
21 '
22 'Einlesen der einzelnen Werte
23 '
24 FOR T%=1 TO Zahl%
25        INPUT " Beobachtungswert "; STR$(T%);": ";Xi!(T%-1)
26 NEXT T%
27 '
28 'Addition der einzelnen Beobachtungswerte
29 '
30 Summe!=0
31 FOR T%=0 TO Zahl%-1
32        Summe!=Summe!+Xi!(T%)
33 NEXT T%
34 '
35 'und Division dieser durch die Anzahl der Werte
36 '
37 Summe!=Summe!/Zahl%
38 '
39 PRINT " Das arithmetische Mittel beträgt:";Summe!
40 END
```

Das obige Programm berechnet den Mittelwert (arithmetische Mittel) für eine bestimmte Anzahl von Werten nach der Formel:

$$\text{Mittelwert} = \frac{1}{n} * \sum_{i=1}^{n} x(i)$$

Es wird über die einzelnen x-Werte aufsummiert und das Ergebnis dieser Addition durch die Anzahl der Werte dividiert. Was liegt näher als die einzelnen x-Werte in einem Array abzulegen? Damit nicht unnötig hoch dimensioniert wird, fragt das

Programm in Zeile 11 nach der Anzahl der Werte, die eingegeben werden sollen und weist diese der Variablen Zahl% (entspricht dem n in der Formel) zu.

Ehe in Zeile 19 der eigentliche DIM-Befehl folgt, wird noch mit einer IF-Abfrage kontrolliert, ob die eingegebene Zahl nicht kleiner (negative Dimensionierung ist nicht möglich!) oder gleich 0 ist. In diesem Fall bricht das Programm mit der Fehlermeldung "Was soll der Quatsch" ab. Ansonsten (ELSE) wird ein Fließkomma-Array mit einfacher Genauigkeit (Xi!) dimensioniert. Der Index einer Feldvariablen läuft bekanntlich von 0 bis ... Aus diesem Grund muß die Anzahl der Dimensionen noch um eins zurückgeschraubt werden, um nicht unnötigerweise Speicherplatz zu vergeuden.

Jetzt zur eigentlichen Neuerung in diesem kleinen Programm: Um nicht für jeden Eingabewert eine eigene INPUT-Anweisung einfügen zu müssen, werden die Werte in einer Schleife mit nur einem einzigen INPUT-Befehl (Zeile 25) eingelesen.

Die Funktionsweise der FOR..NEXT-Schleife im einzelnen: zuerst wird ein Schleifenzähler (die Variable T%) auf den Anfangswert (T%=1) gesetzt. Danach wird die Schleife durchlaufen, bis der Interpreter auf den Befehl NEXT stößt. Er kehrt zur Ausgangsposition (FOR T% = 1 TO Zahl%) zurück, erhöht den Schleifenzähler um eins und überprüft, ob der Endwert der Schleife (steht hinter TO, hier: Zahl%) schon erreicht ist. Ist dies der Fall, wird die Programmausführung hinter der Schleife (NEXT) fortgesetzt, ansonsten wird die Schleife ein weiteres Mal durchlaufen:

```
FOR <Schleifenzähler> = <Anfangswert> TO <Endwert>
....
....
....
NEXT <Schleifenzähler>
```

Da im Programm genau Zahl%-Werte in das Feld eingetragen werden müssen, läuft unsere FOR..NEXT-Schleife logischerweise von 1 bis Zahl%. Den Schleifenzähler verwenden wir als Indexvariable in Xi! (Zeile 25) für die Zuweisung der einzelnen

Werte in das Array. Doch aufgepaßt! Während der Zähler von 1 bis Zahl% läuft, beginnt der Index der Feldvariable bei 0 und endet bei Zahl%-1. Deshalb muß der Indexwert um eins erniedrigt werden (T%-1). Statt dessen hätte man den Schleifenzähler von 0 bis Zahl%-1 laufen lassen können:

```
FOR T%=0 TO Zahl%-1
    <Input>
NEXT T%
```

Warum ich das nicht gemacht habe? Ja, da ist noch eine Kleinigkeit: Der Benutzer soll nämlich gleichzeitig angezeigt bekommen, welchen Beobachtungswert er gerade eingibt. Auch dazu verwendet das Programm wieder den Schleifenzähler T%, hier jedoch mit den Werten von 1 bis ...

Liefe die Schleife von 0 bis Zahl%-1 müßte zur Nummer des jeweiligen Beobachtungswertes noch eine 1 addiert werden, um die der Schleifenzähler ja zuerst verringert wurde. Dies bedeutet zweimaligen Rechenaufwand für den Computer, der auf diese Weise elegent umgangen werden konnte.

Interessant ist noch das STR$(T%) in Zeile 25. T% enthält jeweils den Wert des momentanen Schleifendurchlaufes. Damit nun die Eingabe statt dem Feld Xi!() nicht versehentlich der Variablen T% zugewiesen wird (T% soll ja lediglich als Zähler mit ausgegeben werden), muß T% mit der Funktion STR$() in einen String umgewandelt werden. Dies macht ST-BASIC übrigens automatisch, so daß auch bei einer Eingabe von T% stets STR$(T%) erscheint.

Nach dem Einlesen der einzelnen Werte, müssen diese noch aufsummiert werden. Dies geschieht (wie könnte es auch anders sein) wieder in einer Schleife, diesmal jedoch mit einem Startwert von 0 und einem Endwert von Zahl%-1. Der Bequemlichkeit halber lautet der Schleifenzähler ebenfalls T%. Nach Beendigung der Schleife enthält die Variable Summe! die Summe der einzelnen Beobachtungswerte. Diese muß schließlich noch durch die Anzahl der Beobachtungswerte dividiert werden, um den Mittelwert zu erhalten. Fertig! Starten Sie das Programm, so erscheint auf dem Bildschirm:

```
Anzahl der Werte: 3 (Eingabe)
Beobachtungswert  1: 2
Beobachtungswert  2: 6
Beobachtungswert  3: 4
Das arithmetische Mittel beträgt: 4

OK
```

Möchten Sie die Schleife mit einer anderen Schrittweite als 1 durchlaufen, muß dies über den Befehl STEP hinter der Endbedingung explizit angegeben werden:

```
FOR T% = 0 TO 20 STEP 2
....
....
NEXT T%
```

Hier wird der Schleifenzähler nach jedem Durchlauf um den Wert 2 erhöht, Schrittweiten von kleiner 1 sind ebenso möglich (z.B: 0.5):

```
FOR T% = 1 TO 10 STEP .5
....
....
NEXT T%
```

Der Schleifenzähler kann nach jedem Durchlauf auch erniedrigt werden, wenn eine negative Schrittweite hinter STEP angegeben wird:

```
FOR T% = 20 TO 1 STEP -1
....
....
NEXT T%
```

Der Übersicht zuliebe wird die FOR..NEXT-Struktur wieder auf mehrere Zeilen verteilt, wobei der Schleifeninhalt nach rechts eingerückt wird. Theoretisch ist auch eine einzeilige FOR..NEXT-Schleife denkbar:

```
FOR T%=1 TO Zahl%: PRINT T%: NEXT T%
```

**Anmerkung:** Ist die Startbedingung bereits vor dem ersten Durchlauf größer als die Endbedingung und keine negative Schrittweite angegeben, so wird die Schleife samt Inhalt kein einziges Mal ausgeführt.

Man spricht in diesem Zusammenhang auch von einer abweisenden Schleife, da sie nur dann zur Ausführung gelangt, wenn die Startbedingung erfüllt ist. Andere BASIC-Interpreter (darunter das GfA-BASIC) besitzen eine nicht abweisende For..Next-Schleife, die in jedem Fall mindestens einmal durchlaufen wird. Bei der Übertragung von Programmen eines anderen BASIC-Interpreters in ST-BASIC sollte man diese Tatsache stets im Hinterkopf behalten, um nicht eine böse Überraschung zu erleben.

### Repeat..Until

Eine weitere Schleifenstruktur bildet die REPEAT..UNTIL-Schleife oder zu deutsch: Wiederhole ... Bis-Schleife. Es handelt sich bei dieser Variante wieder um die Gattung der nichtabweisenden Schleifen, da ihr Inhalt auf jeden Fall einmal durchlaufen wird, ehe die eigentliche Schleifenbedingung hinter UNTIL geprüft wird.

```
REPEAT
   <Anweisung 1>
   <Anweisung 2>
   ....
   <Anweisung n>
UNTIL <Bedingung erfüllt>
```

Eine FOR..NEXT-Schleiße kann durch eine REPEAT..UNTIL-Schleife ersetzt werden:

```
FOR T%= 1 to 30
    <Anweisung 1>
    <Anweisung 2>
    ....
    <Anweisung n>
NEXT T%
```

entspricht der REPEAT..UNTIL-Schleife:

```
T%=0
REPEAT
    T%=T%+1' Schleifenzähler mit 'STEP 1'
    <Anweisung 1>
    <Anweisung 2>
    ....
    <Anweisung n>
UNTIL T%=30
```

Dies gilt jedoch nur solange die Startbedingung für die FOR..NEXT-Schleife erfüllt ist. Andernfalls unterscheiden sich beide Schleifen, da FOR..NEXT nicht, REPEAT..UNTIL jedoch einmal durchlaufen wird. Nötigenfalls muß mit einer IF..THEN-Abfrage am Schleifenanfang die Einsprungbedingung überprüft und die Schleife mit EXIT verlassen werden, falls diese nicht erfüllt ist.

Auch dazu wieder ein kleines Beispiel, das Sie schon kennen: Passworteingabe. Die Inputanweisung wird solange wiederholt, bis das Passwort korrekt eingegeben worden ist:

```
100 REPEAT
110   CLS
120   INPUT " Bitte geben Sie Ihr Passwort ein: ";Pa$
130 UNTIL Pa$="ST-BASIC"
140 '
150 'hier folgt dann das eigentliche Programm
160 '
170 END
```

### Endlosschleife

In manchen BASIC-Dialekten gibt es eine spezielle Struktur für die Endlosschleife. Endlos deshalb, weil die Schleife nie auf normalem Weg verlassen werden kann. In ST-BASIC ist dies nicht der Fall, dennoch kann auch hier problemlos eine Endlosschleife zusammengebaut werden. Man bedient sich einfach der REPEAT..UNTIL-Schleife und sorgt dafür, daß die Bedingung

hinter UNTIL nie erfüllt wird. Am besten gibt man gleich den Wahrheitswert falsch an, der in BASIC durch die 0 dargestellt wird:

```
REPEAT
   <Anweisung 1>
   <Anweisung 2>
   ....
   <Anweisung n>
UNTIL 0
```

oder verständlicher:

```
REPEAT
   <Anweisung 1>
   <Anweisung 2>
   ....
   <Anweisung n>
UNTIL Immer_und_ewig
```

Schön und gut! Aber wie kann man eine solche Schleife wieder verlassen? Eigentlich gar nie nicht, es sei denn der Interpreter stößt in der Schleife auf einen EXIT-Befehl:

### EXIT

Mit dem EXIT-Befehl ist es möglich eine Schleife (vorzeitig) zu verlassen, auch wenn ihre Abbruchbedingung noch nicht erfüllt ist. Dies gilt übrigens nicht nur für Endlosschleifen, sondern auch für deren Verwandte.

```
100 REPEAT
110    CLS
120    INPUT " Bitte geben Sie ihr Passwort ein: ";Pa$
130    IF Pa$="ST-BASIC"
140       THEN EXIT
150    ENDIF
160 UNTIL immer_und_ewig
170 ...
180 ...
```

Wird das Passwort richtig eingegeben, verläßt der EXIT-Befehl die Endlosschleife und das Programm wird direkt hinter dem Schleifenende (Zeile 170 ff) fortgesetzt. Zusätzlich kann hinter EXIT noch die Programmstelle (Zeilennummer, Label, ...) angegeben werden, an der das Programm nach dem EXIT fortgesetzt werden soll. Dazu wird dann eine Mischung aus EXIT und GOTO verwendet:

```
EXIT TO <Ziel>
```

<Ziel> muß den Ansprüchen des GOTO-Befehls genügen.

**Anmerkung:** In GFA-BASIC existiert eine eigene Struktur für die Endlosschleife:

```
DO
 <Anweisung 1>
  ....
  <Anweisung n>
LOOP
```

In ST-BASIC muß diese durch

```
REPEAT
   <Anweisung 1>
   ....
   <Anweisung n>
 UNTIL 0
```

ersetzt werden, wenn GFA- in ST-BASIC-Programme umgeschrieben werden sollen, ebenso das EXIT IF <Bedingung> des GFA-BASIC in:

```
IF <Bedingung> THEN EXIT
```

zum vorzeitigen Verlassen einer Schleifenstruktur.

## WHILE..WEND

WHILE..WEND ist - Sie dürfen aufatmen - der letzte Vertreter der Gattung Schleifen. Es handelt sich dabei um eine abweisende Schleife, da bereits vor einem Schleifeneinsprung die hinter WHILE stehende Bedingung überprüft wird. Ist sie nicht erfüllt, wird die Schleife erst gar nicht durchlaufen, sondern das Programm gleich hinter dem Schleifenende fortgesetzt. Andernfalls gelangt der Schleifeninhalt zur Ausführung, bis der Interpreter auf WEND stößt und zum Schleifenanfang zurückkehrt. Dort wird die Bedingung erneut überprüft.

Auch mit WHILE..WEND läßt sich eine Endlosschleife konstruieren. Dann muß die Bedingung hinter WHILE stets erfüllt sein, oder anders ausgedrückt den Wert -1 (wahr!) enthalten:

```
WHILE -1

    <Anweisung 1>
    <Anweisung 2>
    ...
WEND
```

```
100 '*****************************************************************
110 '*                          SIEB.BAS                             *
120 '*---------------------------------------------------------------*
130 '* Autor: Michael Maier    Version: 1.00     Datum: 02.09.1988 *
140 '*      Ein Programm aus dem 'GROSSEN ST-BASIC-BUCH'             *
150 '*        (C) 1988 by DATA BECKER GmbH Düsseldorf                *
160 '*****************************************************************
170 '
180 'Sieb des Eratosthenes
190 '
200 True%=-1:False%=0
210 REPEAT
220     CLS
230     INPUT " Bitte Suchobergrenze/2 eingeben (> 50) ";N%
240 UNTIL N%>50
250 DIM Liste%F(N%)
260 FOR T%=0 TO N%'             Liste mit -1 vorbelegen
270         Liste%F(T%)=True%
280 NEXT T%
290 FOR T%=0 TO N%
300     IF Liste%F(T%) THEN
```

```
310         ' Primzahl ausgeben
320         Prim%=2*T%+3
330         PRINT Prim%,
340         I%=T%+Prim%
350         'Alle Vielfachen davon streichen
360         WHILE I%<N%
370             Liste%F(I%)=False%
380             I%=I%+Prim%
390         WEND
400     ENDIF
410 NEXT T%
420 END
```

Als krönender Abschluß der Schleifen (ich versprech's!) das berühmt berüchtigte Sieb des Eratosthenes, mit dessen Hilfe sich alle Primzahlen, die kleiner als eine gegebene natürliche Zahl N sind, ermitteln lassen.

Dieses Verfahren ist so einfach wie wirkungsvoll: Zuerst wird eine Liste angelegt, und jeder Eintrag in dieser Liste auf den Wert -1 (wahr) gesetzt (Falls Liste%F(X) = -1 <=> X ist eine Primzahl). Beginnend vom Listenanfang (Auf alle Fälle -1!) wird jetzt für jedes Element der Liste geprüft, ob es sich dabei um eine Primzahl handelt. Dann können auch alle Vielfachen dieser Zahl aus der Liste gestrichen werden. Dies geschieht, indem die entsprechenden Einträge auf den Wert 0 (falsch) gesetzt werden. Nach diesem Schema wird Eintrag für Eintrag, bis schließlich alle Nicht-Primzahlen aus der Liste gestrichen worden sind.

Der obige Algorithmus berechnet alle Primzahlen bis zur Suchobergrenze * 2 + 3. Einen Haken hat die ganze Geschichte noch: Der Wert 2 taucht in obiger Liste nicht auf. Deshalb müßte noch eine Zeile eingefügt werden, in der die fehlende Ziffer auf den Bildschirm gedruckt wird:

```
285 Print 2
```

Noch eine weitere Neuerung steckt in diesem kleinen Programm. Ein Komma am Ende einer Printausgabe bewirkt, daß der Cursor an die nächste Tabulatorposition (jede achte Spalte auf dem

Monitor bildet eine solche Tabulatorposition) gefahren und bei einem folgenden PRINT dann an dieser Stelle weitergeschrieben wird.

## 2.11 Alles eine Frage der Routine ...

Nein lieber Leser, Sie lesen völlig richtig. Es ist wirklich alles eine Frage der Routine. Doch mit Routine ist in diesem Fall nicht nur die Praxis gemeint.

Häufig kommt es nämlich vor, daß bestimmte Dinge innerhalb eines Programmes mehrfach durchgeführt werden müssen, jedoch an völlig unterschiedlichen Programmstellen. Ein Beispiel dafür wäre das Sortieren eines Variablenfeldes. Man kann natürlich die entsprechenden Befehle jedesmal erneut in den Programmtext einflechten, doch dies geht auf Kosten des Speicherplatzes. Macht nichts, davon haben wir in unserem ST wirklich genug! Doch wie steht es mit der Mehrarbeit, die durch das immer wiederkehrende Abtippen der entsprechenden Programmzeilen auftritt? Viel praktischer wäre ein Programmteil, der nur ein einziges Mal programmiert werden müßte und dann jedesmal die Arbeit für uns verrichten würde! Einen solchen Programmteil nennt man im Computerjargon - Sie ahnen es schon - eine (Unter-)Routine oder auf gut englisch: Subroutine. Wie funktioniert nun die Sache mit den Unterroutinen?

### Einmal Unterroutine und zurück ...

Eine Unterroutine ist nichts anderes als ein kleiner Programmteil, der eine bestimmte Aufgabe (z.B. das Sortieren eines Variablenfeldes, dessen Speichern auf Diskette oder Festplatte, das Setzen des Datums und der Uhrzeit usw.) erledigt, sobald er innerhalb des Programms aufgerufen wird und anschließend zum Hauptprogramm zurückkehrt. Der Aufruf erfolgt in BASIC mit

```
GOSUB <Ziel>
```

der Rücksprung ins Hauptprogramm stets mit

```
RETURN

<Anweisung 1>
<Anweisung 2>
<Anweisung 3>
....
....
....
GOSUB Unterprogramm
<Anweisung n>
....
....
END
-Unterprogramm
<Anweisung U1>
<Anweisung U2>
....
....
<Anweisung Un>
RETURN
```

Das eigentliche Programm (Hauptprogramm) beginnt mit <Anweisung 1> und endet mit END. Innerhalb des Hauptprogramms erfolgt ein Sprung in das Unterprogramm, das den sinnigen Namen Unterprogramm trägt. Dort werden dann die Anweisungen des Unterprogramms <Anweisungen U1> bis <Anweisung Un> ausgeführt. Stößt der Interpreter auf ein RETURN, so kehrt er in das Hauptprogramm zurück. Die nächste Anweisung hinter dem GOSUB wird ausgeführt, hier wäre dies die Anweisung <n>.

Der wesentlichste Unterschied zwischen dem GOTO-Befehl und dem GOBUB besteht darin, daß bei GOSUB wieder direkt hinter den Unterroutinenaufruf im Hauptprogramm zurückgekehrt werden kann, während dies bei einem GOTO praktisch unmöglich ist, da sich der Interpreter nicht merkt, von wo aus er an den unter <Ziel> angegebenen Programmpunkt gesprungen ist. Bei einem GOSUB (übrigens GO SUBroutine, springe in das Unterprogramm) merkt sich der Interpreter die Stelle des Aufrufs und kehrt mit einem RETURN wieder an die ursprüngliche Stelle im Hauptprogramm zurück.

Als <Ziel> kann eine Zeilennummer (Mircosoft-BASIC läßt grüßen), eine Marke, und die übrigen auch beim GOTO zugelassenen Parameter angegeben werden.

Die Verwendung von Unterprogrammen macht ein Programm kürzer und übersichtlicher. Ein weiterer Vorteil ist die größere Änderungsfreundlichkeit. Stellen Sie sich vor, Sie müssen an zehn Programmstellen Zahlen sortieren, und zwar in aufsteigender Reihenfolge. Aus irgendeinem Grung werden aber die Zahlen plötzlich in absteigender Folge benötigt. Arbeiten Sie mit Unterroutinen, so bedarf es lediglich einer Änderung des entsprechenden Unterprogramms, schon sind Sie fertig. Sonst müßte die Änderung gleich zehnfach im Programm vorgenommen werden. Nicht nur ein Mehraufwand, sondern auch eine Fehlerquelle, da erfahrungsgemäß immer der eine oder andere Sortieralgorithmus im Programm übersehen wird, wenn Programme etwas länger werden.

Als Alternative zu GOSUB stellt ST-BASIC noch

```
ON ... GOSUB
```

zur Verfügung, das in der Syntax der ON..GOTO-Anweisung entspricht. MIT ON..GOSUB wird jedoch in ein Unterprogramm gesprungen, aus dem mit RETURN zurückgekehrt werden kann.

### Prozeduren

Auch bei Prozeduren handelt es sich um um Unterprogramme, jedoch in einer etwas umfassendere,n weil moderneren Erscheinungsart.

Mit einer Prozedur wird nämlich ein neuer Befehl definiert, der dann innerhalb des (Haupt-)Programms aufgerufen werden kann. Auch im Bildschirm-Editor wird der neue Befehl unmittelbar nach seiner Definition zur Verfügung gestellt. Doch eine Prozedur kann noch mehr:

- Parameter können ihr übergeben werden,
- es können lokale Variablen vereinbart werden;
- Parameter können zurückgegeben werden;
- Prozeduren können sich selbst wieder aufrufen (Rekursion)

Dies sind alles Dinge, die bis vor kurzem für einen BASIC-Interpreter undenkbar waren und mit ST-BASIC Realität geworden sind. Doch genug der schönen Worte, schreiten wir zur Tat:

Eine Prozedur wird für den Interpreter als solche gekennzeichnet, indem sie mit dem Befehl

```
DEF PROC <Name>
```

definiert werden. Das Ende eine Prozedur erkennt ST-BASIC wieder an einem RETURN.

```
DEF PROC <Name>
   <Anweisung 1>
   <Anweisung 2>
   <Anweisung 3>
   .....
   .....
   <Anweisung n>
RETURN
```

Auf diese Weise wird eine Prozedur <Name> definiert. Um sie innerhalb des Hauptprogrammes aufzurufen, wird einfach ihr Name <Name> wie ein Befehl in das Programm eingefügt. Stößt der Interpreter bei der Abarbeitung des Programms auf einen unbekannten Befehlsnamen, überprüft er, ob damit nicht vielleicht eine Prozedur gemeint ist. Dann werden die zwischen der Definition der Prozedur (DEF PROC) und dem abschließenden RETURN stehenden Anweisungen ausgeführt und zum Hauptprogramm zurückgekehrt. Dazu ein einfaches Beispiel:

ST-BASIC kennt einen Befehl zum Löschen des Bildschirms: CLS. Der Cursor wird nach dem Löschvorgang in die linke obere Ecke des Monitors gesetzt. Häufig möchte man nicht

gleich den ganzen Bildschirm löschen, sondern lediglich den Cursor in die besagte Ecke des Bildschirms bringen. Dieser Effekt wird mit Home auf der Tastatur ausgelöst, innerhalb eines Programms stellt ST-BASIC jedoch keinen derartigen Befehl zur Verfügung. Also schreiben wir uns einfach einen neuen Befehl dafür:

```
DEF PROC Home
   PRINT CHR$(27);"H"
RETURN
```

Erscheint innerhalb des Hauptprogramms der Befehl Home, so wird der Cursor in die linke obere Monitorecke gesetzt. Doch auch im Bilschirmeditor wird der Befehl Home mit der Cursorpositionierung in die linke obere Bildschirmecke beantwortet. Rätselhaft wird Ihnen warscheinlich die zweite Zeile der Definition vorkommen. Dies ist der eigentliche Befehl zur Positionierung des Cursors an die besagte Position.

Der Monitor des Atari ST ist nämlich in gewisser Weise intelligent. Er versteht bestimmte Befehle, wie Bildschirm löschen, Cursor an eine angegebene Position setzen, eine Zeile löschen oder einfügen, Cursor ein- und ausschalten. Diese Befehle sind einem bekannten Terminal nachempfunden (deshalb spricht man auch von einem VT-52-Emulator) und besginnen stets mit dem Befehl CHR$(27) (Escape, ESC), gefolgt von einem (oder mehreren) Buchstaben. Doch dazu später mehr.

### Parameter übergeben

Im Gegensatz zu einem Unterprogrammaufruf mit GOSUB können einer Prozedur auch ein oder mehrere Parameter übergeben werden. Diese müssen innerhalb von Klammern an den Prozedurenamen angehängt werden:

```
....
<Anweisung 1>
Setcursor(10,10)
....
....
```

```
<Anweisung n>

DEF PROC Setcursor(Spalte,Zeile)
   PRINT CHR$(27);CHR$(Spalte+32);CHR$(Zeile+32)
RETURN
```

Die in diesem kleinen Programmfragment vorkommende Prozedure Setcursor setzt den Cursor auf eine bestimmte Bildschirmposition, die durch die beiden Übergabeparameter Spalte und Zeile eindeutig bestimmt wird. Auch hier muß wieder der VT-52-Emulator des Atari ST herhalten. Die beiden in Klammern angegebenen Parameter Spalte und Zeile sind lokal. Was bedeutet das schon wieder?

### Globale und lokale Variablen

Variablen in BASIC sind normalerweise global definiert. Das bedeutet, daß eine Variable überall im Programm einen bestimmten, einmal zugewiesenen Wert besitzt. Im Gegensatz dazu besitzen lokale Variablen nur innerhalb der Befehlsdefinition (Prozedur) einen bestimmten Wert. Außerhalb dieser Prozedur verwendete (globale) Variablen können einen völlig anderen Inhalt besitzen, obwohl Sie den gleichen Namen tragen. Mit ihren lokal definierten Kollegen haben sie nichts zu schaffen. Ein kleines Beispiel:

```
100 Spalte=20: Zeile=10
110 PRINT "Spalte:";Spalte;" Zeile:";Zeile
120 Setcursor(10,5)
130 PRINT "Spalte:";Spalte;" Zeile:";Zeile
140 END
150 'die Prozedur Setcursor
160 DEF PROC Setcursor(Spalte,Zeile)
170    PRINT CHR$(27);CHR$(Spalte+32);CHR$(Zeile+32)
180    PRINT "In Prozedur: Spalte:";Spalte;" Zeile:";Zeile
190 RETURN
```

An diesem kleinen Beispiel wird deutlich, daß die beiden Variablen Spalte und Zeile innerhalb der Prozedur lokal definiert sind. Obwohl beide auch im eigentlichen Hauptprogramm auftauchen, besitzen sie innerhalb der Prozedur völlig andere Werte. Mit Verlassen der Prozedur erhalten sie dann wieder die ur-

sprünglichen (globalen) Werte zugewiesen. Anders verhält sich die Sache, wenn die beiden Parameter nicht beim Funktionsaufruf übergeben werden:

```
100 Spalte=20: Zeile=10
110 PRINT "Spalte:";Spalte;" Zeile:";Zeile
120 Setcursor
130 PRINT "Spalte:";Spalte;" Zeile:";Zeile
140 END
150 'die Prozedur Setcursor
160 DEF PROC Setcursor
170    PRINT CHR$(27);CHR$(Spalte+32);CHR$(Zeile+32)
180    PRINT "In Prozedur: Spalte:";Spalte;" Zeile:";Zeile
190 RETURN
```

In diesem Fall sind die Variablen Spalte und Zeile global und besitzen folglich auch im Unterprogramm die gleichen Werte wie im Hauptprogramm. Möchten Sie die Variablen innerhalb des Unterprogramms lokal behandelt wissen, so müssen diese mit einem eigenen Befehl als lokale Variablen im Unterprogramm gekennzeichenet werden:

```
LOCAL <Variable>, <Variable>, ...
```

Eine so deklarierte Variable nimmt innerhalb des Unterprogramms Werte an, die bei einer Rückkehr ins Hauptprogramm nicht mehr bekannt sind. Dort auftauchende Variablen können trotz gleichen Namens völlig andere Werte beinhalten. Solange Variablen nicht beim Aufruf einer Prozedur in Klammern übergeben werden oder durch den Befehl LOCAL innerhalb des Unterprogramms als lokale Variablen definiert sind, ist ihr Wert innerhalb des gesamten Programms bekannt.

Werte werden an eine Prozedur übergeben, indem man sie in Klammern hinter den Funktionsaufruf schreibt. Selbstverständlich muß die Anzahl der übergebenen Parameter auch mit der in der Prozedurdefinition übereinstimmen bzw. müssen die richtigen Variablentypen übergeben werden. Andernfalls meldet der Interpreter diesen Fehler, und das Programm bricht ab.

Häufig wird eine Prozedur mit bestimmten Werten aufgerufen, diese führt die entsprechenden Operationen aus und gibt einen oder mehrere Werte an das Hauptprogramm zurück. Um dem Computer mitzuteilen, daß ein bestimmter Wert zurückgegeben werden soll, wird die Variable in der Prozedurdefinition durch ein vorangestelltes R gekennzeichnet:

```
DEF PROC Setcursor(R Spalte, R Zeile)
....
RETURN
```

Wird diese Prozedurdefinition in das kleine Beispielprogramm integriert, so erhalten die Variablen Spalte und Zeile ab dem Prozeduraufruf die neuen Werte 10 und 5, da die Werte des Funktionsaufrufs an das Hauptprogramm zurückgegeben werden, sobald die Prozedur ihre Tätigkeit beendet hat.

### Rekursionen

Im Gegensatz zu den BASIC-Dialekten vergangener Tage ist es möglich, daß sich eine Prozedur auch selbst aufruft. Diesen Vorgang nennt man in der Informatik Rekursion:

```
.....
.....
Male(10,20,100,250)
....
....
DEF PROC Male(x1,y1,x2,y2)
    .....
    .....
    Male(x1,y1,x2,y2)
    .....
    .....
RETURN
```

Dieses kleine Programmfragment verdeutlicht die Funktionsweise einer rekursiv programmierten Prozedur. Zunächst wird eine solche Unterroutine ganz normal aufgerufen und die übergebenen Parameter werden in irgendeiner Weise verarbeitet. Neu ist

dagegen, daß sich diese Funktion innerhalb der Unterroutine selbst aufruft. Wieder werden ihr die (zuvor schon verarbeiteten) Parameter übergeben.

Der erneute Aufruf bewirkt wiederum eine Bearbeitung der angegebenen Parameter, bis die Funktion ein weiteres Mal aufgerufen wird usw. Erst wenn die Parameter dergestalt bearbeitet sind, daß die Funktion ihre Pflicht getan hat, kehrt sie in das Hauptprogramm zurück. Mit Hilfe der rekursiven Programmierung können bestimmte Probleme auf äußerst elegante Art und Weise gelöst werden. Es würde mit Sicherheit den Rahmen dieses Buches sprengen detailliert auf die rekursive Programmierung einzugehen - darüber haben Informatiker schon ganze Bände geschrieben -, aber anhand eines Beispiels soll einmal eine Rekursion aufgezeigt werden.

Variablenfelder sind Ihnen bereits bestens vertraut. Möchte man ein solches Array sortieren, muß ein dafür geeigneter Algorithmus benutzt werden. Informatiker waren auf diesem Gebiet besonders findig. Es existieren eine Unzahl der verschiedensten Sortieralgorithmen, vom einfachen Bubble-, über Shell-, Heapbis hin zu Bucket-Sort. Jeder Algorithmus kann mit bestimmten Vorteilen aufwarten. Ein Vertreter der schnelleren Garde von Sortiermethoden ist der 1962 von C.A.R Hoare entwickelte Quicksort, der - Sie ahnen es schon - die ihm anvertrauten Daten rekursiv sortiert:

```
100 '*****************************************************************
110 '*                          QUICK.BAS                            *
120 '*---------------------------------------------------------------*
130 '* Autor: Michael Maier    Version: 1.00     Datum: 16.02.1987 *
140 '*      Ein Programm aus dem 'GROSSEN ST-BASIC-BUCH'             *
150 '*        (C) 1988 by DATA BECKER GmbH Düsseldorf                *
160 '*****************************************************************
170 '
180 '
190 DIM Feld%(9)
200 RESTORE
210 FOR T%=0 TO 9
220    READ Feld%(T%)
230 NEXT T%
240 '
```

```
250 Quicksort(0,9)
260 '
270 FOR T%=0 TO 9
280     PRINT Feld%(T%);"  ";
290 NEXT T%
300 DATA 3,9,7,2,0,6,1,5,4,8
310 END
320 '
330 DEF PROC Quicksort(Anfang%,Ende%)
340    'die folgende Zeile dient nur zum Testen
350    'und hat mit der eigentlichen Routine nichts
360    'zu tun!
370    PRINT "#";: FOR T%=0 TO 9: PRINT Feld%(T%);" ";: NEXT T%: PRINT
380    'jetzt erst geht's los...
390    LOCAL A%,Z%
400    A%=Anfang%
410    Z%=Ende%
420    X%=Feld%((Anfang%+Ende%)/2)
430    REPEAT
440       WHILE Feld%(A%)<X%
450         A%=A%+1
460       WEND
470       WHILE Feld%(Z%)>X%
480         Z%=Z%-1
490       WEND
500       IF A%<=Z% THEN
510         SWAP Feld%(A%),Feld%(Z%)
520         ' auch die beiden folgenden Zeilen dienen nur zum
530         ' Testen und können entfernt werden
540         FOR T%=0 TO 9: PRINT Feld%(T%);" ";: NEXT T%
550         PRINT ,A%;" ";Z%," ";X%
560         ' hier geht's wieder normal weiter
570         A%=A%+1
580         Z%=Z%-1
590       ENDIF
600     UNTIL A%>Z%
610     IF Anfang%<Z% THEN
620       Quicksort(Anfang%,Z%)
630     ENDIF
640     IF A%<Ende% THEN
650       Quicksort(A%,Ende%)
660     ENDIF
670 RETURN
```

Dieses Programm sortiert ein Integerarray Feld%. Ehe das Feld durch den Aufruf der Funktion Quicksort in Zeile 250 sortiert werden kann, müssen den einzelnen Indices erst verschiedene Werte zugewiesen werden. Dies erledigt die FOR..NEXT-Schleife über eine READ-DATA-Anweisung. Die Werte selbst sind hinter der DATA-Anweisung in Zeile 300 abgelegt. Sobald dies erledigt ist, erfolgt der Aufruf der Quicksortroutine. Die anzugebenden Parameter sind dabei der erste und letzte zu sortierende Index.

Bei jedem Einsprung in die Routine wird zuerst einmal der gesamte Feldinhalt über eine PRINT-Anweisung ausgegeben. Anschließend werden den lokalen Variablen A% und Z% die ebenfalls lokalen Parameter des Funktionsaufrufs Anfang% bzw. Ende% zugewiesen.

Der nächste Schritt besteht nun darin, das Feld in zwei Teile aufzuspalten. Dies geschieht, indem der Variablen X% das in der Feldmitte stehende Element zugewiesen wird, das ab sofort als Vergleichsvariable weiterbenutzt wird. Die erste WHILE-Schleife sucht das erste Element von unten (Anfang), das nicht kleiner als das Vergleichselement ist. Entsprechend wird in der folgenden Schleife das erste Element von oben (Feldende) gesucht, das nicht größer als die Vergleichsvariable X% ist. Die auf diese Weise gefundenen Elemente werden miteinander vertauscht und, da sie theoretisch auch gleich groß sein können, übersprungen.

Apropos vertauschen: In ST-BASIC existert dafür ein eigener Befehl:

```
SWAP <Feld(X)>,<Feld(Y)>
```

vertauscht die Elemente X und Y des Arrays Feld miteinander.

Die REPEAT-Schleife sorgt dafür, daß dieses Spielchen so oft wiederholt wird, bis die untere die obere Grenze überschritten hat. Dann sind in beiden Teilfeldern keine Elemente mehr vorhanden, die in den anderen Teil des Feldes gehören.

Danach werden die beiden Teilfelder sortiert. Dies geschieht, indem sich die Funktion wieder selbst (rekursiv) aufruft und die Sortiergrenzen auf das gewünschte Teilfeld gesetzt werden.Damit das Prinzip noch deutlicher wird, sind in dem Programm zwei Zeilen eingefügt, die den Inhalt des Feldes ausgeben. Dies geschieht bei jedem Einsprung in die Routine Quicksort, sowie nach jedem Austausch zweier Variablen. Um die Ausgaben beider Zeilen optisch trennen zu können, beginnt erstere stets mit dem Zeichen #. Zusätzlich wird nach dem Tausch noch der Inhalt der unteren bzw. oberen Grenze und des Vergleichselements mit ausgegeben.

Starten Sie das Programm, wird zuerst das Ausgangsfeld auf den Monitor geschrieben:

```
#3 9 7 2 0 6 1 5 4 8
```

Das Programm wählt als Vergleichselement die 6. Das erste Element von unten (in diesem Fall von links), das nicht kleiner als 6 ist, ist die 9, von oben (hier von rechts) das erste Element, das nicht größer als das Vergleichselement ist, ist die 4. Beide werden miteinander vertauscht:

```
3 4 7 2 0 6 1 5 9 8          1 8   6
                             A% Z%  X%
```

Noch sind die untere und die obere Grenze nicht überschritten, es wird weitergesucht: von unten stößt der Interpreter auf die 7, von oben auf die 5. Auch diese beiden werden vertauscht:

```
3 4 5 2 0 6 1 7 9 8          2 7   6
                             A% Z%  X%
```

Noch immer nicht überschreitet die untere die obere Grenze. Es heißt weitersuchen. Die beiden nächsten Variablen, die der Vertauschungswut des Interpreters zum Opfer fallen sind 6 und 1:

```
3 4 5 2 0 1 6 7 9 8          5 6   6
                             A% Z%  X%
```

Jetzt ist es soweit, die untere Grenze liegt über der oberen. Also die beiden Teilfelder mit einem rekursiven Aufruf sortieren. Zuerst werden die einzelnen Elemente des Feldes wieder auf dem Monitor ausgegeben:

```
# 3 4 5 2 0 1 6 7 9 8
```

Als Vergleichselement des zweiten Funktionsaufrufes muß die 2 herhalten. Wieder wird das erste Element von unten gesucht, das nicht kleiner als das Vergleichselement ist, in diesem Fall die 3. Vertauscht wird sie mit dem ersten Element von oben, das nicht größer als die 2 ist: 3:

```
1 4 5 2 0 3 6 7 9 8          0  5   2
                             A% Z%  X%
```

Solange die untere noch nicht die obere Grenze überschreitet, muß weitervertauscht werden:

```
1 0 5 2 4 3 6 7 9 8          1  4   2
                             A% Z%  X%

1 0 2 5 4 3 6 7 9 8          2  3   2
                             A% Z%  X%
```

Das Vergleichselement des nächsten Aufrufs ist die 0, die mit der 1 vertauscht wird:

```
# 1 0 2 5 4 3 6 7 9 8

0 1 2 5 4 3 6 7 9 8          0  1   0
                             A% Z%  X%
```

Beim nächsten Aufruf wird das Vergleichselement 2 einmal mit sich selbst vertauscht:

```
# 0 1 2 5 4 3 6 7 9 8

0 1 2 5 4 3 6 7 9 8          2  2   2
                             A% Z%  X%
```

Dieses Spielchen setzt sich fort, bis beide Teilfelder fertig sortiert vorliegen. Anhand dieses kleinen Algorithmus, der übrigens auch in der implementierten Sortierroutine des ST-BASIC Verwendung findet, sieht man sehr schön die Funktionsweise und Eleganz einer rekursiv programmierten Unterroutine. Die Nachteile sollen allerdings auch nicht verschwiegen werden: Häufig ist es sehr schwierig, eine rekursive Routine zu programmieren, da sich kein geeigneter Rekursionsweg finden läßt. Andererseits kostet eine Rekursion bei ihrer Ausführung stets mehr Speicherplatz als ihre iterativ (Gegenteil von rekursiv) durchgeführte Schwester. Warum dieses?

Der Interpreter merkt sich den Punkt, von dem aus er in ein Unterprogramm gesprungen ist. Dazu legt er die entsprechenen Daten in einem bestimmten Speicherbereich ab. Dieser Speicherbereich, den der Prozessor auch bei seiner Arbeit verwendet, heißt Stack. Ruft sich eine Funktion rekursiv auf, so wächst der Stack logischerweise mit jedem Aufruf an. Es sind Problemstellungen denkbar, die nicht rekursiv gelöst werden können, da selbst der Speicherplatz eines Mega ST 4 nicht ausreicht, um die Daten des Stacks aufzunehmen.

### Funktionen

Ähnlich wie Prozeduren, können Sie in ST-BASIC auch Funktionen selbst definieren. Im Gegensatz zu einer Prozedur gibt eine Funktion jedoch stets einen Wert zurück: den Funktionswert.

```
[Funktionsparameter]  --Funktion-->  Funktionswert
```

Dieser kann dann einer Variablen zugewiesen oder gleich auf dem Monitor ausgegeben werden. Die Funktionsparameter können bei Bedarf auch entfallen. Ein Funktionswert wird aber auf jeden Fall zurückgegeben. ST-BASIC unterscheidet zwei Funktionsarten: Ein- und mehrzeilige.

### Einzeilige Funktionen

Ehe eine Funktion benutzt werden kann, muß sie mit

```
DEF FN <Name(Parameter)> = <Ausdruck>
```

definiert werden. Die Anweisung DEF FN weist den Interpreter an, eine Funktion mit dem Namen <Name> einzuführen, der die Argumente <Parameter> übergeben werden. <Ausdruck> enthält die eigentliche Funktionsvorschrift. Als Beispiel möchte ich noch einmal die Wechselkursumrechnung DM in Schilling mißbrauchen. Zuerst die Definition der Funktion:

```
DEF FN Schilling(Betrag!) = INT((Betrag!/.1429*100)+.5)/100
```

Diese Definition kann beliebig in den Programmtext eingestreut werden, da der Interpreter beim Programmstart sämtliche DEF FN selbständig ausführt. Möchten Sie diese Funktion aufrufen, so geht dies über FN, gefolgt von dem Funktionsnamen und dem benötigten Parameter als Argument:

```
Waehrung!= FN Schilling(100)
```

Der in Schilling umgerechnete Betrag von 100 DM wird der Variablen Waehrung! zugewiesen, und kann via PRINT auf den Monitor gebracht werden. Mit

```
PRINT FN Schilling(100)
```

umgeht man die Zuweisung an die Variable und sieht das Ergebnis sofort auf dem Monitor: 699.79

Möchten Sie einen String als Funktionswert erhalten, so muß der Funktionsname mit einem Dollarzeichen ($) am Namensende gekennzeichenet werden:

```
DEF FN Screen$(X$) = CHR$(27)+X$
```

auf diese Weise resultiert ein zwei Zeichen langer String, der mit einer Escape-Sequenz beginnt. Mit ihm kann der VT-52-Emulator des Atari ST angesprochen werden (erinnern Sie sich?).

```
PRINT Screen$("H")
```

setzt den Cursor in die linke obere Ecke des Bildschirms (HOME), ohne diesen jedoch zu löschen. Funktionen können auch andere Funktionen aufrufen. Zum Beispiel die Funktion zur Cursorpositionierung:

```
DEF FN Crsr$(Sp,Ze)= FN Screen$("Y")+CHR$(32+Sp)+CHR$(32+Ze)
```

Möchten Sie den Cursor an die Position (10,10) setzen, genügt der Aufruf

```
PRINT FN Crsr$(10,10).
```

Es versteht sich von selbst, daß die Anzahl und der Typ der übergebenen Parameter mit denen in der Funktionsdefinition übereinstimmen müssen. Auch eine Funktion ohne Übergabe eines Parameters ist denkbar und vor allem möglich!

```
DEF FN Wuerfel= -RND(-6)
```

Diese Funktion liefert Zufallszahlen im Bereich von 1 bis 6. Dazu wird der Zufallsgenerator des Atari mit der Funktion RND (Randomize) angesprochen. Wird ihm als Argument ein negativer Wert übergeben, so liefert der Zufallsgenerator ganzzahlige Zufallszahlen im Bereich von -1 und dem übergebenen Wert. Das Minuszeichen vor der Funktion RND sorgt dafür, daß der Funktionswert positiv wird:

```
-(-4) = +4
```

### Mehrzeilige Funktionen

ST-BASIC erlaubt auch die Definition mehrzeiliger Funktionen. Wieder muß die Umrechnung von DM in Schilling herhalten, um den Unterschied zwischen beiden Funktionsarten deutlich zu machen:

```
100 DEF FN Schilling(Betrag!)
110 RETURN INT((Betrag!/.1429*100)+.5)/100
```

Zwei Unterschiede existieren zur einzeiligen Variante:

- Der Zuweisungsoperator = ist verschwunden
- Die Funktion wird über RETURN verlassen, wobei der Funktionswert, der zurückgegeben werden soll, hinter dem RETURN stehen muß.

Mehrzeilige Funktionen sind - wie sich unschwer erkennen läßt - mit Mehrarbeit verbunden. Als Lohn der Mühe erhält man mehr Möglichkeiten und eine damit verbundene größere Flexibilität:

- Kontrollstrukturen wie IF..THEN..ELSE oder Schleifen können in der Funktionsdefinition vorkommen.
- In Abhängigkeit bestimmter Bedingungen können unterschiedliche Funktionsausdrücke zur Berechnung herangezogen werden.
- Funktionen können rekursiv programmiert werden.

Dazu wieder ein (Parade-)Beispiel: Die Fakultätsfunktion ist in der Mathematik bzw. Statistik wie folgt definiert:

- Die Fakultät von 0 ist 1,
- Die Fakultät von X ist X mal der Fakultät von (X-1).

Die Fakultät von 5 (geschrieben als 5!) berechnet sich folglich als:

$$5*4! = 5*4*3! = 5*4*3*2! \; 5*4*3*2*1! = 5*4*3*2*1 = 120$$

Ein klassisches Rekursionsproblem, das wie folgt gelöst wird:

```
100 DEF FN Fakultaet(X!)' Float, da extrem große Werte!
110   IF FRAC(X!) > 0 THEN' Nachkommastellen vorhanden?
120     PRINT" Fakultät ist nur für natürliche Zahlen definiert!"
130     'EXIT TO <Ziel> mit EXIT zurück
140   ENDIF
150   IF X!=0 THEN
```

```
160     RETURN(1)' 0! = 1
170  ENDIF
180 RETURN FN Fakultaet(X!-1)*X!
```

Als Funktionsargument muß eine Fließkommazahl herhalten, da bei der Fakultätsberechnung sehr schnell extrem große Werte erreicht werden (69! = 1,71 * 10^98 !!!!), die dann nicht mehr in einer Integerzahl gespeichert werden können.

Eine mehrzeilige Funktion kann übrigens auch mit einem EXIT TO <Ziel> verlassen werden. Dies wird dann nötig, wenn eine Zahl mit Nachkommastellen zur Berechnung übergeben wurde, da dies (mathematisch) nicht definiert ist. Die Fakultät kann nur für positive natürliche (ganze) Zahlen einschließlich der Null berechnet werden. Als <Ziel> muß eine Zeilennumer oder eine Marke angegeben werden, an die der Interpreter im Falle des Falles springen kann. Die Funktion FRAC(X!) liefert - ich weiß ich wiederhole mich - die Nachkommastellen einer Zahl, und somit bei vorhandenen Nachkommastellen eine wahre Bedingung.

Die Rekursion selbst dürfte keine Schwierigkeiten mehr machen. Rufen Sie die Funktion mit Fakultaet(5) auf, so geschieht folgendes:

- Solange der Funktionswert noch nicht den Wert 0 erreicht hat, ruft sich die Funktion immer wieder selbst auf, jedoch mit einem um den Wert 1 erniedrigten Funktionswert.
- Ist der Funktionswert 0 erreicht, werden die einzelnen Werte multipliziert und zurückgegeben. Fertig!

Übrigens: ST-BASIC besitzt eine eigene Funktion zur Berechnung der Fakultät einer natürlichen Zahl, so daß man sich das Schreiben der Funktion schenken kann: FACT(X). Im Gegensatz zu Prozeduren ist es nicht möglich Funktionen mit Rückgabe-Parametern auszustatten. Es wird jeweils nur der Funktionswert zurückgegeben!

Eine nützliche Funktion zur Stringmanipulation, die in ST-BASIC jedoch nicht implementiert ist, möchte ich Ihnen nicht

vorenthalten: Insert$

```
100 DEF FN Insert$(Master$,Einf$,Stelle)
110 Master$= LEFT$(Master$,Stelle-1)+Einf$+ MID$(Master$,Stelle)
120 RETURN Master$
```

Die Funktion setzt den String Einf$ ab der Position Stelle in die Zeichenkette Master$ ein. Mit diesen Werkzeugen ausgestattet, können wir (endlich!) zu größeren Taten schreiten! Zuvor bin ich Ihnen allerdings noch die Erklärung eines Befehls schuldig, den ich zwar schon bei Quicksort schon benutzt, aber noch nicht erklärt habe: READ, DATA und RESTORE.

## 2.12 READ, DATA und RESTORE

Möchten Sie ein Array mit bestimmten Anfangswerten versorgen, stehen Ihnen dafür mehrere Möglichkeiten zur Verfügung:

- Jedes Element kann über eine eigene Zuweisung mit dem Anfangswert versorgt werden:

```
A(0)=10
A(1)=3
A(2)=41
....
....
```

- Auch eine manuelle Eingabe der Werte beim Programmstart erweist sich als nicht besonders praktisch:

```
FOR T% = 0 TO Anzahl
   Input A(T%)
NEXT T%
```

- die Werte können jedoch auch hinter der Anweisung DATA abgelegt und über eine READ-Anweisung eingelesen werden.

```
100 REM Array A() mit Anfangswerten versorgen
110 DIM A(10)
120 FOR T%=0 TO 10
130    READ A(T%)
140 NEXT T%
150 '
160 '
170 '
180 DATA 10,20,30,50,60,20,65,0,4,521,87
190 END
```

In der Programmzeile 180 sind genau 11 verschiedene Werte, fein säuberlich durch Komma voneinander getrennt, abgelegt. Stößt der Interpreter bei seiner Arbeit auf den Befehl

```
READ
```

so wird das Element aus der DATA-Zeile gelesen, auf das der DATA-Zeiger weist. Dieser Zeiger sagt dem Computer, welches Element er als nächstes aus der DATA-Zeile holen muß. Nach dem Programmstart wird der Zeiger stets auf den ersten Wert in der ersten DATA-Zeile gesetzt. In unserem Programm ist nur eine Zeile vorhanden, deshalb nimmt der Interpreter den Wert 10.

Anschließend erhöht er den DATA-Zeiger um eine Position, so daß dieser ab sofort auf den nächsten Wert in der Datazeile weist (20). Beim nächsten READ geht es diesem Wert an den Kragen, gleichzeitig wird der Zeiger wieder ein Element weiterbewegt. Dies wiederholt sich bei jedem READ, bis alle Daten eingelesen sind.

Versuchen Sie mit READ mehr Daten aus der Tabelle abzuholen, als in dieser vorhanden sind, muß der Computer feststellen, daß sein DATA-Zeiger irgendwo in die Prärie weist und er meldet sich mit einem OUT OF DATA zu Wort. Zur Manipulation des DATA-Zeigers existiert ein eigener Befehl:

*RESTORE:* RESTORE setzt den DATA-Zeiger auf das erste Element der ersten Datazeile. Aber auch inmitten der Werte kann der Zeiger gesetzt wer-

den, indem hinter RESTORE eine Zeilennummer oder eine Marke angegeben wird.

*RESTORE 500:* Placiert den Zeiger auf das nächste auffindbare Element ab Zeile 500.

*RESTORE Irgendwohin:* Setzt den Zeiger auf das nächste Datenlement hinter dem Label "Irgendwohin".

*ON..RESTORE:* Funktioniert wie ON..GOTO und setzt den Zeiger auf die entsprechende(n) hinter RESTORE angegebene(n) DATA-Zeile(n):

```
100 INPUT " Umsätze für Monat? (1-12) ";Monat%
110 ON Monat RESTORE Januar, Februar, Maerz, April, Mai, Juni ...
120 '
130 FOR T%=1 TO 31
140   READ Umsatz
160   IF Umsatz=-1 THEN
170      EXIT
180   ENDIF
190   PRINT T%;".";Monat%;"      ";Umsatz
200 NEXT T%
210 END
220 '
230 -Januar
240 DATA 5436,4986,45673,12345,23987,23769,2976
250 DATA 7659,3276,4598,5456,5986,3456,-1
260 -Februar
270 DATA 65786,4987,3096,2386,2387,30975,23765,2386
280 DATA 54675,23987,-1
290 -Maerz
300 'und so weiter .....
```

Dieses (zugegebenermaßen) nicht sehr sinnvolle Programm druckt die in jedem Monat getätigten Umsätze aus. Der ON..RESTORE-Befehl sorgt dafür, daß der DATA-Zeiger auf den ersten Umsatzwert des entsprechenden Monats positioniert wird. Anschließend werden die einzelnen Datawerte per READ-Anweisung eingelesen.

Neu ist noch die Überprüfung des eingelesenen Wertes. Beträgt dieser -1, so sind bereits alle Umsätze eingelesen und die Schleife kann verlassen werden. Auf diese Weise werden stets nur soviele Daten eingelesen, wie auch wirklich vorhanden sind. Bei unterschiedlichen Datenmengen ein unabdingbares Muß, damit die Fehlermeldung OUT OF DATA vermieden wird.

Neben Zahlen können selbstverständlich auch Zeichenketten in DATA-Zeilen abgelegt werden. Diese müssen dann in Anführungszeichen stehen:

```
90 DIM A$(12):T%=0
100 REPEAT
110   T%=T%+1
120   READ A$(T%)'  einer STRINGvariable zuweisen
130 UNTIL A$(T%)="Dezember"140 '
150 END
160 '
170 DATA "Januar","Februar","Maerz","April","Mai","Juni"
180 DATA "Juli","August","September","Oktober","November"
190 DATA "Dezember"
```

In die Variable A$() werden die einzelnen Monatsnamen aus den DATA-Zeilen eingetragen. Die REPEAT..UNTIL-Schleife sorgt dafür, daß alle Monatsnamen bis einschließlich "Dezember" eingelesen werden. Ist dies geschehen, wird die Schleife beendet. Somit erspart man sich das Abzählen der einzelnen Werte, das besonders bei längeren Datenkolonnen zweitaufwendig und fehlerbehaftet ist.

## 2.13 BASIC-Allerlei

Mit Riesenschritten nähern wir uns dem nächsten Kapitel, in dem sich alles um die Datenspeicherung auf Diskette drehen wird. Und weil damit auch der Grundkurs in ST-BASIC zu Ende geht, können wir uns gleich an eine etwas schwierigere Kost wagen.

Ehe Daten auf Diskette gespeichert werden können, müssen diese erst einmal erfasst werden. Was liegt also näher, als eine Mini-Adreßverwaltung zu entwickeln, anhand der gleich noch ein paar prinzipielle Dinge besprochen werden können, die ich bisher unterschlagen habe? Was braucht der Mensch?

Bis jetzt waren die Programme relativ kurz, man konnte einfach drauflos programmieren. Je komplexer die Programme allerdings werden, desto mehr Gedanken muß man sich vor dem eigentlichen Programmiervorgang machen. Als Lohn dieser Mühe kann man sich viel Zeit beim Austesten seiner Programme sparen.

BASIC verführt dazu, ein Programm direkt in den Computer zu "hacken" und anschließend solange auszuprobieren bis es funktioniert. Bei Verwendung von Compilersprachen, deren Programme nicht direkt ausführbar sind, sondern erst in eine ausführbare Form gebracht werden müssen, ist diese Problematik nicht so akut. Stellen Sie sich einmal vor, Sie haben ein Programm entwickelt und ehe Sie es testen können, müssen Sie einige Minuten warten, in denen das Programm compiliert wird. Schon bald werden Sie sich freiwillig zuerst überlegen, was dieses Programm können soll, und wie dies bewerkstelligt werden kann, ehe Sie den Computer einschalten. Und auch bei BASIC sollten Sie sich im eigenen Interesse diesen Arbeitsstil angewöhnen.

### Aufgabenstellung festlegen

Der erste Schritt vor dem eigentlichen Programmiervorgang ist die Festlegung der Aufgabenstellung. Gut, daß wir eine Adreßverwaltung schreiben wollen, das ist bekannt. Aber was soll dieses Programm leisten? Profis verwenden hierzu ein Pflichtenheft, in dem genau verzeichnet ist, was das Programm können soll. Für unsere Adressverwaltung könnte dies etwa so aussehen:

1. Adressen erfassen
2. Adresse korrigieren und gegebenenfalls löschen
3. Eine bestimmte Adresse suchen

4. Adressen auf Diskette speichern
5. Adressen von Diskette einlesen
6. Programm verlassen

Als nächstes muß abgeklärt werden, welche Daten vom Programm erfasst werden sollen:

1. Name und Vorname
2. Strasse incl. Hausnummer
3. Postleitzahl und Wohnort
4. Telefonnummer
5. Geburtsdatum

Nachdem abgeklärt wäre, welche Funktionen das Programm enthalten und welche Daten damit erfaßt werden sollen, muß es noch anwenderfreundlich gestaltet werden. Dazu bietet man dem Benutzer nach dem Programmstart alle Funktionen an, die das Programm beinhaltet, und läßt ihn auswählen, welche Funktion er wünscht. Diese Auswahl nennt man im Fachjargon auch MENÜ. Und damit auch die Optik stimmt, realisieren wir den Bildschirmaufbau wie folgt:

- Kopfzeile (HEADER),
- zur Verfügung stehende Funktionen,
- Aufforderung an Benutzer eine Auswahl zu treffen

Nach dem Programmstart wird dieses Bild auf dem Monitor ausgegeben, und der Computer muß warten, bis eine Taste gedrückt wurde. Im nächsten Schritt ist zu überprüfen, ob diese Taste eine bestimmte Funktion auslösen soll oder ob eine falsche Taste gedrückt wurde. Diese muß ignoriert werden, damit das Programm nicht abstürzt. Kann der Taste eine Funktion zugewiesen werden, so ist diese auszuführen (Abbildung 2.1).

Die einzelnen Funktionen selbst legt man in Unterprogrammen (Module) ab, die über ein ON..GOSUB erreicht werden können. Welche Daten gespeichert werden sollen, darüber waren wir uns bereits im Klaren. Doch wie werden diese Daten am günstigsten

im Computer abgelegt? Dumme Frage, ein Array muß dafür herhalten. Die Dimension des Feldes wird in einer Variablen gespeichert, damit bei Bedarf einfach und schnell nachdimensioniert werden kann. Eine Dimension von 100 sollte für den Anfang genügen.

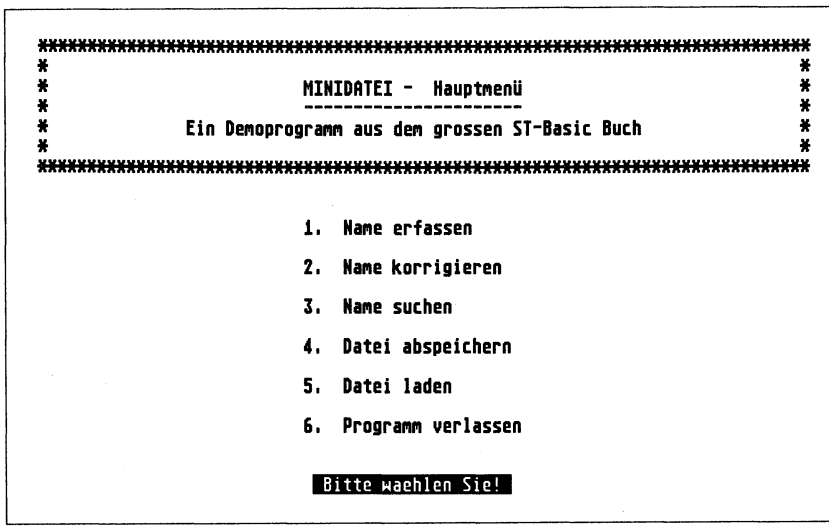



*Abb. 2.1: Das Hauptmenü der Minidatei*

Mit diesen Vorüberlegungen sollte die Umsetzung in ein ausführbares Programm keinerlei Schwierigkeiten mehr machen. Nach der Dimensionierung der String-Arrays, die üblicherweise am Programmbeginn erfolgt, wird das Menü aufgebaut. Dazu wird der Befehl PRINT verwendet bzw. ein naher Verwandter von PRINT, nämlich PRINT AT (sprich: Print Ät).

## PRINT AT

Print At ermöglicht die Ausgabe von Daten an der Bildschirmposition mit dem Koordinatenpaar (Zeile, Spalte). Die Syntax dafür lautet

```
PRINT @(Zeile,Spalte);<Text>
```

Über den Befehl PRINT gibt es nichts Neues sagen. Das Zeichen @ (Klammeraffe genannt) erreichen Sie durch gleichzeitiges Drücken der Tasten <Alternate> und <Ü>. Anschließend wird die Position angegeben, an der die Ausgabe von <Text> (durch ein Semikolon von der Positionsangabe getrennt) erfolgen soll. Die linke obere Ecke des Monitors besitzt dabei die Koordinaten (0,0), die rechte untere Ecke die Koordinaten (24,79).

```
PRINT @(0,1);"*"*78
```

schreibt demnach in die erste Zeile ab Spalte 2 einen aus 78 Sternchen bestehenden String. Mit PRINT At läßt sich jetzt der Bildschirmaufbau problemlos realisieren. Der Optik halber wird die in der untersten Zeile stehende Aufforderung "Bitte wählen Sie" invers dargestellt. Die inverse Darstellung übernimmt der VT-52 Emulator des Atari ST. Eingeschaltet wird sie mit dem Befehl

```
PRINT CHR$(27);"p"
```

während

```
PRINT CHR$(27);"q"
```

dafür sorgt, daß sie wieder abgeschaltet wird. Die in der Zeile 9 definierte Funktion SCREEN$(), der als Argument das entsprechende Zeichen übergeben wird, macht die ganze Sache etwas übersichtlicher.

Nachdem der Bildschirm aufgebaut ist, muß das Programm anhalten und auf die Eingabe des Benutzers warten. Man könnte jetzt auf die Idee kommen, dafür eine INPUT-Anweisung zu benutzen. Sicher eine Möglichkeit, aber es gibt einen eleganteren Weg, der hier auch prompt beschritten werden soll:

### INKEY$

INKEY$ liefert eine vier Byte lange Zeichenkette, die die Daten der zuletzt gedrückten Taste enthält. Dabei wartet die Funktion jedoch nicht auf einen Tastendruck wie INPUT. Wurde keine

Taste gedrückt, liefert INKEY$ als Ergebnis einen leeren String. Folglich muß INKEY$ in einer Schleife solange wiederholt werden, bis eine Taste gedrückt wurde:

```
REPEAT
  ' weist A$ das von INKEY$ gelieferte Ergebnis zu
  A$= INKEY$
UNTIL A$ <> ""' und zwar solange, bis Taste gedrückt
```

Sobald eine Taste gedrückt wurde, enthält A$ eine vier Byte lange Zeichenkette:

1.Byte: Bit 0: Rechte Shift-Taste
Bit 1: Linke Shift-Taste
Bit 2: Control
Bit 3: Alternate
Bit 4: Zustand von Caps Lock

2.Byte: Scancode der Taste

3.Byte: Nicht benutzt

4.Byte: ASCII-Code der Taste

Für uns ist im Augenblick lediglich das vierte Byte interessant, das mit

```
RIGHT$(A$,1)
```

isoliert werden kann. Doch Vorsicht! Solange keine Taste gedrückt wurde, enthält die Variable A$ einen Leerstring. Versucht man mit RIGHT$ das letzte Byte von A$ abzuschneiden, so führt dies unweigerlich zu einer Fehlermeldung, wenn in A$ noch keine Zeichenkette vorhanden ist. Andererseits läßt sich das Angenehme gleich mit dem Nützlichen kombinieren: die Schleife soll nur dann verlassen werden, falls der Benutzer eine korrekte Menüauswahl getroffen hat. Korrekt bedeutet in diesem Zusammenhang, daß eine Zifferntaste von 1 bis 6 gedrückt wurde. Die verbesserte Schleife:

```
REPEAT
```

```
    A$= INKEY$
    IF A$ <> "" THEN
       A$= RIGHT$(A$,1)
    ENDIF
  UNTIL A$>"0" AND A$ <"7"
```

Der Operator AND weist den Computer an, die Gesamtbedingung nur dann als wahr zu akzeptieren, wenn beide Teilbedingungen (A$ > 0, A$ < 7) erfüllt sind. Doch dazu später mehr.Zunächst werden der Variablen A$ die Daten der zuletzt gedrückten Taste zugewiesen. Wurde noch keine Taste betätigt, enthält A$ einen Leerstring. Die IF-Abfrage überprüft, ob schon eine Zeichenkette in A$ vorliegt und isoliert das für uns wertvolle vierte Byte, das anschließend zur Überprüfung des Abbruchkriteriums herangezogen wird. Die Schleife wird immer dann verlassen, wenn eine Taste von 1 bis 6 gedrückt wird. Daß Buchstaben aufgrund ihres ASCII-Wertes auch für Vergleiche herangezogen werden können, ist für Sie bereits Schnee von vorgestern.

A$ enthält nach Abbruch der Schleife die gedrückte Taste in Form eines 1 Byte langen Characters. Doch der hilft uns bei unserem nächsten Problem überhaupt nicht weiter: ON..GOSUB erwartet nämlich eine Zahl (!) von 1 bis 6, und nicht irgendeinen Buchstaben. Also umwandeln! Aber wie?

Mit VAL() (A = VAL(A$()) ginge es, aber das ist zu einfach! Kennen Sie noch einen zweiten Lösungsweg? Richtig, der ASCII-Code, der sich für jeden Buchstaben mit Hilfe der Funktion ASC() ermitteln läßt, muß ran! Wenn Sie sich die im Anhang dieses Buches abgedruckte ASCII-Tabelle einmal in einer ruhigen Minute zu Gemüte führen, werden Sie feststellen, daß die 1 den Code 49, die 2 den Wert 50, usw. zugeordnet bekommen hat.

Ermitteln wir von A$ den Code über ASC(A$) und subtrahieren von diesem Ergebnis den Wert 48, resultieren daraus - je nach gedrückter Taste - die Zahlen von 0 bis 9, die wir für unsere ON..GOSUB-Anweisung verwenden können. Wie gesagt, auch VAL() führt (in diesem Fall sogar schneller) zum Ziel, aber dann hätten Sie nicht so viel über den ASCII-Code gelernt.

Mehr als eine reine Formsache ist das Löschen der Variablen A vorm Schleifeneintritt. Wird dies unterlassen, geschehen gar wundersame Dinge, da die INKEY$-Schleife in Zukunft verlassen wird, ohne daß eine Taste gedrückt werden mußte! Probieren Sie es einmal spaßeshalber aus!

Die Menüauswahl selbst ist in eine Schleifenstruktur eingebettet, die nur dann beendet wird, falls die Taste 6 gedrückt wurde. Die Struktur in "Pseudocode":

```
REPEAT
  <Bildschirmaufbau>
  REPEAT
    <Taste gedrückt?>
  UNTIL <Taste richtige gedrückt>
    <ON <Taste> GOSUB ......>
UNTIL <Programmende>
```

Ach ja, vor der INKEY$-Schleife wird noch der Cursor ausgeschaltet, damit er nicht irgendwo (störenderweise) blinkt. Der Ordnung halber wird er dann nach dem Schleifendurchlauf wieder aktiviert (Der noch kommende INPUT-Befehl soll ruhig seinen blinkenden Cursor haben). Der Menüaufbau wäre abgeschlossen, jetzt kommen die einzelnen Funktionen des Programmes an die Reihe:

Zuerst einmal Adressen erfassen. Auf dem Monitor befindet sich im Augenblick noch die Menüauswahl, die mit einem CLS entfernt wird. Im nächsten Schritt wird ein Formular auf dem Bildschirm aufgebaut, in das die geforderten Daten eingegeben werden werden können. Ist ein Datensatz komplett erfaßt, kann der Benutzer wählen, wie er fortfahren möchte:

1. Eine weitere Adresse eingeben.
2. Eingegebene Adresse korrigieren.
3. In das Hauptmenü zurückspringen.

Wie lassen sich nun diese Funktionen am zweckmäßigsten auf einzelne Prozeduren verteilen? Der Bildschirm wird, da auch die Menüpunkte Adressen korrigieren bzw. Adresse suchen auf die gleiche Maske zurückgreifen sollen, in einer eigenen Prozedur aufgebaut. Auch eine universell gestaltete Eingabeprozedur hat ihre Vorteile: Der Menüpunkt "Adressen korrigieren" kann sie ebenfalls benutzen. Da die Auswahl des Untermenüs jedoch von Fall zu Fall verschieden sein wird, hat eine Abfrage nach der weiteren Vorgehensweise des Benutzers in der Eingabeprozedur nichts mehr zu suchen, sie ist vielmehr eine Sache des eigentlichen Unterprogramms.

Sinn dieser Vorüberlegungen ist, die anfallenden Arbeiten auf verschiedene Prozeduren zu verteilen und somit das Programm nicht unnötig in die Länge zu ziehen. Fassen wir die bisher geklärten Aufgaben des Menüpunktes noch einmal zusammen, um nicht den Überblick zu verlieren:

```
******* Unterprogramm: Adressen erfassen ********

        <Maske aufbauen>
  <Eingabe einer Adresse ermöglichen>
   <Weitere Vorgehensweise abfragen>
```

Diese Grobstrukturierung muß jetzt noch Schritt für Schritt verfeinert werden, ehe mit der eigentlichen Programmierung begonnen werden kann. Eine Prozedur zur Darstellung der Maske zu schreiben, sollte uns vor keine größeren Probleme mehr stellen. Ein Punkt, der bei der Programmierung dieser Funktion gleich mit berücksichtigt werden sollte, ist die Ausgabe einer Überschrift. Dazu wird der Prozedur bei ihrem Aufruf ein (lokaler) String übergeben, in dem die Überschrift enthalten ist. Für's Auge wird er dann genau in die Bildschirmmitte gesetzt:

```
PRINT @(Zeile, (Zeilenlänge-Stringlänge)/2);"....."
```

Dieser Algorithmus zentriert den String. Zunächst wird die Länge der auszugebenden Zeichenkette von der maximalen Zeilenlänge abgezogen. Übrig bleibt die Anzahl der nicht genutzten Zeichen in dieser Zeile. Durch die Division mit 2 werden diese zwischen dem linken und dem rechten Bildschirmrand aufgeteilt, so daß schließlich der auszugebende String genau in der Bildschirmmitte erscheint.

Schwieriger wird die Prozedur zur Adreßeingabe, da hierzu gleich eine ganze Menge Neuerungen benötigt werden. Aber alles schön der Reihe nach!

### Formatierte Eingabe

INPUT bzw. LINE INPUT kennen Sie bereits. INKEY$ ist in diesem Kapitel neu hinzugekommen. Und den absoluten Profibefehl zur Dateneingabe möchte ich Ihnen jetzt servieren: INPUT USING.

Ja, Sie lesen richtig! Bei INPUT USING handelt es sich um einen Profibefehl, der vor allem für die Verwendung in kommerziellen Programme gedacht ist. Wenn seine Handhabung auch etwas komplizierter als die des INPUT bzw. INKEY$ erscheint, soll das niemanden abschrecken! Es ist bekanntlich noch kein Meister vom Himmel gefallen. Und zudem sollten Sie mittlerweile über genügend Hintergrundwissen verfügen, um diesen Befehl zu verstehen.

```
INPUT @(Zeile,Spalte);<Variable>
```

kenne ich schon, werden Sie jetzt vielleicht erfreut feststellen! Das ist doch die Syntax des altbekannten INPUT-Befehls. Und in der Tat, bisher unterscheiden sich beide Befehle noch überhaupt nicht. Doch was jetzt kommt, ist neu:

```
USING <Steuerstring>
```

USING ist ein Befehlswort, das halt dastehen muß. Interessanter wird es beim <Steuerstring>. Er zeichnet dafür verantwortlich, daß bei der Eingabe nur ganz bestimmte Zeichen angenommen

werden. Zeichen, die im <Steuerstring> nicht zugelassen sind, werden vom Computer ignoriert. Soll z.B. die Eingabe einer Telefonnummer erfolgen, so haben Buchstaben darin absolut nichts zu suchen, während Zahlen bei der Erfassung von Namen ignoriert werden müssen. Folgende Zeichen sind im <Steuerstring> von Bedeutung:

| Zeichen | bewirkt ein Zulassen von |
|---|---|
| 0 | Ziffern (0-9) |
| a | Buchstaben (incl. Umlaute) falls Modus "Deutsch" |
| % | Sonderzeichen (excl. Umlaute) falls Modus "Deutsch" |
| ^ | Control-Zeichen |
| +<Zeichen> | <Zeichen> zulassen |
| -<Zeichen> | <Zeichen> verbieten |
| c<Zeichen 1><Zeichen 2> | <Zeichen 1> durch <Zeichen 2> ersetzen |
| u | alle Buchstaben in Großschrift |
| l | alle Buchstaben in Kleinschrift |

Durch Kombination dieser Zeichen wird der Steuerstring zusammengebaut. Aber dies ist nicht weiter kompliziert:

*"0"* Bei der Eingabe sind alle Ziffern von 0 bis 9 zugelassen.

*"0a"* Alle Ziffern und Buchstaben sind zugelassen.

*"0au"* Alle Ziffern und Buchstaben sind zugelassen, Buchstaben werden automatisch in Großschrift umgewandelt.

*"0a+.u"* Wie oben, zusätzlich ist jedoch die Eingabe eines Punktes gestattet.

*"0a+.-au"* Der Buchstabe a ist ab sofort nicht mehr zugelassen.

*"+a+b+c"* Bei der Eingabe werden nur die Buchstaben a, b und c angenommen.

*"+a+b+c+ "* Wie oben, jedoch zusätzlich das Leerzeichen (Space).

*"+a+b+c+ +-+.c.-"* Zusätzlich sind hier noch die beiden Zeichen -

und . zugelassen. Ein Drücken von . bewirkt jedoch, daß ein - in der Eingabe erscheint, da mit c.- der Punkt in einen Strichpunkt umgewandelt wird.

Es bleibt sich letztendlich egal, ob die Buchstaben innerhalb des Steuerstring in Groß- oder in Kleinschrift eingegeben werden. Der INPUT-Befehl wird mit der <Return>-Taste verlassen. Auch INPUT USING wird mit dieser Taste beendet. Zusätzlich können noch weitere Bedingungen im Steuerstring angeben werden, mit denen die Eingabe abgebrochen werden soll:

| Zeichen | Abbruch bei |
|---|---|
| x<ASCII> | Taste mit dem ASCII-Wert <ASCII> |
| s<Scan> | Taste mit dem Code <Scan> |
| < | Überschreitung des linken Randes |
| > | Überschreitung des rechten Randes |

Soll die Eingabe bei Betätigung der Taste <E> beendet werden, kann dies auf zweierlei Art und Weise angegeben werden:

1. Über den ASCII-Code dieser Taste: 101. Der Steuerstring müßte dann so aussehen:

```
"ax"+CHR$(101) bzw. "axe"
```

Die CHR$()-Darstellung des ASCII-Wertes ist immer dann erforderlich, wenn das Zeichen, das den Abbruch auslösen soll, nicht direkt eingegeben werden kann. Ein Beispiel dafür wäre die Taste <ESC>, die den ASCII-Wert 27 besitzt:

```
"x"+CHR$(27)
```

2. Über den Scancode dieser Taste. Dazu ist wieder eine Erklärung fällig: Neben dem ASCII-Code gibt es noch den Scancode, eine Tastennummer. Jede Taste besitzt einen ganz bestimmten (Scan-)Code, anhand dessen sich die gedrückte Taste eindeutig identifizieren läßt. Dies ist vor allem für Tasten, denen kein ASCII-Code (z.B. Cursor links, rechts, usw.)

zugeordnet ist, wichtig. Wir werden später noch einmal darauf zurückkommen. Auch INKEY$ liefert den Scancode der gedrückten Taste, und zwar im zweiten Byte.

Bemerkenswert ist noch, daß die Zifferntasten des separaten Zifferblocks eigene Tastennummern besitzen. Somit kann streng zwischen den Ziffern des Hauptfeldes und denen des Zifferblocks unterschieden werden. Auch für die Scancodes finden Sie im Anhang eine Tabelle.

Doch zurück zum eigentlichen Problem: Die Taste <E> besitzt den Scancode $12 bzw. in dezimaler Notation 18. Um einen Abbruch über diese Taste zu ermöglichen, muß der Steuerstring lauten:

```
"s"+CHR$(18)
```

So, den Steuerstring können wir mit diesem Wissen bereits zusammenbauen. Überlegen wir uns also, welche Eingaben für die einzelnen Bestandteile des Datensatzes erlaubt werden sollen:

NAME

Zur Eingabe des Familiennamens werden auf alle Fälle einmal Buchstaben benötigt. Umlaute können u.U. auch auftreten. Deshalb muß am Programmanfang auf den Modus Deutsch mit MODE "D" umgeschaltet werden. Für Doppelnamen (Bauer-Reichel) sollte der Trennstrich möglich sein. Namenszusätze (Titel) müssen durch ein Leerzeichen (Space) abgesetzt werden können. Berücksichtigt man ferner, daß sämtliche Buchstaben gleich bei der Eingabe in Großschrift erscheinen sollen, so muß der Steuer-String für die Eingabe des Namens "a+ +-u" lauten. Für den Vornamen kann der gleiche Steuer-String benutzt werden, die Umwandlung in Großbuchstaben entfällt jedoch: "a+-+ ".

STRASSE

Zur Eingabe des Straßennamens werden wieder Buchstaben, Ziffern für die Hausnummern, das Leerezichen sowie der Punkt benötigt: "0a+ +-+."

POSTLEITZAHL

Die Postleitzahl besteht aus genau vier Ziffern. Sind alle eingegeben, so soll der Computer den INPUT USING-Befehl verlassen. Dazu dient das > (Exit rechte Randüberschreitung) im Steuer-String: "0>"

ORT

Zur Eingabe des Ortes sind Buchstaben, Zahlen für den evtl. vorhandenen Zustellbezirk (München22), sowie die beiden Trennstriche - (Passau-Neustift) und / (Ingolstadt/Donau) von Nöten: "a0+/+-".

TELEFON

Die Eingabe der Telefonnummer erfordert Ziffern, sowie die beiden Trennstriche zur Abrenzung der Vorwahl. Das Zeichen - wird jedoch - um auch diese Funktion einmal zu verwenden - in den Schrägstrich gewandelt: "0+-+/c-/".

GEBURTSDATUM

Last, not least soll noch das Geburtsdatum erfasst werden. Dazu werden lediglich Ziffern, sowie zur Trennung des Monats und des Jahres Punkte benötigt: "0+."

Der nächste Parameter, den INPUT USING hinter dem Steuerstring - durch ein Komma getrennt - erwartet, ist die <Return-Variable>. In ihr wird vom Interpreter festgehalten, welche Taste das Verlassen der Eingabe bewirkt hat. Sie enthält den Wert 0, falls Return, -1, falls eine rechte Randüberschreitung und -2, falls eine Überschreitung des linken Randes zum Ende der Eingabe geführt hat.

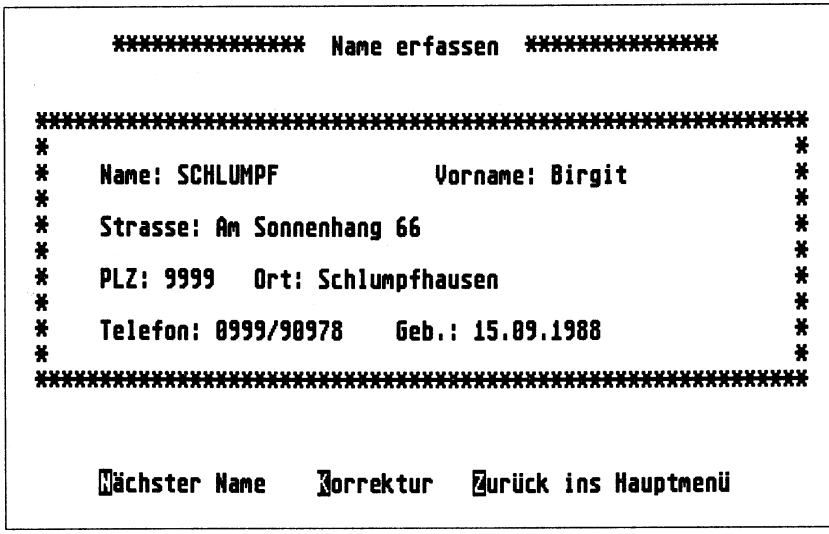



*Abb. 2.2: Maske für Adressen erfassen*

Ansonsten sind - ähnlich wie auch bei INKEY$ - die Shiftkeys, der Scancode, die letzte Cursorposition sowie der ASCII-Wert der Taste, die zum Abbruch der Eingabe geführten hat, darin enthalten.

Die maximale Zeichenlänge, die das Eingabefeld erhalten soll, läßt sich ebenfalls angeben. Erfolgt keine Angabe, beträgt die Länge stets 255 Zeichen. Für unsere Adreßverwaltung sollen jedoch folgende Eingabelängen gelten:

| DatenMax. | Eingabelänge |
|---|---|
| Name | 15 |
| Vorname | 15 |
| Straße | 32 |
| Postleitzahl | 4 |
| Ort | 30 |
| Telefon | 11 |
| Geburtsdatum | 10 |

Das Eingabefeld wird noch in seiner gesamten Länge mit einem

Füllzeichen vorbelegt. Soll nicht der Unterstrich _ dazu verwendet werden, ist der Füllzeichencode als ASCII-Wert (wieder durch Komma getrennt) an die maximale Eingabelänge zu hängen. In unserem Fall erübrigt sich die Angabe, da der Unterstrich verwendet werden soll.

Der letzte Parameter, der bei INPUT USING angegeben werden kann, ist die Cursorpositions-Variable. Wird sie nicht extra aufgeführt, so steht der Cursor beim Aufruf des INPUT USING stets an der ersten Eingabeposition. Damit wäre die Syntax des INPUT USING komplett besprochen. Noch einmal das ganze Ungetüm zum Mitschreiben:

```
INPUT [@(y,x);]["Text";]<Eingabe-STRING-Variable> USING

[<Steuer-String>],[<Länge>],[<Füllzeichencode>],[ <Cursorpositions-
Variable> ]
```

## Logische Verknüpfungen

In ST-BASIC existieren eine ganze Reihe von logischen Operatoren, mit denen sich Bool'sche Operationen (Teilgebiet der Mathematik) ausführen lassen.

### AND

Dieser Operator sorgt einmal dafür, daß zwei Teilbedingungen erfüllt sein müssen, damit die Gesamtbedingung ebenfalls erfüllt ist:

```
IF    X < 3     AND     X > 0     THEN ....
   Bedingung 1       Bedingung 2
```

Die Bedingung hinter dem IF ist nur dann wahr, wenn beide Teilbedingungen wahr sind. Ist nur eine Teilaussage falsch, wird unweigerlich auch die Gesamtaussage falsch. In dieser Funktion wurde der Operator AND schon einmal eingesetzt: in der Menü-Auswahlschleife. Doch dies ist wieder einmal nur die halbe Wahrheit. Der Operator AND beinhaltet nämlich noch eine zweite Funktion: die bitweise AND-Verknüpfung zweier Zahlen miteinander. Angenommen die beiden Variablen A und B ent-

halten die Zahlen 31 und 43, oder in Dualschreibweise ausgedrückt:

```
A: %00011111
B: %00101011
```

Bit für Bit wird jetzt logisch nach folgender Tabelle undiert:

| A | B | A AND B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Als Ergebnis liefert diese Verknüfung nur dann eine 1, wenn beide verknüpften Bits eine 1 enthielten, ansonsten ist das Resultat der Operation stets 0. Auf die beiden Zahlen angewendet, bedeutet dies:

```
A:          %00011111
B:          %00101011
-------------------------
A AND B:    %00001011
```

In Dezimalnotation lautet das Ergebnis der (bitweisen) AND-Verknüpfung zwischen den Zahlen 31 und 43 11. Wofür läßt sich diese AND-Verknüfung sinnvoll einsetzten? Sie wird immer dann herangezogen, wenn bestimmte Bits einer Zahl ausmaskiert, also auf den Wert 0 gesetzt werden müssen. Ein Beispiel macht dies gleich um vieles klarer:

In einer Long-Integer, die sich bekanntlich aus vier Bytes zusammensetzt, interessiert nur der Inhalt des zweiten Bytes. Die übrigen Bits werden nicht benötigt. Also undiert man diese Variable mit einer Konstanten, in der alle Bits des zweiten Bytes den Wert 1 besitzen. Durch das logische UND erhalten die Bits des ersten, dritten und vierten Byte den Wert 0 - ungeachtet ihres vorherigen Inhalts - zugewiesen, sie wurden ausmaskiert. Der Wert des zweiten Bytes bleibt jedoch unverändert!

```
A AND 11111111000000000000000
```

oder in der etwas lesbareren Hexadezimalnotation:

```
A AND $FF0000
```

Diese Eigenschaft der UND-Verknüfung werden Sie noch bei der Überprüfung der Return-Variable des INPUT USING zu schätzen lernen.

## OR

Der nächste Vertreter der logischen Verknüfungen ist das ODER. Werden zwei Teilaussagen miteinander odiert, so ist die Gesamtaussage bereits dann wahr, wenn nur eine der beiden Teilaussagen wahr war.

```
IF A > 2 OR B < 30 THEN ....
```

Bei AND müssen beide Teilbedingungen erfüllt sein, für OR genügt eine wahre Teilaussage, damit als Ergebnis die Aussage wahr resultiert. Auch eine bitweise Odierung zweier Zahlen ist möglich. Während AND jedoch in der Hauptsache dazu benutzt wird, bestimmte Bits auszumaskieren, so dient OR dazu, bestimmte Bits zu setzen:

| A | B | A OR B |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 1 |

Das aus dieser Verknüfung resultierende Bit enthält immer dann den Wert 1, wenn nur eines der beiden Bits eine 1 enthielt. Verknüpfen wir wieder die beiden Zahlen 31 und 43 miteinander, diesmal jedoch mit OR:

```
A:        %00011111
B:        %00101011
-------------------
A OR B:   %00111111
```

In Dezimalnotation lautet das Ergebnis der Verknüfung 63.

## NOT

NOT dreht Ihnen zwar nicht das Wort im Munde, dafür aber die Bits im Computer um. Eine wahre Aussage wird mittels der Negation nämlich zu einer falschen und eine falsche zu einer wahren Aussage:

```
IF NOT (A=B) THEN ...
```

entspricht der Notation

```
IF A <> B THEN ...
```

Die Aussage ist nur dann erfüllt, wenn die Bedingung A=B nicht erfüllt ist, wenn die Variablen A und B also unterschiedliche Werte repräsentieren. Auch die bitweise Negation einer Zahl ist möglich:

```
A   NOT A
-----------
1     0
0     1
```

Enthält B den Wert 43, so liefert die Zuweisung

```
A = NOT B
```

in A den Wert

```
%00101011
NOT %11010100
```

oder in dezimaler Schreibweise ausgedrückt 212. Es handelt sich dabei um das sogenannte Zweierkomplement einer Zahl. Subtrahieren Sie von der mit 8 Bits maximal darstellbaren Zahl 255 die Zahl 43, so erhalten Sie ebenfalls das Ergebnis 212.

```
255 - A = NOT A (Zweierkomplement)
```

## XOR

Das Ergebnis der Exklusiv-Oder-Verknüpfung liefert nur dann eine wahre Aussage, wenn eine Bedingung erfüllt, die zweite

Bedingung jedoch nicht erfüllt ist. Sind beide Aussagen erfüllt oder nicht erfüllt, lautet der Wahrheitswert dieser Verknüpfungsform jeweils falsch. Auch bitweise kann XOR verknüpft werden. Das resultierende Bit ist nur dann eine 1, wenn ein Bit den Wert 0 und das andere Bit den Wert 1 enthielt:

| A | B | A XOR B |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 0 |

Mit den logischen Operatoren, die Sie bisher kennengelernt haben, sind Sie für die meißten Fälle bestens gerüstet. Dennoch existieren in ST-BASIC noch weitere logische Operatoren, auf die ich an dieser Stelle jedoch nicht weitere eingehen möchte, da sie im wesentlichen nur eine Kombination aus verschiedenen Operatoren darstellen. Für nähere Details empfehle ich Ihnen das Handbuch zu Rate zu ziehen.

### "Bitgeschiebe"

Die folgenden beiden Befehl sind relativ schnell erklärt. Ich habe zwar im Augenblick keinerlei Verwendung für sie, dennoch möchte ich sie der Vollständigkeit halber an dieser Stelle vorstellen.

### SHR

Was geschieht, wenn man die Bits der Zahl 56 (%00111000) um eine Position nach rechts verschiebt?

```
%00111000 ---um 1 Bit nach rechts---> %00011100
```

Der Wert halbiert sich! Die Funktion SHR (SHift Right) verschiebt die Bits Zahl A um N Positionen nach rechts:

```
Ergebnis = A SHR N
```

Dies entspricht einer n-maligen (vorzeichenlosen) Division der Zahl durch 2. Im Gegensatz zur normalen Division wird diese Funktion jedoch wesentlich schneller ausgeführt, da ein Computer nichts schneller erledigen kann, als Bits zu verknüpfen oder zu verschieben.

### SHL

Während SHR eine Zahl halbiert, wird sie mit der Funktion SHL verdoppelt. SHL (SHift Left) schiebt die Bits einer Zahl A um genau N-Positionen nach links:

```
Ergebnis = A SHR N
```

A SHR N entspricht der (vorzeichenlosen) Multiplikation mit

```
Ergebnis = A * 2^N
```

### Formatierte Ausgabe mit Print

Ähnlich wie mit INPUT USING Daten formatiert erfasst werden können, dient PRINT USING zur formatierten Ausgabe von Daten. Die Syntax für diesen Befehl lautet:

```
PRINT USING <"Formatmaske">,<Ausgabe(variable)>
```

Zu beachten ist dabei, daß nur Zahlen ausgegeben werden können. Für eine Stringvariable kann dieser Befehl nicht verwendet werden. Möchten Sie dagegen Zahlen sauber untereinander (Rechtsbündig und Komma unter Komma ...) ausgeben, eignet sich PRINT USING hervorragend. Sehen wir uns die <Formatmaske> einmahl genauer an:

```
PRINT USING "###,##",2.8
```

ergibt auf dem Bildschirm:

```
2,80
```

Das Zeichen # in der Formatmaske dient als Platzhalter für eine Ziffer. Für jede Ziffernstelle einer Zahl muß ein solcher Platz-

halter in der Formatmaske vorhanden sein. Kommazahlen werden dergestalt in die Maske eingepaßt, daß der Vorkomma-Anteil links, der Nachkomma-Anteil rechts des Kommas bzw. Punktes in der Formatmaske stehen. Die restlichen, nicht benutzten Nachkommastellen werden noch mit Nullen aufgefüllt, fertig.

Folgende Zeichen besitzen eine Bedeutung in der Formatmaske:

| Zeichen | Bedeutung |
|---|---|
| # | Platzhalter für eine Ziffer. |
| . | Dezimalpunkt an dieser Stelle ausgeben. |
| , | Dezimalkomma an dieser Stelle ausgeben. |
| , | später . Dezimalpunkt an dieser Stelle ausgeben, Tausender durch Komma trennen. |
| . | später , Dezimalkomma an dieser Stelle ausgeben, tausender durch Punkt trennen. |
| ,,, | Tausender durch Komma trennen, Nachkommastellen unterdrücken. Tausender durch Punkt trennen, Nachkommastellen unterdrücken. |
| - | bei negativen Zahlen, Minuszeichen an dieser Stelle ausgeben. |
| + | Vorzeichen (auch +) an dieser Stelle ausgeben, Ausnahme zu +/-: Ein '+' oder '-', das direkt vor dem ersten '#', '*', '.', oder ',' steht, bewirkt die Ausgabe des Vorzeichens direkt vor der ersten gültigen Stelle |
| *<Zeichen> | Füllt die Ausgabemaske vorne mit dem Zeichen <Zeichen> auf, soweit diese Stellen nicht benutzt werden. Nicht benutzte Stellen ('#') werden ansonsten mit Leerzeichen aufgefüllt. Der Unterstrich '_' darf nicht als Füllzeichen benutzt werden. |
| _<Zeichen> | gibt das Zeichen <Zeichen> aus, auch wenn es sich dabei um ein geschütztes Zeichen mit einer Bedeutung für den Formatstring handelt. |
| ^^^ | Exponent an dieser Stelle ausgeben. |

Die übrigen, hier nicht aufgeführten Zeichen, werden so ausgegeben, wie sie sind.

```
PRINT USING ".######,## DM",25875.57
```

ergibt

```
25.875,57 DM
```

auf dem Monitor. Der Punkt dient als Tausender-Separator, das Komma trennt die Nachkommastellen ab.

```
PRINT USING "*0.######,## DM",25875.57
```

ergibt

```
000025.875,57 DM
```

Die führenden, nicht belegten Stellen werden aufgrund der Füllzeichendefinition *0 mit Nullen aufgefüllt. Sie sehen dabei auch, daß auch die Füllzeichendefinition selbst sowie der Tausenderseparator als Ziffernplatzhalter eingesetzt werden können. Das Vorzeichen (+ bzw. -) beansprucht für sich ebenfalls eine Ziffernstelle, falls es nicht durch + bzw. - an einer anderen Stelle innerhalb des Formatstrings definiert ist, auch dann, wenn kein Vorzeichen ausgegeben wird, da es sich um eine positive Zahl handelt.

Mit der Definition

```
PRINT USING "####.##",A!
```

können maximal Zahlen mit drei Vorkommastellen (< 1000) auf den Monitor gebannt werden, da der erste Platzhalter das Vorzeichen aufnimmt. Wird im Formatstring das Kennzeichen jedoch an eine andere Stelle gesetzt. PRINT USING ermöglicht auch die Ausgabe einer Zahl im wissenschaftlichen Format mit Angabe eines Exponenten. Wird im Formatstring dagegen kein Exponent definiert (keine ^ vorhanden), so gibt ST-BASIC die Zahl auf alle Fälle ohne Exponenten aus.

Für den Formatstring gelten noch ein paar Einschränkungen: Insgesamt dürfen maximal 30 Platzhalter (#) verwendet werden, der Unterstrich (_) darf nicht als Füllzeichen verwendet werden und die Länge des Formatstrings darf 253 Zeichen nicht überschreiten. Ein fehlerhafter Formatstring führt zu der Fehlermeldung Syntax Error.

### USING

USING <Formatstring> ist auch ohne einen PRINT-Befehl möglich. Die auf diese Weise definierte Formatmaske wird bei allen folgenden PRINT, LPRINT, PRINT#, und der Funktion STR$ berücksichtigt.

Nach diesem kleinen Ausflug wieder zurück zur Minidatei: Die Prozedur zum Erfassen der Daten muß geschrieben werden. Um für den Benutzer die Eingabe der geforderten Daten zu erleichern, soll mit Hilfe der Cursortasten von einer Eingabeposition zur anderen gesprungen werden können. Die Steuerstrings für die einzelnen Eingaben haben wir bereits definiert. Daran anhängen müssen wir noch die Erlaubnis, daß die Eingabe bei Betätigung einer der beiden Cursortasten <nach oben> oder <nach untenrunter> verlassen werden darf. EXIT by ASCII scheidet für diese Problemstellung aus, da für Cursortasten keinerlei ASCII-Werte reserviert sind. Sie besitzen jedoch einen Scancode:

| Scancode | Bedeutung |
|---|---|
| 72 | Cursor hoch |
| 80 | Cursor runter |

Der Befehl zum Verlassen der Eingabe aufgrund eines bestimmten Scancodes lautet s, gefolgt von dem entsprechenden Code, der im 1-Byte-Format vorliegen muß. Dazu verhilft uns wieder die CHR$()-Funktion. Dieser String ist für alle INPUT USING gleich, und wird deshalb in einer eigenen Zeichenkette abgelegt, die per Stringaddition jeweils an die bereits erstellten Steuerstrings gehängt wird.

```
BACK$="s"+CHR$(72)+"s"+CHR$(80)
```

Die Eingabe wird also bei Betätigung einer der beiden Cursortasten verlassen, in der Return-Variable findet sich dann unter anderem auch der Scancode der Taste, die zu diesem Abbruch geführt hat. Jetzt muß noch festgestellt werden, welche Taste zum Verlassen des INPUT geführt hat.

Relevant ist hier lediglich das zweite Byte (von links nach rechts betrachtet), der vier Bytes langen Return-Variable. Die übrigen Bits müssen weg, um eine IF-Abfrage nach dem Scancode einbauen zu können. Dazu verhilft uns das logische UND, mit dessen Hilfe alle anderen Bits ausmaskiert, d.h. auf den Wert 0 gesetzt werden können. Der Aufbau der Returnvariable noch einmal im einzelnen:

| 1. Byte | 2. Byte | 3. Byte | 4. Byte |
|---|---|---|---|
| Shiftkeys | Scancode | Cursorposition | ASCII-Code |

Die einzelnen Bits des ersten, dritten und vierten Bytes müssen bei der logischen Undierung auf Null gesetzt werden, Byte zwei wird mit acht gesetzten Bits (entspricht dem Wert 255 in Dezimalnotation bzw. $FF in Hexadezima lschreibweise) undiert:

| | 1. Byte | 2. Byte | 3. Byte | 4. Byte |
|---|---|---|---|---|
| AND | 00 | 255 | 00 | 00 |
| bzw. | | | | |
| AND | $00 | $FF | $00 | $00 |

Die Unterroutine selbst wird von oben nach unten abgearbeitet. Mit <Return> bzw. <Cursor nach unten> gelangt der Benutzer somit automatisch in das nächste Eingabefeld, lediglich bei Betätigung der <Cursor nach unten>-Taste muß gewaltsam um eine Eingabeposition hochgesprungen werden. Der Scancode für <Cursor hoch> beträgt 72 bzw. in Hexadezimalnotation $48. Hexadezimal deshalb, da mit dieser Schreibweise das 2.Byte leichter erreicht werden kann. In Dezimalnotation müßte die gesamte Zahl umgerechnet werden. Die Abfrage nach der <Cursor hoch> Taste lautet dann folglich:

```
IF (<Returnvariable> AND $FF) = $480000 THEN
   <... um ein Eingabefeld hochspringen>
ENDIF
```

Vor jedes INPUT USING wird eine Marke gesetzt, die dann über ein GOTO angesprungen werden kann. Eine Kleinigkeit sollte noch berücksichtigt werden:

Erfolgt eine Betätigung der <Cursor hoch>-Taste, während gerade das erste Datenfeld (Name) bedient wird, soll der Cursor in das unterste Eingabefeld (Geburtsdatum) springen, vom untersten soll mit <Cursor runter> in das erste Eingabefeld gesprungen werden. Damit wäre auch die Eingabeprozedur fertiggestellt. Um gleich zwei Fliegen mit einer Klappe zu schlagen, wird ihr eine (lokale) Variable als Parameter übergeben, die bestimmt in welchen Eintrag des Feldes die erfaßten Daten abgelegt werden sollen. Wird die Eingaberoutine vom Unterprogramm "Adressen korrigieren" angesprungen, stellt dieser Parameter dann die Nummer des zu korrigierenden Eintrages dar.

Doch in welchem Eintrag des Arrays soll ein neuer Datensatz abgelegt werden? Ganz einfach, in das nächste freie, d.h. nicht schon mit einer Adresse belegte Feld. Eine Schleife erhöht eine Zählvariable solange, bis ein freier Eintrag im Array gefunden ist. Halt! Was geschieht, wenn das gesamte Variablenfeld bereits bis zum Kragen vollgepackt, also kein freier Eintrag mehr vorhanden ist?

Dann verabschiedet sich der Computer! Also heißt es, diesen Fehler abzufangen: Eine Erfassung der Daten ist nur solange möglich, wie freie Einträge im Array vorhanden und die Gesamtgröße (aus der Dimensionierung ersichtlich und in der Variablen <Groesse> festgehalten) noch nicht überschritten ist:

```
<Zaehler> = 1
WHILE <Array(<Zaehler>)> <> "" AND <Zaehler> < <Groesse>
   <Zaehler> = <Zaehler> + 1
WEND
```

Vor Schleifeneintritt wird wird eine Zählvariable namens <Zaehler> auf den Wert 1 gesetzt, dies entspricht dem zweiten Eintrag im Array. Der erste Eintrag, mit dem Feldindex 0, bleibt unbenutzt. In ihm werden später die Daten erfaßt, nach denen das Feld durchsucht werden soll. Anschließend wird in der abweisenden WHILE-Schleife geprüft, ob dieser Eintrag

schon belegt ist. Gleichzeitig darf das Feldende (<Groesse>) noch nicht erreicht sein. Sind beide Bedingungen erfüllt, wird der Zaehler um 1 erhöht, der nächste Eintrag wird überprüft. Dies wiederholt sich solange, bis ein freier Eintrag im Array gefunden, oder das gesamte Feld bereits durchsucht wurde. Wird kein freier Datensatz im Feld ermittelt, so gibt das Programm den letzten Eintrag - der zwar schon belegt ist, aber was soll man machen? - auf dem Monitor aus.

Der nächste Menüpunkt lautet "Korrektur". An Korrekturarbeiten können zwei Dinge anfallen:

- Tippfehler in einem Datensatz ausbessern
- kompletten Datensatz aus dem Array entfernen

Es wäre ein äußerst komfortabler Zug des Programms, nach einem bestimmten Eintrag suchen zu lassen, der dann abgeändert werden könnte. Ich habe mich allerdings für eine etwas andere Form der Korrektur entschieden.

Der Benutzer erhält den ersten Eintrag im Array vorgesetzt, und kann dann das gesamte Feld durchblättern. Mit der Taste <K> wird die Prozedur Erfassen angesprungen, eine Korrektur ist möglich. Mit <E> kann der soeben auf dem Bildschirm stehende Datensatz getilgt werden. Um für den Benutzer die Auswahl zu erleichern, wurden zwei Feinheiten in das Programm eingebaut:

- In der unteren Bildschirmhälfte wird ein Untermenü angezeigt, anhand dessen die weitere Programmkontrolle erfolgt.

- Der erste (invers dargestellte) Buchstabe wählt die entsprechende Funktion.

- Damit die Funktion auch ausgeführt wird, wenn der Benutzer zwar den richtigen Buchstaben, jedoch in Großschrift statt in Kleinschreibweise, getippt hat, wandelt das Programm den Character (um auch diesen Namen wieder einmal zu verwenden) gleich mit UPPER$() in Großschrift um.

Das Blättern innerhalb des Arrays müßte eigentlich verständlich sein. Es muß lediglich darauf geachtet werden, daß die untere Grenze (erster Datensatz mit Index 1), sowie die obere Grenze (Datensatz mit Index Groesse bzw. wenn Feld noch freie Einträge enthält, der letzte belegte Datensatz) nicht unter- bzw. überschritten werden.

Interessannter wird das Eliminieren eines Eintrages aus dem Array. Theoretisch können zwei Situationen auftreten:

- Der letzte Datensatz des Feldes soll entfernt werden. Dies ist nicht weiter tragisch, man ersetzt in diesem Fall einfach die Elemente mit dem entsprechenden Index durch einen Leerstring.

- Schwieriger wird es, wenn (was der Normalfall sein dürfte) sich der zu löschende Datensatz inmitten des Feldes befindet: Damit keine leeren Einträge innerhalb des Variablenfeldes entstehen (täte unserer Funktion Blättern gar nicht gut!), müssen die folgenden Einträge ab dem zu löschenden Datensatz um eine Indexposition nach unten verfrachtet werden. Der letzte Eintrag mit Index n ist dann in diesem Feld zweimal vorhanden (nämlich an Position n und an Position n-1), und kann einmal entfernt werden (logischerweise aus Position n, weil andernfalls wieder ein leerer Eintrag innerhalb der Liste entstehen würde!). Da dies etwas kompliziert erscheint, vergleichen Sie bitte Abbildung 2.3.

Enthält die Liste nur ein einziges Element das entfernt werden soll, stellt dies lediglich einen Spezialfall der ersten Variante dar. In einer einelementigen Liste ist das Element nämlich gleichzeitig der erste und letzte Datensatz. Und letzte Datensatz kann aus der Liste entfernt werden.

Der letzte (vorläufig) noch verbleibende Menü-Eintrag ist das Auffinden eines bestimmten Namens innerhalb der Liste. Dazu wird über ein eigenständiges INPUT USING der Name eingelesen, in das Array an Indexposition 0 geschrieben und mit Hilfe einer Schleife das gesamte Feld durchsucht.

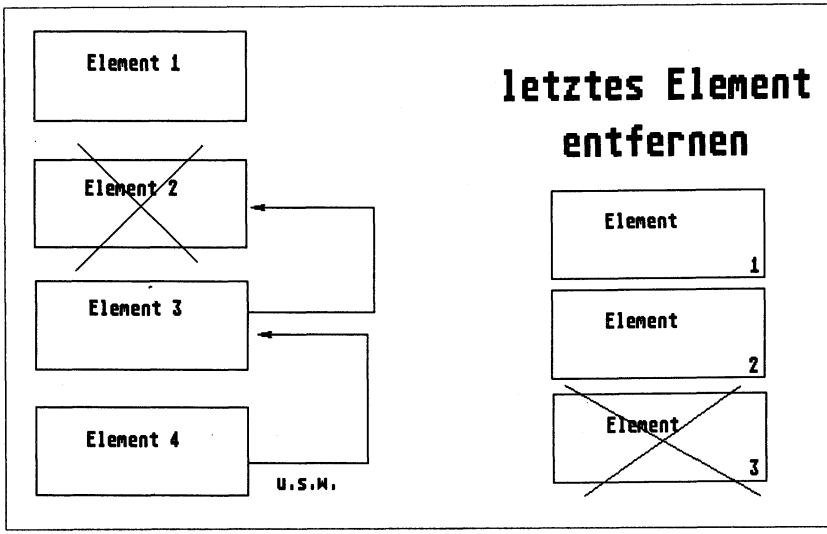



*Abb. 2.3: Element aus der Liste entfernen*

Als Abbruchbedingung gilt:

```
<Suchbegriff> = <Arrayinhalt>
```

bzw. für die Schleifenbedingung negativ formuliert:

```
<Zaehler> = 1
WHILE <Suchbegriff> <> <Arrayinhalt>
   <Zaehler> = <Zaehler> + 1
WEND
```

### Ein Vorgeschmack auf GEM

Sicher sind sie Ihnen schon einmal untergekommen. Jene freundlichen, mit einem Verkehrszeichen ähnlichen Symbol geschmückten Warnmeldungen, die der Atari immer dann zu servieren pflegt, wenn ein Bedienungsfehler aufgetreten ist oder

eine Sicherheitsabfrage durchgeführt werden soll. Der Fachmann nennt sie Alert-Boxen, und im ST-BASIC ist ein entsprechender Befehl für ihren Aufruf gleich mit eingebaut:

### FORM_ALERT

ruft eine Alert-Box auf. Die Syntax tritt in zwei Erscheinungsformen auf:

```
FORM_ALERT(<Default>, <Text>)
```

bzw.

```
FORM_ALERT(<Default>, <Text>, <Button>)
```

<Default> enthält die Nummer des Buttons (1-3), der auch durch eine Betätigung der <Return>-Taste selektiert werden kann. Wird als Defaultwert die 0 angegeben, so bedeutet dies, daß kein Button mit Return selektiert werden kann. <Text> besteht aus einer Zeichenkette, die sich wie folgt zusammensetzt:

```
"[<Nummer>][<Zeile 1>¦<Zeile 2>¦<Zeile 3>][<Button 1>¦<Button
2>¦<Button 3>]"
```

<Nummer> gibt an, welches Symbol (Piktogramm) die Alert-Box tragen soll:

0: Kein Symbol
1: Achtung (!)
2: Fragezeichen (?)
3: Stop

In den zweiten eckigen Klammern, die mit eingegeben werden müssen, steht der Text, der in der Box erscheinen soll. Er darf aus maximal fünf Zeilen bestehen, die voneinander durch das Zeichen | getrennt werden müssen. Das Zeichen | erreichen Sie durch gleichzeitiges Drücken der Tasten <Shift> und <~> (gleich links neben der Taste <Return>). Zu beachten ist dabei, daß die maximale Länge einer Zeile 40 Zeichen nicht überschreiten darf.

Als letztes muß der String noch den Text für die einzelnen Knöpfe enthalten, die dem Benutzer angeboten werden sollen. Diese sind ebenfalls durch das Zeichen | voneinander zu trennen. Es sind maximal drei Buttons erlaubt, der Button <Nummer> kann auch per <Return> selektiert werden. Optisch wird dies durch eine stärkere Umrahmung des Buttons dargestellt. Wird zusätzlich noch <Button> mit angegeben, enthält diese Variable die Nummer des Quittierungsknopfes, der zum Verlassen der Alert-Box geführt hat:

1. = erster Knopf
2. = zweiter Knopf
3. = dritter Knopf

Standen mehrere Möglichkeiten zur Auswahl, kann nach Abbruch der Box der Button abgecheckt werden, der das Verlassen bewirkt hat. Mit FORM_ALERT steht uns ein mächtiges Werkzeug zur Verfügung, um Fehlermeldungen auf den Monitor zu projizieren. Ein paar Beispiele zu ihrer Anwendung:

```
FORM_ALERT(1,[2][Was soll der Quatsch?][Abbruch]")

FORM_ALERT(1,[3][Das sollten Sie|nie wieder tun!][Ende]")


FORM_ALERT(2,[1][Möchten Sie den Datensatz|wirklich löschen?][ Ja
| Nein | Weiß nicht ]", exit)
```

In <Exit> finden Sie nach dem Verlassen der Box die Nummer des Buttons vor, mit dem die Meldung verlassen wurde.

Moment! Irgendetwas stimmt da noch nicht ganz! Es kann nämlich nur der Button selektiert werden, der über Return verlassen werden kann. Die übrigen beiden Knöpfe können nicht selektiert werden, da kein Mauszeiger vorhanden ist! Dieser muß nämlich vor Aufruf der Funktion im Handbetrieb mit

```
MOUSEON
```

eingeschaltet, und nach Beendigung der Funktion mit

```
MOUSEOFF
```

wieder ausgeschaltet werden. Aber jetzt klappt die Sache! Noch eine Anmerkung: Der Atari ST merkt sich, wie oft die Befehle MOUSEON und MOUSEOFF aufgerufen wurden. Wird beispielsweise zweimal der Befehl MOUSEOFF ausgeführt, sind wieder zwei Befehle MOUSEON vonnöten, um den Mauspfeil auf dem Bildschirm erscheinen zu lassen.

Die beiden Menüpunkte "Daten abspeichern" und "Daten laden" werden Gegenstand des nächsten Kapitel dieses Buches sein. In dem jetzt folgenden Listing, sind zwei leere Unterprogramme eingefügt, damit das Programm nicht aussteigt, wenn einer dieser beiden Punkte aufgerufen wird. Aber jetzt endlich das Listing:

```
0  '*****************************************************************
1  '*                          MINIDATA.BAS                         *
2  '*---------------------------------------------------------------*
3  '* Autor: Michael Maier    Version: 1.00     Datum: 15.08.1988 *
4  '*      Ein Programm aus dem 'GROSSEN ST-BASIC-BUCH'             *
5  '*        (C) 1988 by DATA BECKER GmbH Düsseldorf                *
6  '*****************************************************************
7  '
8  '
9  MODE "D"
10 DEF FN Screen$(X$)= CHR$(27)+X$
11 '
12 Groesse%L=100' falls nötig einfach ändern
13 DIM Name$(Groesse%L),Vorname$(Groesse%L),Strasse$(Groesse%L)
14 DIM Plz$(Groesse%L),Ort$(Groesse%L),Tel$(Groesse%L),Geb$(Groesse%L)
15 '
16 Fehler$="[3][Diese Funktion ist leider¦    nicht möglich!!!][Sorry]"
17 '
18 REPEAT
19    CLS
20    PRINT @(0,1);"*"*78
21    FOR Y%=1 TO 5: PRINT @(Y%,1);"*";@(Y%,78);"*": NEXT Y%
22    PRINT @(6,1);"*"*78
23    PRINT @(2,28);"MINIDATEI -  Hauptmenü"
24    PRINT @(3,28);"----------------------"
25    PRINT @(4,16);"Ein Demoprogramm aus dem grossen ST-BASIC Buch"
```

```
26    PRINT @(9,28);"1.  Name erfassen"
27    PRINT @(11,28);"2.  Name korrigieren"
28    PRINT @(13,28);"3.  Name suchen"
29    PRINT @(15,28);"4.  Datei abspeichern"
30    PRINT @(17,28);"5.  Datei laden"
31    PRINT @(19,28);"6.  Programm verlassen"
32    PRINT @(22,29);FN Screen$("p");" Bitte waehlen Sie! ";FN
Screen$("q")
33    PRINT FN Screen$("f")' Cursor ausschalten
34    '
35    A%L=0
36    REPEAT
37      A$= INKEY$
38      IF A$<>"" THEN
39         A%L= ASC( RIGHT$(A$,1))-48
40      ENDIF
41    UNTIL A%L>0 AND A%L<7
42    PRINT FN Screen$("e")' Cursor wieder einschalten
43    ON A%L GOSUB Erfassen,Korrigieren,Suchen,Speichern,Laden
44 UNTIL A%L=6' Schleife wiederholen, bis '6' gedrückt
45 CLS
46 END
47 '
48-Erfassen
49   CLS
50   Header$="***************  Name erfassen  ***************"
51   ' zuerst einmal den ersten freien Eintrag suchen!
52   T%=1
53   WHILE Name$(T%)<>"" AND T%<Groesse%L
54      T%=T%+1
55   WEND
56   Formular(Header$)
57   REPEAT
58      Eingabe(T%)
59      PRINT FN Screen$("f")' Cursor stört hier bloß!
60      PRINT @(19,15);FN Screen$("p");"N";FN Screen$("q");
61      PRINT "ächster Name    ";FN Screen$("p");"K";FN Screen$("q");
62      PRINT "orrektur   ";FN Screen$("p");"Z";FN Screen$("q");
63      PRINT "urück ins Hauptmenü"
64      A$="": INPUT "  ";A$ USING "+n+z+ku>",Ret%L,1,32
65      PRINT FN Screen$("e")'Cursor wieder einschalten
66      IF A$="K" THEN
67         Formular(Header$)
68         Anzeige(T%)
69      ELSE
70         IF A$="N" AND T%<Groesse%L THEN
```

```
71            T%=T%+1
72            Formular(Header$)
73            Anzeige(T%)
74          ENDIF
75        ENDIF
76    UNTIL A$="Z"
77 RETURN
78 '
79-Korrigieren
80 Header$="***************   Name korrigieren   ****************"
81 PRINT FN Screen$("f")
82 T%=1
83 Formular(Header$)
84 REPEAT
85   Anzeige(T%)
86   PRINT @(19,4);FN Screen$("p");"N";FN Screen$("q");"ächster Name  ";
87   PRINT FN Screen$("p");"L";FN Screen$("q");"etzter Name  ";
88   PRINT FN Screen$("p");"K";FN Screen$("q");"orrektur  ";
89   PRINT FN Screen$("p");"E";FN Screen$("q");"ntfernen  ";
90   PRINT FN Screen$("p");"Z";FN Screen$("q");"urück ins Hauptmenü"
91   '
92   A$=""
93   REPEAT
94     A$= INKEY$
95     IF A$<>"" THEN
96       A$= UPPER$( RIGHT$(A$,1))
97     ENDIF
98   UNTIL A$="N" OR A$="L" OR A$="K" OR A$="E" OR A$="Z"
99   '
100   IF (A$="N") AND (T%<Groesse%L) AND (Name$(T%+1)<>"") THEN
101     T%=T%+1
102   ELSE
103     IF A$="N" THEN
104       FORM_ALERT (1,Fehler$)
105     ENDIF
106   ENDIF
107   IF A$="L" AND T%>1 THEN
108     T%=T%-1
109   ELSE
110     IF A$="L" THEN
111       FORM_ALERT (1,Fehler$)
112     ENDIF
113   ENDIF
114   IF A$="K" THEN
115     PRINT FN Screen$("e")
116     PRINT @(19,1);" "*78
```

```
117     Eingabe(T%)
118     PRINT FN Screen$("f")
119   ENDIF
120   IF A$="E" THEN
121     MOUSEON
122     FORM_ALERT (2,"[2][Datensatz wirklich löschen?][Ja|Nein]",But%)
123     MOUSEOFF
124     IF But%=1 THEN
125       Delete(T%)
126     ENDIF
127   ENDIF
128 UNTIL A$="Z"
129 RETURN
130 '
131-Suchen
132 Header$="***************   Name suchen    ****************"
133 Name$(0)="":Vorname$(0)="":Strasse$(0)="":Plz$(0)=""
134 Ort$(0)="":Tel$(0)="":Geb$(0)=""
135 T%=0
136 Formular(Header$)
137 PRINT FN Screen$("e")
138 INPUT @(7,21);Name$(0) USING "a+ +-u",Ret%L,15
139 PRINT FN Screen$("f")
140 T%=1
141 REPEAT
142   WHILE Name$(T%)<>Name$(0) AND T%<Groesse%L
143     T%=T%+1
144   WEND
145   IF Name$(T%)=Name$(0) THEN
146     Anzeige(T%)
147   ELSE
148     FORM_ALERT (1,"[1][Name nicht vorhanden!][ Was soll's ]")
149   ENDIF
150   PRINT @(19,15);FN Screen$("p");"W";FN Screen$("q");
151   PRINT "eitersuchen    ";FN Screen$("p");"N";FN Screen$("q");
152   PRINT "eueingabe    ";FN Screen$("p");"Z";FN Screen$("q");
153   PRINT "urück ins Hauptmenü"
154   A$=""
155   REPEAT
156     A$= INKEY$
157     IF A$<>"" THEN
158       A$= UPPER$( RIGHT$(A$,1))
159     ENDIF
160   UNTIL A$="W" OR A$="N" OR A$="Z"
161   IF A$="W" AND T%<Groesse%L THEN
162     T%=T%+1
```

```
163   ENDIF
164   IF A$="N" THEN
165     EXIT TO Suchen
166   ENDIF
167 UNTIL A$="Z"
168 RETURN
169 '
170-Speichern
171 ' diese Routine folgt noch ...
172 RETURN
173 '
174-Laden
175 ' diese Routine folgt noch ...
176 RETURN
177 '
178 '
179 DEF PROC Formular(Text$)
180    LOCAL T%
181    CLS
182    PRINT @(2,(78- LEN(Text$))/2);Text$
183    PRINT @(5,10);"*"*60
184    FOR T%=1 TO 9: PRINT @(5+T%,10);"*";@(5+T%,69);"*": NEXT T%
185    PRINT @(15,10);"*"*60
186    PRINT @(7,15);"Name: _____     Vorname:_____"
187    PRINT @(9,15);"Strasse: _____"
188    PRINT @(11,15);"PLZ: ____   Ort: _____"
189    PRINT @(13,15);"Telefon: _____   Geb.: _____"
190 RETURN
191 '
192 DEF PROC Anzeige(Nummer%)
193    PRINT @(7,21);Name$(Nummer%);
194    PRINT STRING$(15- LEN(Name$(Nummer%)),"_")
195    PRINT @(7,50);Vorname$(Nummer%);
196    PRINT STRING$(15- LEN(Vorname$(Nummer%)),"_")
197    PRINT @(9,24);Strasse$(Nummer%);
198    PRINT STRING$(32- LEN(Strasse$(Nummer%)),"_")
199    PRINT @(11,20);Plz$(Nummer%);
200    PRINT STRING$(4- LEN(Plz$(Nummer%)),"_")
201    PRINT @(11,32);Ort$(Nummer%);
202    PRINT STRING$(30- LEN(Ort$(Nummer%)),"_")
203    PRINT @(13,24);Tel$(Nummer%);
204    PRINT STRING$(11- LEN(Tel$(Nummer%)),"_")
205    PRINT @(13,44);Geb$(Nummer%);
206    PRINT STRING$(10- LEN(Geb$(Nummer%)),"_")
207 RETURN
208 '
```

```
209 DEF PROC Eingabe(Nummer%)
210   LOCAL Back$="s"+ CHR$(72)+"s"+ CHR$(80)
211   -Nam
212   INPUT @(7,21);Name$(Nummer%) USING "a+ +-u"+Back$,Ret%L,15
213   IF (Ret%L AND $FF0000)=$480000 THEN
214     GOTO Geb
215   ENDIF
216   -Vorname
217   INPUT @(7,50);Vorname$(Nummer%) USING "a+ +-"+Back$,Ret%L,15
218   IF (Ret%L AND $FF0000)=$480000 THEN
219     GOTO Nam
220   ENDIF
221   -Street
222   INPUT @(9,24);Strasse$(Nummer%) USING "0a+ +-+."+Back$,Ret%L,32
223   IF (Ret%L AND $FF0000)=$480000 THEN
224     GOTO Vorname
225   ENDIF
226   -Plz
227   INPUT @(11,20);Plz$(Nummer%) USING "0>"+Back$,Ret%L,4
228   IF (Ret%L AND $FF0000)=$480000 THEN
229     GOTO Street
230   ENDIF
231   -Ort
232   INPUT @(11,32);Ort$(Nummer%) USING "a0+/+-"+Back$,Ret%L,30
233   IF (Ret%L AND $FF0000)=$480000 THEN
234     GOTO Plz
235   ENDIF
236   -Telefon
237   INPUT @(13,24);Tel$(Nummer%) USING "0+-+/c-/"+Back$,Ret%L,11
238   IF (Ret%L AND $FF0000)=$480000 THEN
239     GOTO Ort
240   ENDIF
241   -Geb
242   INPUT @(13,44);Geb$(Nummer%) USING "0+."+Back$,Ret%L,10
243   IF (Ret%L AND $FF0000)=$480000 THEN
244     GOTO Telefon
245   ELSE
246     IF (Ret%L AND $FF0000)=$500000 THEN
247       GOTO Nam
248     ELSE
249     ENDIF
250   ENDIF
251 RETURN
252 '
253 DEF PROC Delete(Nummer%)
254 LOCAL Anzahl%=1
```

```
255 ' Anzahl der vorhandenen Datensätze ermitteln
256 WHILE (Name$(Anzahl%+1))<>"" AND Anzahl%<Groesse%L
257   Anzahl%=Anzahl%+1
258 WEND
259 IF Nummer%=Anzahl% THEN
260    Loesche(Nummer%)
261 ELSE
262   WHILE (Nummer%<>Anzahl%)
263     Name$(Nummer%)=Name$(Nummer%+1)
264     Vorname$(Nummer%)=Vorname$(Nummer%+1)
265     Strasse$(Nummer%)=Strasse$(Nummer%+1)
266     Ort$(Nummer%)=Ort$(Nummer%+1)
267     Plz$(Nummer%)=Plz$(Nummer%+1)
268     Tel$(Nummer%)=Tel$(Nummer%+1)
269     Geb$(Nummer%)=Geb$(Nummer%+1)
270     Nummer%=Nummer%+1
271   WEND
272   Loesche(Nummer%)
273 ENDIF
274 RETURN
275 '
276 DEF PROC Loesche(Nummer%)
277   Name$(Nummer%)="":Vorname$(Nummer%)="":Strasse$(Nummer%)=""278
Plz$(Nummer%)="":Ort$(Nummer%)="":Tel$(Nummer%)=""
279   Geb$(Nummer%)=""
280 RETURN
```

# 3. Dateiverwaltung

Daten erfassen ist die eine, sie auch nach dem Abschalten des Computers nicht zu verlieren die andere Sache. Eine Möglichkeit ein Programm mit Daten zu versorgen, kennen Sie bereits: READ und DATA. Doch für die Erstellung einer Adreßverwaltung ist diese Methode denkbar ungeeignet.

Diese Daten legt man am besten auf Diskette in sogenannten Dateien (Files, gesprochen: "Feils") ab. Von dort können Sie bei Bedarf eingelesen, verarbeitet, und wieder abgespeichert werden.Zwei Arten der Datenspeicherung sind hierbei besonders zu unterscheiden:

- Sequentielle Dateien
- Relative Dateien

sowie eine Mischform aus sequentieller und relativer Dateiverwaltung, die

- Indexsequentiellen Dateien (ISAM)

## 3.1 Dateien auf Diskette

Bis jetzt war schon eine ganze Menge von sequentiellen Dateien die Rede, was dahinter steckt habe ich allerdings noch nicht verraten. Das wird jetzt nachgeholt:

### Sequentielle Dateien

In sequentiellen Dateien werden die Daten einfach hintereinander abgelegt. Damit die einzelnen Elemente bei einem späteren Einlesen voneinander unterschieden werden können, trennt sie der Computer durch ein Carriage Return, das bekanntlich den ASCII-Wert 13 besitzt. Die einzelnen Elemente können unter-

schiedliche Längen besitzen, ja sogar unterschiedliche Variablentypen können in beliebiger Reihenfolge in eine solche Datei geschrieben werden.

Diese Art der Datenspeicherung hat natürlich auch einen Haken: Da die einzelnen Elemente in der Datei unterschiedlich lang sein können, muß die Datei stets komplett in den Speicher des Computers gelesen werden. Ferner sollte man sich genau merken, welche Daten man in diese Datei geschrieben hat, sonst könnte es bei einem späteren Einlesen Probleme geben. (Stellen Sie sich vor, in einer Datei sind 50 Adreßen gespeichert, und als nächster Wert wurde das Datum der Erfassung in die Datei geschrieben!)

Ändern sich bestimmte Daten, muß die Datei komplett neu auf Diskette geschrieben werden, da einzelne Einträge nicht einfach abgeändert werden können.

### Random-access-Dateien (relative Dateien)

Im Gegensatz zur Speicherung in sequentieller Form werden die zu speichernden Daten in relativen Dateien in einzelnen Datensätzen (Records) festgehalten. Ein solcher Datensatz kann mehrere Elemente enthalten (z.B. Name, Vorname, Straße, ...), die Länge eines Records darf von einem zuvor festgelegten Wert jedoch nicht abweichen. Dafür ist es dann möglich, einen ganz bestimmten Datensatz innerhalb der Datei anzusprechen und einzulesen, was bei sequentiellen Dateien nicht der Fall ist.

Auch diese Form der Datenspeicherung ist nicht ohne Nachteile: Jeder Record besitzt innerhalb der Datei die angegebene Länge, unabhängig davon, wie lang die in ihm enthaltenen Daten tatsächlich sind. Deshalb benötigt eine Random-access-Datei im Normalfall wesentlich mehr Speicherplatz als ihre Artgenossin, die sequentielle Datei. Ferner sollte eine relative Datei vor ihrer Benutzung eingerichtet werden, das heißt alle Datensätze werden bereits auf der Diskette angelegt, die einzelnen Records enthalten jedoch nur Leerzeichen (Spaces).

**Fazit:** Eine relative Datei erfordert wesentlich mehr Vorüberlegungen, einen etwas größeren Programmieraufwand und mehr Speicherplatz auf Diskette. Dafür kann ein einzelner Datensatz gezielt angesprochen werden.

### ISAM-Dateien

Bei der indexsequentiellen (abgekürzt ISAM) Dateiverwaltung werden die beiden Formen - sequentielle und relative Dateien - geschickt kombiniert. Die Idee, die hinter diesem Verfahren steckt, ist folgende:

Sequentielle Dateien werden komplett in den Computer gelesen, ehe sie bearbeitet werden können; in relativen Dateien werden lediglich bestimmte Datensätze eingelesen. Was geschieht nun, wenn Daten in einer Datei gesucht werden müssen? Bei der sequentiellen Datei, die sich bereits komplett im Computer befindet, geht dies unheimlich schnell. Die Suche in relativen Dateien dauert dagegen sehr lange, da Datensatz für Datensatz eingelesen werden muß, bis die betreffenden Daten gefunden sind. Und der Zugriff auf Diskette benötigt seine Zeit, da erst die Mechanik in der Diskettenstation in Gang gesetzt werden muß.

Nun könnte man natürlich die gesamte Datei in den Speicher holen, aber was wäre dabei gewonnen? In diesem Fall könnte man gleich eine sequentielle Datei bemühen!

Folglich greift man zu einem Trick: Die gesamte Datei wird in relativer Form abgespeichert. Gleichzeitig werden die Suchbegriffe, nach denen die Datei durchstöbert werden soll, in einer sequentiellen Datei abgelegt, die Indexdatei genannt wird. Jeder Eintrag in dieser Indexdatei besteht aus zwei Teilen:

```
<Suchbegriff> + <Datensatznummer>
```

Die Indexdatei liest man (zu Programmbeginn) komplett in den Speicher. Werden jetzt bestimmte Daten benötigt, durchsucht man einfach die Indexdatei im Computer. Wird der Suchbegriff gefunden, muß nur noch der Record eingelesen werden, dessen

Nummer hinter dem Suchbegriff vermerkt ist. Und schon hat man die gewünschten Daten im Speicher. Dadurch erspart man sich das zeitraubende Einlesen eines jeden Datensatzes, bis der gewünschte Suchbegriff gefunden ist.

Werden neue Daten erfaßt, muß die Indexdatei erweitert werden und zwar um den neuen Suchbegriff und die Nummer, unter der der neue Record in der Random-access-Datei zu finden ist. Wird ein Record aus der Datei gelöscht, muß die Indexdatei ebenfalls um den entsprechenden Suchbegriff gekürzt werden.

Im Extremfall können natürlich mehrere Indexdateien (eine für Namen, eine für Geburtsdatum ...) zu einer relativen Datei gehören. Dann ist es möglich, die Datei nach den unterschiedlichsten Daten zu durchforsten.

## 3.2 Ohne Kanäle geht gar nichts

Ehe auch nur ein einziges Byte auf Diskette oder Harddisk gespeichert werden kann, muß erst ein Kanal geöffnet werden, über den der weitere Datenaustausch zwischen dem Computer und dem gewünschten Speichermedium abgewickelt werden kann. Einen solchen Kanal kann man sich wie eine Telefonleitung zwischen dem Computer und der Diskettenstation vorstellen, die Verbindung zwischen beiden wird über den Befehl

```
OPEN
```

hergestellt. OPEN besitzt folgende Syntax:

```
OPEN <"Dateimodus">,<Kanalnummer>,<"Dateiname">
```

Der erste Parameter, den der Interpreter hinter einem OPEN erwartet ist der Dateimodus. Zur Verfügung stehen dabei:

| Dateimodus | Bedeutung |
|---|---|
| "I" (Input) | sequentielle Datei lesen |
| "O" (Output) | sequentielle Datei schreiben |
| "A" (Append) | an bestehende sequentielle Datei anhängen |
| "F"(Files) | Inhaltsverzeichnis |
| "P" (Printer) | Drucker |
| "R" | Random-Acess-Zugriff (relative Datei) |
| "C" | Konsole, Bildschirm und Tastatur |
| "K" (Keyboard) | Befehle an Tastaturprozessor |
| "M" (Midi) | MIDI-Port |
| "V" | RS232-Schnittstelle |

Keine Angst, für unsere Zwecke reichen die ersten drei Dateitypen erst einmal völlig aus, später kommen dann die übrigen Spielarten hinzu.

Um einen Kanal zu öffnen, dessen Sinn und Zweck das Schreiben in eine sequentielle Datei ist, muß der Parameter "O" als Dateityp angegeben werden. Ein auf diese Weise geöffneter Kanal kann später dann nur zum Erstellen einer sequentiellen Datei benutzt werden, es ist also nicht möglich, über diesen Kanal Daten aus einer Datei einzulesen.

Ein Kanal wird also stets mit einer spezifischen Aufgabe (Lesen einer sequentiellen Datei, Schreiben einer sequentiellen Datei, Ausgabe der Daten an den MIDI-Port ...) betraut. Da es möglich ist, innerhalb eines Programms mehrere Kanäle zu öffnen, muß ein Kriterium existieren, anhand dessen die einzelnen Kanäle unterschieden werden können. Deshalb wird als nächster Parameter eine Kanalnummer beim Befehl OPEN angegeben. Diese Nummer, anhand der die Daten dann sicher auf die Reise geschickt werden können, ist eine Zahl im Bereich von 1 bis 16, die beliebig (aber nicht doppelt!) vergeben werden kann.

OPEN "O",1 öffnet einen Kanal mit der Nummer 1, der die Aufgabe besitzt, die Daten in sequentieller Form einer Datei anzuvertrauen. Gleichzeitig kann ein Kanal über OPEN "I",2 geöffnet werden. Seine Aufgabe besteht darin, Daten einzulesen. Möchten Sie innerhalb des Programms Daten einlesen, muß der Kanal mit der logischen Dateinummer 2, zum Speichern von Daten der Kanal mit der Nummer 1 benutzt werden.

Damit die in einer Datei gespeicherten Daten zu einem späteren Zeitpunkt auch wieder eingelesen werden können, muß als dritter Parameter hinter OPEN noch ein Dateiname angegeben werden. Dieser darf maximal aus acht Buchstaben bestehen, wobei zusätzlich noch eine sogenannte Extension (bestehend aus drei Buchstaben) angehängt wird. Der eigentliche Dateiname und die Extension sind durch einen Punkt voneinander getrennt. Im Gegensatz zu den Befehlen LOAD, SAVE, und NEW wird die Extension jedoch nicht selbstständig vom Computer vergeben, sondern muß im Dateinamen mit enthalten sein.Fassen wir noch einmal zusammen:

OPEN öffnet einen Kanal, über den der Datenverkehr abgewickelt werden kann. Drei Parameter müssen angegeben werden:

*Dateimodus*: Gibt die Aufgabe des Kanals an (Lesen, Schreiben, ...)

*Kanalnummer*: Dient der Identifizierung und dem Ansprechen des Kanals, sie besteht aus einer Zahl im Bereich von 1 bis 16.

*Dateiname*: Unter diesem Namen werden die Daten auf Diskette abgelegt. Er darf aus maximal acht Buchstaben bestehen, wobei zusätlich noch drei Buchstaben für eine Extension erlaubt sind.

## 3.3 Noch ein Print, aber mit Write geht's auch

Ein Kanal wäre jetzt also geöffnet, die Daten können auf die Reise geschickt werden. Aber welchen Befehl kann dazu benutzen? PRINT gibt die ihm anvertrauten Daten auf dem Bildschirm aus, doch da wollen wir sie in diesem Fall nicht haben. Ein anderer Vertreter der Gattung PRINT kann diese Aufgabe jedoch bestens erledigen:

```
PRINT#<Kanalnummer>,<Daten>
```

PRINT# arbeitet genauso wie Print, nur schreibt er die Daten nicht auf den Monitor (das geht zwar auch ...), sondern in den unter <Kanalnummer> angegebenen Kanal. Wurde eine Datei zum Schreiben geöffnet, kann sie per PRINT# mit Daten versorgt werden. Die Kanalnummer ist in jedem Fall anzugeben:

```
OPEN "O",1,"BEISPIEL.DAT"
   PRINT#1, "irgendwas"
```

entspricht:

```
OPEN "O",5,"BEISPIEL.DAT"
   PRINT#5, "irgendwas"
```

Sie sehen schon, welche Kanalnummer Sie bei OPEN auch verwenden, die Ausgabe muß unbedingt auf den richten Kanal erfolgen. Wird versucht, die Ausgabe auf einen nicht geöffneten Kanal zu leiten, ernten Sie eine Fehlermeldung!

```
WRITE#
```

Write# besitzt die gleiche Syntax wie PRINT#, und gibt ebenfalls Daten auf dem angegebenen Kanal aus. Im Unterschied zu PRINT# werden bei WRITE# jedoch die ausgegebenen Zeichenketten in Anführungszeichen gesetzt. Dies birgt einen gewaltigen Vorteil in sich: Das Komma dient bei einem INPUT als Trennzeichen, d.h. für den INPUT-Befehl ist die Eingabe dann beendet, wenn es auf ein Komma stößt (vgl. INPUT A,B!). Enthält ein String nun Kommata, so wird bei einem Input die Zeichenkette lediglich bis zum ersten Komma angenommen.Bei einem in Anführungszeichen stehenden String werden Kommata jedoch nicht als Trennzeichen angesehen! Folglich kann mit einem Input die gesamte Zeichenkette eingelesen werden, auch wenn sie Kommata enthält.

Ferner schreibt WRITE# bei Daten, die durch Komma voneinander getrennt sind, das Komma auch wirklich auf den angegebenen Kanal (Input macht dies nicht!). Somit können mehrere Variablen mit nur einem einzigen WRITE# abgespeichert wer-

den. Ein INPUT-Befehl kann sie zu einem späteren Zeitpunkt wieder problemlos einlesen, da für ihn ein Komma als Trennzeichen gilt:

```
OPEN "O", 1, "NAME.BAS"
A=14:B=20:C=77:D=14
WRITE #1, A,B,C,D
....
....
```

Ergibt in der Datei NAME.DAT:

```
14,20,77,14
```

Von dort können Sie dann mit einem Input-Befehl wieder abgeholt werden. So, die Daten können wir jetzt in eine Datei schreiben. Doch ehe das Programm verlassen wird, muß die geöffnete Datei wieder geschlossen werden. Andernfalls sind die Daten verloren. Zum Schließen einer Datei dient der Befehl

```
CLOSE <Dateinummer>,<Dateinummer>,...
```

Dabei werden die Dateien mit der Nummer <Dateinummer> wieder geschlossen. Wird keine Dateinummer hinter dem CLOSE angegeben, schließt ST-BASIC sämtliche Dateien, die zu diesem Zeitpunkt geöffnet sind.

## 3.4 Sequentielle Dateien einlesen

Die Daten sind jetzt auf der Diskette, doch eines Tages benötigen wir sie wieder. Zum Einlesen von Daten wird ein naher Verwandter des INPUT-Befehls benutzt:

```
INPUT #<Kanalnummer>,<Variable(nliste)>
```

arbeitet wie INPUT, holt sich seine Eingabe jedoch nicht von der Tastatur, sondern aus dem Kanal <Kanalnummer>. Das Einlesen ist für den Interpreter beendet, sobald er auf ein Carriage Return (CHR$(13)) bzw. ein Komma stößt.

```
OPEN "I", 1, "NAME.DAT"
INPUT #1,A,B,C,D
CLOSE 1
```

liest die mit WRITE# auf Diskette geschriebene Datei wieder in den Speicher des Computers. Übrigens, es existiert auch ein LINE INPUT#-Befehl, der Kommata erlaubt und nur Carrige Return als Ende der Eingabe betrachtet.

Soweit ist noch alles klar. Schwieriger wird es, wenn man nicht mehr weiß, wie viele Daten in die Datei geschrieben wurden. Will man nämlich in einer Schleife mittels INPUT# solange Daten von Diskette einlesen, bis sich die Datei komplett im Speicher des Atari ST befindet (bis Input# einen Leerstring ("") liefert, da keine weiteren Daten mehr vorhanden sind), meldet sich der Interpreter zu Wort, sobald das Ende der Datei überschritten wurde.

Zum Einlesen einer Datei, von der man nicht mehr weiß, wie viele Elemente man in ihr bereits erfaßt hat, scheidet diese Methode folglich aus. Da man sich allerdings unmöglich merken kann, wie viele Einträge in einer Datei vorhanden sind (in diesem Fall könnte man die Daten in einer FOR...NEXT-Schleife einlesen), existiert in ST-BASIC eine eigene Funktion, die mitteilt, wann das File-Ende (Datei-Ende) erreicht ist. Diese Funktion lautet:

```
EOF(<Dateinummer>)
```

EOF() (End Of File: Datei-Ende) liefert den Wahrheitswert falsch, solange das File-Ende der Datei, die über <Dateinummer> angesprochen wird noch nicht erreicht ist. Andernfalls erhält man den Wahrheitswert wahr.

Für die Konstruktion einer Abbruchbedingung ist diese Funktion augenblicklich noch nicht brauchbar. Sie liefert nämlich immer dann den Wert falsch (0) und führt dadurch zu einem Verlassen der Schleife, wenn die Schleife noch ein weiteres Mal durchlaufen werden müßte, weil das Datei-Ende noch nicht erreicht ist. Deshalb muß der von dieser Funktion gelieferte

Wahrheitswert erst einmal umgedreht, d.h. negiert werden, ehe er als Abbruchkriterium in die Schleife eingebaut werden kann. Dies erledigt die Funktion NOT:

```
OPEN "I",1,"DATEI.NAM"
  WHILE NOT EOF(1)' bis zum Datei-Ende wiederholen
      INPUT #1, Name$
  WEND
CLOSE 1
```

Das soeben Gesagte gilt zumindest für sequentielle Dateien, bei relativer Dateiverwaltung ist das Verfahren etwas anders. Aber dazu kommen wir später noch!

Nach diesem theoretischen Teil schreiten wir wieder zur Praxis: Die Mini-Adreßverwaltung ist noch nicht ganz fertig! Ihr fehlen noch die Unterroutinen zum Laden und Speichern der Adreßen auf Diskette. Zuerst einmal zum Abspeichern der Daten: In einer Schleife wird die gesamte Liste vom ersten bis zum letzten Eintrag durchlaufen. Die einzelnen Daten werden dann mit PRINT# (Komma tauchen nirgends auf, deshalb wird kein Write# benötigt) in die sequentielle Datei verfrachtet. Beim Einlesen werden dann die Daten (logischerweise wieder in der gleichen Reihenfolge, wie sie in die Datei geschrieben wurden) in das Feld geladen, bis das Abbruchkriterium der Schleife, das Datei-Ende, erreicht ist. Anschließend kehrt die Unterroutine wieder in die Menüauswahl zurück.

Speichern:

```
OPEN "O",1,"MINIADR.DAT"
  T%=1
  WHILE Name$(T%)<>""
     PRINT #1,Name$(T%)
     PRINT #1,Vorname$(T%)
     PRINT #1,Strasse$(T%)
     PRINT #1,Plz$(T%)
     PRINT #1,Ort$(T%)
     PRINT #1,Tel$(T%)
     PRINT #1,Geb$(T%)
     T%=T%+1
  WEND
```

```
CLOSE 1
RETURN
'
```

Laden:

```
OPEN "I",5,"MINIADR.DAT"
  T%=1
  WHILE NOT EOF(5)
     INPUT #5,Name$(T%)
     INPUT #5,Vorname$(T%)
     INPUT #5,Strasse$(T%)
     INPUT #5,Plz$(T%)
     INPUT #5,Ort$(T%)
     INPUT #5,Tel$(T%)
     INPUT #5,Geb$(T%)
     T%=T%+1
  WEND
  CLOSE 5
RETURN
```

## 3.5 Files kopieren

Häufig kommt es vor, daß bestimmte Dateien kopiert werden müssen; sei es, daß Sie eine Sicherheitskopie wünschen, sei es, daß Sie mit einer RAM-Disk arbeiten, die zuerst mit den entsprechenden Dateien versorgt werden muß.

ST-BASIC besitzt dafür einen eigenen Befehl: COPY. Seine Syntax lautet:

```
COPY <Quelldatei> TO <Zieldatei>
```

Doch wir schreiben uns eine dazu eigene kleine Prozedur! Warum? Weil ich Ihnen dabei zwei weitere Befehle vorführen kann!

Der Algorithmus, den wir dafür nutzen, ist denkbar einfach: Sowohl die Quelldatei als auch die Zieldatei werden als sequentielle Files betracht. Aus der Quelldatei wird nun solange Byte

für Byte eingelesen und in die Zieldatei geschrieben, bis das Ende der Quelldatei erreicht ist. Danach kehrt die Prozedur (in das Hauptprogramm?) zurück.

Der Input#-Befehl versagt kläglich, wenn man versucht, ein einzelnes Byte aus einer Datei einzulesen. Das Eingabeende ist für ihn erst mit Erreichen eines Carriage Return bzw. eines Kommas gegeben. Was geschieht nun, wenn im ganzen File kein einziges Komma oder Carriage Return vorkommt? Der Interpreter wird sich unter Umständen verabschieden, da ein String aus maximal 32000 Zeichen bestehen darf. Und wenn die Datei länger war, ja dann ...

Der Input#-Befehl ist für unsere Zwecke also nicht brauchbar. Ein anderer Befehl zu Einlesen von Daten muß her! Hier ist er:

```
INPUT$(<Anzahl Zeichen>,<Dateinummer>)
```

ließt genau <Anzahl Zeichen> aus der Datei mit dem Kanal <Dateinummer> ein. Wird als <Anzahl Zeichen> der Wert 1 angegeben, so erhält man genau 1 Byte aus der entsprechenden Datei.

An dieser Stelle möchte ich gleich das fehlerhafte Listing im Handbuch zu ST-BASIC (Omikron.-BASIC Version 3.00, Seite 81) korrigieren. Die nachfolgende Prozedur gibt den gesamten Inhalt einer Datei Name$ auf dem Bildschirm aus:

```
1000 DEF PROC Type(Name$)
1010 OPEN "I",1,Name$
1020  WHILE NOT EOF(1)
1030    PRINT INPUT$(1,1);
1040  WEND
1050  CLOSE 1
1060 RETURN
```

In Zeile 1010 wird eine sequentielle Datei mit der Kanalnummer 1 zum Lesen geöffnet. Zeile 1030 liest genau ein Byte ein (INPUT$) und gibt es mit PRINT gleich wieder auf dem Monitor aus. Die Schleife sorgt dafür, daß die gesamte Datei Byte für Byte ausgegeben wird, bis das File-Ende erreicht ist. An-

schließend wird der Kanal wieder geschlosssen und die Prozedur verlassen. So ähnlich muß auch unsere Prozedur zum Kopieren einer Datei aufgebaut werden. Nur darf hier die Ausgabe der einzelnen Daten nicht an den Monitor erfolgen, sondern gleich in eine andere Datei. Diese muß zum (sequentiellen) Schreiben geöffnet werden und eine andere Dateinummer tragen:

```
DEF PROC Filecopy(Von$,Nach$)
   OPEN "I",1,Von$
   OPEN "O",2,Nach$
   WHILE NOT EOF(1)
     PRINT #2,INPUT$(1,1);
   WEND
   CLOSE 2
   CLOSE 1
RETURN
```

Aber diese Routine arbeitet viel zu langsam! Wesentlich schneller geht es, wenn größere Datenmengen auf einmal eingelesen werden können. Klingt paradox? Die Erklärung dafür ist aber ganz einfach: Die Diskettenstation arbeitet mit einer aufwendigen Mechanik. Ein Tonkopf (bzw. zwei Tonköpfe bei einer doppelseitigen Floppy) wird an die Spur gefahren, auf der das gewünschte Byte aufgezeichnet wurde. Da sich die Diskette dreht, muß er solange warten, bis sich das Byte gerade unter dem Tonkopf befindet, dann erst kann es eingelesen werden. Werden gleich mehrere Bytes auf einmal eingelesen, geht dies wesentlich schneller von statten, da sich der Tonkopf sowieso schon an der richtigen Stelle befindet, er muß die nächsten Bytes nur noch "mitnehmen". Zugegeben, dies ist eine stark stilisierte Erklärung, aber ich hoffe, sie ist dafür leicht verständlich.

Die maximale Größe für einen Einlesevorgang beträgt 32000 Byte, da ein String nicht mehr Buchstaben fassen kann. Doch schon bekommen wir wieder Schwierigkeiten! Die Funktion EOF() hilft uns jetzt nämlich nicht mehr weiter. Bei einem gleichzeitigen Einlesen von mehreren Bytes kann das Datei-Ende überschritten werden, ohne daß dies von der Funktion EOF() abgefangen werden könnte. Stellen Sie sich nur einmal vor, in der Datei sind noch 2000 Bytes einzulesen. EOF() bzw. NOT

EOF() liefert als Ergebnis wahr, d.h. das File-Ende ist noch nicht erreicht - aber für INPUT$(32000,1) sind nicht mehr genug Daten vorhanden! Was tun?

Ein Lösungsweg führt über die Länge der Datei, d.h. die in diesem File enthaltenen Bytes. Könnte man diese Länge ermitteln, bräuchte man nur in einer (abweisenden) Schleife solange jeweils 32000 Bytes einlesen, bis keine 32000 Bytes mehr bis zum Datei-Ende übrig bleiben. Diese könnte man dann noch auf einen Schlag kopieren, und schon ist die gesamte Datei kopiert. Auch zum Ermitteln der Länge eines bestimmten Files gibt es eine Funktion:

```
LOF(<Dateinummer>)
```

LOF() (Length Of File) liefert bei sequentiellen Dateien, die zum Lesen geöffnet sind, die Länge der Datei in Bytes. Das Datei-Endezeichen EOF wird dabei mitgerechnet. Deshalb muß ein Byte von der Länge abgezogen werden, das in unserem Fall nicht mitkopiert zu werden braucht.

Ist eine sequentielle Datei zum Schreiben geöffnet, liefert die Funktion LOF() die Länge der Daten zurück, die bereits auf Diskette geschrieben wurden. Befinden sich noch Daten im Diskettenpuffer (der Computer speichert die Daten erst einmal im Diskettenpuffer, ehe sie auf Diskette verfrachtet werden, damit bei einem Diskettenzugriff gleich größere Datenpakete auf die Reise geschickt werden können), so werden diese nicht berücksichtigt. Handelt es sich um eine relative Datei, liefert LOF() als Ergebnis die Anzahl der in dieser Datei gespeicherten Datensätze zurück.

Mit LOF() kann also die Länge der Datei ermittelt werden. Diese wird in einer Variablen (L) festgehalten. Befinden sich mehr als 32000 Bytes in der Datei, werden in einer Schleife 32000 Bytes eingelesen und gleich wieder in die Zieldatei geschrieben (die Datei wird also häppchenweise kopiert). Anschließend wird die Länge (L) um die bereits eingelesene Anzahl von Bytes erniedrigt und die Schleifenbedingung erneut überprüft.

Erst wenn die Datei weniger als 32000 noch zu kopierende Bytes enthält, wird die Schleife verlassen (bzw. gar nicht abgearbeitet), und die restlichen Bytes (in der Variablen L enthalten) werden kopiert. Hier nun die kleine Prozedur:

```
0  '*****************************************************************
1  '*                          FILECOPY.BAS                         *
2  '*---------------------------------------------------------------*
3  '* Autor: Michael Maier    Version: 1.00     Datum: 10.06.1988 *
4  '*      Ein Programm aus dem 'GROSSEN ST-BASIC-BUCH'             *
5  '*        (C) 1988 by DATA BECKER GmbH Düsseldorf                *
6  '*****************************************************************
7  '
8  '
9  ' die folgende Routine kopiert ein File
10 '
11 ' Aufruf: Filecopy("A:\DATEI_1.BAS", "D:\DATEI_2.BAS")
12 '                       ..von..           ..nach..
13 '
14 DEF PROC Filecopy(Von$,Nach$)
15    LOCAL L
16    OPEN "I",1,Von$
17    OPEN "O",2,Nach$
18    L= LOF(1)
19    WHILE L>32000' mehr passt in keinen String
20       PRINT #2, INPUT$(32000,1);
21       L=L-32000
22    WEND
23    ' EOF nicht mitkopieren => L-1
24    PRINT #2, INPUT$(L-1,1);
25    CLOSE 2
26    CLOSE 1
27 RETURN
```

## 3.6 Die File-Selector-Box und deren Verwaltung

Die kleine Adreßdatei, die wir entwickelt haben, kann jetzt die erfassten Daten auf Diskette schreiben, und von dort auch wieder einlesen. Diese Datei, in der sich die Adreßen befinden, trägt den Namen MINIADR.DAT.

Jetzt könnte man auf die Idee kommen, mehrere Adreßdateien anzulegen (z.B. für Freunde, Geschäftspartner usw.), die allesamt von dem Programm verwaltet werden sollen. Dann muß aber der Name für die Datei abgeändert werden können, damit mehrere Adreßdateien mit der Minidatei verwaltet werden können. Die einfachste Möglichkeit, dies zu bewerkstelligen, besteht darin, mit einem INPUT nach dem Dateinamen zu fragen, dann die entsprechende Datei ganz normal mit OPEN zu öffnen und die darin enthaltenen Adreßen auszulesen.

```
INPUT "Welche Datei möchten Sie einlesen? ";Name$
OPEN "I",1, Name$
 ....
 ....
```

So oder ähnlich würde dieses Problem auf anderen Computern gelöst werden, nicht jedoch auf dem Atari ST und schon gar nicht in ST-BASIC! Wozu besitzen wir schließlich GEM? GEM bietet eine äußerst komfortable Möglichkeit zur Eingabe bzw. der Auswahl eines schon bestehenden Dateinamens vor dem Einlesen oder Abspeichern einer Datei an: Die File-Selector-Box (Dateiauswahlbox).

In der ersten Zeile dieser Box steht unter INDEX der sogenannte Pfadname. Was aber ist ein Pfad? Er gibt wie ein normaler Pfad auch den Weg zu einem Ziel an, in unserem Fall ist dies der Weg zu einer Datei. Moderne Computer gestatten es nämlich, den Inhalt einer Diskette in Ordner zu unterteilen. In diesen Ordnern können dann entweder weitere Ordner, oder die gewünschten Dateien angelegt werden. Damit der Computer - oder besser gesagt das Betriebssystem - eine entsprechende Datei lesen kann, muß es wissen, in welchem Ordner sie sich befindet. Und dafür gibt es den Pfadnamen. Aber das ist noch nicht alles! Der Pfad besteht noch aus weiteren Komponenten:

```
<Laufwerksbezeichnung>:\<Ordner 1>\<Ordner 2>\*.BAS
```

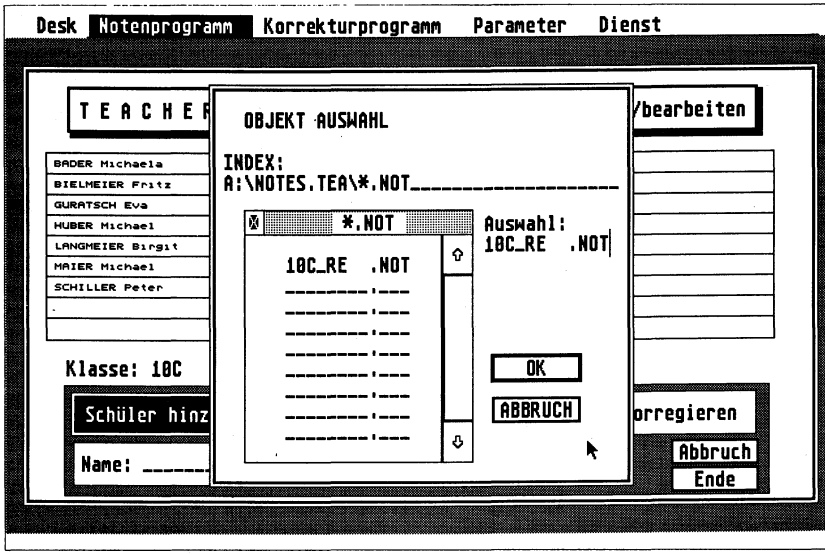



*Abb. 3.1: Die File-Selector-Box*

Der erste Buchstabe des Pfades gibt das gerade aktive Laufwerk an. Im Normalfall wird die Laufwerkskennung A sein (das ist die eingebaute Diskettenstation), aber auch B, C, D (üblicherweise die RAM-Disk) usw. sind möglich.

Von der Laufwerksbezeichnung durch einen Doppelpunkt abgetrennt, folgt der eigentliche Pfad, d.h. die Ordnerverschachtelung. Im einfachsten Fall (die Datei befindet sich in keinem Ordner, sondern im Hauptdirectory) lautet der Pfad dann:

```
<Laufwerksbezeichnung>:\*.*
A:\*.*
```

Möchten Sie dagegen eine Datei einladen, die sich im Ordner DOKUMENT.SDO befindet, muß der Pfad erweitert werden:

```
A:\DOKUMENT.SDO\*.*
```

Jetzt erhalten Sie sämtliche Dateien, die sich im Ordner DOKUMENT.SDO befinden. Aber auch ein Ordner im Ordner ist möglich. Dann muß der Weg vom Hauptdirectory zum innersten Ordner im Pfad angegeben werden. Die einzelnen Ordnernamen werden wieder durch den Querstrich "\" (Backslash) voneinander getrennt:

```
A:\DOKUMENT.SDO\FORMULAR.SDO\*.*
```

gibt sämtliche Dateien im Ordner FORMULAR.SDO, der sich im Ordner DOKUMENT.SDO befindet aus. Der letzte Teil des Pfades gibt den Namen an. Dieser besteht wiederum aus zwei Teilen: dem eigentlichen Namen und einem Extender (Extension), der durch den Punkt vom eigentlichen Namen getrennt ist. Üblichweise gibt der Extender den Dateityp an:

| Extender | Typ |
|---|---|
| BAS | BASIC-Programm |
| PRG | ausführbaren GEM-Programm |
| ... | |
| ... | |

Aber davon war schon einmal die Rede! Neu ist dagegen, daß der Name im Indexfeld (die erste Eingabezeile der Fileselector-Box) durch sogenannte Wildcards (Joker) ersetzt werden kann. Dies sind Platzhalter, die entweder einen Buchstaben, oder gar einen kompletten Namen ersetzen. Mit ihrer Hilfe wird es möglich, eine Auswahl verschiedener Dateien anzuzeigen. Dazu benutzt man die beiden Zeichen * und ?.

Das Fragezeichen ? steht als Platzhalter genau für einen Buchstaben oder genau ein beliebiges Zeichen, während der Stern * eine ganze Zeichenkette ersetzt. Ein paar Beispiele zur Veranschaulichung:

*H?MMEL.PRG* Zeigt alle (Programm-)Dateien an, die die angegebenen Buchstaben enthalten, wobei das zweite Zeichen (repräsentiert durch das Fragezeichen) beliebig ist. Zum Beispiel HIMMEL.PRG, HUMMEL.PRG, HAMMEL.PRG usw.

*H*.PRG* Zeigt alle (Programm-)dateien an, die mit einem H beginnen, wobei die restlichen Buchstaben bis zum Beginn der Extension beliebig sind. Zum Beispiel HALLO.PRG, HILBERT.PRG usw.

**.BAS* Zeigt sämtliche Dateien an, die mit der Extension BAS ausgestattet sind.

**.** Zeigt gänzlich alle Dateien an, egal welche Extension sie besitzen.

ST-BASIC bietet einen eigenen Befehl, um die File-Selector-Box auf den Bildschirm zu bringen:

```
FILESELECT(<Pfadname>,<Dateiname>,<Flag>)
```

In der Stringvariablen <Pfadname> muß der Pfadname angegeben werden, der zu Beginn des Aufrufes im Indexfeld der Box angezeigt werden soll. Auf jeden Fall muß darin ein korrekter Pfad enthalten sein! Der "Minimalpfad" lautet: A:*.*. Möchten Sie nur BASIC-Programme in der Box zur Auswahl stellen, so erreichen Sie dies durch Abändern des Extenders: A:*.BAS. Als Dateiname kann - muß jedoch nicht - ein Name angegeben werden, der dann nach dem Aufruf der Box im Eingabefeld unter AUSWAHL: erscheint. Auf diese Weise kann z.B. ein Vorschlag des Dateinamens angegeben werden. Wurde vom Benutzer ein File ausgewählt, so enthält diese Stringvariable den ausgewählen Dateinamen.

In der Variablen <Flag> wird nach Beendigung der Auswahl vom Computer mitgeteilt, ob der Benutzer OK, oder Abbruch angeklickt hat. Im ersten Fall enthält <Flag> dann der Wert 1, im letzeren den Wert 0. Damit der Benutzer auch mit der Maus arbeiten kann, muß sie vor Aufruf der Funktion ein- und anschließend wieder ausgeschaltet werden. Hier nun ein ganz einfaches Beispielprogramm zum Aufruf der Box:

```
' Minimalpfad vor Aufruf der Box angeben
Pfad$="A:\*.*"
MOUSEON' Maus einschalten
```

```
FILESELECT (Pfad$,Name$,Button)
MOUSEOFF
IF Button THEN
  PRINT "Datei ";Name$;" ausgewählt"
ELSE
  PRINT "Abbruch wurde angeklickt!"
ENDIF
....
....
```

Nachdem der Benutzer seine Auswahl getätigt hat, kann die Datei Name$ mit OPEN geöffnet und die entsprechenden Daten können eingelesen oder abgespeichert werden.

Eine etwas komfortablere Verwaltung der Fileselector-Box gestattet die folgende Prozedur:

```
0  '*****************************************************************
1  '*                          DO_FILE.BAS                          *
2  '*---------------------------------------------------------------*
3  '* Autor: Michael Maier    Version: 1.00     Datum: 16.09.1988 *
4  '*      Ein Programm aus dem 'GROSSEN ST-BASIC BUCH'             *
5  '*        (C) 1988 by DATA BECKER GmbH Düsseldorf                *
6  '*****************************************************************
7  '
8  '
9  ' die hier aufgeführte Prozedur DO_FILE kann in eigene Programme
10 'eingebaut werden, und dient der Verwaltung der FILESELECTOR.BOX
11 '
12 F_Name$="":F_Pfad$="":Ext$="*.*"
13 Do_File(F_Name$,F_Pfad$,Ext$,F_Name$,F_Pfad$,Ext$,Taste%)
14 ' Just for Info ... (für den der's wissen will)
15 IF Taste%=0 THEN
16   FORM_ALERT (1,"[1][Taste ABBRUCH gedrückt!][ Abbruch ]")
17 ELSE
18   FORM_ALERT (1,"[1][Taste OK gedrückt][ Weiter ]")
19 ENDIF
20 '
21 END
22 '
23 ' jetzt folgt die eigentliche Prozedur
24 '
25 DEF PROC Do_File(Name$,Path$,Post$,R Name$,R Path$,R Post$,R Tas%)
26    ' Lokale Variablen verwenden, damit die Prozedur auch
27    ' in anderen Programmen einsetzbar ist.
```

```
28    LOCAL R_Path$=Path$,R_Name$=Name$,T%,Drive%,Pointer%L,Ret%L
29    IF Path$="" THEN ' kein Pfad angegeben, dann einen basteln
30       Path$=" "*64'   GEMDOS Konvention Folge leisten
31       Pointer%L= LPEEK( VARPTR(Path$))+ LPEEK( SEGPTR +28)
32       ' anschließend aktuellen Pfadnamen holen
33       GEMDOS (,71, HIGH(Pointer%L), LOW(Pointer%L),0)
34       ' und zurechtstutzen
35       Path$= LEFT$(Path$, INSTR(Path$+ CHR$(0), CHR$(0))-1)
36       GEMDOS (Drive%,25)' aktuelles Laufwerk ermitteln
37       Path$= CHR$(65+Drive%)+":"+Path$+"\"
38    ENDIF
39    IF Post$="" THEN ' ohne Extension kann man schlecht arbeiten
40       Path$=Path$+"*.*"' Pauschalextension anhängen
41    ELSE
42       Path$=Path$+Post$
43    ENDIF
44    PRINT CHR$(27);"f"' Cursor aus- und Maus einschalten
45    MOUSEON
46    FILESELECT (Path$,Name$,Ret%L)
47    MOUSEOFF
48    PRINT CHR$(27);"e"' Cursor wieder einschalten
49    IF Ret%L=0 THEN '     Abbruch angeklickt
50       Path$=R_Path$'   alte Vorgaben zurückkopieren
51       Name$=R_Name$
52       Tas%=0
53    ELSE
54       ' letzten Backslash '\' suchen
55       T%= LEN(Path$)
56       WHILE T%>0 AND MID$(Path$,T%,1)<>"\"
57          T%=T%-1
58       WEND
59       Path$= LEFT$(Path$,T%)
60       Tas%=1
61    ENDIF
62 RETURN
```

Die jetzt folgenden Erklärungen zur Funktionsweise der Prozedur Do_File erfordern etwas weiterreichende Kenntnisse, als Sie sie bis jetzt besitzen, vor allem über das Betriebssystem. Dennoch sollten Sie sich nicht entmutigen lassen und die folgenden Zeilen lesen. Zudem erfahren Sie darin auch gleich, wie die Prozedur angewendet werden kann.

Der Prozedur werden drei Stringvariablen bei ihrem Aufruf übergeben, die restlichen Parameter sind lediglich Rückgabewerte. Die zu übergebenden Stringvariablen lauten (in korrekter Reihenfolge):

1. Pfadname
2. Dateiname (ohne Extender)
3. Extender für den Dateinamen

Gleich nach dem Aufruf der Prozedur werden der Pfadname und der Dateiname in zwei lokale Variablen gerettet. ST-BASIC gestattet die Deklaration einer lokalen Variable mit der Zuweisung eines Wertes zu verbinden. Von diesem Effekt wurde hier Gebrauch gemacht. Im nächsten Schritt wird überprüft, ob beim Aufruf der Prozedur ein Pfadname übergeben wurde. Ist dies nicht der Fall, muß die Prozedur einen eigenen Pfadnamen zusammenbasteln. Und dazu wird das Betriebssystem des Atari ST mißbraucht.

Die Funktion GEMDOS() (mit Parameter 71) liefert den aktuellen Pfadnamen. Sie weist diesen jedoch nicht einfach einer Variablen zu, sondern erwartet einen Zeiger auf einen bestimmten Speicherbereich, in den der Pfad dann geschrieben wird. Eine GEMDOS-Konvention sagt aus, daß dieser Pfadname maximal 64 Zeichen lang sein kann. Also füllen wir zuerst einmal einen String (Path$) mit 64 Leerzeichen. Im nächsten Schritt muß noch ermittelt werden, wo sich der String im Speicher befindet. Die auf diese Weise erhaltene Adreße (Zeiger) kann der GEMDOS-Routine übergeben werden und schon enthält die Stringvariable Pfad$ den aktuellen Pfadnamen (die Laufwerkskennung ist darin noch nicht enthalten!).

Dieser wird durch ein Nullbyte (CHR$(0)) abgeschlossen. Während es aber in der Sprache C üblich ist, eine Zeichenkette mit einem Nullbyte zu beenden, kann ST-BASIC mit diesem Nullbyte überhaupt nichts anfangen. Also muß es aus dem String entfernt werden! Dies geht am besten, indem man seine Position mit INSTR() sucht, und mit LEFT$ alle links von dieser Position (deshalb das -1) abschneidet.

Die nächste GEMDOS-Funktion (Nr. 25) weist der Variablen Drive% die Nummer des gerade aktiven Laufwerks zu. Dabei gilt:

| Inhalt von Drive% | entspricht Laufwerk |
|---|---|
| 0 | A |
| 1 | B |
| 2 | C |

Als Integer nützt uns das aktuelle Laufwerk aber überhaupt nichts! Im Pfadnamen benötigen wir einen Buchstaben. Der ASCII-Code hilft uns auch hier wieder einmal aus der Patsche: CHR$(65) liefert den Buchstaben 'A', CHR$(66) den Buchstaben B. Addieren wir zu dem Argument dieser Funktion einfach das ermittelte Laufwerk (Drive%), resultiert daraus eine Laufwerksbezeichnung von A bis ..., je nach Inhalt der Variablen Drive%: CHR$(65+Drive%). Jetzt kann der (fast) entgültig Pfad zusammengesetzt werden:

```
CHR$(65+Drive%)+":"+Path$+"\"
```

Auch um den letzten Backslash (\) müssen wir uns selbst kümmern. Mit String-Addition ist dies aber kein Problem. An den auf diese Weise erhaltenen Pfadnamen muß noch die Extension angehängt werden. Ist kein Extender als Parameter übergeben worden, so wird einfach die Pauschal-Extension *.* an den Pfad gehängt. Vor dem Aufruf der File-Selector-Box noch den Cursor aus- aber die Maus einschalten! Wurde vom Benutzer Abbruch angeklickt, so werden die ursprünglichen Parameter wiederhergestellt und an das aufrufende Hauptprogramm zurückgegeben. Ebenso wird das Flag auf 0 gesetzt und zurückgegeben. Hat der Benutzer dagegen den Button OK angeklickt, so muß der letzte Teil des Pfadnamens (die zuvor an diesen gehängte Extension) wieder entfernt werden. Dies wird in einer Schleife erledigt. Angefangen vom Stringende wird jedes Zeichen untersucht, bis der erste Backslash entdeckt wird. Nach diesem wird der Pfad dann abgeschnitten.

Ein Algorithmus zum Abschneiden der Extension, der mir gut gefallen hat, findet sich im Programmierhandbuch zum ST-BASIC: Die Funktion INSTR sucht ein beliebiges Zeichen innerhalb eines Strings und gibt seine Position zurück. Dabei beginnt sie mit der Suche beim ersten Zeichen. In diesem Fall muß jedoch ab dem Stringende gesucht werden! Deshalb wird der String mit MIRROR$ einfach umgedreht, und schon kann INSTR zum Suchen des letzten Backslash verwendet werden:

```
Path$= LEFT$(Path$,LEN(PATH$)-INSTR(MIRROR$(Path$)+"\","\")
```

Der auf diese Weise ermittelte Pfad, der Dateiname incl. Extender, sowie der Wert 1 für den OK-Button werden noch zurückgegeben. Fertig! Möchten Sie jetzt die ausgewählte Datei öffnen (z.B. um sie einzulesen), geschieht dies mit:

```
OPEN "I",1, F_Pfad$+F_Name$
```

Was geschieht aber, wenn die zurückgegebene Datei gar nicht auf der Diskette existiert? Dann erhält man prompt eine Fehlermeldung! Diese wäre ja noch zu verkraften, schlimmer ist jedoch, daß sich der Interpreter dabei verabschiedet. Gute Programme machen dies nicht! Folglich muß der Fehler abgefangen werden. ST-BASIC bietet dafür eine Möglichkeit an:

## 3.7 Fehler abfangen

Tritt während der Programmabarbeitung ein Fehler auf, kann statt der Ausgabe der Fehlermeldung und des Programmabbruchs in eine eigene Fehlerbehandlungsroutine gesprungen werden. Dort ist es dann möglich nachzufragen, welcher Fehler aufgetreten ist, und eine entsprechende Meldung auszugeben. Die Anweisung für ST-BASIC bei Auftreten eines Fehlers an ein vorbestimmtes Ziel (Marke, Zeilennummer) zu springen lautet:

```
ON ERROR GOTO <Ziel>
```

und wird normalerweise an den Programmanfang gesetzt. Möchten Sie die Fehlerbehandlung abschalten, genügt der Befehl:

```
ON ERROR GOTO 0
```

d.h. nach einem Fehler wird das Programm wieder wie gewohnt abgebrochen. Auch ein CLEAR (Löschen aller Variableninhalte), RUN oder NEW schaltet die Fehlerbehandlung mit ON ERROR GOTO wieder ab. Eine Änderung des Programmcodes bewirkt ebenfalls ein Abschalten dieser Funktion.

Um innerhalb der Fehlerbehandlungsroutine überhaupt prüfen zu können, welcher Fehler aufgetreten ist, existiert eine Systemvariable (Variable, die vom Computer verwaltet wird) namens

```
ERR
```

in der die Nummer des Fehlers festgehalten wird, sowie

```
ERR$
```

die den Text der Fehlermeldung enthält, der ohne ein ON ERROR GOTO ausgegeben worden wäre. In

```
ERL
```

wird schließlich noch die Nummer der Zeile festgehalten, in der der Fehler aufgetreten ist. Eine Liste der einzelnen Fehlermeldungen und der dazugehörigen Nummern finden Sie im Anhang dieses Buches.

Die Meldung File not found (Datei auf der Diskette nicht vorhanden) trägt die Nummer 53. Besitzt die Variable ERR also den Inhalt 53, handelt es sich bei dem aufgetretenen Fehler um eine nicht gefundene Datei:

```
ON ERROR GOTO Fehler
....
....
Do_File(Pfad$,Name$,Ext$,Pfad$,Name$,Ext$,Flag%)
IF Flag% THEN
   OPEN "I",1,Pfad$+Name$
     WHILE NOT EOF(1)
       ......
         ......
```

```
    WEND
   CLOSE
 ENDIF
 ....
 .....
-Fehler
 IF ERR=53 THEN
 ' Angegebene Datei nicht vorhanden!
   FORM_ALERT(1,"[3][Ich kann die Datei nicht lesen!][Abbruch]")
      ELSE
      ' war anscheinend ein anderer Fehler ...
     ENDIF
      RESUME NEXT
      ....
      ....
```

Tritt ein Fehler auf, so springt der Computer in die Routine Fehler. Dort wird zuerst geprüft, ob es sich bei diesem Fehler um eine nicht gefundene Datei handelt (Fehlernummer 53 in der Systemvariablen ERR). Ist dies der Fall, wird eine Alertbox mit einer Fehlermeldung ausgegeben. Ansonsten müßte noch eine Fehlermeldung hinter ELSE folgen (es können ja noch andere Fehlerquellen auftreten!). Der Befehl RESUME veranlaßt den Computer schließlich, mit der Programmabarbeitung fortzufahren:

```
RESUME
```

Den Befehl RESUME gibt es in drei Variationen:

*1. RESUME <Ziel>* Verläßt die Fehlerbehandlungsroutine und setzt die Programmausführung bei <Ziel> fort.

*2. RESUME NEXT* Verläßt die Fehlerbehandlungsroutine und setzt die Programmausführung hinter der Zeile fort, in der der Fehler aufgetreten ist.

*3. RESUME* Verläßt ebenfalls die Fehlerbehandlungsroutine, versucht dann allerdings noch einmal, den Befehl auszuführen, der den Sprung in die Fehlerroutine verursacht hat.

Möchten Sie die Fehlerroutine testen, kann ein Fehler auch auf Befehl ausgelöst werden:

```
ERROR <Fehlernummer>
```

erzeugt einen Fehler mit der angegebenen Fehlernummer. Dabei ist zu beachten, daß ERROR innerhalb des Programms stehen muß, damit die Fehlerbehandlungsroutine angesprungen wird, andernfalls (im Direktmodus) werden die erzeugten Fehler immer auf dem Monitor ausgegeben.

## 3.8 Backup-Dateien

Mit den bisher besprochenen Werkzeugen können auf einfachste Weise mehrere Dateien mit der Adreßverwaltung bearbeitet werden. Man wählt dazu einfach die gewünschte Datei in der File-Selector-Box aus, liest sie in den Speicher und ändert sie bei Bedarf ab. Vor dem Verlassen des Programms muß dann diese Datei noch auf Diskette zurückgeschrieben werden.

Dabei wird jedoch die alte Datei zerstört, da sie vom Programm einfach mit dem neuen Inhalt überschrieben wird. (Innerhalb eines Directories können keine zwei Dateien den gleichen Namen tragen). Möchte man beide Dateien behalten, muß die alte Adreßdatei umbenannt werden. Es hat sich eingebürgert, vor dem Überschreiben der Datei erst eine sogenannte BACKUP-Datei anzufertigen. Diese trägt zwar den gleichen Dateinamen, ihr Extender wird jedoch in BAK (Abkürzung für BACKUP) umgetauft. Wird nun die Adreßdatei zurückgeschrieben, wird die alte Datei nicht zerstört, da sie einen anderen Namen besitzt (der sich zwar nur in der Extension unterscheidet, aber das macht ja nichts!).

Da bereits eine BACKUP-Datei auf der Diskette vorhanden sein kann (von einem früheren Zurückschreiben der Datei), muß diese erst noch entfernt werden. Hier die benötigten Befehle:

```
KILL <Name>
```

Löscht die Datei <Name> von der Diskette. Gehen Sie deshalb schon in Ihrem eigenen Interesse mit diesem Befehl sorgsam um, denn eine einmal gelöschte Datei ist in den meisten Fällen unwiederbringlich verloren. Befindet sich die zu löschende Datei in einem Ordner, so muß der Pfad mit angegeben werden.

```
NAME AS
```

Dient zum Umbenennen einer Datei. Die Syntax lautet:

```
NAME <alter Name> AS <neuer Name>
```

NAME "OM_BASIC.PRG" AS "ST_BASIC.PRG" tauft die Datei OM_BASIC.PRG in ST_BASIC.PRG um. Auch hier muß der Pfad vor dem Dateinamen angegeben werden, wenn sich die umzubenennende Datei in einem Ordner befindet.

```
COPY
```

kopiert eine Datei, und war nach der Syntax:

```
COPY <Quelldatei> TO <Zieldatei>
```

COPY "A:\ST_BASIC.PRG" TO "D:\ST_BASIC.PRG" kopiert die Datei ST_BASIC von Laufwerk A in Laufwerk D (Ramdisk). COPY akzeptiert auch Wildcards (*, ?) im Namen <Quelldatei>, nicht jedoch in der <Zieldatei>!

```
BACKUP
```

Als letzer Vertreter der Kopierbefehle, hier der Befehl, der eine Kopie von der angegebenen Datei anfertigt, die dann den Extender BAK trägt.

```
BACKUP "MINIADR.DAT"
```

Fertigt eine Kopie der Datei MINIADR.DAT mit dem Namen MINIADR.BAK.

Ehe eine Backupdatei angefertigt wird, muß erst geprüft werden, ob sich eine Datei mit dem angegebenen Namen auf Diskette befindet (andernfalls kann man sich die Arbeit sparen. Wovon sollte man auch ein Backup erstellen?). Ist dies der Fall, wird eine evtl. von dieser Datei schon bestehende Backupdatei gelöscht und anschließend die neue Backupdatei gezogen.

Die folgende Prozedur übernimmt die Anferigung einer Backupdatei. Sie benötigt jedoch eine Funktion, die feststellt, ob eine bestimmte Datei auf Diskette vorhanden ist. Da ST-BASIC keine derartige Funktion bereitstellt, habe ich selbst eine entwickelt: FN EXIST(Filename$) liefert als Ergebnis den Wert -1 (wahr), wenn die angegebene Datei auf der Diskette vorhanden ist, andernfalls schreibt sie den Wert 0 (falsch) zurück. Die genaue Erklärung dieser Funktion erfolgt zu einem späteren Zeitpunkt, da weitergehende Kenntnisse zu ihrem Verständnis nötig sind!

Zuerst die Prozedur "Rename", die bei Bedarf eine BACKUP-Datei anfertigt:

```
0  '*****************************************************************
1  '*                          RENAME.BAS                           *
2  '*---------------------------------------------------------------*
3  '* Autor: Michael Maier    Version: 1.00     Datum: 16.09.1988 *
4  '*      Ein Programm aus dem 'GROSSEN ST-BASIC BUCH'             *
5  '*        (C) 1988 by DATA BECKER GmbH Düsseldorf                *
6  '*****************************************************************
7  '
8  '
9  ' Die hier aufgeführte Prozedur benötigt die Funktion EXIST.BAS
10 ' Ist ein File mit dem angegebenen Namen auf der Diskette vor-
11 ' handen, so wird eine BACKUP-Datei erstellt. Eine bereits
12 ' existierende BACKUP-Datei wird zuvor gelöscht.
13 '
14 ' Aufruf der Prozedur:  Rename(<Filename>)
15 '
16 '
17 DEF PROC Rename(Filename$)
18    IF FN Exist%L(Filename$) THEN
19       ' Datei existiert bereits => BACKUP auch schon vorhanden?
```

```
20       IF FN Exist%L( LEFT$(Filename$, INSTR(Filename$,"."))+"BAK")
21          ' dann BACKUP einfach löschen ...
22          THEN KILL LEFT$(Filename$, INSTR(Filename$,"."))+"BAK"
23       ENDIF
24       ' jetzt BACKUP anfertigen
25       BACKUP (Filename$)
26    ENDIF
27 RETURN
```

Der Vollständigkeit halber hier auch gleich das Listing der Funktion EXIST, ohne die obige Prozedur nicht arbeitet! (Erklärung folgt später!)

```
0  '*********************************************************************
1  '*                          EXIST.BAS                                *
2  '*-------------------------------------------------------------------*
3  '* Autor: Michael Maier    Version: 1.00     Datum: 16.09.1988 *
4  '*      Ein Programm aus dem 'GROSSEN ST-BASIC BUCH'                 *
5  '*        (C) 1988 by DATA BECKER GmbH Düsseldorf                    *
6  '*********************************************************************
7  '
8  '
9  ' Die folgende Funktion überprüft, ob ein File 'Name$' auf der
10 ' Diskette vorhanden ist
11 ' File vorhanden       => Returnwert ist '-1'   ('wahr')
12 ' File nicht vorhanden => Returnwert ist '0'   ('falsch')
13 '
14 IF NOT FN Exist%L("EXIST.BAS") THEN
15    FORM_ALERT (1,"[3][Datei nicht vorhanden!][ Abbruch ]")
16 ELSE
17    FORM_ALERT (1,"[1][Alles in Ordnung][ OK ]")
18 ENDIF
19 END
20 '
21 DEF FN Exist%L(Name$)
22    LOCAL T%
23    '
24    OPEN "F",1,Name$,55
25    IF EOF(1) THEN
26    ' Keine Files auf der Diskette vorhanden
27    ' => Fehlermeldung (0) zurückgeben
28       CLOSE 1
29       RETURN (0)
30    ELSE
31       ' DTA-Buffer bereitstellen
```

```
32        FIELD 1,30 AS Buffer$,14 AS File$
33        GET 1,0
34        ' Datei vorhanden => '-1' zurückgeben
35        IF Name$= LEFT$(File$, INSTR(File$, CHR$(0))-1) THEN
36           CLOSE 1
37           RETURN (-1)
38        ENDIF
39     ENDIF
40 CLOSE 1
41 ' File war nicht auf der Diskette
42 RETURN (0)
```

## 3.9 Wir sorgen für Ordnung auf der Diskette

Bei der Verwaltung der File-Selector-Box war schon einmal die Rede von ihnen, den Ordnern, die mithelfen, bei dem ganzen Wust von Dateien auf einer Diskette nicht den Überblick zu verlieren. Doch einen Ordner vom Programm aus zu erzeugen, war für uns bis jetzt nicht möglich. Dem soll in diesem Kapitel abgeholfen werden!

### Ordner erzeugen

Um einen Ordner zu erzeugen, wird in ST-BASIC der Befehl

```
MKDIR <Ordnername>
```

benutzt. Er erstellt einen Ordner mit der Bezeichnung <Ordnername> auf dem aktuellen Laufwerk.

Ein Ordner stellt zunächst einmal einen Eintrag im Inhaltsverzeichnis der Diskette, dem Directory, dar. Wird der Ordner geöffnet bzw. der Pfad um einen Ordner erweitert, erscheint ein Unterinhaltsverzeichnis (Subdirectory), in dem die in diesem Ordner enthaltenen Dateien aufgeführt sind.

### Ordner löschen

Ein Ordner kann natürlich auch wieder von der Diskette gelöscht werden! Dazu dient der Befehl

```
RMDIR <Ordnername>
```

Das Subdirectory (Unterinhaltsverzeichnis) wird aus dem (Haupt-)Directory entfernt.

### Immer auf dem richtigen Pfade

Existieren auf einer Diskette mehrere Ordner und somit logischerweise auch mehrere Unterinhaltsverzeichnisse (Subdirectories), muß mit Hilfe eines entsprechenden Pfades jedes Unterverzeichnis angesprochen werden können, um die gewünschte Datei aus einem Ordner einzulesen.

Enthält der Ordner DOKUMENT.SDO die Datei BRIEF.SDO, so schlägt jeder Versuch fehl, die Datei mit

```
OPEN "I",1,"BRIEF.SDO"
```

einzulesen, solange Sie sich noch im Hauptinhaltsverzeichnis befinden. Wird dagegen der Pfadname mit angegeben, wird der Inhalt des Subdirectories erreicht und kann eingelesen werden:

```
OPEN "I",1,"A:\DOKUMENT.SDO\BRIEF.SDO"
```

Um diese doch recht umständliche Prozedur abzukürzen (stellen Sie sich einmal vor, Sie möchten im Ordner DOKUMENT.SDO, den Ordner BRIEFE.SDO, der sich wiederum im Ordner GESCHAEFT.SDO befindet, ansprechen, um daraus verschiedene Dateien zu lesen, und Sie müßten jedesmal den vollen Dateinamen als Pfad angeben!) kann ein Unterinhaltsverzeichnis zum aktuellen Verzeichnis ernannt werden. Alle Operationen beziehen sich dann ab sofort auf dieses Verzeichnis, obgleich kein Pfad angegeben wird.

Zum Umstellen auf ein Subdirectory dient der Befehl

```
CHDIR <Ordnername>
```

CHDIR "DOKUMENT.SDO" erklärt den Inhalt des Subdirectories DOKUMENT.SDO zum aktuellen Verzeichnis, auf das sich dann ab sofort alle Diskettenbefehle beziehen. Ein SAVE speichert die ihm anvertraute Programmdatei in diesen Ordner, der Befehl FILES (Inhalsverzeichnis ausgeben) gibt nur noch den Inhalt dieses Ordners aus. Möchten Sie wieder in das Hauptdirectory zurückkehren, so genügt als <Ordnername> die Angage zweier Punkte gefolgt von einem Backslash:

```
CHDIR "..\"
```

und schon befinden Sie sich wieder im Hauptdirectory. Gleichzeitig kann das aktuelle Laufwerk mit CHDIR umgestellt werden, indem der Pfad um die Laufwerkskennung

```
CHDIR "A:\DOKUMENT:SDO"
```

erweitert oder nur die Laufwerkskennung alleine angegeben wird:

```
CHDIR "D:"
```

setzt das Defaultlaufwerk (aktuelles Laufwerk). Ab sofort wird bei einem Diskettenzugriff das Laufwerk (Ramdisk) D angesprochen.

## 3.10 Relative Dateien

Eine relative oder Random-access-Datei besteht, wie Sie bereits wissen, aus einzelnen Datensätzen. Ähnlich wie sich eine Kartei aus mehreren Karteikarten, die wiederum verschiedene Daten tragen, zusammensetzt, enthält auch ein Datensatz verschiedene Angaben:

```
Name + Vorname + Straße + Plz + Ort + Telefon
```

ergibt z.B. einen Datensatz für eine Adreßdatei. Damit einzelne Datensätze (Records) anschließend eingelesen werden können, müssen alle Records einer Datei die gleiche Länge besitzen. Diese Datensatzlänge wird als vierter Parameter beim Befehl OPEN angegeben. Der Dateimodus ist jetzt nicht mehr "I" oder "O", sondern "R". Dies gilt sowohl für das Lesen als auch für das Schreiben der Datei:

```
OPEN "R",1,"MINIDDR.REL",<Datensatzlänge>
```

### Datensatz einrichten

Zum Einrichten eines Datensatzes existiert der Befehl

```
FIELD <Dateinummer>,<Länge> AS <Buffervariable>,...
```

Der erste Parameter hinter FIELD ist die Dateinummer, die auch beim Befehl OPEN angegeben wurde. Anschließend werden die einzelnen Puffervariablen (Zwischenspeicher), aus denen sich wiederum der gesamte Record zusammensetzt, sowie die Anzahl der Zeichen, die jede einzelne Puffervariable enthalten soll, angegeben. Die Gesamtlänge der Puffervariablen darf die beim OPEN angegebene Datensatzlänge nicht überschreiten. Ein Beispiel verdeutlicht das soeben Gesagte wohl am besten.

Ein Datensatz soll aus folgenden Daten bestehen:

| Daten | Länge |
|---|---|
| Name | 15 |
| Vorname | 15 |
| Strasse | 20 |
| Ort | 20 |

Die Datensatzlänge beträgt 70, folglich muß der Befehl zum Öffnen der Datei lauten:

```
OPEN "R",1,"MINIADR.REL",70
```

Anschließend wird der Record mit FIELD eingerichtet:

```
FIELD 1,15 AS Na$, 15 AS Vorna$, 20 AS Stra$, 20 AS Ort$
```

Die einzelnen Puffervariablen können (nach dem Einlesen eines Datensatzes, ansonsten ergibt es natürlich keinen Sinn!) wie normale Variablen auch abgefragt werden (in IF, Schleifen, usw.), oder mit PRINT ausgegeben werden:

```
IF Na$ = "Arthur" THEN ....
WHILE Na$ <> ""
   ....
WEND

PRINT Na$,Vorna$,Stra$,Ort$
```

Die Zuweisung eines Wertes an eine Puffervariable darf nur mit MID$()=, LSET oder RSET erfolgen. Zumindest die Zuweisung mit MID$()= müßte Ihnen mittlerweile geläufig sein. Die zwei anderen Zuweisungsformen werden wir jetzt besprechen:

### LSET und RSET

Die Syntax entspricht der von LET , im Gegensatz zu LET ist ein LSET bzw. RSET aber unbedingt nötig, wenn eine Puffervariable mit einem Wert versorgt werden soll:

```
LSET Na$ = "MAIER"
```

weist der Variablen Na$ den Inhalt MAIER zu. Das Besondere dabei ist, daß der Stringausdruck in die Variable linksbündig eingesetzt wird. Nicht benutzte Zeichen (Differenz zwischen der Länge der Puffervariable und der zugewiesenen Zeichenkette) werden mit Leerzeichen aufgefüllt (Na$ = "MAIER" + CHR$ (32)*(15-LEN("MAIER"))). Enthält die Zeichenkette mehr Buchstaben, als die Puffervariable aufnehmen kann, so werden die überhängenden Buchstaben einfach abgeschnitten.

Im Gegensatz zu LEST setzt RSET den Stringausdruck nicht links-, sondern rechtsbündig in die Puffervariable ein. Auch hier werden überflüssige (nicht benötigte) Zeichen der Puffer-

variable mit Leerzeichen aufgefüllt. Ein zu langer Stringausdruck wird ebenfalls abgeschnitten. Sobald der Record mit den nötigen Daten versorgt ist, kann er auf Diskette geschrieben werden. Zum Lesen und Schreiben des Puffers gibt es zwei eigene Befehle.

### Puffer auf Diskette schreiben und wieder lesen

```
PUT <Dateinummer>, <Datensatznummer>
```

schreibt die sich im Puffer befindenden Daten in die geöffnete Datei, und zwar an die Position <Datensatznummer>.

```
GET <Dateinummer>, <Datensatznummer>
```

Liest die an der Position <Datensatznummer> stehenden Daten aus der Datei mit der Kanalnummer <Dateinummer> in den Puffer. Dort können sie dann aus den einzelnen Puffervariablen abgeholt werden.

Ehe die graue Theorie an einem Beispiel anschaulich dargestellt werden soll, muß noch ein Problem abgeklärt werden. Sollen nämlich neben Zeichen auch noch Zahlen in einer relativen Datei abgespeichert werden, müssen diese erst auf eine genau vorgegebene Länge gebracht werden (für die Puffervariable). Zudem handelt es sich bei dem Puffer um Stringvariablen, denen keine Zahlen zugewiesen werden können. Deshalb gibt es in ST-BASIC Funktionen, die Zahlen in einen String mit 2, 4, 6, oder 10 Bytes umwandeln. Die so umgewandelten Zahlen können dann einer Puffervariablen mit LSET bzw. RSET zugewiesen werden.

| Funktion | konvertiert in eine | in einen |
|---|---|---|
| MKI$(Zahl) | 16 Bit Integerzahl | 2 Byte String |
| MKIL$(Zahl) | 32 Bit Integerzahl | 4 Byte String |
| MKS$(Zahl) | Single-Float | 6 Byte String |
| MKD$(Zahl) | Double-Float | 10 Byte String |

und andersherum geht's mit CVx:

| Funktion | konvertiert einen | in eine |
|---|---|---|
| CVI(String) | 2 Byte String | 16 Bit Integer |
| CVIL(String) | 4 Byte String | 32 Bit Integer |
| CVS(String) | 6 Byte String | Single-Float |
| CVD(String) | 10 Byte String | Double-Float |

Die beiden Funktionen X$ = MKI$(Zahl) und Zahl = CVI(X$) heben sich also wieder auf.

## 3.11 Minidatei - diesmal relativ

Ich habe die Ihnen bereits aus dem zweiten Teil dieses Buches bestens bekannte Miniadreßverwaltung noch ein weiteres Mal umgearbeitet, diesmal jedoch für relative Datenspeicherung!

Ein Record setzt sich aus den Komponenten

| Daten | Länge |
|---|---|
| Name | 15 |
| Vorname | 15 |
| Strasse | 32 |
| Plz | 4 |
| Ort | 30 |
| Tel | 11 |
| Geb | 10 |

zusammen und erhält folglich die Datensatzlänge 117. Ehe Adreßen erfasst werden können, muß eine (leere) Adreßdatei eingerichtet werden. Diese Aufgabe übernimmt Menüpunkt 4. Die Auswahl, bzw. Neueingabe eines Dateinamens erfolgt über die File-Selector-Box, die in der Do_File-Routine verwaltet wird. Befindet sich bereits eine Datei mit dem angegebenen Namen auf der Diskette, sorgt die Prozedur Rename dafür, daß vor dem Einrichten der neuen Datei erst einmal ein Backup angefertigt wird. Sicher ist sicher!

Anschließend wird der Pfad gesetzt (falls die Datei in einem Ordner angelegt werden soll) und der von der Prozedur Do_File zurückgegebene Dateiname (Name$) als Filename verwendet. Die

Datei enthält so viele Datensätze, wie in der Variablen "Gro", mit der auch die einzelnen Felder dimensioniert wurden, angegeben ist.

Neu ist auch die Prozedur Speichern(Nummer%), die den Variableninhalt des Index Nummer% auf Diskette schreibt. Nach dem Öffnen der Datei Name$ wird erst einmal der Datensatz mit FIELD eingerichtet. Da sämtliche Variablen, aus denen sich der Puffer zusammensetzt, in einer Anweisung untergebracht werden müssen, aus drucktechnischen Gründen jedoch maximal 72 Zeichen in einer Zeile stehen dürfen, müssen die letzten beiden Puffervariablen zwei verschiedene Datensatzkomponenten aufnehmen. Doch das macht gar nichts! Auf diese Weise kann ich Ihnen wenigstens demonstrieren, mit welchen Befehlen eine Puffervariable gefüllt werden darf.

So, der Puffer wäre definiert, jetzt muß er noch mit den Daten versorgt werden, die auf Diskette geschrieben werden sollen. LSET und MID$()= erledigen diese Aufgabe. Anschließend wird der komplette Record auf der Diskette abgespeichert und der Kanal geschlosssen, damit es keine Konflikte mit anderen geöffneten Kanälen gibt. Da in relativen Dateien einzelne Records abgespeichert werden können, wird dies gleich beim Erfassen bzw. Korrigieren von Adreßen erledigt.

Und weil wir gerade von der Adreßerfassung sprechen: Damit ein Aufruf der Prozedur Speichern(Nummer%) erfolgen kann, wird ein Dateiname benötigt. Deshalb wird zuerst einmal eine Unterroutine "Taufe" aufgerufen, die abklärt, ob in Name$ bereits ein Dateiname gespeichert ist (vom Anlegen oder Einlesen einer Datei). Ansonsten erscheint die File-Selectorbox, und ein Dateiname kann angegeben werden. Auch die Routine zum Einlesen einer Datei mußte geändert werden. Zuerst werden sämtliche Variableninhalte gelöscht, damit nicht zwei Dateien miteinander vermischt werden. Dies könnte man durchaus einer FOR...NEXT-Schleife

```
FOR T%=1 TO Gro ' Dimensionierung
   Loesche(T%)
NEXT T%
```

überlassen, die dann allerdings mit Sicherheit unnötige Arbeit verrichten würde. Enthält eine Variable nämlich keine Daten mehr, so ist das Datei-Ende bereits erreicht, und die Schleife kann verlassen werden. Deshalb wurde hier einer (abweisende) WHILE...WEND-Schleife der Vorzug gegeben. In einer REPEAT...UNTIL-Schleife werden dann - beginnend vom ersten Record - alle Datensätze eingelesen, bis ein nicht belegter Datensatz im Puffer steht. Dies ist das Zeichen dafür, daß die Datei schon komplett eingelesen ist. Die Schleife kann abgebrochen und der letzte (leere) Datensatz gelöscht werden. Das Einlesen der gesamten Datei erfolgt deshalb, da in einer relativen Datei, die sich auf Diskette befindet, die Suche eines Namens relativ lange dauert.

Ist eine Datei eingelesen, besitzen alle Variablen ihre maximal erlaubten Längen, da die nicht benötigten Stellen im Puffer mit Leerzeichen aufgefüllt wurden. Um bei der Suche nach einem bestimmten Namen nicht in Teufels Küche zu geraten (MÜLLER <> MÜLLER), wird die Suchvorgabe ebenfalls bis zur maximal erlaubten Länge (15 Zeichen) mit Leerzeichen aufgefüllt. Eine äußerst noble Geste wäre es natürlich, wenn die Leerzeichen am Ende einer Variablen gleich nach dem Einlesen abgeschnitten würden. Möchten Sie das Programm in dieser Richtung ergänzen, sollten Sie folgendes beachten:

- Im Prinzip kann mit INSTR() die Position des ersten Leerzeichens ermittelt und mit LEFT$ die Zeichenkette zurechtgestutzt werden. Dieses Verfahren funktioniert aber nur dann, wenn der Datensatz auch wirklich belegt war. Ein Eintrag, der nur Leerzeichen enthält, führt zu einem Rückgabewert der Funktion INSTR() von 1. Das Leerzeichen muß aber abgeschnitten werden, also LEFT$( ...,INSTR()-1), und daraus resultiert dann ein Wert 0, der auch prompt zu einer Fehlermeldung führt.

- Enthält andererseits der Datensatz kein einziges Leerzeichen, weil sein Inhalt alle Stellen in Beschlag nimmt, sieht es übel aus! INSTR() liefert den Wert 0 und siehe da, eine Fehlermeldung auf dem Monitor!

- Ferner können einzelne Datensätze durchaus rechtmäßigerweise Leerzeichen enthalten, die nur als Trennzeichen dienen, z.B. ein Straßenname: "Am Sonnenhang 99" enthält gleich zwei Leerzeichen, hinter denen dann allerdings noch wichtige Daten folgen. Die Anweisung

```
Strasse$ = LEFT$(Strasse$, INSTR(Strasse$,CHR$(32))-1)
```

  würde den Straßennamen verstümmeln. Übrig bliebe lediglich das Wörtchen "Am", mit dem ein Postbote herzlich wenig anfang kann, muß er einen Brief mit dieser Adreße zustellen. Abhilfe kann hier wieder - wenn es schon unbedingt die Funktion INSTR() sein muß - ein Umdrehen der Zeichenkette mit MIRROR$ schaffen. Dann müssen Sie allerdings noch die von INSTR() ermittelte Position von der maximalen Variablenlänge subtrahieren, um sie in LEFT$() als Parameter benutzen zu können.

Sie sehen schon, es ist gar nicht so einfach, die Leerzeichen wieder aus dem Variableninhalt zu entfernen, da geht es schon wesentlich schneller, bei einem Vergleich die nicht mit Leerzeichen aufgefüllte Variable mit Spaces aufzublasen. Und wenn die Leerzeichen nicht anderweitig stören oder gar Fehler verursachen, warum sollte man sie dann entfernen? Aber versuchen Sie es ruhig! Übung macht bekanntlich den Meister!

Und hier nun das Listing der relativen Adreßverwaltung, die übrigens nicht hundertprozentig gegen mögliche Fehlbedienung geschützt ist; schließlich soll sie nur als Beispiel dienen, und keiner Datenbank Konkurrenz machen:

```
0  '*****************************************************************
1  '*                        MINIDATR.BAS                           *
2  '*---------------------------------------------------------------*
3  '* Autor: Michael Maier    Version: 1.00     Datum: 15.08.1988 *
4  '*      Ein Programm aus dem 'GROSSEN ST-BASIC BUCH'             *
5  '*        (C) 1988 by DATA BECKER GmbH Düsseldorf                *
6  '*****************************************************************
7  '
8  '
9  MODE "D"
10 DEF FN Screen$(X$)= CHR$(27)+X$
```

```
11 '
12 Gro%L=100' falls nötig einfach ändern
13 DIM Name$(Gro%L),Vorname$(Gro%L),Strasse$(Gro%L)
14 DIM Plz$(Gro%L),Ort$(Gro%L),Tel$(Gro%L),Geb$(Gro%L)
15 '
16 Fehler$="[3][Diese Funktion ist leider¦    nicht möglich!!!][Sorry]"
17 Mel$="[1][Diese Datei enthält"+ STR$(Gro%L)+"¦    Datensätze!][OK]"
18 Fehler_2$="[3][Ich kann die Datei¦  nicht finden!][Sorry]"
19 '
20 REPEAT
21    CLS
22    PRINT @(0,1);"*"*78
23    FOR Y%=1 TO 5: PRINT @(Y%,1);"*";@(Y%,78);"*": NEXT Y%
24    PRINT @(6,1);"*"*78
25    PRINT @(2,28);"MINIDATEI -  Hauptmenü"
26    PRINT @(3,28);"----------------------"
27    PRINT @(4,16);"Ein Demoprogramm aus dem grossen ST-BASIC Buch"
28    PRINT @(9,28);"1.  Name erfassen"
29    PRINT @(11,28);"2.  Name korrigieren"
30    PRINT @(13,28);"3.  Name suchen"
31    PRINT @(15,28);"4.  Datei anlegen"
32    PRINT @(17,28);"5.  Datei laden"
33    PRINT @(19,28);"6.  Programm verlassen"
34    PRINT @(22,29);FN Screen$("p");" Bitte waehlen Sie! ";FN
Screen$("q")
35    PRINT FN Screen$("f")' Cursor ausschalten
36    '
37    A%L=0
38    REPEAT
39      A$= INKEY$
40      IF A$<>"" THEN
41         A%L= ASC( RIGHT$(A$,1))-48
42      ENDIF
43    UNTIL A%L>0 AND A%L<7
44    PRINT FN Screen$("e")' Cursor wieder einschalten
45    ON A%L GOSUB Erfassen,Korrigieren,Suchen,Anlegen,Laden
46 UNTIL A%L=6' Schleife wiederholen, bis '6' gedrückt
47 CLS
48 END
49 '
50-Erfassen
51   CLS
52   ' einen Namen braucht die Datei auch noch
53   Taufe(Button%)
54   ' bei 'Abbruch' oder fehlendem Namen zurück zum Hauptmenü
55   IF Button%=0 OR Name$="" THEN
```

```
56      RETURN
57    ENDIF
58    Header$="***************  Name erfassen  ***************"
59    ' zuerst einmal den ersten freien Eintrag suchen!
60    T%=1
61    WHILE Name$(T%)<>"" AND T%<Gro%L
62      T%=T%+1
63    WEND
64    Formular(Header$)
65    REPEAT
66      Eingabe(T%)
67      PRINT FN Screen$("f")' Cursor stört hier bloß!
68      PRINT @(19,15);FN Screen$("p");"N";FN Screen$("q");
69      PRINT "ächster Name    ";FN Screen$("p");"K";FN Screen$("q");
70      PRINT "orrektur   ";FN Screen$("p");"Z";FN Screen$("q");
71      PRINT "urück ins Hauptmenü"
72      A$="": INPUT "  ";A$ USING "+n+z+ku>",Ret%L,1,32
73      PRINT FN Screen$("e")'Cursor wieder einschalten
74      IF A$="K" THEN
75        Formular(Header$)
76        Anzeige(T%)
77      ELSE
78        IF A$="N" AND T%<Gro%L THEN
79          Speichern(T%)
80          T%=T%+1
81          Formular(Header$)
82          Anzeige(T%)
83        ENDIF
84      ENDIF
85    UNTIL A$="Z"
86    Speichern(T%)' damit auch keine Daten verloren gehen
87  RETURN
88  '
89-Korrigieren
90  Header$="***************   Name korrigieren   ***************"
91  PRINT FN Screen$("f")
92  T%=1
93  Formular(Header$)
94  REPEAT
95    Anzeige(T%)
96    PRINT @(19,4);FN Screen$("p");"N";FN Screen$("q");"ächster Name  ";
97    PRINT FN Screen$("p");"L";FN Screen$("q");"etzter Name  ";
98    PRINT FN Screen$("p");"K";FN Screen$("q");"orrektur  ";
99    PRINT FN Screen$("p");"E";FN Screen$("q");"ntfernen  ";
100   PRINT FN Screen$("p");"Z";FN Screen$("q");"urück ins Hauptmenü"
101   '
```

```
102   A$=""
103   REPEAT
104     A$= INKEY$
105     IF A$<>"" THEN
106       A$= UPPER$( RIGHT$(A$,1))
107     ENDIF
108   UNTIL A$="N" OR A$="L" OR A$="K" OR A$="E" OR A$="Z"
109   '
110   IF (A$="N") AND (T%<Gro%L) AND (Name$(T%+1)<>"") THEN
111     T%=T%+1
112   ELSE
113     IF A$="N" THEN
114       FORM_ALERT (1,Fehler$)
115     ENDIF
116   ENDIF
117   IF A$="L" AND T%>1 THEN
118     T%=T%-1
119   ELSE
120     IF A$="L" THEN
121       FORM_ALERT (1,Fehler$)
122     ENDIF
123   ENDIF
124   IF A$="K" THEN
125     PRINT FN Screen$("e")
126     PRINT @(19,1);" "*78
127     Eingabe(T%)
128     Speichern(T%)
129     PRINT FN Screen$("f")
130   ENDIF
131   IF A$="E" THEN
132     MOUSEON
133     FORM_ALERT (2,"[2][Datensatz wirklich löschen?][Ja|Nein]",But%)
134     MOUSEOFF
135     IF But%=1 THEN
136       Delete(T%)
137     ENDIF
138   ENDIF
139 UNTIL A$="Z"
140 RETURN
141 '
142-Suchen
143 Header$="***************   Name suchen    ***************"
144 Name$(0)="":Vorname$(0)="":Strasse$(0)="":Plz$(0)=""
145 Ort$(0)="":Tel$(0)="":Geb$(0)=""
146 T%=0
147 Formular(Header$)
```

```
148 PRINT FN Screen$("e")
149 INPUT @(7,21);Name$(0) USING "a+ +-u",Ret%L,15
150 PRINT FN Screen$("f")
151 T%=1
152 REPEAT
153   WHILE Name$(T%)<>(Name$(0)+ SPACE$(15- LEN(Name$(0)))) AND
T%<Gro%L
154     T%=T%+1
155   WEND
156   IF Name$(T%)=Name$(0)+ SPACE$(15- LEN(Name$(0))) THEN
157     Anzeige(T%)
158   ELSE
159     FORM_ALERT (1,"[1][Name nicht vorhanden!][ Was soll's ]")
160   ENDIF
161   PRINT @(19,15);FN Screen$("p");"W";FN Screen$("q");
162   PRINT "eitersuchen    ";FN Screen$("p");"N";FN Screen$("q");
163   PRINT "eueingabe    ";FN Screen$("p");"Z";FN Screen$("q");
164   PRINT "urück ins Hauptmenü"
165   A$=""
166   REPEAT
167     A$= INKEY$
168     IF A$<>"" THEN
169       A$= UPPER$( RIGHT$(A$,1))
170     ENDIF
171   UNTIL A$="W" OR A$="N" OR A$="Z"
172   IF A$="W" AND T%<Gro%L THEN
173     T%=T%+1
174   ENDIF
175   IF A$="N" THEN
176     EXIT TO Suchen
177   ENDIF
178 UNTIL A$="Z"
179 RETURN
180 '
181-Anlegen
182 CLS
183 Pfad$="A:\":Name$="MINIDAT.REL":Ext$="*.REL"
184 Do_File(Name$,Pfad$,Ext$,Name$,Pfad$,Ext$,Taste%)
185 IF Taste% AND Name$<>"" THEN
186   CLS
187   ' aktuellen Pfad zum Directory machen
188   CHDIR (Pfad$)
189   ' und überprüfen, ob schon eine solche Datei vorhanden
190   Rename(Name$)
191   ' dann Datei mit dem angegeben Namen anlegen
192   OPEN "R",1,Name$,117
```

```
193     FIELD 1,117 AS Buffer$
194     LSET Buffer$= SPACE$(117)
195     FOR T%=1 TO Gro%L
196       PUT 1,T%
197     NEXT T%
198   CLOSE 1
199   FORM_ALERT (1,Mel$)
200 ENDIF
201 RETURN
202 '
203-Laden
204 CLS
205 IF Pfad$="" THEN
206 ' noch kein Pfad verhanden => Pfad zuweisen
207   Pfad$="A:\":Name$="MINIDAT.REL":Ext$="*.REL"
208 ENDIF
209 Do_File(Name$,Pfad$,Ext$,Name$,Pfad$,Ext$,Taste%)
210 IF Taste% AND Name$<>"" THEN
211   CLS
212   CHDIR (Pfad$)' neues Hauptdirectory
213   IF NOT FN Exist%L(Name$) THEN
214     FORM_ALERT (1,Fehler_2$)
215   ELSE
216     Reset_All' vorhandene Daten löschen
217     OPEN "r",1,Name$,117
218     FIELD 1,15 AS Na$,15 AS Vo$,32 AS St$,34 AS Ort$,21 AS Rest$
219     ' aus drucktechnischen Gründen mußten die letzten Daten
220     ' leider in je einer Buffervariable zusammengefasst werden
221     'Ort$ => PLZ$ (4 Stellen) und Ort$(30 Stellen)
222     'Rest$ => 11 Stellen Telefon$, 10 Stellen Geb$
223     T%=0
224     REPEAT
225       T%=T%+1
226       GET 1,T%
227       Name$(T%)=Na$
228       Vorname$(T%)=Vo$
229       Strasse$(T%)=St$
230       Plz$(T%)= LEFT$(Ort$,4)
231       Ort$(T%)= MID$(Ort$,5)
232       Tel$(T%)= LEFT$(Rest$,11)
233       Geb$(T%)= MID$(Rest$,12)
234     UNTIL Name$(T%)= SPACE$(15) OR T%=Gro%L
235     Loesche(T%)
236     CLOSE 1
237   ENDIF
238 ENDIF
```

```
239 RETURN
240 '
241 DEF PROC Formular(Text$)
242  LOCAL T%
243  CLS
244  PRINT @(2,(78- LEN(Text$))/2);Text$
245  PRINT @(5,10);"*"*60
246  FOR T%=1 TO 9: PRINT @(5+T%,10);"*";@(5+T%,69);"*": NEXT T%
247  PRINT @(15,10);"*"*60
248  PRINT @(7,15);"Name: _____      Vorname: _____"
249  PRINT @(9,15);"Strasse: _____"
250  PRINT @(11,15);"PLZ: ____   Ort: _____"
251  PRINT @(13,15);"Telefon: _____   Geb.: _____"
252 RETURN
253 '
254 DEF PROC Anzeige(Nummer%)
255  PRINT @(7,21);Name$(Nummer%);
256  PRINT STRING$(15- LEN(Name$(Nummer%)),"_")
257  PRINT @(7,50);Vorname$(Nummer%);
258  PRINT STRING$(15- LEN(Vorname$(Nummer%)),"_")
259  PRINT @(9,24);Strasse$(Nummer%);
260  PRINT STRING$(32- LEN(Strasse$(Nummer%)),"_")
261  PRINT @(11,20);Plz$(Nummer%);
262  PRINT STRING$(4- LEN(Plz$(Nummer%)),"_")
263  PRINT @(11,32);Ort$(Nummer%);
264  PRINT STRING$(30- LEN(Ort$(Nummer%)),"_")
265  PRINT @(13,24);Tel$(Nummer%);
266  PRINT STRING$(11- LEN(Tel$(Nummer%)),"_")
267  PRINT @(13,44);Geb$(Nummer%);
268  PRINT STRING$(10- LEN(Geb$(Nummer%)),"_")
269 RETURN
270 '
271 DEF PROC Eingabe(Nummer%)
272  LOCAL Back$="s"+ CHR$(72)+"s"+ CHR$(80)
273  -Nam
274  INPUT @(7,21);Name$(Nummer%) USING "a+ +-u"+Back$,Ret%L,15
275  IF (Ret%L AND $FF0000)=$480000 THEN
276     GOTO Geb
277  ENDIF
278  -Vorname
279  INPUT @(7,50);Vorname$(Nummer%) USING "a+ +-"+Back$,Ret%L,15
280  IF (Ret%L AND $FF0000)=$480000 THEN
281    GOTO Nam
282  ENDIF
283  -Street
284  INPUT @(9,24);Strasse$(Nummer%) USING "Oa+ +-+."+Back$,Ret%L,32
```

```
285  IF (Ret%L AND $FF0000)=$480000 THEN
286    GOTO Vorname
287  ENDIF
288  -Plz
289  INPUT @(11,20);Plz$(Nummer%) USING "0>"+Back$,Ret%L,4
290  IF (Ret%L AND $FF0000)=$480000 THEN
291    GOTO Street
292  ENDIF
293  -Ort
294  INPUT @(11,32);Ort$(Nummer%) USING "a0+/+-"+Back$,Ret%L,30
295  IF (Ret%L AND $FF0000)=$480000 THEN
296    GOTO Plz
297  ENDIF
298  -Telefon
299  INPUT @(13,24);Tel$(Nummer%) USING "0+-+/c-/"+Back$,Ret%L,11
300  IF (Ret%L AND $FF0000)=$480000 THEN
301    GOTO Ort
302  ENDIF
303  -Geb
304  INPUT @(13,44);Geb$(Nummer%) USING "0+."+Back$,Ret%L,10
305  IF (Ret%L AND $FF0000)=$480000 THEN
306    GOTO Telefon
307  ELSE
308    IF (Ret%L AND $FF0000)=$500000 THEN
309      GOTO Nam
310    ELSE
311    ENDIF
312   ENDIF
313 RETURN
314 '
315 DEF PROC Delete(Nummer%)
316 LOCAL Anzahl%=1,T%,Dummy%=Nummer%' zu löschenden Record merken
317 ' Anzahl der vorhandenen Datensätze ermitteln
318 WHILE (Name$(Anzahl%+1))<>"" AND Anzahl%<Gro%L
319   Anzahl%=Anzahl%+1
320 WEND
321 IF Nummer%=Anzahl% THEN
322    Loesche(Nummer%)
323    Speichern(Nummer%)
324 ELSE
325   WHILE (Nummer%<>Anzahl%)
326     Name$(Nummer%)=Name$(Nummer%+1)
327     Vorname$(Nummer%)=Vorname$(Nummer%+1)
328     Strasse$(Nummer%)=Strasse$(Nummer%+1)
329     Ort$(Nummer%)=Ort$(Nummer%+1)
330     Plz$(Nummer%)=Plz$(Nummer%+1)
```

```
331     Tel$(Nummer%)=Tel$(Nummer%+1)
332     Geb$(Nummer%)=Geb$(Nummer%+1)
333     Nummer%=Nummer%+1
334   WEND
335   Loesche(Nummer%)
336   FOR T%=Dummy% TO Nummer%
337     Speichern(T%)
338   NEXT T%
339 ENDIF
340 RETURN
341 '
342 DEF PROC Loesche(Nummer%)
343   Name$(Nummer%)="":Vorname$(Nummer%)="":Strasse$(Nummer%)=""
344   Plz$(Nummer%)="":Ort$(Nummer%)="":Tel$(Nummer%)=""
345   Geb$(Nummer%)=""
346 RETURN
347 '
348 DEF PROC Reset_All' löscht sämtliche Datensätze im RAM
349   LOCAL T%=1
350   WHILE Name$(T%)<>"" AND T%<Gro%L
351     Loesche(T%)
352     T%=T%+1
353   WEND
354 RETURN
355 '
356 DEF PROC Do_File(Name$,Path$,Post$,R Name$,R Path$,R Post$,R Tas%)
357    ' Lokale Variablen verwenden, damit die Prozedur auch
358    ' in anderen Programmen einsetzbar ist.
359    LOCAL R_Path$=Path$,R_Name$=Name$,T%,Drive%,Pointer%L,Ret%L
360    IF Path$="" THEN ' kein Pfad angegeben, dann einen basteln
361       Path$=" "*64'   GEMDOS Konvention Folge leisten
362       Pointer%L= LPEEK( VARPTR(Path$))+ LPEEK( SEGPTR +28)
363       ' anschließend aktuellen Pfadnamen holen
364       GEMDOS (,71, HIGH(Pointer%L), LOW(Pointer%L),0)
365       ' und zurechtstutzen
366       Path$= LEFT$(Path$, INSTR(Path$+ CHR$(0), CHR$(0))-1)
367       GEMDOS (Drive%,25)' aktuelles Laufwerk ermitteln
368       Path$= CHR$(65+Drive%)+":"+Path$+"\"
369    ENDIF
370    IF Post$="" THEN ' ohne Extension kann man schlecht arbeiten
371       Path$=Path$+"*.*"' Pauschalextension anhängen
372    ELSE
373       Path$=Path$+Post$
374    ENDIF
375    PRINT CHR$(27);"f"' Cursor aus- und Maus einschalten
376    MOUSEON
```

```
377     FILESELECT (Path$,Name$,Ret%L)
378     MOUSEOFF
379     PRINT CHR$(27);"e"' Cursor wieder einschalten
380     IF Ret%L=0 THEN '     Abbruch angeklickt
381        Path$=R_Path$'   alte Vorgaben zurückkopieren
382        Name$=R_Name$
383        Tas%=0
384     ELSE
385        ' ersten Backslash '\' suchen
386        T%= LEN(Path$)
387        WHILE T%>0 AND MID$(Path$,T%,1)<>"\"
388           T%=T%-1
389        WEND
390        Path$= LEFT$(Path$,T%)
391        Tas%=1
392     ENDIF
393 RETURN
394 '
395 DEF PROC Speichern(Nummer%)
396   ' schreibt Datensatz 'Nummer%' auf Diskette
397   OPEN "R",1,Name$,117
398     FIELD 1,15 AS Na$,15 AS Vo$,32 AS St$,34 AS Pl$,21 AS Rest$
399     ' auch hier wieder aus drucktechnischen Gründen die
400     ' Zusammenfassung mehrerer Daten in eine Buffervariable
401     LSET Na$=Name$(Nummer%)
402     LSET Vo$=Vorname$(Nummer%)
403     LSET St$=Strasse$(Nummer%)
404     LSET Pl$=Plz$(Nummer%)
405     ' leider nicht anders zu machen ...
406     MID$ (Pl$,5)=Ort$(Nummer%)
407     LSET Rest$=Tel$(Nummer%)
408     MID$ (Rest$,12)=Geb$(Nummer%)
409     ' und auf Diskette schreiben
410     PUT 1,Nummer%
411   CLOSE 1
412 RETURN
413 '
414 DEF PROC Taufe(R Taste%)
415   IF Name$="" THEN
416      Name$="MINIDAT.REL"
417      Pfad$="A:\":Ext$="*.REL"
418      Do_File(Name$,Pfad$,Ext$,Name$,Pfad$,Ext$,Taste%)
419      IF Taste% AND Name$<>"" THEN
420         CHDIR Pfad$
421      ENDIF
422   ELSE
```

```
423       Taste%=1
424    ENDIF
425 RETURN
426 '
427 DEF FN Exist%L(Name$)
428    LOCAL T%
429    '
430    OPEN "F",1,Name$,4
431    IF EOF(1) THEN
432    ' Keine Files auf der Diskette vorhanden
433    ' => Fehlermeldung (0) zurückgeben
434       CLOSE 1
435       RETURN (0)
436    ELSE
437       ' DTA-Buffer bereitstellen
438       FIELD 1,30 AS Buffer$,14 AS File$
439       GET 1,0
440       ' Datei vorhanden => '-1' zurückgeben
441       IF Name$= LEFT$(File$, INSTR(File$, CHR$(0))-1) THEN
442          CLOSE 1
443          RETURN (-1)
444       ENDIF
445    ENDIF
446 CLOSE 1
447 ' File war nicht auf der Diskette
448 RETURN (0)
449 DEF PROC Rename(Filename$)
450    IF FN Exist%L(Filename$) THEN
451       ' Datei existiert bereits => BACKUP auch schon vorhanden?
452       IF FN Exist%L( LEFT$(Filename$, INSTR(Filename$,"."))+"BAK")
453          ' dann BACKUP einfach löschen ...
454          THEN KILL LEFT$(Filename$, INSTR(Filename$,"."))+"BAK"
455       ENDIF
456       ' jetzt BACKUP anfertigen
457       BACKUP (Filename$)
458    ENDIF
459 RETURN
```

## 3.12 Warum das Rad noch einmal erfinden?

Programmieren ist mit Sicherheit eine schöne Sache, aber warum soll man sich den Kopf über Probleme zerbrechen, die andere schon lange gelöst haben? Oder warum soll man für jedes neue Programm wieder die gleichen Funktionen schreiben, die man

schon einmal entwickelt hat, und die sich prächtig bewährt haben? Ist es nicht viel sinnvoller, bestimmte Funktionen, die man für ein professionelles Programm nun einmal benötigt, in einer eigenen Datei unterzubringen, die einfach an das Programm angehängt werden kann? Auf diese Weise erspart man sich eine Menge Arbeit!

Solche Dateien, in denen alle möglichen Funktionen vorhanden sind, die bei Bedarf dann nur noch aufgerufen werden müssen, heißen Libraries.

Library bedeutet zunächst einmal nichts anderes, als Bibliothek. In Compilersprachen sind sie gang und gebe, was wäre z.B. C ohne eine Library? Nichts, denn dann gäbe es nicht einmal einen Befehl zur Datenein- oder -ausgabe! Und obwohl ST-BASIC durch eine Vielzahl von Befehlen glänzen kann, wird es durch die Verwendung von Befehlsbibliotheken noch einfacher, gute Programme zu entwickelen. Im Lieferumfang des Interpreters befindet sich beispielsweise eine GEM-Library, mit deren Hilfe GEM-Funktionen - wir werden darauf später noch einmal ausführlich zu sprechen kommen - aufgerufen werden können. Sie als Programmierer müssen sich keine Gedanken mehr machen, wie GEM-Funktionen zu programmieren sind, da sie - bedingt durch die Library - nur noch aufgerufen werden müssen!

Zum Abschluß dieses Kapitels über Dateiverwaltung möchte auch ich Ihnen eine äußerst nützliche Library servieren, mit deren Hilfe die File-Selector-Box verwaltet, eine Backup-Datei erstellt und überprüft werden kann, ob sich die angegebene Datei auf der Diskette befindet.

Wie arbeitet man nun mit einer solchen Library? Ganz einfach:

- Sie erstellen zunächst einmal ihr Programm, wobei Sie die Funktionen, die Ihnen die Library zur Verfügung stellt, bereits aufrufen können.

- Da sie vom Interpreter während der Programmabarbeitung jedoch benötigt werden, wird die gesamte Library einfach an das Programm gehängt. Jetzt sind die Funktionen definiert, und können ausgeführt werden.
- Das auf diese Weise erhaltene Programm wird - zusammen mit der Library - auf Diskette abgespeichert. Von nun an kann es ganz normal geladen und sofort ausgeführt werden.

Beim Zusammenbasteln der beiden Programmteile sind folgende Regeln zu beachten:

- Der Befehl, mit dessen Hilfe zwei Programme zusammengehängt werden können, lautet MERGE <Programmname>. Ein Programmteil (dies wird im Normalfall Ihr selbstgeschriebenes Programm sein, das noch die Funktionen der Library benötigt) muß sich bereits im Speicher des Computers befinden.
- Der Befehl MERGE <Programmname> sorgt nun dafür, daß das genannte Programm (Name der Library) an das schon bestehende Programm gehängt wird.
- Zusammengehängt ist eigentlich der falsche Ausdruck, da beide Teile miteinander verschmolzen werden. MERGE fügt nämlich das (übrigens im ASCII-Code, also mit SAVE,A abgespeicherte) Programm so in das Programm im Speicher ein, wie es die Zeilennummern erfordern. Ein Beispiel macht dies sofort anschaulicher:

Folgendes Programm befinde sich im Speicher:

```
10 <Zeile 1>
20 <Zeile 2>
44 <Zeile 3>
50 <Zeile 4>
60 END
```

Auf der Diskette befindet sich das Programm <Datei.BAS>:

```
1  <Datei.BAS, Zeile 1>
2  <Datei.BAS, Zeile 2>
11 <Datei.BAS, Zeile 3>
22 <Datei.BAS, Zeile 4>
51 <Datei.BAS, Zeile 5>
```

ergibt "zusammengemerged":

```
1  <Datei.BAS, Zeile 1>
2  <Datei.BAS, Zeile 2>
10 <Zeile 1>
11 <Datei.BAS, Zeile 3>
20 <Zeile 2>
22 <Datei.BAS, Zeile 4>
44 <Zeile 3>
50 <Zeile 4>
51 <Datei.BAS, Zeile 5>
60 END
```

Deshalb dürfen in beiden Programm keine gleichen Zeilennummern vorkommen! Die Library 'DAT_LIB.BAS' beginnt mit Zeile 1000. Soll Sie an Programme angehängt werden, die Zeilennummern von größer 1000 besitzen, muß zuerst ein RENUM durchführt werden, ehe die Library neu abgespeichert wird. Vergessen Sie dabei aber auf keinen Fall, daß das Programm auf Diskette im ASCII-Format vorliegen muß, damit der Befehl MERGE benutzt werden kann.Hier nun die versprochende Library, deren Funktionen Sie schon kennen:

```
1000 '*****************************************************************
1010 '*                          DAT_LIB.BAS                          *
1020 '*---------------------------------------------------------------*
1030 '* Autor: Michael Maier    Version: 1.00     Datum: 16.09.1988 *
1040 '*      Ein Programm aus dem 'GROSSEN ST-BASIC BUCH'             *
1050 '*        (C) 1988 by DATA BECKER GmbH Düsseldorf                *
1060 '*****************************************************************
1070 '
1080 '
1090 ' Die hier aufgeführten Prozeduren und Funktionen können in eigene
1100 ' Programme eingebaut werden, damit Sie das Rad nicht noch einmal
1110 ' erfinden müssen:
1120 '
```

```
1130 ' Prozedur DO_FILE(): verwaltet die FILE-SELECTORBOX
1140 ' ------------------------------------------------
1150 '
1160 ' benötigte Parameter:      Name$: Dateiname, der nach dem Aufruf
1170 ' ===================            der Funktion im Eingabefeld
1180 '                                 'AUSWAHL' stehen soll.
1190 '                           Path$: gewünschter Pfadname, wird kein
1200 '                                  Name angegeben, benutzt die
1210 '                                  Funktion den aktuellen Pfad.
1220 '                           Post$: Extension für den Pfad: ("*.*")
1230 '                           Die nächten drei Parameter geben die
1240 '                           ausgewählten Zeichenketten zurück
1250 '                           Tas%:  Nummer des Buttons, der ange-
1260 '                                  klickt wurde.
1270 '                                  '0' => 'ABBRUCH'
1280 '                                  '1' => 'OK'
1290 '
1300 '
1310 DEF PROC Do_File(Name$,Path$,Post$,R Name$,R Path$,R Post$,R Tas%)
1320    ' Lokale Variablen verwenden, damit die Prozedur auch
1330    ' in anderen Programmen einsetzbar ist.
1340    LOCAL R_Path$=Path$,R_Name$=Name$,T%,Drive%,Pointer%L,Ret%L
1350    IF Path$="" THEN ' kein Pfad angegeben, dann einen basteln
1360       Path$=" "*64'   GEMDOS Konvention Folge leisten
1370       Pointer%L= LPEEK( VARPTR(Path$))+ LPEEK( SEGPTR +28)
1380       ' anschließend aktuellen Pfadnamen holen
1390       GEMDOS (,71, HIGH(Pointer%L), LOW(Pointer%L),0)
1400       ' und zurechtstutzen
1410       Path$= LEFT$(Path$, INSTR(Path$+ CHR$(0), CHR$(0))-1)
1420       GEMDOS (Drive%,25)' aktuelles Laufwerk ermitteln
1430       Path$= CHR$(65+Drive%)+":"+Path$+"\"
1440     ENDIF
1450     IF Post$="" THEN ' ohne Extension kann man schlecht arbeiten
1460        Path$=Path$+"*.*"' Pauschalextension anhängen
1470     ELSE
1480        Path$=Path$+Post$
1490     ENDIF
1500     PRINT CHR$(27);"f"' Cursor aus- und Maus einschalten
1510     MOUSEON
1520     FILESELECT (Path$,Name$,Ret%L)
1530     MOUSEOFF
1540     PRINT CHR$(27);"e"' Cursor wieder einschalten
1550     IF Ret%L=0 THEN '     Abbruch angeklickt
1560        Path$=R_Path$'   alte Vorgaben zurückkopieren
1570        Name$=R_Name$
1580        Tas%=0
```

```
1590     ELSE
1600        ' ersten Backslash '\' suchen
1610        T%= LEN(Path$)
1620        WHILE T%>0 AND MID$(Path$,T%,1)<>"\"
1630           T%=T%-1
1640        WEND
1650        Path$= LEFT$(Path$,T%)
1660        Tas%=1
1670     ENDIF
1680 RETURN
1690 '
1700 ' ************************************************************
1710 '
1720 ' Funktion: FN EXIST(<Dateiname>)
1730 ' ------------------------------
1740 '
1750 ' Die folgende Funktion überprüft, ob ein File 'Name$' auf der
1760 ' Diskette vorhanden ist
1770 ' File vorhanden       => Returnwert ist '-1'   ('wahr')
1780 ' File nicht vorhanden => Returnwert ist '0'   ('falsch')1790 '
1800 ' Aufruf: [<Variable>] = FN EXIST(<Dateiname>)
1810 ' ============================================
1820 '
1830 DEF FN Exist%L(Name$)
1840    LOCAL T%
1850    '
1860    OPEN "F",1,Name$,55
1870    IF EOF(1) THEN
1880    ' Keine Files auf der Diskette vorhanden
1890    ' => Fehlermeldung (0) zurückgeben
1900       CLOSE 1
1910       RETURN (0)
1920    ELSE
1930       ' DTA-Buffer bereitstellen
1940       FIELD 1,30 AS Buffer$,14 AS File$
1950       GET 1,0
1960       ' Datei vorhanden => '-1' zurückgeben
1970       IF Name$= LEFT$(File$, INSTR(File$, CHR$(0))-1) THEN
1980          CLOSE 1
1990          RETURN (-1)
2000       ENDIF
2010    ENDIF
2020 CLOSE 1
2030 ' File war nicht auf der Diskette
2040 RETURN (0)
2050 '
```

```
2060 ' ************************************************************
2070 '
2080 ' Prozedur: RENAME(<Dateiname>)
2090 ' ----------------------------
2100 '
2110 ' Die Prozedur RENAME überprüft, ob ein File mit dem angegebenen
2120 ' Namen auf der Diskette vorhanden ist.
2130 ' Dann wird davon eine BACKUP-Datei gezogen, eine schon vorhandene
2140 ' BACKUP-Datei wird von der Diskette gelöscht.
2150 '
2160 ' ACHTUNG: Diese Prozedur benötigt die Funktion EXIST(), aus
2170 ' ======== dieser Library!!
2180 '
2190 ' Aufruf der Prozedur:  Rename(<Filename>)
2200 ' ------------------------------------------------------------
2210 '
2220 DEF PROC Rename(Filename$)
2230    IF FN Exist%L(Filename$) THEN
2240       ' Datei existiert bereits => BACKUP auch schon vorhanden?
2250       IF FN Exist%L( LEFT$(Filename$, INSTR(Filename$,"."))+"BAK")
2260          ' dann BACKUP einfach löschen ...
2270          THEN KILL LEFT$(Filename$, INSTR(Filename$,"."))+"BAK"
2280       ENDIF
2290       ' jetzt BACKUP anfertigen
2300       BACKUP (Filename$)
2310    ENDIF
2320 RETURN
```

## 3.13 Schwarz auf Weiß - Druckerausgabe

Bis jetzt wurden sämliche Ausgaben entweder auf den Monitor geleitet oder über einen mit OPEN geöffneten Kanal auf Diskette geschrieben. Oft ist es jedoch wünschenswert, die Ausgaben schwarz auf weiß auf einem Blatt Papier vorliegen zu haben, sie also auf einem Drucker auszugeben. Wie muß man dies nun anstellen?

Wenn Sie noch einmal ein paar Seiten zurück, an den Anfang dieses Kapitels blättern, so werden Sie feststellen, daß mit

```
OPEN "P",<Kanalnummer>
```

ein Kanal geöffnet werden kann, der die Daten, die auf diesen Kanal geleitet werden, an den Drucker schickt:

```
OPEN "P",4
   PRINT #4,"Diese Zeile erscheint auf dem Drucker!"
CLOSE 4
```

Mehr wäre zu diesem Thema fast nicht zu sagen! Oder doch? Es gibt nämlich noch ein paar weitere Befehle, die eine Ausgabe auf dem Drucker ermöglichen:

LPRINT

Entspricht dem Befehl PRINT, sendet die Ausgaben jedoch sofort an den Drucker:

```
LPRINT "Diese Zeile erscheint auf dem Drucker!"
```

Um diesen Befehl benutzen zu können, muß zuvor kein OPEN erfolgen, da die Ausgabe stets auf dem Druckerport (Anschluß für den Drucker) landet. Noch einen Vorteil bringt dieser Befehl mit sich:

Epson-Drucker und Epson-kompatible weigern sich normalerweise standhaft die Umlaute des Atrai ST korrekt auf das Papier zu bringen, sie werden vom Computer einfach verschluckt! Dafür erscheinen sie immer dann, wenn eigentlich eine geschweifte Klammer ausgedruckt werden soll. Ein Textverarbeitungsprogramm löst dieses Problem, indem die ASCII-Werte der problematischen Zeichen vor der Ausgabe an den Drucker bei Bedarf entsprechend umgewandelt werden. Nur Sie arbeiten in BASIC und nicht mit einem Textverarbeitungsprogramm! Auch die Idee, sämtiche Ausgaben statt auf den Druckerkanal in eine Diskettendatei zu lenken, um sie dann mit einem Textverarbeitungsprogramm einzulesen und auszudrucken, wird Sie mit Sicherheit nicht sehr begeistern können. Also was tun? Die Lösung findet sich im Handbuch zum ST-BASIC: Man nehme den Befehl MODE LPRINT, und sorge dafür, daß der Modus auf DEUTSCH gestellt wird. Dies geschieht mit

```
MODE LPRINT "D"
```

und bewirkt, daß Umlaute für Epson-Drucker vor der Ausgabe erst einmal in den richtigen ASCII-Code gewandelt werden. Wohlgemerkt, dieser Befehl hat nur auf LPRINT Einfluß, eine Ausgabe, die mittels PRINT# an den Adreßaten gelangt, wird nicht umgesetzt. Sollte Ihr Drucker dagegen ab sofort, d.h. nach Verwendung dieses Befehls, den Ausdruck verstümmeln, empfiehlt es sich, einen anderen Modus für LPRINT zu verwenden. Zur Auswahl stehen ja noch GB und USA.

```
CMD <Kanalnummer>
```

sorgt dafür, das alle Daten, die ausgegeben werden (z.B. mit PRINT oder LPRINT) in die Datei <Kanalnummer> umgeleitet werden. Möchten Sie beispielsweise dem Anwender die Wahl einräumen, ob er die da kommenden Ausgaben auf dem Monitor (spart Papier und Farbband) oder lieber auf dem Drucker wünscht, müßten eigentlich zwei getrennte Routinen zur Ausgabe entwickelt werden: Eine für die Ausgabe auf dem Bildschirm, in der PRINT-Befehle verwendet werden, und eine für die Druckerausgabe, die sich des Befehls LPRINT bedient.

```
PRINT "Hallo"
```

läßt Hallo auf dem Monitor erscheinen, sobald die Ausgabe jedoch umgeleitet wird, erscheint sie auf dem Papier:

```
OPEN "P",1'       Kanal zum Drucker öffnen
  PRINT "Hallo"'  Es lebe der Monitor
  CMD 1'          Ab sofort alles auf den Drucker
  PRINT "Hallo"'  erscheint auf dem Drucker
CLOSE 1
```

Mit CMD kann ein Listing auch in eine Diskettendatei geschrieben werden, um sie später in ein Textverarbeitungsprogramm einzubinden. Dazu lenkt man LLIST (arbeitet wie LIST, gibt seine Daten jedoch an den Drucker weiter!), in eine zum Schreiben geöffnete Datei:

```
OPEN "O",4,"DATEI.BAS"
  CMD 4
  LLIST '100-200 wenn nur ein Teillisting gewünscht
CLOSE 4
```

Für die Ausgabe sollten Sie deshalb LLIST und nicht das gewöhnliche LIST bemühen, da LLIST keine Steuerzeichen für den Monitor in die Ausgabedatei schreibt, die das Textverarbeitungsprogramm, mit dem der ganze Sermon weiterverarbeitet werden soll, verärgern könnten. Gleiches gilt auch für die anderen Befehle, deren vorangestelltes L dafür Sorge trägt, daß die Ausgabe auf den Drucker gelenkt wird (DUMP - LDUMP)

PRINT @(Zeile, Spalte) ermöglicht die genaue Positionierung eines Textes auf dem Bildschirm. Der Klammeraffe bewirkt das Erzeugen eines Steuerzeichens für den Monitor, der dem Cursor Beine macht und ihn an die entsprechende Stelle springen läßt. LPRINT @() funktioniert dagegen nicht, da der Drucker die Steuerzeichen des Monitors nicht kennt. Aber auch er verwendet Steuerzeichen, die für die Gestaltung des Ausdrucks von enormer Wichtigkeit sind.

Alle Steuerzeichen für den Drucker beginnen, sofern es sich um einen Epson- oder kompatiblen Drucker handelt, mit einer Escape-Sequenz. Dies ist für den Drucker gleichsam das Achtung, daß jetzt interessante Daten zu erwarten sind, die z.B. eine Umstellung der Schriftart auslösen.

Zum Einschalten der Eliteschrift (mein NEC behält seine Schrift bei, ändert dafür jedoch die Zeichenbreite auf 12 cpi (Zeichen pro Zoll) ab, während ein FX-80/85 direkt auf Elite (ebenfalls mit einer Breite von 12 cpi) umstellt), lautet der Steuercode

```
CHR$(27);CHR$(77);
```

bzw.

```
CHR$(27);"M";
```

der an den Drucker weiterzuleiten (LPRINT, PRINT#, ...) ist.

## 3.14 Mehrzeilige Suchvorgaben mit OR mit AND

Mit dieser Version unserer Adreßverwaltung ist es nur möglich eine Adreßdatei nach dem Familiennamen zu durchsuchen. Sind in einer Datei aber mehrere Datensätze enthalten, die den gleichen Namen tragen (z.M. MAIER, weil er so selten ist), müssen Sie Eintrag für Eintrag weiterblättern, bis der richtige Datensatz endlich gefunden ist. Professioneller wäre es auf jeden Fall, auch den Vornamen bei der Suche mitangeben zu können, wobei die Datei dann auch nach dem Vornamen durchsucht wird. Stimmen Familien- und Vorname eines Datensatzes überein, scheint der richtige Eintrag gefunden worden zu sein und kann auf dem Monitor ausgegeben werden.

```
REPEAT
   .....
   .....
UNTIL Name$(0) = Name$(T%) AND Vorname$(0) = Vorname$(T%)
```

Auch die andere Alternative der Datensuche kann bisweilen ganz nützlich sein: Sie möchten alle Einträge herausfischen, die mit Familiennamen Huber oder mit Vornamen Peter heißen. Dann muß OR heran:

```
REPEAT
  .....
  .....
UNTIL Name$(0) = Name$(T%) OR Vorname$(0) = Vorname$(T%)
```

# 4. Das Betriebssystem des Atari ST

Obwohl ST-BASIC an sich eine Unmenge von Befehlen zur Verfügung stellt, kommt man als Programmierer nicht um das Betriebssystem des Atari ST herum. Möchte man nämlich innerhalb eines Programms eine Diskette formatieren oder das Inhaltsverzeichnis einer solchen einlesen, läßt sich dies nur über das Betriebssystem (Operating System) bewerkstelligen.

Ferner existieren in ST-BASIC bestimmte Variablen, deren Werte nur abgefragt werden können (eine Zuweisung ist nicht möglich!) und die Auskunft über die Position des Mauszeigers, das Datum, die Uhrzeit, die Cursorzeile, usw. geben. Auch von ihnen werden Sie in diesem Kapitel hören.

## 4.1 Systemvariablen

Systemvariablen stellen in ST-BASIC Variablen dar, deren Namen reserviert sind, und denen - im Gegensatz zu anderen Variablen kein Wert zugewiesen werden kann. Häufig kann man in diesem Zusammenhang auch von Funktionen lesen, da Systemvariablen einen Wert liefern (auch Funktionen machen dies). Während man Funktionen jedoch ein Argument - das in Klammern hinter dem Funktionsnamen stehende Funktionsargument - übergibt, erhalten Systemvariablen kein Argument, auf Grund dessen sie ihre Berechnungen durchführen können.

### MOUSEX

Diese Variable enthält die X-Koordinate der momentanen Mausposition. Eine Bewegung der Maus in horizontale (X-) Richtung bewirkt eine automatische Änderung des Variableninhalts.

## MOUSEY

In MOUSEY legt der Interpreter die Position des mauszeigers in vertikaler Richtung (Y-Koordinate) ab.

## MOUSEBUT

Neben den Koordinaten des Mauszeigers besitzt die Maus auf ihrer Oberseite noch zwei Tasten, die Maustasten oder Mousebuttons. Die Systemvariable MOUSEBUT gibt Aufschluß über den Zustand dieser beiden Tasten. Ihr Inhalt im einzelnen:

| Inhalt von MOUSEBUT | Bedeutung |
|---|---|
| 0 | kein Kopf gedrückt |
| 1 | linker Knopf gedrückt |
| 2 | rechter Knopf gedrückt |
| 3 | beide Knöpfe (gleichzeitig) gedrückt. |

Wie lassen sich diese Informationen nun nutzen? Stellen Sie sich einmal vor, Sie haben ein Bild mit einem Zeichenprogramm erstellt, das als Maske für ein Programm dienen soll. In dieser Maske sind - in Anlehnung an GEM - Rechtecke mit Symbolen vorhanden, die angeklickt werden müssen, damit die gewünschte Reaktion seitens des Programms ausgelöst wird. Um nun feststellen zu können, welches Rechteck angeklickt wurde, müssen die Systemvariablen herangezogen werden. In einer Schleife kann zunächst auf einen Knopfdruck der linken Maustaste gewartet werden:

```
REPEAT
 ......
 ......
 ......
UNTIL MOUSEBUT=1
```

Die leere Schleife würde an sich schon genügen, da die Koordinatenposition der Maus problemlos auch noch später mit MOUSEX und MOUSEY abgefragt werden kann. Doch was geschieht, wenn der Benutzer nur schnell eine Box anklickt und anschließend die Maus auf eine ganz andere Position setzt? Die Systemvariablen liefern falsche Koordinaten. Deshalb sollte man

die Koordinaten gleich innerhalb der Schleife einer anderen Variablen anvertrauen. Zudem ist ein X bzw. Y schneller in einer Abfrage hingeschrieben, als der Name der Systemvariablen.

```
REPEAT
  Y%=MOUSEY
  X%=MOUSEX
UNTIL MOUSEBUT=1
```

und anschließend die Abfrage auf ein bestimmtes Rechteck, hier mit den X-Koordinaten 100,150 und den Y-Koordinaten 30, 90, d.h. die Eckpunkte besitzen die Koordinaten:

| | |
|---|---|
| linke obere Ecke: | (100,30) |
| rechte obere Ecke: | (150,30) |
| linke untere Ecke: | (100,90) |
| rechte untere Ecke: | (150,90) |

```
IF X% >= 100 AND X% <= 150 THEN
   IF Y% >= 30 AND Y% <= 90 THEN
      GOSUB Unterroutine
   ENDIF
ENDIF
```

In der IF-Abfrage wird also geprüft, ob sich die Mauskoordinaten innerhalb des betreffenden Rechtecks befinden. Innerhalb bedeutet, daß die X-Position der Maus zwar größer gleich der linken Begrenzungslinie des Rechtecks ist, zugleich jedoch kleiner gleich der Begrenzungslinie der rechten Rechteckseite sein muß (deshalb wieder das logische AND). Das soeben Gesagte muß natürlich auch noch auf die Y-Position der Maus zutreffen, damit eindeutig feststeht, ob sich die Maus auch wirklich in diesem Rechteck befand.

### TIMER

Timer enthält die Zeit seit dem Einschalten des Computers, und zwar in 1/200 Sekunden. Möchten Sie z.B. die Laufzeit für eine bestimmte Berechnung stoppen, so gelingt dies mit dieser Variablen:

```
Zeit=TIMER' aktuellen Stand merken
FOR T%=1 TO 30000
 PRINT T%
NEXT T%
PRINT (TIMER-Zeit)/200' Zeit in Sekunden
```

### TIME$

Systemvariablen kann man keine Werte zuweisen! Doch keine Regel ohne eine Ausnahme, wie Sie gleich sehen werden: TIME$ liefert die aktuelle Uhrzeit im Format "HH:MM:SS", also Stunden (HH) im 24-Stunden Format, Minuten und Sekunden, wobei die einzelnen Komponenten durch einen Doppelpunkt voneinander getrennt werden.

```
PRINT TIME$
```

Ergibt auf dem Bildschirm die aktuelle Uhrzeit. Wer jedoch nicht im Besitze eines neuen MEGA ST ist, der mit einer Batteriegepufferten Echtzeituhr aufzuwarten vermag, muß nach jedem Einschalten des Rechners die Uhrzeit neu stellen. Die Uhrzeit wird nämlich beim Abspeichern einer Datei auf Diskette oder Harddisk im Inhaltsverzeichnis festgehalten, und wenn die Uhrzeit nicht richtig eingestellt ist, steht darin dann das falsche Datum mit der falschen Zeit.

Aus diesem Grund kann der Systemvariablen TIME$ (ebenso wie ihrer Schwester DATE$) die aktuelle Zeit zugewiesen werden. Möchten Sie die Zeit beispielsweise auf 10.30 Uhr stellen, geben Sie einfach die Zeichenkette

```
TIME$="10:30:00"
```

ein, und die Uhrzeit ist richtig eingestellt. Der Inhalt von TIMER wird durch eine solche Zuweisung nicht berührt.

### DATE$

Während TIME$ die Uhrzeit ergibt, liefert DATE$ das jeweilige Datum. Das Format wird von dem jeweils eingestellten Modus beeinflußt: Modus "D" ergibt das Format TT.MM.JJ, während MODUS "USA" das Format TT/MM/JJ erzwingt. T steht dabei für Tag, M für Monat, und J für Jahr. Auch DATE$ kann das aktuelle Datum einfach zugewiesen werden.

### PI

Die Konstante PI ( ) steht für die in der Mathematik gebräuchliche Kreiszahl 3.14159265..... Dabei wird die Konstante PI als Fließkommaazahl mit doppelter Genauigkeit behandelt. Wünschen Sie lediglich einfache Genauigkeit, so kann die Konstante entweder einer Single-Float

```
PI!=PI
```

zugewiesen und mit PI! weitergerechnet werden, oder es kann ein PI mit einfacher Genauigkeit definiert werden:

```
CSNG(PI)
```

### CSRLIN

Diese Variable enthält die Zeile, in der sich der Cursor momentan befindet. Die oberste Zeile ist die Nummer 1, die unterste Zeile die Nummer 25.

### ERR,ERL,ERR$

kennen Sie bereits! Sie werden zum Errorhandling (Fehlerbehandlung) in ST-BASIC herangezogen, und liefern die Fehlernummer (ERR), die Zeile, in der der Fehler aufgetreten ist (ERL), und schließlich die Fehlermeldung im Klartext: ERR$.

## 4.2 TOS, GEMDOS, BIOS und XBIOS

Obwohl die Überschrift einen Ausschnitt aus einem chinesischen Telefonbuch vermuten lassen könnte, handelt es sich nicht um einen solchen. Vielmehr sind mit diesen zugegebenermaßen auf den ersten Blick gar wunderlich anmutenden Namen oder besser gesagt Abkürzungen die Bezeichnungen für das Betriebssystem des Atari ST gemeint. Aber was ist ein Betriebssystem? Für einen Einsteiger, der seinen Computer das erste Mal in Betrieb nimmt, scheint es eine Selbstverständlichkeit zu sein, daß auf Tastendruck bestimmte Reaktionen ausgelöst werden und mit der Maus quer über den Bildschirm gefahren werden kann, daß sich - wie im Falle des Atari ST - Fenster öffnen, verschieben, vergrößern oder wieder verkleinern lassen.

Der Prozessor, also das Herz des Computers, kann zwar seine Bits und Bytes durch den Arbeitsspeicher jonglieren, da er ihn auf mannigfache Weise zu adressieren versteht, er beherrscht auch die Manipulation einzelner Bits oder gar arithmetische Operationen. Bei allen Ein- und Ausgaben muß er sich jedoch auf die Hilfe anderer Peripheriebausteine stützen, wie auf einen Floppycontroller für den Datenverkehr mit dem Medium Floppy-Disk sowie andere Bausteine, die für ihn die Daten besorgen und wieder auf die Reise schicken.

Die Formalitäten, die dabei abzuwickeln sind, müssen dem Computer erst beigebracht werden. Er muß wissen, wie er eine bestimmte Schnittstelle ansprechen kann, und wie er wiederum eine bestimmte Reaktion seitens der Peripheriebausteine zu beantworten hat. Diese Aufgabe übernimmt für den Computer das Betriebssystem, das nach dem Chef von Atari - Jack Tramiel - TOS (Tramiel Operating System) getauft wurde.

Dieses Betriebssystem befindet sich - zumindest in neueren Computern der ST-Reihe - in ROMs, dabei handelt es sich um Datenspeicher, die im Computer Platz finden und ihren Inhalt nie vergessen. Selbst dann nicht, wenn der Strom abgestellt wird. TOS selbst setzt sich aus einem CP/M-68K-Derivat (was für ein Name!), und einem GEM (Graphics Environment Manager)-Teil zusammen. Das Derivat wickelt dabei die Kommunikation mit

der Außenwelt ab - und selbst die Tastatur zählt dabei zur Peripherie. Im Gegensatz zu seinem Vorbild, dem CP/M-68K, wurde es jedoch um einige Funktionen erweitert, da die Verwaltung der MIDI-Schnittstelle sowie die Steuerung der Maus ebenfalls vom Computer gewährleistet sein müssen.

Wie andere Betriebssysteme auch - man denke dabei an MS-DOS, das Betriebssystem der IBM-kompatiblen Computerwelt - unterteilt sich das Betriebssystem in einen allgemeinen und eine gerätespezifischen Teil. Der allgemeine Teil im ST trägt den Namen GEMDOS (GEM/Disk-Operating-System; dieser Name ist jedoch irreführend, da er mit GEM absolut nichts am Hut hat), während BIOS (Basic Input-Output System) mit den gerätespezifischen Aufgaben betraut ist. Die zusätzlichen Funktionen des Atari ST sind in einem dritten Modul des TOS untergebracht, dem XBIOS (eXtended BASIC Input-Output System).

Der GEM-Teil des Betriebssystems besteht aus dem GEM-Dektop (neudeutsch: Benutzeroberfläche) mitsamt den dazugehörenden Hilfsprogrammen, den Accessories, sowie dem GEM-VDI (Virtual Device Interface) und dem GEM-AES (Application Environment System). Dem VDI obliegen sämtlichen elementaren Grafikfunktionen, wie das Zeichnen von Linien oder Kreisen, das Einstellen der Strichstärke und des Füllmusters usw.

AES ist dagegen für die grafische Kommunikation zwischen Mensch und Maschine zuständig. Es sind AES-Routinen, die das Dektop (Benutzeroberfläche) aufbauen, das nach dem Einschalten des Atari sichtbar wird. GEM, die aufgesetzte Benutzeroberfläche des TOS, setzt die Befehle des Benutzers in eine für TOS verständliche Sprache um. Innerhalb von TOS wird das GEMDOS, das den Vorsitz im Triumvirat der drei Teilmodule hat, mit der Ausführung des entsprechenden Befehls betraut. Und GEMDOS wiederum beschäftigt seine beiden Knechte BIOS und XBIOS. Soviel zur Theorie und dem Aufbau des Betriebssystems (vgl. Abbildung 4.1). In diesem Kapitel wird es vor allem das TOS sein, das uns beschäftigt. GEM kommt zu einem späteren Zeitpunkt an die Reihe.

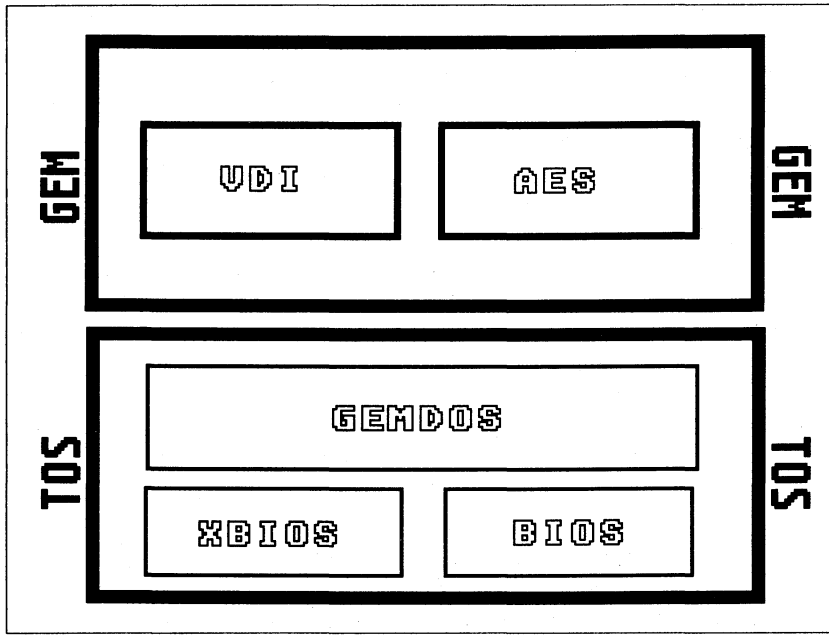



*Abb.: 4.1 Das Betriebssystem des Atari ST*

## 4.3 GEMDOS

Um eine GEMDOS-Funktion aufzurufen, wird der Befehl

```
GEMDOS([Rückgabewert],<Funktionsnummer>,[Parameter])
```

benutzt. Da GEMDOS eine ganze Palette von Routinen anbietet, wird jede Routine durch eine Funktionsnummer, die sich von 0 bis 87 erstreckt, gekennzeichnet. Auch ein oder mehrere Parameter können bei Bedarf (durch Komma getrennt) angegeben werden. Die einzelnen Funktionen, die unter ST-BASIC eingesetzt werden können (R bezeichnet den Rückgabewert):

*GEMDOS(R,2,Zeichen)* *Cconout*
Diese Routine gibt ein Zeichen auf dem Monitor aus. Auch hierfür ist der PRINT-Befehl besser geeignet.

*GEMDOS(R,3)* *Cauxin*
Liest ein Zeichen von der seriellen Schnittstelle (RS232) ein.

*GEMDOS(R,4,Zeichen)* *Cauxout*
Gibt ein Zeichen auf der seriellen Schnittstelle aus.

*GEMDOS(R,5,Zeichen)* *Cprnout*
Gibt ein Zeichen auf dem Drucker aus.

*GEMDOS(R,10,HIGH(Buffer),LOW(Buffer))* *Cconrs*
Gestattet, eine Zeile in den Puffer zu lesen. Puffer ist dabei ein Zeiger auf den eigentlichen Puffer für die Eingabe. Die ersten beiden Bytes dieses Puffer enthalten die maximale Eingabelänge, sowie die tatsächlich eingegebe Zeichenzahl. Die Tastenkombination <Contro>l<C> bewirkt einen Programmabbruch während der Funktionsausführung. Ein Beispiel für den Einsatz:

```
Eingabe(10, Inhalt$)
....
END
DEF PROC Eingabe (Laenge%,R Text$)
   Local Adr
   ' maximale Eingabelänge, sowie Rest der Zeichenkette
Text$=CHR$(Laenge%)+CHR$(0)*(Laenge%+2)
       ' berechnet die Adresse der Zeichenkette        Adr =
Lpeek( VARPTR(Text$))+ Lpeek(SEGPTR+28)
       GEMDOS(,10,HIGH(Adr),LOW(Adr))
       Text$=MID$(Text$,3, ASC(MID$(Text$,2)))
    RETURN
```

Die kleine Procedure liest einen String mit der maximalen Eingabelänge von 10 Zeichen ein und gibt ihn an das aufrufende Hauptprogramm zurück. Vor dem ersten Zeichen stehen bekanntlich die maximale Eingabelänge sowie als zweites Byte die tatsächlich eingegebene Anzahl von Zeichen. Die maximale Eingabelänge wird vor dem Aufruf der Routine als erstes Byte im String abgelegt, die restlichen Zeichen (2 + maximale Eingabelänge) werden einfach mit einem Füllbyte vorbelegt, damit nicht

anschließend wichtige Speicherbereiche versehentlich überschrieben werden. Da die Routine einen Zeiger auf den String erwartet, wird dieser berechnet und in Adr festgehalten. Nach der Eingabe wird ab der dritten Position die im 2. Byte angegebene Buchstabenzahl aus dem String herausgeschnitten und zurückgegeben.

*GEMDOS(R,14,Laufwerk)* *Dsetdrv*
Setzt das aktuelle Laufwerk, wobei Laufwerk folgende Werte annehmen kann:

0 Laufwerk A
1 Laufwerk B
2 Laufwerk C

In R ist die Nummer des Laufwerks enthalten, das vor dem Aufruf der Funktion aktiv war. Entspricht dem Befehl:

```
CHDIR CHR$(65+Laufwerk)+":"
```

*GEMDOS(R,16)* *Cconos*
Ermittelt, ob der Bildschirm Daten ausgeben kann. Ist dies der-Fall (immer!!) so steht in R der Wert $FFFF, ansonsten wird der Wert 0 zurückgegeben (nie!).

*GEMDOS(R,17)* *Cprnos*
Ermittelt, ob der Drucker bereit ist, ansonsten wird als Ergebnis der Wert 0 geliefert.

*GEMDOS(R,18)* *Cauxis*
Ergibt den Wert 0, wenn kein Zeichen an der seriellen Schnittstelle anliegt.

*GEMDOS(R,19)* *Cauxos*
Ergibt den Wert 0, wenn keine Daten über die serielle Schnittstelle geschickt werden können.

*GEMDOS(R,25)* *Dgetdrv*
Ermittelt die Nummer (vgl. Dsetdrv) des aktuellen Laufwerks.

*GEMDOS(,26,HIGH(Buffer),LOW(Buffer))* *Fsetdta*
Setzt die Adresse des DTA-Buffers (Disk-Transfer-Adress). Dieser 44-Byte große Puffer dient als Zwischenspeicher für Diskettenoperationen (z.B. Directory einlesen). Sein Aufbau im einzelnen:

| Byte | Inhalt |
|---|---|
| 1-21 | für GEMDOS reserviert |
| 22 | Datei-Attribut |
| 23-24 | Uhrzeit der Erstellung |
| 25-26 | Datum der Erstellung |
| 27-30 | Größe der Datei in Bytes |
| 31-44 | Name der Datei |

Auch bei OPEN "F",1,"*.*" erhält man mit GET 1,Datensatz die oben angegebenen 44 Bytes, dann muß jedoch ein entsprechender Puffer mit FIELD 1,... angelegt werden, aus dem die Daten abgeholt werden können. Diese Methode ist für ST-BASIC wesentlich praktischer.

*GEMDOS(R,47)* *Fgetdta*
Diese Funktion ermittelt die Adresse des DTA-Puffers.

*GEMDOS(R,48)* *Sversion*
Ermittelt die Versionsnummer des Betriebssystems.

*GEMDOS(R,54,HIGH(Buffer),LOW(Buffer),Drv)* *Dfree*
Entspricht im wesentlichen dem Befehl FRE(CHR$(65+Drv) +":") (freier Speicherplatz auf der Diskette in Laufwerk Drv ermitteln) und liefert als Ergebnis des Funktionsaufrufs einen Zeiger auf einen aus vier Langworten bestehenden Puffer:

- Anzahl der freien Cluster (b_free)
- Gesamtclusterzahl der Diskette (b_total)
- Anzahl Bytes eines Sektors (b-secsize)
- Zahl der Sektoren je Cluster (2) (b_clsize)

Damit kann dann sowohl der freie, als auch der belegte Speicherplatz auf der Diskette ermittelt werden:

| | |
|---|---|
| Freier Speicherplatz: | b_free * b_secsize * b_clsize |
| Diskettenkapazität: | b_total * b_secsize * b_clsize |
| Belegter Platz: | Diskettenkapazität - freier Platz |

Drv enthält die Nummer des Laufwerks, von dem der freie Speicherplatz ermittelt werden soll:

0 aktuelles Laufwerk (Defaultlaufwerk) benutzen
1 Laufwerk A
2 Laufwerk B
.
.

*GEMDOS(R,57,HIGH(Adresse),LOW(Adresse))* *Dcreate*
Legt ein Subdirectory (Ordner) auf der Diskette an. Der Name des Ordners darf aus maximal 8 Zeichen zuzüglich 3 Zeichen für die Extension (durch Punkt vom Namen abgetrennt) bestehen. Der Pfadname eines bereits existierenden Ordners kann ebenfalls angegeben werden. Dann erzeugt die Funktion Dcreate einen Ordner innerhalb des anderen. Adresse verweist auf einen String, der nach C-Manier mit einem Nullbyte abgeschlossen sein muß, und den Ordnernamen enthält. Die Adresse kann wieder mit

```
Adresse = LPEEK( VARPTR(STRING$)) + LPEEK( SEGPTR+28)
```

ermittelt werden. Entspricht dem Befehl MKDIR in ST-BASIC

*GEMDOS(R,58,HIGH(Adresse),LOW(Adresse))* *Ddelete*
Entfernt das Subdirectory von der Diskette, dessen Name in dem String, auf den der Zeiger Adresse verweist, durch ein Nullbyte abgeschlossen steht. Sind in dem zu löschenden Ordner noch Dateien vorhanden, wird eine Fehlermeldung zurückgegeben. In diesem Fall sind alle in diesem Subdirectory existierenden Dateien vor dem Löschen des Ordners zu entfernen. Entspricht RMDIM

*GEMDOS(R,59,HIGH(Adresse),LOW(Adresse))* *Dsetpath*
Diese Funktion ernennt einen Ordner (Folder) zum aktuellen Directory. Der neue Pfadname steht ab Adresse in einem String, und wird durch ein Nullbyte abgeschlosssen. Entspricht dem Befehl CHDIR.

*GEMDOS(R,60,HIGH(Adresse),LOW(Adresse),Atr)* *Fcreate*
Legt eine neue Datei an, wobei der Name ab Adresse in einem String stehen muß (Nullbyte!!). Der Rückgabewert ist ein sogenanntes File-Handle, das bei folgenden Schreib- und Lesebefehlen anzugeben ist. Konnte die neue Datei erzeugt werden, so ist die Handle-Nummer >= 6, ansonsten wird eine negative Zahl zurückgegeben. Der Fehlercode -36 kann darauf hindeuten, daß keine weiteren Dateien angelegt werden können, da das Inhaltsverzeichnis bereits voll ist. Dieser Befehl entspricht dem OPEN "O",... in St-BASIC. Als Attribut Atr können folgende Werte angegeben werden:

0 Datei kann gelesen und beschrieben werden.
1 Datei kann nach dem Schließen nur noch gelesen werden.
2 Es wird eine versteckte Datei (Hidden-File) erzeugt, die im Directory nicht erscheint.
4 System-Datei erzeugen.
8 Diskettenname schreiben.

Eine bereits auf der Diskette vorhandene Datei erhält die Länge 0, d.h. ihr Inhalt wird gelöscht. Zum Anlegen eines Ordners (Attribut 16) muß die Funktion Dcreate verwendet werden.

*GEMDOS(R,61,HIGH(Adresse),LOW(Adresse),Mod)* *Fopen*
Öffnet eine Datei. Ein Modus Mod von 0 entspricht einem Open "I", Mod = 1 entspricht OPEN "A", (Schreiben) und MOD = 2 bewirkt ein Öffnen zum gleichzeitigen Lesen und Schreiben der Datei. Ansonsten vgl. die Parameter bei Fcreate.

*GEMDOS(R,62,Handle)* *Fclose*
Schließt die Datei mit der Nummer Handle, dem Rückgabeparameter bei Erzeugen (Fcreate) oder Öffnen (Fopen) einer Datei.

*GEMDOS(R,63,Handle,HIGH(Anzahl),LOW(Anzahl),HIGH (Adresse),LOW(Adresse))* *Fread*
Liest Anzahl Bytes aus der Datei mit der Nummer Handle und legt diese ab Adresse in einem Puffer (String) ab. R enthält anschließend entweder die gelesenen Bytes oder aber eine Fehlermeldung, falls dies nicht geglückt ist.

*GEMDOS(R,64,Handle,HIGH(Anzahl),LOW(Anzahl),HIGH (Adresse),LOW(Adresse))* *Fwrite*
Schreibt die Anzahl Bytes auf dem Puffer (String), auf den der Zeiger Adresse weist, in die Datei mit der Nummer Handle.

*GEMDOS(R,65,HIGH(Adresse),LOW(Adresse))* *Fdelete*
Löscht eine Datei von der Diskette. R enthält eine Fehlernummer bzw. 0, falls alles geklappt hat. Entspricht dem Befehl KILL "<Datei>".

*GEMDOS(R,66,HIGH(Anzahl),LOW(Anzahl), Handle,Modus)* *Fseek*
Während üblicherweise eine Datei nur sequentiell geschrieben werden kann, gestattet diese Funktion einen Zeiger innerhalb einer Datei an eine bestimmte Position zu setzen. Drei Parameter werden benötigt:

- Anzahl der Bytes, um die der Zeiger verschoben wird.
- Handle-Nummer der Datei von Fopen (Fcreate).
- Modus, wobei hier drei Modi möglich sind:

  0 - Vom Dateianfang aus verschieben.
  1 - Von der aktuellen Position aus verschieben.
  2 - Vom Datei-Ende rückwärts verschieben.

Bei Modus 1 muß Anzahl positiv, bei Modus 2 negativ oder positv (je nach Richtung) und bei Modus 3 schließlich stets negativ sein, da vom Datei-Ende aus zurückgegangen wird. Wird über das Datei-Ende hinausgegangen, enthält R eine Fehlermeldung.

*GEMDOS(R,67,HIGH(Adresse),LOW(Adresse),Modus, Attribut)* *Fattrib*

Diese Funktion gestattet es, das Attribut einer Datei zu lesen oder zu ändern. Man übergibt ihr die Adresse des Dateinamens, den Modus (0 => Attribut ermitteln, 1 => Attribut setzen) und das neue Attribut:

0 Datei les- und beschreibbar.
1 Datei nur lesbar, kann jedoch nicht beschrieben werden.
2 Datei ist verborgen (Hidden-File).
3 Systemdatei.
8 Diskettenname (Disk Label).
16 Ordner.

Die Attribute von Ordnern und Diskettennamen können mit dieser Funktion nicht geändert werden. Ein Bespiel für die Anwendung:

```
DEF PROC Attribut(Datei$,Attribut%)
    LOCAL Dummy%, Adr
    Datei$=Datei$+CHR$(0)' damit er nicht abstürzt
    Adr= LPEEK(SEGPTR+28)+ LPEEK( VARPTR(Datei$))
    GEMDOS(Dummy%,HIGH(Adr),LOW(Adr),1,Attribut%)
    IF Dummy% < 0 THEN ' hat nicht ganz geklappt!
       FORM_ALERT(1,"[3][   Fehler!][ Abbruch ]")
    ENDIF
RETURN
```

*GEMDOS(R,69,Device)* *Fdup*

*GEMDOS(R,70,Device,Handle)* *Ffcorce*

Beide Routinen leiten eine Ein- oder Ausgabe von GEMDOS-Routinen auf den Standard-Ausgabekanal um. Device bezeichnet dabei:

1 Tastatur und Bildschirm.
2 Serielle Schnittstelle.
3 Drucker.

*GEMDOS(R,71,HIGH(Adresse),LOW(Adresse),Drv)* *Dgetpath*
Schreibt in einen 64 Byte großen Puffer, der ab Adresse zu finden ist, den aktuellen Pfadnamen für das Laufwerk Drv. Die Laufwerksbezeichnung am Pfadanfang selbst wird nicht in den Puffer geschrieben, sondern muß selbst hinzugefügt werden.

*GEMDOS(R,72,HIGH(Anzahl),LOW(Anzahl))* *Malloc*
Diese Funktion fordert Anzahl von Bytes Speicherplatz an. In R wird die Adresse des reservierten Speicherbereichs zurückgegeben. Diese Funktion entspricht im wesentlichen dem Befehl MEMORY, wobei mit Malloc angeforderte Speicherbereiche nicht durch ein CLEAR gelöscht werden.

*GEMDOS(R,73,HIGH(Adresse),LOW(Adresse))* *Mfree*
Gibt den mit Malloc angeforderten Speicherblock wieder frei.

*GEMDOS(R,78,HIGH(Adresse),LOW(Adresse),Atr)* *Fsfirst*
Sucht den ersten Eintrag im Directory mit dem Attribut Atr. Dieser wird in den DTA-Puffer übertragen und kann dort abgeholt werden. Diese Funktion sollte in ST-BASIC stets durch ein OPEN "F",... umgangen werden.

*GEMDOS(R,79)* *Fsnext*
Sucht den nach Aufruf von Fsfirst nächsten Eintrag im Directory. Auch dieser Befehl kann durch OPEN "F",... umgangen werden.

*GEMDOS(R,86,0,HIGH(alt),LOW(alt),HIGH(neu),*
*LOW(neu))* *Frename*
Benennt eine Datei um. Durch NAME ... AS ersetzten!

*GEMDOS(R,87,HIGH(Adresse),LOW(Adresse),*
*Handle,Modus)* *Fdattime*
Diese Routine gestattet das Ändern des Datums und der Zeit einer Datei. Dazu übergibt man dem Aufruf die Handle-Nummer des geöffneten Files (Fopen), sowie die Adresse eines 4 Byte großen Puffers (String), in den die Daten geschrieben (Modus=0) oder dessen Inhalt die alten Daten ersetzen sollen (Modus=1).

## 4.4 Das BIOS

Das Bios (BASIC Input- Output System) ist für die normalen Ein- und Ausgaberoutinen zuständig und wird über die Funktion

```
BIOS([Rückgabewert],<Funktionsnummer>[,Parameter])
```

aufgerufen. Folgende BIOS-Routinen sind im Betriebssystem des Atari ST implementiert:

*BIOS(,0,HIGH(Adresse),LOW(Adresse))* *Getmpb*
Diese Routinen führt die Speicherverwaltung des GEMDOS durch. Sie liefert eine Adresse auf einen Speicherblock, mit dem Memory-Parameterblock (MPB). Dieser Block wiederum enthält drei weitere Zeiger, die jeweils auf eine weitere Struktur verweisen. Ohne genaue Betriebssystemkenntnisse geht hier gar nichts!

*BIOS(R,1,Device)* *Bconstat*
Die Funktion prüft, ob ein Zeichen vom jeweiligen Device anliegt. Folgende Werte sind für Device anzugeben:

1 Serielle Schnittstelle (AUX, RS232).
2 Tastatur und Bildschirm (nicht in ST-BASIC verwenden!).
3 MIDI-Port.

*BIOS(R,2,Device)* *Bconin*
Liest ein Zeichen vom Gerät Device ein. In ST-BASIC sollte diese Funktion nicht für die Tastatur verwendet werden.

0 Parallele Schnittstelle (Centronics-Port, Drucker!).
1 Serielle Schnittstelle (RS232).
2 Tastatur und Bildschirm.
3 MIDI-Schnittstelle.

*BIOS(,3,Device,Zeichen)* *Bconout*
Gibt ein Zeichen auf dem jeweiligen Gerät aus. Zur Zeichenausgabe sollte in ST-BASIC der Bildschirm gemieden werden.

0 Parallele Schnittstelle (Centronics-Port, Drucker!).
1 serielle Schnittstelle (RS232).
2 Tastatur und Bildschirm.
3 MIDI-Schnittstelle.
4 Tastaturprozessor (Vorsicht!).

*BIOS(R,4,Flag,HIGH(Adresse),LOW(Adresse),Sektorzahl, Sektornummer,Drive)* *Rwabs*
Für Flag = 0 werden Sektorzahl Sektoren ab dem logischen Sektor Sektornummer in den ab Adresse stehenden Puffer geschrieben. Flag = 1 sorgt dafür, daß der Pufferinhalt auf Diskette geschrieben wird, Flag = 2 bzw. Flag = 3 arbeiten wie 1 und 2, ignorieren jedoch einen erfolgten Diskettenwechsel.

*BIOS(R,5,Nummer,HIGH(Adresse),LOW(Adresse))* *Setexc*
Diese Funktion ändert einen Exeptionvektor um. Nummer beinhaltet dabei die Nummer des zu ändernden Vektors. Die Adresse der Routine, die den alten Vektor ersetzten soll, muß ab Adresse stehen. Möchten Sie einen der 256 Exeption-Vektoren des 68000-Prozessors oder einen der 8 GEM-Vektoren lesen, muß als Nummer der Wert -1 angegeben werden.

*BIOS(R,6)* *Tickcal*
Liefert die Zeit in Millisekunden, die zwischen zwei Timeraufrufen vergangen ist.

*BIOS(R,7,Drive)* *Getbpb*
Ermittelt die Adresse des BIOS-Parameter-Blocks für das angegebene Laufwerk. Der Block besteht aus 9 Integerzahlen:

Recsiz Sektorgröße in Bytes.
Clsiz Cluster-Größe in Sektoren.
Clsizb Cluster-Größe in Bytes.
Rdlen Länge des Directories in Sektoren.
Fsiz Größe der File-Allocation-Table (FAT) in Sektoren. Die FAT gibt Auskunft, in welchen Clustern eine Datei abgespeichert wurde und dient dazu, die folgenden Cluster zu finden.
Fatrec Sektornummer der FAT-Kopie.

| | |
|---|---|
| Datrec | Sektornummer der ersten Daten-Clusters. |
| Numcl | Gesamtzahl der Datencluster auf der Diskette. |
| Bflags | Diverse Flags. |

Für Drive gelten dabei wieder folgende Werte:

0 Laufwerk A.
1 Laufwerk B.
2 Laufwerk C (Festplatte).

*BIOS(R,8,Device)* *Bcostat*
Stellt fest, ob das Ausgabegerät bereit ist, das nächste Zeichen zu empfangen. Die Rückgabewerte sind dabei:

1 Das Gerät ist zur Ausgabe bereit.
0 Das Gerät ist (noch) nicht bereit.

Folgende Devices können angesprochen werden:

0 Parallele Schnittstelle (Centronics-Port, Drucker!).
1 Serielle Schnittstelle (RS232).
2 Tastatur und Bildschirm.
3 MIDI-Schnittstelle.
4 Tastaturprozessor.

*BIOS(R,9,Drive)* *Mediach*
Stellt fest, ob ein Wechsel der Diskette auf dem angegebenen Laufwerk erfolgt ist. Rückgabewerte:

0 Die Diskette wurde sicher nicht gewechselt.
1 Diskettenwechsel könnte erfolgt sein.
2 Diskettenwechsel ist mit Sicherheit erfolgt

*BIOS(R,10)* *Drvmap*
Ermittelt die Laufwerkskonfiguration, wobei ein Bitvektor in R resultiert, in dem jedes gesetzte Bit für ein jeweils angeschlossenes Laufwerk steht:

Bit 0 => Laufwerk A angeschlossen
Bit 1 => Laufwerk B angeschlossen
Bit 2 => Laufwerk C angeschlossen

*BIOS(R,11,Status)* *Kbshift*
Ermittelt bzw. verändert den Status der Tastatur-Sondertasten. Ist die Variable Status nicht negativ, so werden die Sondertasten gemäß dem Bitmuster von Status neu gesetzt, ein Status von -1 liefert einen Bitvektor nach folgender Tabelle:

Bit 0 => Rechte Shift-Taste.

Bit 1 => Linke Shift-Taste.

Bit 2 => <CONTROL>-Taste.

Bit 3 => Alt-Taste.

Bit 4 => Caps-Lock aktiv.

Bit 5 => CLR/HOME (rechte Maustaste).

Bit 6 => INSERT (linke Maustaste).

Bit 7 => Nicht verwendet, immer 0.

## 4.5 Das XBIOS

Das XBIOS (eXtended Basic Input-Output System) gestattet die zusätzlichen Funktionen, die der Atari ST beinhaltet. Der Bildschirmspeicher, die Verwaltung der MIDI-Schnitt-stelle und der Maus oder das Setzen der Bildschirmfarben, all diese Dinge gehen auf das Konto des erweiterten BIOS. Sein Aufruf erfolgt über:

```
XBIOS([Rückgabewert],<Funktionsnummer>[,Parameter])
```

*XBIOS(,0....)* *Initmous*
Diese Routine initialisiert die Mausroutinen, wobei die Funktion nicht unbedingt mit GEM verträglich ist.

*XBIOS(R,1,Menge)* *Ssbrk*
Reserviert Speicherplatz für ROM-Module

*XBIOS(R,2)* *Physbase*
Ermittelt die Adresse des Bildschirms, der gerade aktiv ist.

*XBIOS(R,3)* *Logbase*
Ermittelt die Adresse des Bildschirms, auf den gerade ausgegeben wird.

*XBIOS(R,4)* *Getrez*
Ermittelt die Auflösung des Bildschirms:

0 Niedrige Auflösung (Lores) 320*200 Pixel, 16 Farben.
1 Mittlere Auflüsung (Midres) 640*200 Pixel, 4 Farben.
2 Hohe Auflösung (Hires) 640*400 Pixel, 2 Farben.

*XBIOS(,5,HIGH(log_adr),LOW(log_adr),HIGH (pys_adr),LOW(pys_adr),Auflösung)* *Setscreen*
Diese Funktion gestattet das Abändern der Bildschirmparameter. Log_adr, sowie Pys-adr enthalten die Adressen des logischen und physikalischen Bildschirms, die über XBIOS(2) und XBIOS(3) ermittelt werden können. Über Auflösung können folgende Auflösungen eingestellt werden:

0 Niedrige Auflösung (Lores) 320*200 Pixel, 16 Farben.
1 Mittlere Auflösung (Midres) 640*200 Pixel, 4 Farben.
2 Hohe Auflösung (Hires) 640*400 Pixel, 2 Farben.

*XBIOS(,6,HIGH(Adresse),LOW(Adresse))* *Setpalette*
Adresse verweist auf die (gerade) Anfangsadresse einer Farbtabelle mit 16 Farben. Für jede Farbe enthält diese Tabelle eine Integer-Zahl (Word). Vgl. PALETTE in ST-BASIC

*XBIOS(R,7,Farbe,Farbwert)* *Setcolor*

Mit dieser Funktion kann eine Farbe gezielt geändert werden. Dem Parameter Farbe ist die Farbnummer (1-15) zu übergeben, der Farbwert kann mit 0 bis $777 neu gesetzt werden. Möchte man den Farbwert einer einzelnen Farbe ausgeben, so ist dem Parameter Farbe der negative Wert -1 zu übergeben.

*XBIOS(R,8,HIGH(Adresse),LOW(Adresse),0,0,Drv,Sektor, Spur,Seite,Anzahl)* *Floprd*

Diese Funktion liest einzelne Sektoren vom Laufwerk A (Drv = 0) usw. in den Puffer, auf den der Zeiger Adresse weist. Als Sektor ist der erste zu lesende Sektor (0-9) anzugeben, die Spur reicht von 0-79 bei 80-Spur-Laufwerken (bzw. darüber hinaus bei "fett" formatierten Disketten bis max. 82). Seite enthält den Wert 0 für Seite 1 bzw. 1 für Seite 2 (nur 720 KByte Floppies: SF 314). Mit Anzahl kann die Zahl der nacheinander zu lesenden Sektoren angegeben werden.

*XBIOS(R,9,....)* *Flopwr*

Die Parameter dieser Funktion entsprechen der von XBIOS(8), hierbei werden die Pufferdaten jedoch auf Diskette geschrieben.

*XBIOS(R,10,HIGH(Adresse),LOW(Adresse),0,0,Drv,Sektor zahl,Spur,Seite,Interleave,$-789B,$4321,Formatwert)*

*Ffopfmt*

Diese Funktion formatiert eine Spur. Das Formatieren einer Diskette ist deshalb wichtig, damit anschließend Daten auf die Diskette aufgezeichnet und auch wieder eingelesen werden können. Folgende Parameter sind anzugeben:

| Parameter | Bedeutung |
|---|---|
| Adresse | Zeigt auf einen Buffer (String), der die Daten zum Formatieren einer Spur (Track) enthält. Für 9 Sektoren muß dieser Buffer zumindest 8 Kbyte groß sein. |
| Drv | Nummer des Laufwerkes (0 = A, 1 = B) |
| Sektorzahl | Anzahl der Sektoren je Spur. Normalerweise 9, ein Wert von 10 sollte allerdings immer möglich sein. |
| Spur | Nummer der zu formatierenden Spur |
| Seite | Diskettenseite (0 bzw. 1) |
| Interleave | bestimmt die Reihenfolge, in der die einzelnen Sektoren auf Diskette aufgezeichnet werden. Interleave = 1 => Reihen |

| Parameter | Bedeutung |
|---|---|
| | folge der Sektoren: 1-2-3-4-5-6-7-8-9. Interleave = 2 => Reihenfolge: 1-3-5-7-9-2-4-6-8. Üblicherweise wird ein Interleavefaktor von '1' benutzt. |
| Formatwert | Dieser Wert wird als Datenbytes auf die Diskette geschrieben. Atari empfiehlt $E5E5, der Wert F0 bis F5 dürfen nicht benutzt werden, da der Floppy-Disk-Controller diese als Befehle interpretiert. |

Als Rückgabewert erhält man eine 0, wenn die Formatierung ohne Fehler durchgeführt werden konnte. Ist der Rückgabewert negativ (z.B. -16: Bad Sectors), so ist während der Formatierung ein Fehler aufgetreten. Im Puffer findet sich dann eine Liste, die alle defekten Sektoren in diesem Track enthält.

*XBIOS(R,11)* *Getdsb*
Nicht benutzt, liefert stets den Wert 0.

*XBIOS(,12,Länge,HIGH(Adresse),LOW(Adresse))* *Midiws*
Schreibt die in dem ab Adresse abgelegten Daten (String!) auf den Midi-Port.

*XBIOS(R,14,Device)* *Iorec*
Diese Funktion ermittelt einen Zeiger auf den Puffer-Datensatz für die Ein- und Ausgabe auf dem Device:

0 RS232 Eingabe- und Ausgabepuffer.

1 Tastatur.

2 MIDI.

Der Puffer selbst besitzt folgende Struktur (bis auf Ibuf besteht der Puffer aus Integerzahlen):

| Variable | Bedeutung |
|---|---|
| Ibuf (Long!) | Zeiger auf den Eingabebuffer |
| Ibufsize | Größe des Eingabebuffers in Bytes |
| Ibufhd | nächste Schreibposition im Buffer |
| Ibuftl | Adresse, ab der man den Buffer lesen kann |

| Variable | Bedeutung |
|---|---|
| Ibuflow | Solange Bufferinhalt kleiner als diese Variable können Daten empfangen werden. |
| Ibufhi | Eingabebuffer voll, es können keine weiteren Daten empfangen werden. |

*XBIOS(,15,Baud,Kom_Par,Usart-Reg,Empf_Stat,Trans_Stat,Sysnchr_Stat)* *Rsconf*

Mit dieser Funktion kann die RS232 (serielle) Schnittstelle konfiguriert werden. Die Bedeutung der Parameter:

| Parameter | Bedeutung |
|---|---|
| Baud | Baudrate (0-15). Zur Verfügung stehen:<br>19200,9600,4800,3600,2400,2000,1800,<br>1200,600,300,200,150,134,110,75,50 |
| Kom_Par | Kommunikationsparameter, mit den Modi:<br>0 - kein Handshake<br>1 - XON/XOFF<br>2 - RTS/CTS<br>3 - XON/XOFF u. RTS/CTS (Modi 1 und 2) |
| Usart_Reg | Wert des USART-Registers im MFP:<br>Bit 1: 0 = odd,<br>1 = even parity<br>Bit 2: 0 = no parity,<br>1 = parity<br>Bit 3,4: 0-3:synchron, 1 Stopbit,<br>1.5 Stopbits, 2 Stopbits<br>Bit 5,6: 0-3: 8,7,6,5 Bits<br>Bit 7: 0: Frequenz nicht teilen,<br>1: Frequenz teilen |
| Empf_Stat | Bit 0: 1 = RS232-Empfänger ein.<br>Bit 1: SCR-Zeichen übertragen |
| Trans_Stat | Bit 0: 1 = RS232-Sender ein<br>Bit 1,2: 0 = Ausgang hochohmig,<br>1 = H,<br>2 = L,<br>3 = Ausgang auf Eingang<br>Bit 3: 1 = Break senden<br>Bit 5: 1 = Empfänger nach Zeichenempfang einschalten |

*XBIOS(R,16,HIGH(Norm),LOW(Norm),HIGH(Shift),*
*LOW(Shift),HIGH(Lock),LOW(Lock))* *Keytbl*
Mit dieser Funktion kann die Tastaturbelegung des Atari ST geändert werden. Das TOS verwaltet drei Tabellen, in denen die Scancodes der einzelnen Tasten, sowie die dazugehörigen ASCII-Werte verzeichnet sind. Eine Tabelle für Tasten ohne Shift-Taste, eine für Tastenbelegung mit Shift-Taste, sowie eine Tabelle für aktiviertes Caps-Lock. Jede dieser Tabellen enthält 128 Bytes. Oben angegebene Parameter sind Zeiger auf die entsprechenden Tabellen. Wird statt der Adresse der Wert -1 übergeben, bedeutet dies, daß die entsprechende Tabelle nicht geändert wird. Möchten Sie die Tastaturbelegung ändern, müssen nur drei Strings, die die neuen Tabellen enthalten, angelegt werden und die Zeiger auf diese Strings mit XBIOS(R,16...) dem Betriebssystem untergeschoben werden. Die Umbelegung kann mit XBIOS(,14) wieder rückgängig gemacht werden.

*XBIOS(R,17)* *Random*
Diese Funktion gibt in R eine 24-Bit Zufallszahl zurück.

*XBIOS(,18,HIGH(Adresse),LOW(Adresse),HIGH(Ser),*
*LOW(Ser),Typ,Exec)* *Protobt*
Diese Funktion erzeugt oder ändert einen Bootsektor. Adresse gibt an, wo der Bootsektor im Speicher zu finden ist (Größe: 512 Bytes). Ser ist die neue 24-Bit-Seriennummer der Diskette bzw. -1 wenn die Nummer nicht geändert werden soll.

Als Diskettentyp sind folgende Parameter relevant:

0 40 Spuren, einseitig (180 KByte)
1 40 Spuren, zweiseitig (360 KByte) (IBM-Format)
2 80 Spuren, einseitig (360 KByte) (SF 354)
3 80 Spuren, zweiseitig (720 KByte) (SF 314)

-1 => Diskettentyp wird nicht geändert

Exec gibt an, ob der Bootsektor ausführbar sein soll:

0 Bootsektor ist nicht ausführbar
1 Bootsektor ist ausführbar
-1 => Bootsektor wird nicht geändert

Der Bootsektor einer Systemdiskette trägt folgenden Aufbau:

| Byte | Inhalt | 40 SS | 40 DS | 80 SS | 80 DS |
|---|---|---|---|---|---|
| 1-2 | | Sprung zum Bootprogramm | | | |
| 3-8 | | "LOADER" | | | |
| 9-11 | | Seriennummer | | | |
| 12-13 | BPS | 512 | 512 | 512 | 512 |
| 14 | SPC | 1 | 2 | 2 | 2 |
| 15-16 | RES | 1 | 1 | 1 | 1 |
| 17 | NFATS | 2 | 2 | 2 | 2 |
| 18-19 | NDIRS | 64 | 112 | 112 | 112 |
| 20-21 | NSECTS | 360 | 720 | 720 | 1440 |
| 22 | MEDIA | 252 | 253 | 248 | 249 |
| 23-24 | SPF | 2 | 2 | 5 | 5 |
| 25-26 | SPT | 9 | 9 | 9 | 9 |
| 27-28 | NSIDES | 1 | 2 | 1 | 2 |

Dabei haben die Abkürzungen folgende Bedeutungen:

BPS In einem Sektor vorhandene Bytes.
SPC Sektoren je Cluster.
RES Anzahl der reservierten Sektoren am Disk-Anfang.
NFATS Anzahl der FATs auf der Diskette.
NDIRS Erlaubte Zahl der Directoy-Einträge.
MEDIA Media Descriptor Byte, nicht benutzt.
SPF Anzahl der Sektoren in jeder FAT.
SPT Anzahl der Sektoren je Spur (9).
NSIDES Anzahl der Diskettenseiten.

Nachdem eine Diskette formatiert wurde, muß mittels dieser Funktion ein Bootsektor erzeugt und anschließend mit der Funktion Flopwr auf die Diskette geschrieben werden. Siehe dazu auch die Routine Format, die anschließend besprochen wird.

*XBIOS(R,19,HIGH(Adresse),LOW(Adresse),0,0,Laufwerk, Sektor,Spur,Seite,Anzahl)* *Flopver*
Prüft Sektoren auf deren Lesbarkeit. Die Syntax entspricht der Funktion Floprd.

*XBIOS(R,,20)* *Scrdmp*
Erstellt eine Hardcopy des Bildschirminhalts

*XBIOS(R,21,Funktion,Geschwindigkeit)*
Schaltet den Cursor aus und ein, ferner kann die Blinkrate eingestellt werden:

| Funktion | bewirkt |
|---|---|
| 0 | Cursor ausschalten |
| 1 | Cursor einschalten |
| 2 | Cursor blinkend |
| 3 | Cursor nicht blinkend |
| 4 | Blinkrate setzen |
| 5 | Blinkgeschwindigkeit erfragen |

*XBIOS(,22,HIGH(Zeit),LOW(Zeit))* *Settime*
Setzt die Uhrzeit und das Datum. Das Format setzt sich wie folgt zusammen:

| Bits | Inhalt |
|---|---|
| 0-4 | Sekunden in Zweierschritten (0-29) |
| 5-10 | Minuten (0-59) |
| 11-15 | Stunden (0-23) |
| 16-20 | Tag (1-31) |
| 21-24 | Monat (1-12) |
| 25-31 | Jahr (+ 1980) (0-119) |

*XBIOS(R,23)* *Gettime*
Holt die Uhrzeit und das Datum in obigem Format.

*XBIOS(,24)* *Bioskeys*
Die mit der Funktion XBIOS(16) vorgenommenen Änderungen werden wieder rückgängig gemacht.

*XBIOS(,25,Länge,HIGH(Adresse),LOW(Adresse))* *Ikbytedws*
Leitet Befehle, die ab Adresse in einem Puffer stehen, an den Tastaturprozessor weiter. Länge ist die Anzahl der im Puffer vorhandenen Daten minus 1.

*XBIOS(,26,Nummer)* *Jidisint*
Sperrt einen Interrupt des MFP.

*XBIOS(,27,Nummer)* *Jenabit*
Gibt einen Interrupt des MFP wieder frei.

*XBIOS(R,28,Daten,Registernummer)* *Giaccess*
Schreibt in das angegebene Register des Soundchips bzw. liest das Register aus. Die Registernummer umspannt die Werte von 0 bis 15, ist zusätzlich Bit 7 in der Registernummer gesetzt, handelt des sich um einen schreibenden Zugriff.

*XBIOS(,29,Bitnummer)* *Offgibit*
Löscht ein Bit im Port A des Soundchips.

*XBIOS(,30,Bitnummer)* *Ongibit*
Setzt ein Bit im Prot A des Soundchips

*XBIOS(,32,HIGH(Adresse),LOW(Adresse))* *Dosound*
Spielt eine Klangfolge, die sich in einer Tabelle ab Adresse im Speicher befindet, ab.

*XBIOS(,33,Einstellung)* *Setprt*
Stellt die Druckerdaten ein:

| Bit | nicht gesetzt | gesetzt |
|---|---|---|
| 0 | Matrix- | Typenraddrucker |
| 1 | Farb- | SW-Drucker |
| 2 | ATARI- | EPSON-Drucker |
| 3 | Draft- | NLQ-Modus |
| 4 | Centronics | RS232 |
| 5 | Endlos- | Einzelblattpapier |

*XBIOS(R,34)* *Kbdvbase*
Liefert einen Zeiger auf eine Tabelle von Zeigern:

- Midi-Eingabe
- Tastatur-Fehler
- Midi-Fehler
- IKBD-Status
- Maus-Routinen
- Uhrzeit-Routine
- Joystick-Routine

*XBIOS(R,35,Verzögerung,Geschwindigkeit)* *Kbrate*
Diese Routine stellt - sofern nicht der Wert -1 angegeben wird - die Verzögerungszeit und die Wiederholgeschwindigkeit der Tasten-Repeat-Funktion ein. In R liefert die Funktion die bisher aktiven Werte zurück: In den Bits 0-7 die Wiederholgeschwindigkeit, in 8-15 die bisherige Verzögerung.

*XBIOS(,26,HIGH(Adresse),LOW(Adresse))* *Prtblk*
Eine weitere Routine zur Hardcopy-Erstellung, in der Handhabung jedoch auch um einiges komplizierter.

*XBIOS(,37)* *Vsync*
Wartet auf einen vertikalen Bildrücklauf (vertical blank)

*XBIOS(,38,HIGH(Adresse),LOW(Adresse))* *Supexec*
Führt ein Maschinenprogramm, das sich ab Adresse im Speicher befindet, im Supervisormodus aus.

*XBIOS(,39)* *Puntaes*
Löscht bei von Diskette gebooteten Betriebssystemen das AES. Der Befehl muß aus dem AUTO-Ordner heraus aufgerufen werden.

## 4.6 Betriebssystemprogrammierung

Anhand zweier Beispiele möchte ich Ihnen nun den Einsatz der verschiedenen Betriebssystemroutinen einmal näher verdeutlichen. Darüber hinaus erfahren Sie noch ein paar nützliche Informationen.

### Diskette formatieren

Ehe man auf einer Diskette Daten speichern kann, muß diese erst formatiert werden. Dabei wird ein Gerüst auf die Diskette gebracht, in das die Daten später eingetragen werden. Nach der Formatierung besteht die Diskette aus einzelnen Spuren (Tracks), die sich als Untereinheit noch einmal in logische Sektoren untergliedern lassen. Da eine Diskette mit einer unterschiedlichen Zahl von Spuren und Sektoren, einseitig und zweiseitig formatiert werden kann, gibt es noch eine Instanz auf jeder Diskette, die Bootsektor genannt wird. Schließlich werden in die FAT (File Allocation Table), die übrigens gleich zweimal auf jeder Diskette vorhanden ist, noch die bereits belegten Diskettenteile eingetragen, damit nicht ein bestimmter Sektor zweimal beschrieben wird (wäre peinlich!).

GEM, die grafische Benutzeroberfläche des Atari ST, bietet eine komfortable Möglichkeit die Diskette zu formatieren, solange man sich im Desktop befindet. Möchte man eine Diskette von einem Anwenderprogramm aus formatieren, hat man den Aufruf der dafür benötigten Routinen im Betriebssystem schon selbst vorzunehmen! Also schreiten wir zur Tat!

Der erste Schritt, den die neue Procedure auszuführen hat, ist das Anlegen eines Puffers. Dieser Puffer wird vom TOS zum Formatieren der Diskette benötigt. Sein Inhalt ist nicht weiter interessant. Zu beachten ist jedoch, daß der Puffer auf keinen Fall zu kein sein darf. Für unsere Zwecke sollte eine Größe von 11 KByte vollkommen ausreichen. Im nächsten Schritt müssen wir dafür Sorge tragen, daß der so angelegte Puffer auf keinen Fall wieder verschoben wird. Wieso sollte dies geschehen?

Stellen Sie sich dazu einmal den Speicher des Computers vor, der als eine eindimensionale Liste aufgefaßt werden kann. Eindimensional deshalb, weil alle Bytes hintereinander angeordnet sind. Irgendwo in diesem Speicher legt der Computer seine Zeichenketten ab. Damit aber nicht unnötig Speicherplatz vergeudet wird, werden die verschiedenen Strings auch hintereinander abgelegt. Weist man einer Zeichenkette nun einen neuen Inhalt zu, der die bisherige Länge übersteigt, müßte der Computer eigentlich drei Schritte ausführen:

1. Die nachfolgenden Zeichenketten nach hinten schieben, um Platz für die zusätzlichen Zeichen zu schaffen.
2. Die alte durch die neue Zeichenkette ersetzen.
3. Da sich durch diese Aktion auch die Adressen aller anderen Zeichenketten geändert haben, müßten auch diese an die neuen Werte angepaßt werden.

Das macht natürlich kein Computer, da der Arbeitsaufwand dafür viel zu groß ist! Also wird die neue Zeichenkette einfach an das Ende des Stringspeichers gesetzt. Die alte Zeichenkette verbleibt weiterhin im Speicher, ist jedoch nur noch Makulatur, da sie nicht mehr benötigt wird (die neue steht ja in einem separaten Speicherbereich).

Doch irgendwann ist auch der größte Speicherplatz aufgebraucht, wenn man in irgendeiner Form manipulierte Zeichenketten einfach neu im Speicher anlegt. Dann heißt es aufräumen, und die ganzen nicht mehr benötigten String-Leichen zu entfernen. Diesen Aufräumvorgang, der äußerst zeitintensiv sein kann, nennt der Fachmann Garbage Collection. Eine vom ST-BASIC durchgeführte Garbage Collection können wird jedoch hier absolut nicht gebrauchen, da die Adresse des geschaffenen Puffers dem Betriebssystem übergeben werden muß. Und wehe, diese Adresse stimmt nicht mehr.

Ein Trick hilft uns aus der Patsche: Wir erzwingen eine Garbage Colletion. Dann muß der Interpreter seinen Speicher aufräumen, und die Adresse des Puffers wird vom Interpreter anschließend nicht mehr angetastet. Eine Garbage Collection erzwingen läßt sich mit dem Befehl

```
FRE("")
```

wobei ein Ergebnis zurückgeliefert wird (der nach dem Aufräumen des Stringspeichers zur Verfügung stehende Speicherplatz), das wir getrost ignorieren können. Deshalb die Zuweisung an die Variable Dummy. Der Rest ist Routine, und das im wahrsten Sinne des Wortes! Mit der XBIOS-Routine 10 formatieren wir die gesamte Diskette, und zwar Spur für Spur. Dabei erschien es mir sinnvoll, bei einer zweiseitigen Diskette nicht beide Seiten nacheinander, sondern abwechselnd zu formatieren. Deshalb die beiden ineinander verschachtelten FOR...NEXT-Schleifen.

Konnte eine Spur nicht korrekt formatiert werden (soll ja vorkommen), liefert die Funktion einen Fehlercode zurück, der als negativer Wert kenntlich ist. Der Benutzer soll auch etwas davon haben, deshalb drucken wir eine entsprechende Fehlermeldung auf den Monitor.

Der nächste Schritt gilt der Erstellung des Bootsektors. Da das Betriebssystem die verschiedenen Disketten anhand einer Seriennummer unterscheidet, sollte eine Zufallszahl, bestehend aus 24-Bit, diese neue Seriennummer bilden. Auch das erledigt wieder eine XBIOS-Routine für uns! Mit der so gewonnenen Seriennummer läßt sich der Bootsektor mittels der XBIOS-Funktion 19 (Protobt) erstellen. Dabei wird ein Pauschal-Bootsektor kreiert und anschließend (den genauen Aufbau eines Bootsektors kennen wir ja) auf unser Format abgeändert.

Den Befehl MKI$() kennen Sie bereits aus dem Kapitel über die Dateiverwaltung. Er wandelt ein Integer-Wort in eine zwei Byte lange Zeichenkette. Somit war es uns möglich, auch Zahlen in einer relativen Datei zu verwenden. Wie wird nun die Zahl in einen zwei Byte langen String gewandelt?

Der Computer arbeitet im Dualsystem. Dieses Dualsystem setzt sich aus digitalen Informationseinheiten, die Bits genannt werden, zusammen. Acht solcher Bits faßt man wiederum zu einem Byte zusammen. Damit lassen sich Zahlen von 0 bis 255 darstellen. Was nun, wenn der benötigte Zahlenbereich diesen Wert überschreitet? Dann muß ein zweites Byte herhalten, das die Wertigkeit 256 besitzt. Da dieses Byte um jeweils eine Einheit erhöht wird, wenn die Kapazität des anderen Bytes ausgeschöpft ist, nennt man es auch High-Byte, im Gegensatz zu seinem Kollegen, das konsequenterweise mit Low-Byte bezeichnet wird. Und genau in dieses Format (man möchte es nicht glauben!) wandelt MKI$ die ihm anvertraute Zahl.

Also noch mal: MKI$ wandelt ein Integer-Word in einen 2 Byte langen String, der folgendes Format trägt:

```
HIGHBYTE  LOWBYTE
```

Testen Sie es einmal selbst, indem Sie eine Zahl konvertieren, den resultierenden String zerlegen (Left$(..,1) bzw. Right$(..,1)) und den jeweiligen ASCII-Wert ermitteln. Mit der Formel

```
ASC(HIGHBYTE) * 256 + ASC(LOWBYTE)
```

erhalten Sie dann wieder Ihre ursprüngliche Zahl. Genau in dieser Form legt der Computer nun seine Zahlen ab, deshalb kann diese Funktion samt ihren Verwandten und Bekannten auch anderweitig eingesetzt werden. Ein Beispiel finden Sie in der Formatier-Routine, in der mit ihrer Hilfe die Zahl der Diskettenseiten, sowie die Zahl der Sektoren auf der Diskette in den Puffer des Bootsektors geschrieben wird.

Eine nicht unwesentliche Kleinigkeit fehlt noch! MKI$ wandelt seine Argumente in das Format HIGH-LOW-Byte. Dies ist das Format, in dem auch der 68000-Prozessor seine Daten ablegt. Doch nicht alle Prozessoren arbeiten nach diesem Strickmuster.

Tja, während ATARI zumindest bei der Wahl des Betriebssystems mit TOS ihr eigenes Süppchen gekocht hat, hat es ein zur Welt der MS-DOS-Rechner kompatibles Aufzeichnungsformat

gewählt. Dies bedeutet, daß der ST Disketten, die mit einem MS-DOS-Rechner aufgezeichnet wurden, zumindest lesen kann. Der umgekehrte Vorgang ist nicht möglich! (Versuchen Sie jetzt aber nicht mit Gewalt eine 5¼"-Diskette eines IBM-kompatiblen PCs in das 3½"-Laufwerk des ST zu pressen! Das geht unter Garantie schief! Die Diskettengrößen müssen schon übereinstimmen, wenn die Geschichte auch funktionieren soll.)

Die in MS-DOS-Rechnern verwendeten Prozessoren (Intel läßt grüßen) bevorzugen ein anderes Datenformat, als dies der 68000 des Atari ST tut: LOW-HIGH-Byte. Die Konsequenz daraus: Der Prozessor des Atari ST muß alle Daten erst einmal umdrehen, ehe sie auf Diskette geschrieben werden können, und bei einem Lesevorgang wiederholt sich dieses Spielchen (Der Prozessor des ST kann ja das Format, mit dem die Daten auf Diskette geschrieben werden, nicht gebrauchen!). Deshalb wird mittels MIRROR$ die Reihenfolge der beiden Bytes vor dem Eintrag in den Puffer erst einmal umgedreht.

Ist der Bootsektor erst einmal erstellt, kann er an seine Position auf die Diskette geschrieben werden! Die Arbeit ist beendet, und somit auch die Erklärung der Procedure. Die benötigten Parameter entnehmen Sie bitte dem Listing. Mit dieser Routine können Sie übrigens auch eine Diskette fett formatieren, das heißt mit 82 oder gar 83 Tracks und 10 Sektoren je Spur (vorausgesetzt Ihr Laufwerk macht das mit!).

```
100 '*****************************************************************
110 '*                          FORMAT.BAS                           *
120 '*---------------------------------------------------------------*
130 '* Autor: Michael Maier    Version: 1.00     Datum: 16.09.1988 *
140 '*      Ein Programm aus dem 'GROSSEN ST-BASIC-BUCH'             *
150 '*        (C) 1988 by DATA BECKER GmbH Düsseldorf                *
160 '*****************************************************************
170 '
180 '        ================================================
190 '        = Die folgende Procedure formatiert eine Diskette =
200 '        ================================================210 '
220 ' Aufruf:         Format(drv%,sides%,tracks%,spt%)
230 ' ======
240 '
250 ' Parameter:           drv%:   Nummer der Laufwerks
```

```
260 ' =========                    0 => Laufwerk A
270 '                              1 => Laufwerk B
280 '                       sides% 1 => einseitige Disk (SF 354)
290 '                              2 => zweiseitige Disk (SF 314)
300 '                       tracks% Anzahl der Spuren (80 - 83)
310 '                       spt%   Sektoren je Spur (9 bzw. 10)
320 '
330 '
340 DEF PROC Format(De%,Sid%,Trk%,St%)
350  LOCAL T%,S%,Du%L,A%L,Nsects%,Buffer$
360  '
370  ' genügend Platz für 10 Sektoren reservieren
380  '
390  Buffer$= SPACE$(11000)
400  '
410  ' Damit der Buffer nicht mehr verschoben wird !
420  '
430  Du%L= FRE("")
440  '
450  ' jetzt die Adresse berechnen
460  '
470  A%L= LPEEK( VARPTR(Buffer$))+ LPEEK( SEGPTR +28)
480  FOR T%=0 TO Trk%-1
490   FOR S%=0 TO Sid%-1
500    ' Spur formatieren, und zwar abwechselnd Seite 1 / Seite 2
510    XBIOS(Du%,10,HIGH(A),LOW(A),0,0,De%,St%,T%,S%,1,$-789B,$4321,0)
520    ' negative Zahl als Rückgabeparameter => Fehler ist aufgetreten
530    IF Du%<0 THEN
540       PRINT
550       PRINT "Formatierfehler auf Seite:";S%;" Track:";T%
560    ENDIF
570   NEXT S%
580  NEXT T%
590  '
600  ' Anzahl der Sectoren auf dieser Diskette mach der Formel
610  ' Anzahl = Anzahl Tracks * Anzahl Seiten * Steps je Track
620  '
630  Nsects%=Trk%*Sid%*St%
640  '
650  ' 24-Bit Zufallszahl für Seriennummer
660  '
670  XBIOS (Du%L,17)
680  '
690  ' Bootsektor im Buffer erzeugen
700  '
710  Buffer$= SPACE$(513)
```

```
720  A%L= LPEEK( SEGPTR +28)+ LPEEK( VARPTR(Buffer$))
730  XBIOS (,18, HIGH(A%L), LOW(A%L), HIGH(Du%L), LOW(Du%L),3,0)740  '
750  ' auf unsere Werte abändern, jedoch in LOW-, HIGH-Byte!
760  '
770  MID$ (Buffer$,20,2)= MIRROR$( MKI$(Nsects%))
780  MID$ (Buffer$,25,2)= MIRROR$( MKI$(Sid%))
790  '
800  ' und auf die Diskette schreiben, fertig
810  '
820  XBIOS (,9, HIGH(A%L), LOW(A%L),0,0,0,1,0,0,1)
830  '
840 RETURN
```

### Inhaltsverzeichnis einer Diskette einlesen

Es gibt zwei Wege in ST-BASIC, das Inhaltsverzeichnis einer Diskette innerhalb eines eigenen Programms einzulesen. Einmal über die Verwendung der beiden Betriebssystemroutinen

```
FSFIRST
```

und

```
FSNEXT
```

sowie über den Befehl OPEN "F", der zwar eigentlich nichts mit dem Betriebssystem am Hut hat, dennoch aber in ähnlicher Weise funktioniert. Die Verwendung der beiden Routinen FSFIRST und FSNEXT ist im Kapitel über die GEM-Programmierung bei der eigenen File-Selector-Box realisiert. Hier geht es nun vornehmlich darum - das Handbuch schweigt sich zu diesem Thema nämlich aus -, wie mit OPEN "F" ein Directory eingelesen werden kann.

Obwohl nach OPEN "F" der Befehl GET benutzt wird, um die einzelnen Einträge einzulesen, hat die beim OPEN als Schußlicht angegebene Zahl 63 nichts mit der Datensatzlänge zu tun, wie man vielleicht vermuten könnte. Sie ist vielmehr das Ergebnis einer Kombination der möglichen Dateiattribute, die bei der Suche berücksichtigt werden sollen:

| Dateityp | Wert |
|---|---|
| schreibgeschützte Dateien | 1 |
| verborgene Dateien (hidden files) | 2 |
| Systemdateien | 4 |
| Diskettenname | 8 |
| Ordner | 16 |
| Datei korrekt geschlossen | 32 |
| (ohne Bedeutung) | |

Möchten Sie nach dem Diskettennamen fahnden, so brauchen Sie nur die Zahl 8 anzugeben. Auch eine Kombination verschiedener Dateitypen ist durch Addition der dazugehörigen Werte möglich. Der Dateiname wird üblicherweise durch die beiden Jokerzeichen *.* ersetzt. Dadurch werden alle Einträge ausgeworfen, die die angegebenen Attribute tragen. Möchten Sie dagegen nur Programme mit der Endung .BAS angezeigt wissen, so ist als Dateiname *.BAS einzusetzen.

Die erste Aufgabe, die das Programm zu erledigen hat, besteht darin, zu überprüfen, ob überhaupt Dateien (mit den passenden Attributen bzw. Namen) auf dieser Diskette vorhanden sind. Dies kann geschehen, indem man testet, ob das Datei-Ende (die Diskettennamen werden zu einer Datei gehörig aufgefaßt) noch nicht erreicht ist:

```
NOT EOF(<Kanal>)
```

Jetzt liest man (am besten in einer Endlosschleife) Eintrag für Eintrag mit GET ein. Solange nach einem GET das File-Ende nicht überschritten ist, handelt es sich um einen gültigen Dateinamen, der ausgegeben werden kann. Andernfalls sollte man die Endlosschleife mit EXIT verlassen.

GET wiederum legt die Daten in einem mit FIELD definierten Puffer ab. Dieser muß 44 Byte groß sein. Sein Aufbau entspricht dem des DTA-Puffers, in den FSFIRST und FSNEXT ihre Daten verfrachten. Also steht ab Byte 31 der Dateiname, abgeschlossen von einem Nullbyte, das vor der Ausgabe mit PRINT noch entfernt wird. Fertig!

```
0  '*****************************************************************
1  '*                          DIR_1.BAS                            *
2  '*---------------------------------------------------------------*
3  '* Autor: Michael Maier    Version: 1.00     Datum: 16.09.1988 *
4  '*      Ein Programm aus dem 'GROSSEN ST-BASIC BUCH'             *
5  '*        (C) 1988 by DATA BECKER GmbH Düsseldorf                *
6  '*****************************************************************
7  '
8  '           Inhaltsverzeichnis einer Diskette lesen
9  '          ****************************************
10 '
11 '              Zuerst einmal die 'Datei' öffnen
12 '
13 OPEN "F",1,"*.*",63
14 ' sind überhaupt Einträge vorhanden?
15 IF NOT EOF(1) THEN
16   ' Feld zum Einlesen mit GET dimensionieren
17   ' der Dateiname steht ab Byte 31, und wird mit einem
18   ' Nullbyte 'CHR$(0)' abgeschlossen  ( 'C' läßt grüßen!)
19   FIELD 1,30 AS Buffer$,14 AS Name$
20   T%=0
21   REPEAT
22     GET 1,T%
23     ' Wenn eingelesener Datensatz gültig, dann ausgeben
24     IF NOT EOF(1) THEN
25         ' String nur bis zum Nullbyte berücksichtigen
26         PRINT LEFT$(Name$, INSTR(Name$, CHR$(0))-1)
27     ELSE
28        ' Ende des Directories erreicht => Schleife beenden
29        EXIT
30     ENDIF
31     ' nächsten Eintrag einlesen
32     T%=T%+1
33   UNTIL 0
34 ENDIF
35 CLOSE 1
```

# 5. Grafikprogrammierung

Grafik ist ein Thema, das fast jeden Computerfan irgendwann einmal beschäftigt, besonders dann, wenn sein Rechner auch noch mit einer gestochen scharfen Auflösung glänzen kann. Jeder von Ihnen wird wohl schon ein Malprogramm in der Hand gehalten haben, mit dessen Hilfe der Atari als Zeichenmaschine eingesetzt werden kann. Solche Malprogramme arbeiten im Gegensatz zu Zeichenprogrammen im Normalfall pixelorientiert, d.h. sie manipulieren in irgendeiner Form den Bildschirminhalt. Diese Manipulation - einmal durchgeführt - kann anschließend nicht mehr rückgängig gemacht werden.

Im Gegensatz zu den pixelorientierten Programmen gibt es objektorientierte Programme die alle Objekte in zweifacher Hinsicht verwalten. Einmal - wie ihre Kollegen auch - in pixelorientierter Form, ein zweites Mal in Form einer Objektbeschreibung. Eine so gefertigte Grafik kann dann editiert werden, ohne dabei andere Objektinhalte zu beinflussen. Der Haken an der Sache ist, daß solche Programme schwieriger zu programmieren sind. Ohne Vektoren und Matrizen geht es nicht! Deshalb finden Sie auch ein wenig Matrizenrechnung in diesem Kapitel. Aber fangen wir ganz einfach an:

## 5.1 Einfache Grafikbefehle

Es ist nicht mein Ansinnen, an dieser Stelle das Handbuch des ST-BASIC in irgendeiner Form wiederzukauen. Vielmehr sollen ein paar Befehle zur Grafikprogrammierung exemplarisch herausgegriffen werden.

### Die Grafik-Auflösungen des Atari ST

Je nachdem, welcher Monitor an den Computer angeschlossen (Farbe oder Schwarz-Weiß) und welche Grafikstufe gerade eingestellt ist, besitzt der Computer eine andere Auflösung. Unter Auflösung versteht man die Anzahl der Punkte, die auf dem Bildschirm dargestellt werden. Der Atari ST verfügt über drei verschiedene Auflösungsstufen:

**640*400 Punkte in HIRES (s/w)**
**640*200 Punkte in 4 Farben**
**320*200 Punkte in 16 Farben**

Während Sie also im Farbbetrieb zwischen zwei Auflösungen wählen können, ist dies beim Anschluß eines Schwarz-Weiß-Monitor nicht möglich. Ihm ist die höchste Stufe der Grafik, die HIgh RESolution (Hires) vorbehalten. Jeder dieser Punkte beansprucht im einfachsten Fall - das ist in Hires - genau ein Bit, womit wir bei einer Bildschirmgröße von 32000 Byte angelangt wären. Zur Farbdarstellung sind nun zusätzlich Informationen über die einzelnen Farben nötig, die dann auf Kosten der Auflösung gehen (deshalb auch die geringere Auflösung im Farbbetrieb).

Die obere linke Ecke des Bildschirms hat dann die Koordinaten (0,0), die rechte untere Ecke - in HIRES - (639,399). Ausnahme: Bei einem geöffneten Fenster ist die linke obere Ecke des Fensters der (logische) Nullpunkt. Diese Koordinaten werden für alle weiteren Befehle benötigt. Ich selbst beziehe mich bei folgenden Überlegungen - zumindest was die Koordinaten angeht - stets auf die höchste Auflösungsstufe des Schwarz-Weiß-Monitors.

Jede Grafik besteht aus einem Raster einzelner Punkte, die entweder gesetzt (schwarz) oder nicht gesetzt (hell) sind. Ein besonderer Effekt ist durch verschiedene Dichten möglich: Graustufen, die mit der Anzahl der gesetzten Punkte immer dunkler werden. Deshalb setzt sich eine Linie ebenso wie ein Drei- oder Viereck aus Punkten zusammen, die nur eng genug aneinandergereiht werden müssen.

### Punkte, Linien und Rechtecke

Ein Punkt wird an der Koordinate (X,Y) mit Befehl

```
DRAW X,Y
```

gesetzt. Als konsequente Folgerung aus obigen Überlegungen kann mit DRAW auch gleich eine Linie (bestehend aus einzelnen Punkten) gezogen werden:

```
DRAW X0,Y0 TO X1,Y1
```

Dabei ist zu beachten, daß DRAW die über

```
MODE =
LINE COLOR =
LINE STYLE =
```

eingestellten Werte heranzieht, um die Arbeit auszuführen. Möchte man ein Rechteck (Quadrat) auf dem Bildschirm darstellen, so können über DRAW die vier Eckpunkte verbunden werden. Einfacher geht es aber auf alle Fälle mit:

```
BOX X0,Y0 TO X1,Y1
```

und dessen Vetter:

```
BOX X,Y,Breite,Höhe
```

Beide Befehle zeichnen ein Rechteck (Quadrat, wenn alle vier Seiten die gleiche Länge besitzen) auf den Bildschirm. Im ersteren Fall werden zwei schräg (diagonal) gegenüber liegenden Ecken angegeben, im letzteren die linke obere des Rechtecks, sowie dessen Breite und Höhe. Dies entspricht (ausgedrückt in der ersten Syntax):

```
BOX X,Y TO X+Höhe,Y+Breite
```

Eine umschlossene Fläche kann auch mit einem Muster oder einer Farbe ausgefüllt werden. Dazu dient der Befehl:

```
FILL X,Y,Grenzfarbe
```

Die Wirkung kann man sich wie einen Farbeimer vorstellen, der an einer innerhalb der Begrenzung liegenden Stelle (X,Y) ausgegossen wird und die umrahmte Fläche füllt. Eine Lücke in der Begrenzung führt zu einem Zerstören des restlichen Bildschirminhaltes, da auch er ausgefüllt wird. Als Grenzfarbe wird der Wert -1 empfohlen, da GEM ansonsten nicht immer zufriedenstellende Ergebnisse liefert. Da der ausfüllende Algorithmus eine gewisse Zeit benötigt, bis er eine Fläche gefüllt hat, hat man einen Befehl implementiert, der Rechtecke gleich bei deren Darstellung mit einer Füllfarbe versieht:

```
PBOX X0,Y0 TO X1,Y1
PBOX X,Y,Breite,Höhe
```

Auch ein Rechteck mit abgerundeten Ecken (runde Ecken?) ist möglich:

```
RBOX X0,Y0 TO X1,Y1
RBOX X,Y,Breite,Höhe
```

und dessen gefülltes Pendant:

```
PRBOX X0,Y0 TO X1,Y1
PRBOX X,Y,Breite,Höhe
```

Das Füllmuster und die Füllfarbe legen die beiden Befehle

```
FILL STYLE =
FILL COLOR =
```

fest.

## 5.2 BITBLT

Dieser Befehl, der übrigens "Bit-Blit" gesprochen wird, kopiert Speicherbereiche, d.h. Bildschirmausschnitte. Folgende Varianten sind möglich:

1. Vom Bildschirm auf den Bildschirm (kopieren).
2. Vom Bildschirm in den Speicher.
3. Vom Speicher auf den Bildschirm.
4. Vom Speicher in den Speicher.

Fangen wir von oben an. Um einen Ausschnitt (Rechteck) von einer Bildschirmposition an eine andere Position zu kopieren, muß zuerst einmal der betreffende Bildschirmausschnitt festgelegt werden, dessen Inhalt an die neue Position kopiert werden soll:

```
BITBLT X0,Y0,Breite0,Höhe0 TO X1,Y1,Breite1,Höhe1
```

kopiert das Rechteck mit den Koordinaten der linken oberen Ecke (X0,Y0), der Breite0, sowie der Höhe0 an die hinter TO angegebene Position. Beide Rechtecke müssen dazu nicht unbedingt die gleiche Größe besitzen. Kopiert wird immer das kleinste Rechteck, das u.U. auch die Höhe 0 aber die Breite 1 haben kann. Zusätzlich kann noch ein Modus angegeben werden, der vorschreibt, wie ein Punkt im Quellrechteck (S) mit einem Punkt des Zielrechtecks (D) (logisch) verknüpft werden soll: BITBLT X0,Y0,B0,H0 TO X1,Y1,B1,H1,Modus

Folgende Modi sind möglich:

| Modus | resultierende Verknüpfung |
|---|---|
| 0 | =0 |
| 1 | S AND D |
| 2 | S AND (NOT D) |
| 3 | S (Standardmodus) replace |
| 4 | (NOT S) AND D |
| 5 | D |
| 6 | S XOR D |
| 7 | S OR D |

| Modus | resultierende Verknüpfung |
|---|---|
| 8 | NOT (S OR D) |
| 9 | NOT (S XOR D) |
| 10 | OT D |
| 11 | OR (NOT D) |
| 12 | OT S |
| 13 | NOT S) OR D |
| 14 | OT (S OR D) |
| 15 | 1 |

BITBLT kann aber nicht nur Bildschirmbereiche an eine andere Position kopieren, sondern diese auch vor Überschreiben in den Speicher des Computers retten. Dieser Befehl ist von enormer Bedeutung, gerade bei der im nächsten Kapitel folgenden GEM-Programmierung. Die durch Formulare und sonstigen Objekte überdeckten Flächen werden einfach vor dem Überschreibe-Vorgang in den Speicher gehieft. Dazu muß jedoch erst einmal Speicherplatz angefordert werden. Dies übernimmt die Funktion

```
Adresse = MEMORY(<Anzahl Byte>)
```

Es wird die angegebene Anzahl Byte reserviert. Als Ergebnis liefert die Funktion dann die Adresse, ab der der reservierte Bereich zu finden ist. Bliebe noch zu klären, wie ermittelt werden kann, wie viele Byte für einen BITBLT-Block im Speicher reserviert werden müssen. Ganz einfach, das berechnet sich nach der Formel:

```
Benötigte Byte = (Breite+15) SHR 4 * Höhe * 2 + 6
```

Jetzt kann der Block in den reservierten Bereich transferiert werden:

```
BITBLT X,Y,Breite,Höhe TO Adresse
```

Wird der Bildschirm in diesem Bereich durch irgendwelche Vorgänge zerstört (z.B. ein Fenster wird geöffnet und überschreibt dadurch den bisherigen Inhalt), kann der Bildschirm später wieder problemlos restauriert werden, indem man die Byte ab Adresse im Speicher wieder auf den Bildschirm kopiert:

```
BITBLT Adresse TO X,Y,Breite,Höhe [,Modus]
```

Zusätzlich kann diese Syntax dazu verwendet werden, im Speicher Grafiken zu erstellen und diese dann auf dem Bildschirm zu bringen. Dabei ist jedoch das von BITBLT verwendetete Format zu berücksichtigen, in dem Speicherbereiche abgelegt werden. Die ersten Worte ab Beginn des reservierten Bereichs enthalten nämlich drei Worte:

Anzahl der Planes*2 (Ebenen)
Breite des Ausschnitts in Pixel
Höhe des Ausschnitts in Pixel

Diese Daten müssen jedoch nur von Hand gesetzt werden, wenn eine Grafik manuell im Speicher aufgebaut wird. Ansonsten erledigt diese Arbeit der BITBLT-Befehl für Sie, der die Daten in den Speicher manövriert. Mit der letzten Syntaxvariation kann auch vom Speicher in den Speicher kopiert werden. Der Sinn der ganzen Angelegenheit liegt darin, Bilder oder Bildschirmteile, die im HIRES-Modus, also in Schwarz-Weiß erstellt worden sind, in einen Farbbildschirm zu kopieren:

```
BITBLT Bereich1 TO Bereich2,COLOR Farbe
```

Die Farbe,die als Parameter anzugeben ist, ist die Farbe, die ein gesetzter Bildschirmpunkt im Farbbildschirm erhalten soll.

## 5.3 Bildschirm laden und speichern

Wie Sie seit dem letzten Kapitel wissen, kann ein bestimmter Bildschirmbereich oder auch der gesamte Bildschirminhalt in den Speicher kopiert, und dort archiviert werden. Zumindest bis zum Ausschalten des Computers. Dann ist er leider verloren. Ab zu und kommt es jedoch vor, daß man den Bildschirminhalt auf

Diskette abspeichern und ihn zu einem späteren Zeitpunkt wieder laden möchte. Dies kann z.B. sein, weil man gerade eine tolle Grafik auf dem Monitor erstellt hat, die man ganz gerne vor der Vernichtung bewahren möchte.

Auf der anderen Seite kommt es natürlich auch vor, daß ein mit einem Malprogramm erstelltes Bild als Vorspann in ein eigenes Programm integriert werden soll. Dazu speichert man den Bildschirm als 32000-Byte (also nicht komprimiert) ab, und lädt ihn mit einem entsprechenden Befehl am Programmstart ein. Es ist ganz einfach, den aktuellen Bildschirminhalt auf Diskette abzuspeichern:

```
BSAVE "Dateiname"
```

speichert den Bildschirm (32000 Byte) unter Dateiname auf Diskette ab. Möchte man ihn wieder einladen, genügt der Befehl:

```
BLOAD "Dateiname"
```

und die in der Datei Dateiname abgelegte Grafik (Bildschirminhalt) wird wieder geladen. Übrigens: Das Zeichenprogramm DOODLE speichert seine Bilder in unkomprimierter Form ab, so daß Bilder, die mit DOODLE abgespeichert wurden, einfach mit BLOAD eingeladen werden können. Jetzt darf ich Ihnen natürlich nicht verschweigen, daß bei BLOAD und BSAVE noch zusätzliche Parameter angegeben werden können, die dann erweiterte Möglichkeiten bieten:

```
BLOAD "Dateiname",<Adresse>
```

lädt die Datei Dateiname an die angegebene Adresse. Der dadurch überschriebene Speicherplatz wurde hoffentlich zuerst mit

```
MEMORY
```

reserviert, damit kein Unglück geschieht. Auch BSAVE kann die ab einer gewissen Adresse folgenden Daten in eine Datei schreiben. Dann muß jedoch noch angegeben werden, wie viele Byte abgespeichert werden sollen:

```
BSAVE "Dateiname",<Adresse>,<Anzahl>
```

Mit BLOAD und BSAVE ist es möglich, Maschinenprogramme in den Speicher an eine bestimme Adresse zu laden., man kann Variableninhalte auf Diskette speichern, indem man gleich den ganzen Speicherbereich auf Diskette auslagert usw.

## 5.4 Objekte verschieben und drehen

Gehen wir bei den folgenden Überlegungen einmal von einem Punkt aus, da es sich hierbei um die einfachte Art eines grafischen Objekts handelt. Statt des einzelnen Punktes können Sie natürlich auch die Eckpunkte eines Rechtecks oder den Mittelpunkt eines Kreises für folgenden Operationen heranziehen.

### Objekt verschieben

Da jeder Punkt durch seine Koordinaten (X,Y) genau bestimmt ist, kann er durch einfache Addition bzw. Subtraktion verschoben werden (vgl. Abbildung 5.1).

In einer Formel ausgedrückt lautet das:

```
Xneu = Xalt + Weite
Yneu = Yalt + Weite
```

Nach der Verschiebung (Translation) besitzt der Punkt (Xalt,Yalt) die Koordinaten (Xneu,Yneu). Man kann dieses Verschieben nun auf die Spitze treiben und das Objekt erst einmal zum Ursprung hin verschieben, ehe es im zweiten Schritt dann an seine neue Position gesetzt wird. Somit dürfte es kei-

nerlei Schwierigkeiten mehr machen, ein Rechteck mit dem Mauszeiger an eine neue Position zu verschieben oder eine sonstige Translation durchzuführen.

### Objekt drehen

Um ein Objekt drehen zu können (Rotation), werden trigonometrische Funktionen eingesetzt. Während wir bei der Verschiebung einen bestimmten Punkt mittels Addition an seine neue Position verfrachtet haben, muß er bei der Rotation um einen bestimmten Winkel gedreht werden (vgl. Abbildung 5.2).

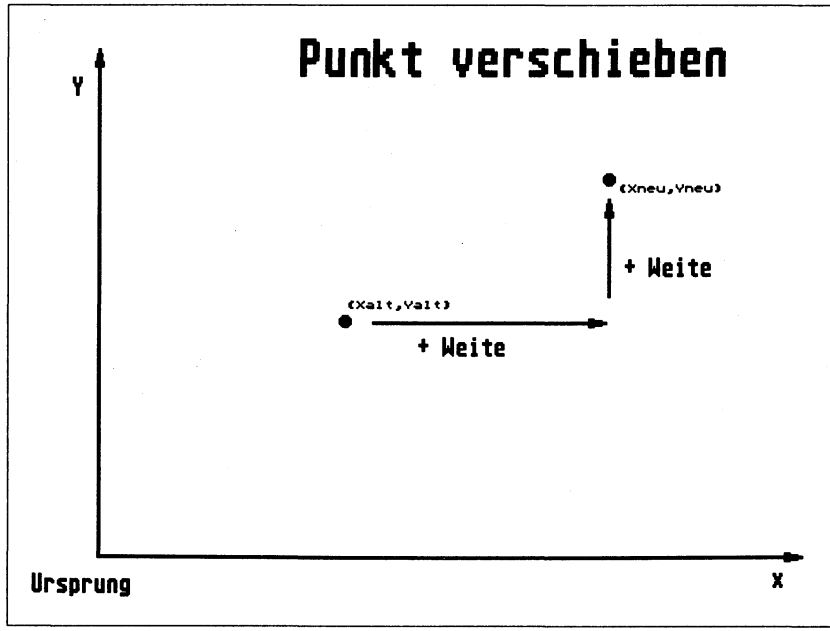



*Abb. 5.1: Punkt verschieben*

Ich möchte Ihnen und mir die mathematische Herleitung der nun folgenden Rotationsgleichung ersparen. Deshalb hier einfach die benötigte Formel:

```
Xneu = Xalt * cos(ß) - Yalt * sin(ß)
Yneu = Xalt * sin(ß) + Yalt * cos(ß)
```

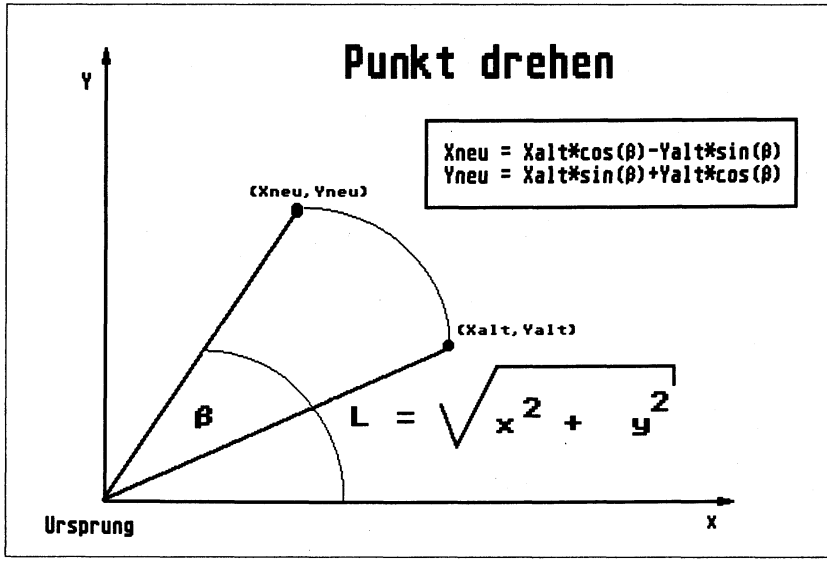



*Abb. 5.2: Punkt drehen*

# 6. GEM

GEM, der Graphics Environment Manager im Atari ST, stellt eine Vielzahl verschiedener Routinen zur Verfügung, die von einfachen Grafikbefehlen (im VDI) bis zur komfortablen Arbeit mit Ressourcen (RCS) reichen. Solche Resourcen sind Dateien, in denen verschiedene Objekte enthalten sind, die auf dem Bildschirm sichtbar gemacht werden können. GEM übernimmt dann die vollständige Kontrolle über ein solches Formular (bestehend aus mehreren Objekten) und gestattet das Selektieren einzelner Objekte (z.B. Boxen) mit der Maus oder die Eingabe von Text in das Formular. Erst wenn ein bestimmtes Abbruchkriterium erfüllt ist (z.B. Abbruch angeklickt), erhält das eigentliche Programm die Kontrolle zurück. Von dort aus läßt sich dann überprüfen, welche Eingaben der Benutzer im Formular getätigt hat.

Damit GEM die Kontrolle über ein solches Formular übernehmen kann, müssen bestimmte Konventionen eingehalten werden. Der Computer muß wissen, in welcher Beziehung die einzelnen Objekte zueinander stehen oder wie die einzelnen Objekte bzw. das aus diesen resultierende Formular auszusehen hat. Deshalb besitzt jedes Element einen 24 Byte großen Eintrag in einer sogenannten Objektliste, in dem das letzte bzw. nächste Objekt, die Koordinaten (X,Y bzw. Breite und Höhe) des Objekts, sowie andere Informationen festgehalten werden. Diese Objektliste wiederum kann als Datei auf Diskette ausgelagert und bei Bedarf (zum Programmstart) wieder eingelesen werden. Da es relativ mühsam und vor allem äußerst umständlich ist, solche Objektlisten zu erstellen, kann man diese Aufgabe auch von einem Ressource Construction Set erledigen lassen. Um all diese Dinge wird es in diesem Kapitel gehen.

## 6.1 Arbeiten mit dem RCS

Die einzelnen Objekte innerhalb eines Formulars werden in einer Objektliste zusammengefaßt, wobei jedes Objekt einen Eintrag von 24 Byte Größe in Anspruch nimmt, in dem sämtlichen Informationen über dieses Objekt enthalten sind. Diese Objektliste wird als Baumstruktur im Speicher verwaltet.

Informatiker nennen diese Listenform deshalb Baum, da sie sich - ähnlich einem (echten) Baum - in verschiedene Ebenen untergliedern läßt, nämlich in eine Wurzel (Root), in Äste, sowie als deren Ausläufer in Zweige. Ein Beispiel macht dies sehr schnell deutlich:

Stellen Sie sich einmal eine große Box vor, die zwei kleinere Boxen enthält. In der rechten der beiden kleineren Boxen, sei eine weitere Box enthalten. Dieses Gebilde besteht aus drei verschiedenen Ebenen:

- Große Box (erste Ebene),
- die beiden kleineren Boxen (zweite Ebene),.
- die Box, in der kleinen Box (dritte Ebene).

Die erste Ebene bezeichnet man als Wurzel, da auf ihr alle weitere Ebenen aufbauen, sie ist gleichsam der Grundstock für das gesamte Formular. Als Äste sind die beiden kleineren Boxen anzusehen, die sich innerhalb der großen Box befinden, und in der gleichen Ebene liegen. Diese beiden Objekte werden auch als Kinder (Children) bezüglich der großen Box der ersten Ebene, die sinnigerweise dann Eltern- (Parent) Objekt genannt wird.

Wie lassen sich derartige Familienverhältnisse nun ermittel? Ganz einfach! Ein Objekt, das sich innerhalb eines anderen Objekts befindet, dessen Ränder also nicht überschreitet, ist diesem (Eltern-)Objekt untergeordnet. Verschiebt man ein solches Elternobjekt, führt dies logischerweise auch zu einer Verschiebung des Kindes! Der ganze Sachverhalt noch einmal als Picture:

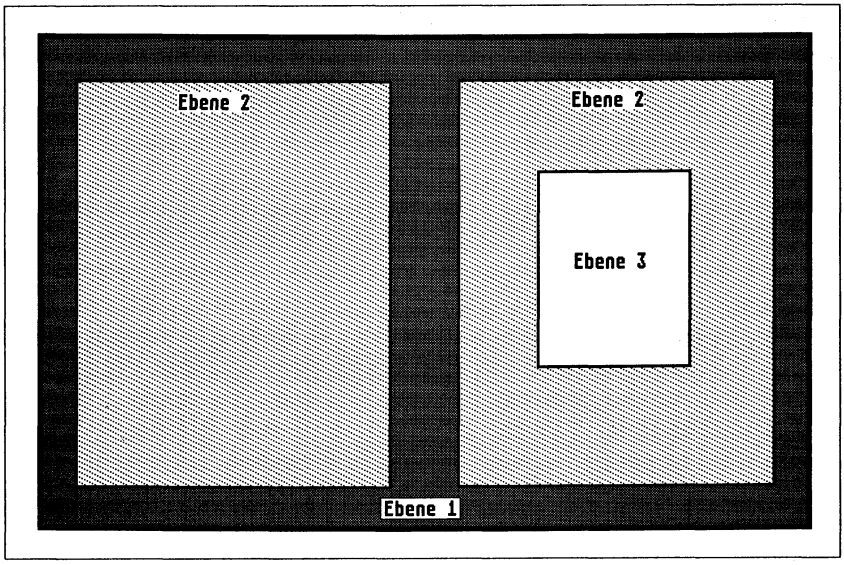



*Abb. 6.1: Ein Objektbaum mit drei Ebenen*

Einen solchen Objektbaum selbst durchzurechnen, ist wirklich der reinste Horror! Aber mit einem Resource Construction Set, das quasi einen Werkzeugkasten für geplagte GEM-Programmierer darstellt, wird das Erstellen von Formularen zu einem echten Kinderspiel. Ein solches Programm sollten Sie sich also auf alle Fälle besorgen, wenn Sie Ihre Programme mit GEM-Formularen schmücken möchten. Das RCS von Digital Research aus dem Entwicklungspaket gib's fast umsonst. Nerven Sie diesbezüglich einmal Ihren Atari-Fachhändler!

Andererseits wissen auch die Firmen um die Nöte beim Erstellen von Resourcen, und legen deshalb manchen Programmiersprachen gleich ein RCS bei. So können Sie selbstverständlich auch das RCS des Megamax-C-Compilers "mißbrauchen". Ich persönlich - nein, jetzt kommt keine Schleichwerbung - benutze bei meiner täglichen Arbeit verschiedene RCS, die gegeneinander ausgespielt werden können. Auf diese Weise können dann Objektbäume konstruiert werden, die es eigentlich gar nicht geben kann!

Für Besitzer eines MEGA-ST ist das RCS von Digital Research in der alten Version (1.xx) übrigens nicht geeignet, da es mit Blitter-TOS nicht zusammenarbeitet (oder das Blitter-TOS nicht mit dem Programm, das Ergebnis bleibt gleich). Abhilfe kann hier nur das Booten der älteren TOS-Version von Diskette schaffen.

Nun aber zum Umgang mit dem RCS von Digital Research, da es wohl das preiswerteste und deshalb am weitesten verbreitete Programm zur Resourceerstellung sein dürfte. Nach dem Laden des Programm finden Sie in der ersten Zeile die Menüpunkte DESK, FILE, OPTIONS und GLOBAL. Am rechten Rand befinden sich das Klemmbrett (Clipboard) zum Zwischenspeichern und der Mülleimer (Trash) zum Löschen von Objekten. Diese beiden Piktogramme werden im Fachchinesisch auch Icons genannt. Ferner erscheinen auf dem Monitor noch zwei Fenster. Das obere, längliche Fenster enthält die verschiedenen Baumarten, die mittels des RCS konstruiert werden können, und sollen deshalb ab sofort Objektfenster genannt werden. Das untere Window (Fenster) dient der Bearbeitung eines Formulars, deshalb soll es Arbeitsfenster heißen. Um ein bestimmten Objekt innerhalb des Arbeitsfensters verändern zu können, muß es erst einmal mit der Maus aus dem Objektfenster transportiert werden. Klicken Sie dazu das entsprechende Symbol an (z.B. FREE), und ziehen bei gedrückter linker Maustaste das Piktogramm in das Arbeitsfenster. Dieser Vorgang heißt übrigens Draggen.

Jetzt öffnet sich eine Box, mittels der Sie dem Objekt einen (neuen) Namen verleihen können. (Tree1 ist absolut nichtssagend!) Wozu der Aufwand mit dem Benennen eines Objekts?

Jedes Objekt trägt innerhalb der Objektliste eine Nummer, anhand der es eindeutig identifiziert werden kann. Um nun ein Objekt aus dem Programm heraus ansprechen bzw. abfragen zu können, welches Objekt der Benutzer selektiert (angeklickt) hat (der Computer kann Ihnen ja nicht mitteilen, daß das kleine schraffierte und schattierte Rechteck, das sich in der linken oberen Hälfte des Wurzelobjekts befindet, angeklickt wurde),

muß sich der Computer einer Variante bedienen, die für ihn im Bereich des Möglichen liegt; der Übergabe einer Zahl (Nummer), die das betreffende Objekt kennzeichnet.

Wird ein Objektbaum nachträglich bearbeitet, kann es vorkommen, daß sich dabei auch die Eltern-Kind-Beziehungen ändern, oder andersherum ausgedrückt, daß Objekte neue Nummern zur Identifikation zugewiesen bekommen. Doch dies kann zu ernsthaften Problemen führen. Möchten Sie nämlich überprüfen, ob beispielsweise Objekt Nummer 2 angeklickt wurde, und Sie bewerkstelligen dies mit einer IF...THEN-Abfrage (Ret% enthalte die Nummer des selektierten Objekts)

```
IF Ret% = 2 THEN
   ......
   <hier folgt die Reaktion seitens des Programms>
   ......
ENDIF
```

geht das solange gut, wie das betreffende Objekt keine neue Nummer zugewiesen bekommt. Aber wehe, Sie ändern den Objektbaum ab! Dann wird ihr schönes Programm nichts anderes als Makulatur, da die Nummern in den IF-THEN-Abfragen nicht mehr mit den neuen Nummern im Baum übereinstimmen. Als einziger Ausweg bleibt Ihnen das umständliche Ändern der Abfragen im Programmtext. Viel Spaß!

In der Programmiersprache C - und GEM ist auch ein Produkt dieser Sprache! - gibt es eine praktische Einrichtung, den sogenannten Präprozessor. Dieser durchsucht - ehe der Compiler das Programm zu Gesicht bekommt - den gesamten Quelltext und ersetzt bestimmte Ausdrücke durch zuvor definierte Anweisungen. Die Präprozessor-Anweisung

```
#define Tree1 0
```

bewirkt ein Ersetzen innerhalb des Programmtextes von Tree1 durch eine Null 0. Überall, wo der Präprozessor auf die Zeichenkette Tree1 stößt, entfernt er diese und plaziert statt dessen den neuen Text (die Null) in das Programm. Das RCS wiederum - und deshalb erzähle ich Ihnen das Ganze - nutzt diese Eigen-

schaft des Preprozessor und generiert ein HEADER-File, das mit dem Extender .h geschmückt ist. Diese Datei enthält - raten Sie mal! - Präprozessor-Anweisungen in Form von #define. Hinter jedem #define steht der Name des Objekts und die dazugehörende Objektnummer. Maßgeschneidert für C-Programmierer! Diese können nämlich dann einfach den Namen, der dem Objekt verpaßt wurde, im Programm verwenden, um ein bestimmtes Objekt anzusprechen. Und genau diesen Namen - damit schließt sich der Kreis wieder - können Sie in obiger Dialogbox des RCS angeben.

Doch eine derart komfortable Einrichtung wie einen Preprozessor haben wir in BASIC leider nicht zur Verfügung. Also muß man sich anderweitig helfen. Man weist die Objektnummer einfach einer Variablen zu, und den Variablennamen gibt man in der Dialogbox des RCS an.

Sollten Sie also mit einem RCS arbeiten, das eine Datei für einen C-Präprozessor generiert, muß diese Datei entsprechend umgewandelt und in das Programm eingebaut werden. (Möglichst an den Anfang, damit die Variablen gleich am Programmstart mit den korrekten Werten versorgt werden und nicht das gesamte Formular aus Wurzelobjekten besteht):

```
#define Tree1 55
```

wird in BASIC zu

```
Tree1 = 55
```

Ab sofort kann dann das Objekt über den Variablenaufruf Tree1 angesprochen werden, da dort die Objektnummer gespeichert ist. Für alle diejenigen, die zufällig ein RCS besitzen, das die HEADER-Dateien nicht nur in eine für C, sondern auch für GfA-BASIC verständliche Form bringen können (ein solches RCS liegt dem GfA-BASIC 3.00 bei, aber auch Application Systems Heidelberg hat dem Megamax C-Compiler (?!?) ein RCS spendiert, das diese Ausgabevariante bietet), sei das Einbinden der Datei in das Programm über LOAD Block *.* in der Menüleiste

empfohlen. Der Extender im Pfad muß dann allerdings in .LST abgeändert werden, damit die Datei in der File-Selektor-Box des ST-BASIC angezeigt wird.

Verpassen Sie also dem Objekt einen (sinnvollen) Namen und drücken Return, oder klicken auf OK. Jetzt kann das entsprechende Objekt im Arbeitsfenster weiter bearbeitet werden. Im Obektfenster stehen folgende Objekte zur weiteren Verarbeitung zur Verfügung (von links nach rechts):

### Unknown (Fragezeichen)

Bei diesem Objekt handelt es sich um keine richtige Baumart, sondern vielmehr um eine unbekannte Objektstruktur! Diese tritt immer dann auf, wenn das zu einer RCS-Datei gehörende DEF-File, in dem die Arten der Objekte festgelegt sind, vom RCS nicht geladen werden kann. In diesem Fall wird das Objekt als unbekannt tituliert und muß vor einer weiteren Bearbeitung erst einmal in eine andere Art umbenannt werden.

### Free

Dieses Objekt stellt den Grundbaustein für Formulare aller Art dar. Die einzelnen Kinder dieses Objekts sind dabei in ihrer Positionierung frei wählbar. Dies hat den Nachteil, daß Baumstrukturen, die in hoher Auflösung erstellt wurden, bei einer anderen Auflösungsstufe vollkommen verschoben aussehen können. Möchten Sie, daß die Objekte in ein bestimmtes Raster positioniert werden, so müssen Sie

### Dialog

wählen. Das dabei verwendete Raster ist von der augenblicklich benutzten Grafikauflösung abhängig. Bei hoher Auflösung werden die Objekte so plaziert, daß die X-Koordinate durch acht und die Y-Koordinate durch sechzehn teilbar ist.

### Menu

Die Baumart wird zur Konstruktion der Pull-Down-Menüs benutzt, die vom oberen Bildschirmrand herunterklappen.

### Alertbox

Mit dieser Objekt lassen sich Alertboxen (Alarmboxen) kreieren, die Warnmeldungen oder Rückfragen ausgeben können (vgl. auch FORM_ALERT in ST-BASIC).

Möchten Sie jetzt ein eigenes Objekt erstellen, draggen Sie zuerst einmal die gewünschte Baumart vom Objekt- in das Arbeitsfenster. Sobald dies geschehen ist, stehen im Objektfenster die einzelnen Objekte zur Verfügung, die in dieser Baumart benutzt werden können. Auch diese müssen wieder in das Parent-Objekt gezogen werden. Dort können Sie dann vergrößert (linke untere Ecke anklicken, Maustaste gedrückt halten und Maus verschieben) bzw. verkleinert werden. Aber auch ein Neupositionieren ist möglich, indem Sie den Mauszeiger inmitten des Objekts fahren, die Maustaste drücken und das Objekt verschieben. Vorsicht! Befindet sich das betreffende Objekt innerhalb der Umrandung eines anderen, so ändert sich die gesamte Baumstruktur, wenn der Rahmen verlassen wird! Innerhalb des Vater-Objekts kann es dagegen problemlos hin- und hergeschoben werden. Nun zu den einzelnen Objektarten, die benutzt werden können:

### Button (Taste)

Dieses Objekt funktioniert wie eine Taste. Es kann mit der Maus angeklickt bzw. unter bestimmten Umständen auch via <Return>-Taste selektiert werden.

### String (Zeichernfeld)

Dieses Objekt kann einen Text speichern, und sollte benutzt werden, wenn man Text in einer Dialogbox benutzen will (z.B. eine Überschrift).

```
Ftext (EDIT: _____)
```

Dieser Text kann später (vom Benutzer) editiert werden.

### Fboxtext

Wie Ftext, jedoch wird der Text in einer Box präsentiert, die bei Bedarf auch ein Füllmuster enthalten kann.

### Ibox

Diese Box wird als Vaterobjekt für andere Objekte benutzt, die dann in einer eigenen Ebene (innerhalb der Ibox) liegen.

### Box

Box verleiht anderen Objekten einen entsprechenden Rahmen, wobei die Box auch mit einem Muster ausgefüllt werden kann.

### Text

Es handelt sich um einen Text, der im Gegensatz zu Ftext jedoch nicht editierbar ist.

### Boxchar

Dieses Objekt beinhaltet einen Buchstaben, und kann mit der Maus (oder mit <Return>) selektiert werden. Gerade für die Symbole <Pfeil hoch> oder <Pfeil runter> ist diese Objektform geradezu prädestiniert.

### Boxtext

Jetzt dürfen Sie dreimal raten, was dieses Objekt darstellt! Richtig, einen Text, der von einer Box umschlossen wird.

### Icon

Icons - über diesen Namen sind wir schon einmal gestolpert - werden beispielsweise auf dem Desktop dargestellt, wo sie die Gestalt einer Diskettenstation oder eines Abfalleimers annehmen. Aber auch im RCS selbst finden sie sich: als Clipboard (Klemmbrett) und Abfalleimer.

### Image

Wird zur optischen Gestaltung von Bäumen benutzt und kann im Gegensatz zu Icons nicht selektiert werden!

### Title (Menütitel)

Diese Objektart ist nur innerhalb eines Menübaumes zugelassen und erscheint deshalb auch nur im Objektfenster, wenn ein Menübaum editiert wird. Ehe eine weitere Baumstruktur bearbeitet werden kann, muß das Arbeitsfenster erst wieder geschlossen werden. Dazu klicken Sie in die linke obere Ecke des Arbeitsfenster, das Schließfeld (Closer) einmal kurz und kräftig an. Im Objektfenster erscheinen nun wieder die einzelnen Baumarten.

Auch das Duplizieren einzelner Objekte ist möglich. Drücken Sie dazu die Shift-Taste, ehe Sie das gewünschte Objekt im Arbeitsfenster anklicken. Mit gedrückter Maustaste ziehen Sie eine Kopie des entsprechenden Objekts. Die einzelnen Objekte können auch noch anderweitig manipuliert werden, indem Sie durch Anklicken geöffnet und ihre Objektattribute abgeändert werden. Die einzelnen Attribute und deren Wirkungsweisen sollen an dieser Stelle jedoch nicht weiter besprochen werden, da Sie im nächsten Kapitel, wenn es um die interne Struktur der Resourcen geht, noch einmal ausführlichst behandelt werden.

Mit diesem Wissen ausgestattet dürfte es eigentlich keinerlei Probleme mehr machen, eigene Resourcen mit dem RCS zu erstellen. Spielen Sie ein bißchen mit dem RCS herum. Auch andere Programme (z.B. K-Resource) arbeiten ähnlich. Es ist jedoch unmöglich, alle RCS, die auf dem Markt erhältlich sind, ausführlich zu beschreiben.

Im Lieferumfang des ST-BASIC bzw. des Omikron.BASIC befindet sich übrigens auch ein RCS, das allerdings (zumindest auf meiner Diskette) immer noch nicht vollständig fertiggestellt und deshalb mehr oder weniger unbrauchbar ist (eher mehr!).

## 6.2 GEM-Programmierung unter ST-BASIC

In diesem Kapiel - das kann ich Ihnen schon jetzt versprechen, wird es ziemlich rund gehen! Aber lassen Sie sich davon ja nicht abschrecken, denn hat man sich erst einmal an diese Art der Programmierung gewöhnt, möchte man sie um nichts in der Welt mehr vermissen (ehrlich!).

### Application anmelden

Die Vorgehensweise ist immer die gleiche: Zuerst muß einmal das GEM - die grafische Benutzeroberfläche des TOS - aktiviert werden. Dies wäre im Prinzip relativ aufwendig, gäbe es nicht eine Library auf der ST-BASIC-Diskette, die die Programmierung unter GEM stark vereinfacht. Möchten Sie also unter GEM programmieren, muß zuerst einmal die Datei

```
GEMLIB.BAS
```

geladen werden. Jetzt steht Ihnen eine Fülle von Befehlen zum GEM-Handling zur Verfügung. Der erste Befehl, mit dem eine Application angemeldet wird lautet:

```
Appl_Init
```

Mit diesem Befehl wird GEM aktiviert. Ehe das Programm wieder seine Arbeit beendet, muß GEM wieder verlassen werden,

möchten Sie nicht (unangenehme) Systemabstürze riskieren. Auch dazu stellt die Library GEMLIB.BAS einen Befehl zur Verfügung:

```
Appl_Exit
```

Nachdem GEM aktiviert wurde, kann es losgehen:

### GEM aktivieren

1. Schritt: RCS-Datei von Diskette laden
2. Schritt: Adresse des Baumes besorgen
3. Schritt: Form_Dial aufrufen (Speicher retten)
4. Schritt: Formular auf dem Bildschirm darstellen
5. Schritt: Kontrolle über das Formular an GEM übergeben.
6. Schritt: Form_Dial aufrufen (Speicher freigeben)

### GEM deaktivieren

Dieses "Schema F" sollten Sie sich genau einprägen, da eine Programmierung von Dialogboxen stets in dieser Reihenfolge abläuft.

Als Programmierprojekt habe ich mir in diesem Kapitel zum Ziel gesetzt, eine eigene File-Selector-Box zu erstellen, die Sie dann auch in Ihre eigenen Programme einbinden können. Dazu muß natürlich erst einmal eine entsprechendes Resource-Datei mit Hilfe eines RCS erstellt werden. Also ran, schnappen Sie sich ein Resource Construction Set, und basteln die Box zusammen (vgl. Abbildung 6.2).

Zuerst müssen wir uns einmal im klaren sein, welche Baumstruktur für die File-Selector-Box verwendet werden soll. Dialog drängt sich förmlich auf, aber auch Free ist bestimmt nicht verkehrt. Draggen Sie also das Icon Dialog aus dem Objektfenster in das Arbeitsfenster, und verpassen Sie ihm anschließend gleich den sinnigen Namen FILSEL. Damit ein Name eingegeben wer-

den kann, muß die Shift-Taste bei gleichzeitiger Betätigung der Buchstaben-Taste gedrückt werden. Das Programm mag bei der Eingabe von Resource-Namen eben nur Großbuchstaben.

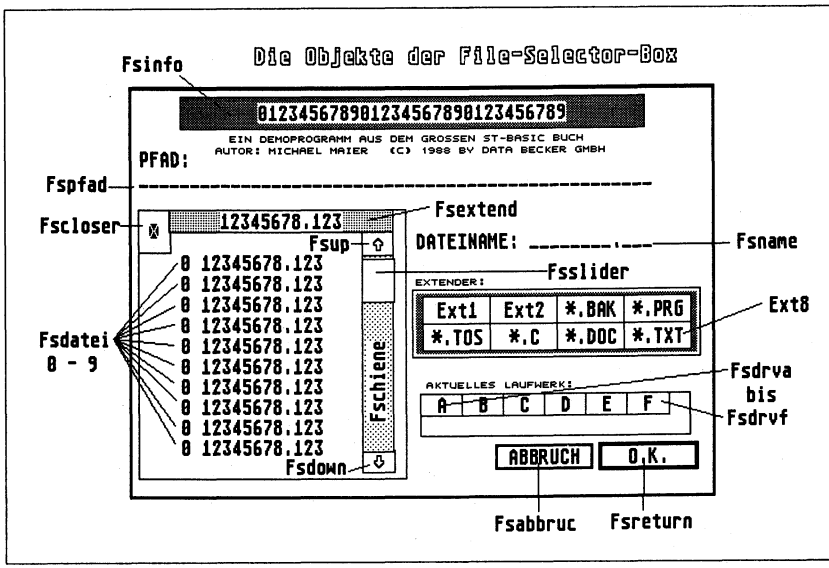



*Abb. 6.2: Die einzelnen Objekte der Dialogbox*

Somit ist unsere Dialogbox schon einmal geöffnet. Jetzt müssen die einzelnen Objekte eingetragen werden. Fangen wir dazu ganz oben an: Im oberen Boxbereich soll die Infoanzeige erfolgen. Als Objekttyp benutzen wir BOXTEXT, und taufen unseren Schützling in Anlehnung an seine Funktion Fsinfo. Sobald sich das Objekt in der Dialogbox befindet, richten Sie es nach Abbildung 6.2 aus. Mit einem Doppelklick auf das Objekt kann eine Dialogbox des Resource Construction Set geöffnet werden, in der die einzelnen Flags und Parameter für dieses Objekt gesetzt werden können. Für die Infoanzeige werden keinerlei Flags benötigt. Trotzdem muß unter Justify (Ausrichtung des Textes innerhalb der Box) das Feld C angeklickt werden. Dadurch erreichen Sie, daß der Text genau in die Mitte zwischen den linken und rechten Rand des Objekts gesetzt wird. In diesem Zusammenhang spricht man auch von zentrieren oder englisch

center, woher auch die Abkürzung C in der Dialogbox herrührt. Mit Border kann dann die Stärke der Umrahmung eingestellt werden, Background sorgt für den richtigen Hintergrund. Sie können selbstverständlich ein beliebiges Füllmuster verwenden, oder auch den weißen Hintergrund belassen.

In der letzten Zeile muß dann hinter PTEXT noch der Text eingetragen werden, der in der Box erscheinen soll. Nun werden Sie natürlich entgegnen, daß Sie im Augenblick noch gar keine Vorstellung darüber haben, welcher Text in diesem Objekt erscheinen soll bzw. der Text ja je nach Bedarf umgeändert werden soll. (Datei laden, Datei Speichern, usw.). Wozu also bereits jetzt einen Text in die Resource-Datei eintragen? Ganz einfach: Der jetzt eingetragene Text dient lediglich als Platzhalter, er wird dann vor dem Aufruf durch den aktuellen Text ersetzt. Dies mag zwar paradox klingen, hat aber einen tieferen Sinn: versucht man nämlich, einen Text in ein Objekt zu plazieren, der länger als der in diesem Objekt reservierte Platz ist, werden dadurch unweigerlich andere Daten überschrieben (schütten Sie einmal zwei Liter Wasser in einen Topf, der nur einen Liter Fassungsvermögen besitzt). Und das führt im Normalfall zu einem Systemabsturz. Um dem vorzubeugen, trägt man einfach einen genügend langen Text als Füllstring in ein Objekt, und schon ist die Gefahr gebannt. Tragen Sie also einen Füllstring hinter PTEXT ein. Mein Vorschlag wäre, entweder 30 mal ein beliebiger Buchstabe, oder - gleich zum Mitzählen - die Ziffern 0 bis 9. Ein Klick auf OK oder eine Betätigung der Return-Taste sorgt dafür, daß die Dialogbox wieder verlassen werden kann.

Jetzt ist das Objekt Pfad an der Reihe. Es hat keine weitergehenden Funktionen und steht quasi nur zur Zierde in der Dialogbox. Draggen Sie dazu das Objekt Text in das Arbeitsfenster und ändern den Text (Objekt zweimal anklicken) in PFAD um. Wichtiger ist die Eingabezeile für den Pfadnamen, der als Objektnamen Fspfad tragen soll. Er besteht aus einem Objekt

Ftext, das als Flag Editable spendiert bekommt. Hinter Text ist 50 mal ein Zeichen als Platzhalter für den Pfadnamen einzugeben, hinter Template 50 mal der Buchstabe P.

Danach geht's ans Konstruieren des Fensters, in dem die einzelnen Dateinamen angezeigt werden sollen. Dazu nehmen Sie sich erst mal ein Objekt Box und ziehen dieses in das Arbeitsfenster. Anschließend fahren Sie den Mauspfeil zur rechten unteren Ecke der Box, drücken die linke Maustaste und vergrößern das Objekt nach Ihrem Geschmack (in Abbildung 6.2 ist dieses Objekt etwas größer gezeichnet). Anschließend sind folgende Objekte als Kinder dieser Box einzutragen.

| Art | Name | Flags | Sonstiges |
|---|---|---|---|
| Boxchar | Fscloser | Touchexit | Als Buchstabe hinter CHAR den 'Closer' (Control E) |
| Boxtext | Fsextend | Touchexit | Justify: Center<br>Background: gepunktet<br>PTEXT: 12345678.123 |
| Boxchar | Fsup | Touchexit | CHAR: Pfeil nach oben (Control A) |
| Boxchar | Fsdown | Touchexit | CHAR: Pfeil nach unten (Control B) |
| Box | Fschiene | Touchexit | Background: gepunktet, reicht von 'Fsup' bis 'Fsdown' |
| Box | Fsslider | Touchexit | Kind von 'Fschiene' Die Größe stellt das Programm ein |
| Text | Fsdatei0<br>bis<br>Fsdatei9 | Selectable<br><br>Touchexit | PTEXT: '0 12345678.123'<br>Radio Button |

Sind alle Objekte richtig plaziert, bringen Sie das Vaterobjekt auf die richtige Größe, so daß es eine Umrahmung für sämtliche darin enthaltenen Objekte darstellt. Somit wäre der schwierigste Teil bereits geschafft. Die übrigen Objekte sind reine Formsache.

In der nachfolgenden Abbildung ist der schrittweise Aufbau der File-Selector-Box verdeutlicht:

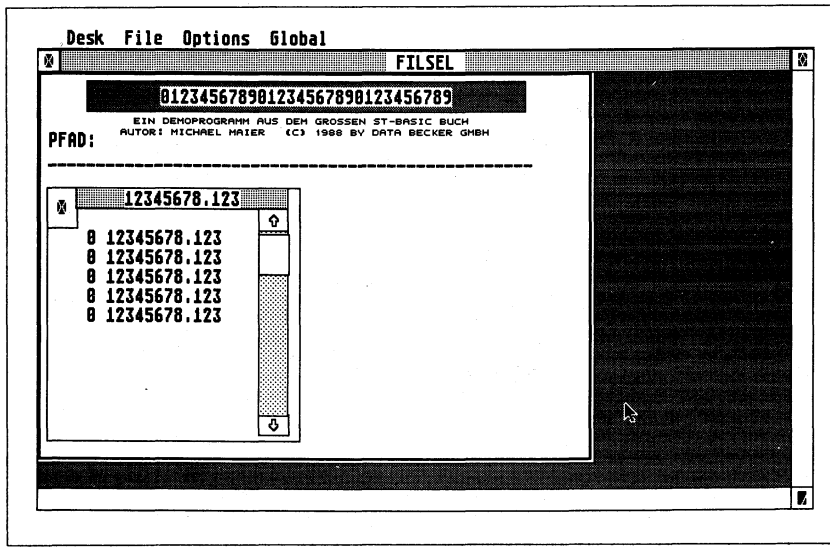



*Abb. 6.3: Die File-Selector-Box nimmt Gestalt an*

| Art | Name | Flags | Sonstiges |
|---|---|---|---|
| Text | --- | --- | PTEXT: Dateiname: |
| Ftext | Fsname | Editable | TEXT: 12345678.123<br>Template: 'pppppppp.ppp' |
| Box | --- | --- | umschließt die einzelnen Knöpfe zur Auswahl des Laufwerkes. Normal nicht sichtbar, Border Color: 0 |
| Boxchar | Fsdrva<br>bis<br>Fsdrvf | Selectable | CHAR: A bis F<br>Radio Button<br>Kinder von obiger Box! |
| Box | --- | --- | umschließt die einzelnen Knöpfe zur Auswahl eines Extenders |
| Boxtext | Ext1<br>bis<br>Ext8 | Selectable<br>Touchexit | Enthält die einzelnen Extender<br>Radio Button<br>letztes Objekt muß unbedingt 5 Zeichen lang sein, da es vom Programm aus geändert wird. |
| Button | Fsabbruc | Selectable<br>Exit | Ptext: Abbruch |
| Button | Fsreturn<br>Exit | Selectable | Ptext: OK |

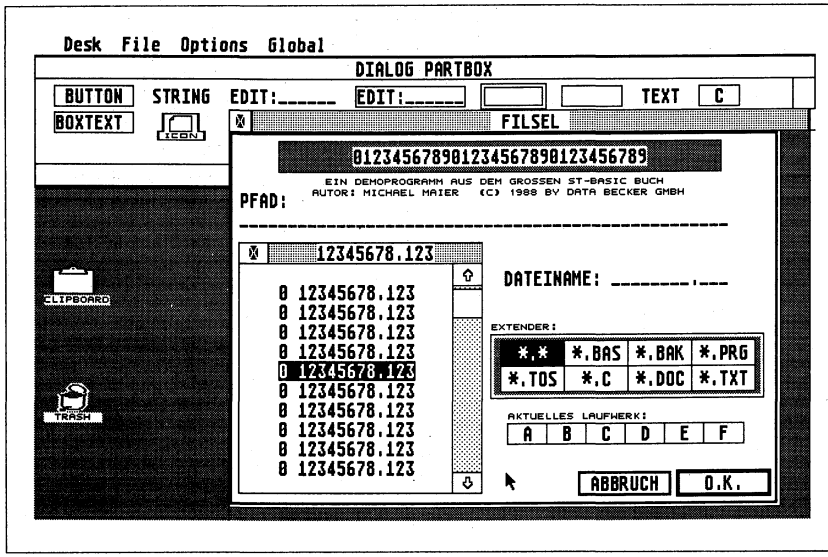



*Abb. 6.4: Die fertige File-Selector-Box*

Ist die Dialogbox fertig erstellt, so speichern Sie diese auf Diskette ab, und binden anschließend die vom Resource erstellte HEADER-Datei in das Programm ein. Wenn Sie die in obiger Tabelle angegebenen Objektnamen übernommen haben, so genügt ein Ausbessern der verschiedenen Objektnummern im Listing zur File-Selector-Box, falls einzelne Nummern abweichen sollten.

### Aufbau eines Objektbaumes

Gehen wir für unsere weiteren Überlegungen einmal davon aus, daß die Resourcedatei bereits geladen ist und sich im Speicher des Computers befindet. Da jedes Objekt einen 24-Byte langen Eintrag sein Eigen nennt, und die Objekte der Reihe nach durchnummeriert werden, wobei das Wurzelobjekt (Root) die Nummer Null zugewiesen bekommt, können die Daten eines einzelnen Objekts nach der Formel:

**Wurzelobjekt + 24 * Objektnummer**

errechnet werden. Zum Ermitteln der Adresse des Wurzelobjekts stellt die GEM-Library wieder eine Funktion bereit. Sie lautet:

```
Rsrc_Gaddr(re_gtype,re_gindex,re_gaddr)
```

Der Übergabeparameter re_gindex repräsentiert dabei die Nummer des Objekts, in re_gaddr befindet sich nach dem Aufruf der Procedure die Adresse des Wurzelobjekts (re_gtype = 0), die für alle weiteren Schritte von enormer Bedeutung ist (z.B. um die Adresse des Eintrags für ein bestimmten Objekts zu berechnen). Wurde der Objektbaum bzw. das Wurzelobjekt beispielsweise Filsel getauft, wird die Adresse dieses Objekts mit dem Aufruf von

```
Rsrc_Gaddr(0, Filsel,Tree)
```

ermittelt. Die Variable Tree (Baum) enthält nach dem Aufruf die Objektadresse. Nachdem nun also die Adresse eines Eintrages in der Objektliste ermittelt werden kann, wäre es nicht uninteressant zu erfahren wie ein derartiger Eintrag überhaupt aussieht. Voila, da ist er:

| Offset | Inhalt |
|---|---|
| +0 | nächstes Objekt |
| +2 | Anfangsobjekt |
| +4 | Endobjekt |
| +6 | Objekttyp |
| +8 | Objektflags |
| +10 | Objektstatus |
| +12 | Objektspezifikation |
| +16 | X-Koordinate des Objekts |
| +18 | Y-Koordinate des Objekts |
| +20 | Objektbreite |
| +22 | Objekthöhe |

Um also den Objekttyp eines Objekts zu ermitteln oder gar umzubiegen, addiert man (daher die Bezeichnung Offset) zur Adresse des Objekts (errechenbar aus Adresse des Wurzelobjekts + 24 * Objektnummer) den angegebenen Offset, der hier den Wert 6 beträgt:

```
(Tree + 24 * Objektnummer) + 6
```

Auf diese Weise kann problemlos jedes Objekt manipuliert werden. Von wegen manipulieren! Dazu benötigt man wieder ein paar Befehle, die ich Ihnen hier vorstellen möchte.

### Peek, Pokeq und deren Verwandte (Speicheroperationen)

Möchten Sie ein Byte aus einer bestimmten Speicherzelle des Computers lesen, muß dies mit

```
Inhalt = PEEK(Adresse)
```

geschehen, wobei Adresse die Adresse des Byte angibt, das der Variablen Inhalt zugewiesen werden soll. Im Gegensatz dazu ist es auch möglich eine bestimmte Speicherstelle mit einem Wert zu versorgen. Das geht mit:

```
POKE Adresse,Wert
```

wobei Wert in die Speicherstelle Adresse verfrachtet wird. Peek und Poke lesen bzw. schreiben jeweils ein Byte. Zum Lesen und Schreiben von Worten (zwei Byte!) sind sie nicht geeignet und müssen das Feld für

```
Inhalt = WPEEK(Adresse)
```

zum Lesen eines Wortes und

```
WPOKE Adresse,Wert
```

zum Schreiben eines Wortes (= 16 Bit, d.h. in zwei aufeinanderfolgende Byte) räumen. Dabei ist zu beachten, daß die Adresse bei WPOKE unbedingt gerade sein muß, da ansonsten die CPU (der Prozessor des Atari ST) einen Adreßfehler meldet, der Ihnen 3 Bomben auf dem Monitor beschert.

Keine Angst, der Computer legt seine Worte schon so im Speicher ab, daß sie stets mit einer geraden Adresse beginnen. Solange Sie also keinen Fehler machen, bleiben Sie von den Bomben verschont!

Als dritte Alternative kann noch ein Langwort (4 Byte) auftreten, das ebenfalls in den Speicher gepackt und von dort wieder gelesen werden kann. Dazu dient der Befehl

```
Inhalt = LPEEK(Adresse)
```

Zum Schreiben wird

```
LPOKE Adresse,Wert
```

benutzt. Auch diese Adresse muß wieder gerade sein, damit kein Adreßfehler gemeldet wird. Nachdem wir schon die Adresse eines Objekteintrages in der Baumstruktur der Resource ermitteln können, ist nun auch deren Manipulation Tür und Tor geöffnet. Der Inhalt eines Objekteintrages setzt sich aus Worten (mit Ausnahme der 4-Byte langen Objektspezifikation) zusammen, kann also mit WPEEK und WPOKE abgeändert werden.

Möchten Sie also den Objekttyp ermitteln, lesen Sie einfach die entsprechende Speicherstelle (Verzeihung: Speicherstellen, da ein Wort in zwei aufeinanderfolgenden Byte plaziert wird):

```
Inhalt = WPEEK((Tree + 24 * Objektnummer) + 6)
```

liefert in der Variablen Inhalt den Objekttyp. Die verschiedenen Objekttypen haben Sie schon alle bei der Beschreibung des RCS kennengelernt. Deshalb hier nur eine Tabelle, die die Typennummern (den Inhalt) der verschiedenen Objektarten enthält:

| Objekttypennummer | Objekt |
|---|---|
| 20 | Box |
| 21 | Text |
| 22 | Boxtext |
| 23 | Image |
| 24 | Progdef (Ein Objekt das vom Programmierer selbst definiert werden kann) |
| 25 | Ibox |
| 26 | Button |
| 27 | Boxchar |
| 28 | String |
| 29 | Ftext |
| 30 | Fboxtext |
| 31 | Icon |
| 32 | Title |

Auch die übrigen Daten eines Objekteintrages in der Baumstruktur sind codiert. Deshalb hier der Reihe nach ihre Bedeutung:

### Objektflags

Die Objektflags entscheiden darüber, welche Eigenschaften ein Objekt besitzen soll. Jedes Bit repräsentiert dabei den Zustand eines bestimmten Flags:

| Bit | Zustand |
|---|---|
| 0 | SELECTABLE |
| 1 | DEFAULT |
| 2 | EXIT |
| 3 | EDITABLE |
| 4 | RADIO-BUTTON |
| 5 | LASTOB |
| 6 | TOUCHEXIT |
| 7 | HIDETREE |
| 8 | INDIRECT |

Da Sie sehr wahrscheinlich mit den verschiedenen Zuständen nicht allzuviel anzufangen wissen, kommt gleich eine kurze Erklärung der einzelnen Bedeutungen:

### SELECTABLE

Dieses Objekt kann vom Benutzer mit der Maus angeklickt werden, wobei es daraufhin invertiert (schwarz) erscheint.

### DEFAULT

Dieses Objekt kann auch durch eine Betätigung der <Return>-Taste selektiert werden. Innerhalb einer Baumstruktur ist jedoch darauf zu achten, daß maximal ein Objekt als Default gekennzeichnet werden darf. Im allgemeinen wird ein Knopf, der das Verlassen des Formulares ermöglicht, dieses Attribut tragen.

### EXIT

Wird ein Knopf, den dieses Attribut schmückt, angeklickt, so erhält das Programm die Kontrolle über das Formular zurück, die zuvor beim AES lag (Details folgen noch).

### EDITABLE

Dieses Objekt kann in irgendeiner Form vom Benutzer editiert werden (bei Text der Fall).

### RADIO-BUTTON

Radio-Buttons bestehen aus Gruppen von Knöpfen (mindestens zwei Knöpfe sind notwendig), die in der gleichen Ebene liegen müssen und eine ganz besondere Eigenschaft an den Tag legen: Sobald ein Knopf aus dieser Gruppe angeklickt wird, wird der zuvor invertierte (schwarze) Knopf wieder in normale Darstellungsart gebracht. Es kann also nur jeweils ein Knopf aus der ganze an Gruppe selektiert werden.

### LASTOB

Dieses Flag zeigt an, daß es sich um das letzte Element in der Baumstruktur (sequentielle Liste) handelt. Je Baum ist nur ein solches Objekt gestattet.

### TOUCHEXIT

Dieses Flag sorgt dafür, daß die Kontrolle an das Programm (Application) zurückgegeben wird, sobald sich der Mauszeiger auf dem betreffenden Objekt befindet und die Maustaste gedrückt wird. Im Gegensatz zu EXIT braucht die Maustaste jedoch nicht erst losgelassen zu werden, um die Kontrolle zurückzugeben.

### HIDETREE

Ein mit diesem Attribut versehenes Objekt wird unsichtbar gemacht. Bei einer erneuten Darstellung dieser Objekte auf dem Monitor sind sie nicht mehr im Formular sichtbar, sind aber wohl noch vorhanden! Ein Löschen des Flags bewirkt, daß das entsprechende Objekt ab sofort wieder sichtbar ist (nach Aufruf der Funktion OBJC_DRAW versteht sich!).

### INDIRECT

Dieses Flag gibt an, daß die Objektspezifikation nicht der tatsächliche Wert, sondern lediglich ein Zeiger auf diesen Wert darstellt.

### Objektstatus

Der Objektstatus bestimmt die Gestalt und die Darstellung eines Objekts. Die Bedeutung der einzelnen Bits:

| Bit | Bedeutung |
|---|---|
| 0 | SELECTED |
| 1 | CROSSED |
| 2 | CHECKED |
| 3 | DISABLED |
| 4 | OUTLINED |
| 5 | SHADOWED |

### SELECTED

Ist Bit 0 gesetzt, so wird das Objekt als vorbelegt gekennzeichnet, d.h. der Zustand des Objekts ist bereits aktiv. Auf diese Weise können Objekte invers dargestellt werden, ehe Sie mit der Maus angeklickt werden.

### CROSSED

Dieses Flag kann nur bei ...BOX...-Objekten verwendet werden und bewirkt, daß ein X in das Objekt gezeichnet wird.

### CHECKED

In das Objekt wird ein kleines Häckchen (vgl. Menüleiste, in der Einträge als abgehakt gekennzeichnet werden können) gezeichnet.

### DISABLED

Das Objekt wird schwach gezeichnet, um anzuzeigen, daß der Benutzer es ab sofort nicht mehr anwählen kann (bei Text und in Menüleisten).

### OUTLINED

Um das Objekt wird ein weiterer Rahmen gesetzt.

### SHADOWED

Das Objekt wird an der rechten unteren Ecke mit einem Schatten verziert.

### Objektspezifikation

Die Objektspezifikation ist (für uns) immer dann interessant, wenn ein Text- bzw. ein sonstiges Eingabeobjekt vorliegt. In diesen Fällen enthält die Objektspezifikation einen Zeiger auf eine objektspezifische Datenstruktur, die TEDINFO-Struktur genannt wird. Für andere Objekttypen enthält sie entweder Zeiger auf verschiedenartige Datenstrukturen, die hier nicht weiter interessieren sollen, oder sonstige Informationen über die Farbe und die Dicke des Randes.

### Die TEDINFO-Struktur

Diese Datenstruktur beinhaltet Informationen, die von GEM für die Ein- und Ausgabe eines Textes benötigt werden. Sie findet bei den Objekttypen

```
TEXT
BOXTEXT
FTEXT
FBOXTEXT
```

Verwendung, wobei ihre Adresse innerhalb des Objekteintrages in der Objektspezifikation abgeholt werden kann. Da es sich bei der Objektspezifikation um einen Long-Wert handelt, lautet der Befehl zum Lesen der Adresse LPEEK(). Doch nun zum Aufbau der TEDINFO-Struktur:

| Offset | Inhalt |
|---|---|
| +0 | te_ptext |
| +4 | te_ptmplt |
| +8 | te_pvalid |
| +12 | te_font |
| +14 | nicht benutzt (reserviert) |
| +16 | te_just |
| +18 | te_color |
| +20 | nicht benutzt (reserviert) |
| +22 | te_thickness |
| +24 | te_txtlen |
| +26 | te_tmplen |

### te_ptext

Beinhaltet einen Zeiger, der auf den auszugebenden Text verweist (Adresse des Textes). Ist das erste Zeichen ein Klammeraffe (@), so werden die nachfolgenden Zeichen als Leerzeichen angesehen.

### te_ptmplt

enthält die Adresse eines Textes, der als Schablone bzw. als Eingabemaske fungiert. Eine Eingabeposition, die der Benutzer editieren können soll, muß das Zeichen "_" tragen. An Stelle des "_" kann dann der Benutzer ein anderes Zeichen eingeben, wobei die Auswahl der eingebbaren Zeichen durch te_pvalid eingeschränkt werden kann, um. z.B. nur Großbuchstaben oder Ziffern zuzulassen.

### te_pvalid

Adresse eines Strings, der die Eingabe einschränkt, und somit bestimmt, welche Zeichen bei der Eingabe zugelassen sind. Die erlaubten Zeichen werden für jede Eingabeposition durch einen Code bestimmt:

| Code | erlaubt sind |
|---|---|
| 9 | nur Ziffern |
| A | Großbuchstaben und Leerzeichen |
| a | Groß- Kleinbuchstaben und Leerzeichen |
| N | Ziffern, Großbuchstaben und Leerzeichen |
| n | Ziffern, Groß- Kleinbuchstaben und Leerzeichen |
| F | TOS-Filenamen, sowie '?', '*', ':' |
| p | TOS-Filenamen, sowie '?', '*', ':', '\' |
| P | TOS-Filenamen, sowie '\', ':' |
| X | alle Zeichen |

### te_font

enthält die Nummer des Zeichensatzes, der verwendet werden soll:

3 => normaler Zeichensatz
5 => verkleinerter Zeichensatz

### te_just

gibt die Ausrichtung an, mit der der Text formatiert werden soll:

0 => linksbündig
1 => rechtsbündig
2 => zentriert (Mitte)

### te_color

bestimmt die Farbe, nach folgender Tabelle:

| Wert | Farbe |
|---|---|
| 0 | weiß |
| 1 | schwarz |
| 2 | rot |
| 3 | grün |
| 4 | blau |
| 5 | cyan |
| 6 | gelb |
| 7 | magenta |

| Wert | Farbe |
|---|---|
| 8 | weiß |
| 9 | schwarz |
| 10 | hellrot |
| 11 | hellgrün |
| 12 | hellblau |
| 13 | hellcyan |
| 14 | hellgelb |
| 15 | hellmagenta |

wobei folgende Bitmaske verwendet wird:

```
($) rrrr zzzz smmm ffff
```

Dabei bedeutet:

r: Randfarbe
z: Zeichenfarbe
s: Schreibmodus
    0 => transparent
    1 => deckend
m: Füllmodus
    0 => keine Füllung
    7 => deckende Füllung
    1-6 => Füllstufen mit steigender Dichte
f: Füllfarbe

### te_thickness

Legt die Dicke der Rechteckumrandung fest. Es gilt:

| | |
|---|---|
| 0 | kein Rand |
| 1-127 | Dicke des inneren Randes |
| 128- | Dicke des äußeren Randes, als negative Zahl zu interpretieren: -1 bis -127. |

### te_txtlen

Enthält die Länge des Strings, auf den te_ptext verweist. Da das den String abschließende Nullbyte (CHR$(0)) mitgerechnet wird, muß die Länge um eins größer als die tatsächliche Zeichenzahl angegeben werden.

### te_tmplen

enthält die Länge des Strings, auf den die in te_ptmplt abgelegte Adresse verweist. Auch hier muß das Nullbyte am Stringende mitgezählt werden.

Ein kleines Beispiel soll noch einmal die Funktion der ersten drei Langworte verdeutlichen:

```
te_ptext  zeige auf '1522'+CHR$(0)
te_ptmplt zeige auf 'Preis ÖS __.__'+CHR$(0)
te_pvalid zeige auf '9999'+CHR$(0)
```

so wird bei der Darstellung des Objektes der Text mit der Schablone gemischt:

```
Preis ÖS 15.22
```

Vom Benutzer können lediglich Ziffern eingegeben werden, da die zugelassenen Zeichen durch te_pvalid ("9999") auf Ziffern eingeschränkt werden.

### EDIT-Objekte

Mit diesem Wissen ausgestattet, werden wir zwei Prozeduren entwickeln, die das Eintragen eines Strings in ein Objekt bzw. das Auslesen eines solchen ermöglichen. Dabei ist natürlich zu beachten, daß die Routinen nur auf die Objekte TEXT, BOXTEXT, FTEXT und schließlich FBOXTEXT angewendet werden, da nur bei ihnen die Objektspezifikation auf eine TEDINFO-Struktur verweist. Als Übergabeparameter soll einmal

die Nummer des Objekts, in das der String geschrieben bzw. aus dem der String abgeholt werden soll sowie der eigentliche String angegeben werden:

```
Put_Text(Objektnummer, "Text")
```

zum Schreiben eines Textes, sowie

```
Get_Text(Objektnummer, Variable$)
```

zum Lesen eines (eingegeben) Textes, der im Rückgabeparameter Variable$ zu finden sein wird.

Wie kommt man nun an den Text bzw. an dessen Adresse heran? Dazu wird zuerst einmal die Adresse des Eintrages in der Objektliste für das gewünschte Objekt ermittelt. Dies sollte keinerlei Schwierigkeiten bereiten (wurde ja schon oft genug vorgeführt!), da jeder Eintrag 24 Byte umfaßt, und die Adresse des Wurzelobjekts in der Variablen Tree gespeichert sein sollte. Diese Adresse ermittelt die Funktion RSRC_GADDR(0,0,Tree). Die Formel zur Adreßermittlung lautet dann:

```
Adresse = Tree + 24 * Objektnummer
```

Zu der auf diese Weise ermittelten Adresse muß noch der Offset für die Objektspezifikation addiert werden. Dieser Offset beträgt - wenn Sie noch einmal um ein paar Seiten zurückblättern - 12. In der Objektspezifikation (einem Lang-Wert), findet sich die Adresse der Tedinfostruktur. Mit LPEEK() kann diese ermittelt werden:

```
Tedinfo= LPEEK(Adresse+12)
```

wobei Adresse mit obiger Formel ermittelt wurde. Die ersten vier Byte (wieder ein Lang-Wert) der TEDINFO-Struktur verweist auf die Adresse, an der der eigentliche Text zu finden ist.

```
Text_Adresse= LPEEK(Tedinfo + 0)
```

wobei (strenggenommen) der Offset von 0 hinzuaddiert wurde, der jedoch völlig bedeutungslos ist, und deshalb auch weggelassen werden kann.

Jetzt erst findet sich in der Variablen Text_Adresse die eigentliche Adresse des Textes. Von dort kann dieser nun abgeholt werden. Mit PEEK() wird Zeichen für Zeichen gelesen. Als Abbruchkriterium für unsere Schleife (in der Peek() steht) dient das Nullbyte, das nach guter alter C-Konvention und GEM-Manier einen String abzuschließen hat. Das Nullbyte kann wahlweise an den (ST-BASIC-)String gehängt, aber auch weggelassen werden. Für die Stringmanipulation eher störend, wird das Nullbyte doch immer dann benötigt, wenn dieser String in irgendeiner Form an das Betriebssystem übergeben werden soll. Die folgende Schleife erhöht einen Schleifenzähler T% solange, bis das Nullbyte erreicht ist, hängt dieses Nullbyte jedoch nicht an den String:

```
Text$=""' enthält den Text
T%= 0' Die Zählvariable
' Schleife wiederholen, solange das Nullbyte
' noch nicht erreicht ist (abweisende Schleife!)
WHILE PEEK(Text_Adresse+T%) <> 0
  Text$=Text$+ CHR$( PEEK(Text_Adresse+T%))
  T%=T%+1
WEND
```

Damit der zeichenweise ermittelte String an den aufrufenden Programmteil zurückgegeben wird, definieren wir den Parameter als Rückgabewert durch ein vorangestelltes R:

```
DEF PROC Get_Text(Nummer%, R Text$)
  LOCAL Adr=Tree+24*Nummer%
  LOCAL Tedinfo= LPEEK(Adr+12)
  LOCAL Text_Adr= LPEEK(Tedinfo)
  LOCAL T%=0
  ....
  hier steht die Schleife von oben
  ....
RETURN
```

Ähnlich aufbauen müssen wir auch die zweite Procedure, die einen Text in ein Objekt eintragen soll. In diesem Fall muß jedoch (falls noch nicht vorhanden) ein Nullbyte an das Stringende gehängt werden. Dieses Nullbyte soll bereits in der als Parameter übergebenen Zeichenkette enthalten sein.

**Tip:** Fehlt das Nullbyte am Stringende, können Sie mit einer an Sicherheit grenzenden Wahrscheinlichkeit mit einen Systemabsturz rechnen. Um dies zu verhindern, kann eine Sicherheitsabfrage in die Procedure integriert werden, die - bei Bedarf - ein Nullbyte an das Stringende hängt:

```
IF RIGHT$(Text$,1) <> CHR$(0)
  Text$=Text$+ CHR$(0)
ENDIF
.....
hier folgt dann die Schleife zum Eintragen der
Zeichenkette in das gewünschte Objekt
.....
```

Das Schreiben eines Strings in das Objekt verläuft analog, im Gegensatz zum Lesen wird hier Zeichen für Zeichen in eine Zahl (ASCII-Code) konvertiert und mit POKE in die ab Adresse folgenden Speicherstellen abgelegt:

```
DEF PROC Put_Text(Nummer%,Text$)
  LOCAL Adr=Tree+24*Nummer%
  LOCAL Tedinfo= LPEEK(Adr+12)
  LOCAL Text_Adr= LPEEK(Tedinfo)
  LOCAL T%=0
  ' solange das Stringende noch nicht erreicht ist
  ' Zeichen für Zeichen mit POKE schreiben
  WHILE T% < LEN(Text$)
    POKE Text_Adr+T%, ASC( MID$(Text$,T%+1,1))
    T%=T%+1
  WEND
RETURN
```

Damit Sie nicht eine böse Überraschung durch einem Systemabsturz erleben, darf die so eingetragene Zeichenkette keinesfalls länger als der im RCS für diesen Eintrag vorbelegte String sein, da ansonsten unter Umständen wichtige Daten im RCS-File vernichtet werden können!

### Formularverwaltung

Was bisher nur kurz angerissen wurde und dann einer Unmenge Theorie weichen mußte, soll in diesem Kapitel zu seinem Recht kommen: die Formularverwaltung, d.h. jenes "Schema F", nach dem die Verwaltung von Formularen abgewickelt wird. Hier noch einmal - zur Erinnerung - die einzelnen Schritte:

1. Das auf Diskette abgelegte RCS-File wird geladen. Dazu dient die Routine RSRC_LOAD, die im AES implementiert ist.

2. Die Adresse des gerade geladenen Objektbaumes wird mit einem Aufruf von RSRC_GADDR ermittelt.

3. Optional, also wahlweise, kann jetzt der Aufruf von FORM_CENTER erfolgen, der dafür Sorge trägt, daß die Koordinaten des Formulars so bestimmt werden, daß es sich in der Mitte des Bildschirms befindet.

4. FORM_DIAL sorgt dafür, daß Bildschirmspeicherplatz reserviert wird.

5. FORM_DIAL wird ein zweites Mal aufgerufen. Im Unterschied zum ersten Aufruf bewirkt ein geänderter Parameter, daß ein sich ausdehnendes Rechteck auf den Bildschirm gezeichnet wird (rein optischer Effekt!).

6. Jetzt kann das Formular mit OBJC_DRAW gezeichnet werden.

7. Nachdem das Formular vollständig auf dem Bildschirm dargestellt ist, übergibt man die Kontrolle über das ganze Formular an das AES mit der Funktion FORM_DO. Erst wenn ein mit dem Attribut EXIT, TOUCHEXIT oder DEFAULT tituliertes Objekt durch ein Betätigen von Return oder Anklicken mit der Maus selektiert wird, gibt das AES die Kontrolle an die Application zurück, die nun zu entscheiden hat, was weiter geschehen soll. Dies ist möglich, indem der Code, d.h. die Objektnummer des zum Abbruch führenden Objekts zurückgegeben wird, die von der Application genau zu analisieren ist.

8. Hat das Formular seine Pflicht und Schuldigkeit getan, können die ersten Schritte zur Demontage des selbigen anlaufen: FORM_DIAL sorgt dafür, daß der Eindruck eines kleiner werdenden Rechtecks auf dem Monitor erscheint.

9. Schließlich wird mit FORM_DIAL noch der beim ersten Aufruf der Funktion occupierte Speicherbereich für den Bildschirm freigegeben. Da GEM keinerlei Restauration der vom Formular während seines Aufrufes bedeckten Fläche vorsieht, muß die Application den Bildschirm neu aufbauen. Dies kann durch ein Zwischenspeichern des betroffenen Bildschirmbereichs mittels BITBLT (Bit-Blit) geschehen, der nach Beendigung der Arbeit mit dem Formular wieder auf den Bildschirm gebracht wird.

Die zur Formularverwaltung benötigten Routinen im einzelnen:

**Rsrc_Load(Re_Lname$,Re_Lreturn)**

Diese Funktion lädt eine Resource-Datei in den Speicher des Computers, wobei in der Variablen Re_Lname$ der Name der Resourcedatei stehen muß. In der Variablen Re_Lreturn teilt der Computer mit, ob der Ladevorgang erfolgreich durchgeführt werden konnte. Enthält diese Variable den Wert 0, so ist ein Fehler beim Laden der Datei aufgetreten, konnte die Datei ordnungsgemäß geladen werden, so enthält die Variable eine 1.

Es empfiehlt sich, nach dem Funktionsaufruf erst einmal die Variable zu überprüfen, und wenn die Datei nicht ordnungsgemäß geladen werden konnte, das Programm zu beeenden:

```
Rsrc_Load(Name$,Ret)

IF RET = 0 THEN
   ' Fehlermeldung ausgeben
   FORM_ALERT(1,[3][Fatal-Error][ Sorry ]")
   ' GEM wieder abmelden!
   Appl_Exit
   END
ENDIF
```

### Rsrc_Gaddr(Re_Gtype,Re_Gindex,Re_Gaddr)

Diese Funktion - aber das wissen Sie ja bereits - ermittelt die Adresse einer Datenstruktur bzw. eines Objekts im Speicher. Re_Gtype kennzeichnet die Art der Struktur, die gesucht werden soll. Es bedeuten dabei:

```
0 => Baumstruktur
1 => Objekt
```

Unter Re_Gindex wird die Nummer des gesuchten Objekts übergeben. Die Rückantwort des Funktionsaufrufs ist die Adresse der gesuchten Objektstruktur in Re_Gaddr, die für ein Wurzelobjekt üblicherweise in der Variablen Tree gespeichert wird. Mit der Adresse des Wurzeleintrages (Root) können dann die einzelnen Objektstrukturen errechnet werden.

### Form_Center(Tree,X_Obj,Y_Obj,W_Obj,H_Obj)

Berechnet die Koordinaten für ein Formular, damit die Dialogbox genau in der Bildschirmmitte erscheint. Die Adresse des Objektbaumes ist in der Variablen Tree anzugeben, als Rückantwort liefert die Funktion:

| | |
|---|---|
| X_Obj | X-Koordinate der linken oberen Ecke, |
| Y_Obj | Y_Koordinate der linken oberen Ecke, |
| W_Obj | enthält die Breite der Dialogbox, |
| H_Obj | enthält die Höhe der Dialogbox. |

### Form_Dial(Flag,X_Obj,Y_Obj,W_Obj,H_Obj)

Diese Funktion ist ein wahres Multitalent, und erledigt gleich vier verschiedene Arbeiten, die sich nach dem Inhalt von Flag richten:

0 => Bildschirmspeicherplatz reservieren,
1 => ausdehnendes Rechteck zeichnen,
2 => schrumpfendes Rechteck zeichnen,
3 => Speicherplatz wieder freigeben.

Die übrigen vier Parameter besitzen die gleichen Bedeutungen, wie in obiger Funktion, nämlich die Koordinaten der linken oberen Ecke, sowie die Breite und Höhe der Dialogbox.

### Objc_Draw(Start,Tiefe,X_Obj,Y_Obj,W_Obj,H_Obj,Tree)

Stellt ein Formular auf dem Bildschirm dar, wobei

| | |
|---|---|
| Start | den Index des ersten zu zeichnenden Objekts, |
| Tiefe | die Anzahl der zu zeichnenden Ebenen (max: 8), |
| X_Obj | die X_Koordinate der linken oberen Ecke, |
| Y_Obj | die Y_Koordinate der linken oberen Ecke, |
| W_Obj | die Breite des Formulars, |
| H_Obj | die Höhe des Formulars, |
| Tree | die Adresse der Baumstruktur, |

enthält. Dabei ist zu beachten, daß Bildschirmbereiche, die durch den Aufbau des Formulars überschrieben werden, von GEM nicht selbstständig gerettet werden! Dies muß vom Programmierer selbst in die Hand genommen werden.

### Form_Do(Start,Tree,Return)

Mit dem Aufruf dieser Funktion wird die Kontrolle des Formulars der Application entzogen und an AES übergeben. Start enthält den Index des Objekts, auf das der Cursor (senkrechter Strich innerhalb eines Eingabefeldes!) positioniert werden soll. Enthält das Formular keine editierbaren Textfelder, muß der Wert auf Null gesetzt werden. Tree ist wieder die Adresse unseres Objektbaumes, in der Variablen Return wird die Nummer (Index) des Objekts zurückgeliefert, das den Exit bewirkte. Dementsprechend kann dann innerhalb des Programms fortgefahren werden.

**Vorsicht:** Ehe diese Funktion aufgerufen wird, sollten Sie sich vergewissern, ob es innerhalb des Objektbaumes mindestens ein Objekt gibt, mit dem die Kontrolle an die Application zurückgegeben werden kann, das also mit einem der drei Attribute EXIT, TOUCHEXIT oder DEFAULT geschmückt ist. Andernfalls gibt es keine Möglichkeit, dem AES die Kontrolle über das Formular wieder zu entziehen (peinlich, denn dann hilft nur noch ein Ausschalten des Computers, und das Programm im Speicher ist futsch, es sei denn Sie haben es in weiser Voraussicht zuvor abgespeichert).

### Objekt-Attribute abfragen und setzen

Ehe AES die Kontrolle nach einem Aufruf von FORM_DO wieder an die Applikation zurückgibt, kann der Benutzer Text in Edit-Felder eingeben, er kann verschiedene Objekte anklicken (z.B. Radio-Boxen) und auf diese Weise verschiedene Einstellungen vornehmen. Es liegt nun am Programm, nach der Aktivierung eines Exit-Objekts die verschiedenen Einstellungen zu überprüfen und die getätigten Eingaben abzuholen, ehe das Formular wieder in der Versenkung verschwindet.

Solange sämtliche Boxen, die angeklickt werden können das zierende Attribut eines irgendwie gearteten Exit tragen, kann eine weitergehende Überprüfung entfallen. Trotzdem müssen alle se-

lektierten (schwarzen) Objekte wieder in den Ausgangszustand gebracht werden, damit sie bei einem erneuten Aufruf des Formulars wieder angeklickt werden können und nicht schon von vornherein selektiert sind. (Wird z.B. das Objekt Fertig zum Verlassen des Formulars angeklickt, so führt dies unweigerlich zu einem Invertieren der Box. Würde das Formular erneut aufgerufen, so bliebe das zuvor selektierte Objekt dank des gesetzten Bits Null (= SELECTED) im Objektstatus schwarz, und das Formular könnte nur durch zweimaliges Anklicken erneut verlassen werden.) Demzufolge müssen alle Objekte nach einem Aufruf des Formulars wieder restauriert und in den Ausgangszustand gebracht werden. Die Theorie dazu wurde schon einmal an anderer Stelle in diesem Buch dargestellt. Aus diesem Grund kann ich hier gleich auf die Praxis eingehen:

Ein gesetztes Bit 0 im Objektstatus sorgt dafür, daß das entsprechende Objekt im Formular invertiert dargestellt wird. Möchte man ein solches Objekt wieder in den Ausgangszustand bringen (deselektieren), muß lediglich dieses verflixte Bit ausmaskiert werden. Zum Ausmaskieren eines Bits dient die logische Verknüpfung AND. Da die restlichen Bits nicht angetastet werden dürfen, setzen wir sie in der Maske einfach auf den Wert 1, nur das erste Bit enthält eine 0 und trägt dafür Sorge, daß als Ergebnis der Undierung dieses Bit stets den Wert 0 besitzt:

```
Irgendetwas AND %11111110
```

setzt das erste Bit (Bit 0) in jedem Fall auf den Wert 0, die übrige Bitmaske bleibt dabei unverändert. Der Übersichtlichkeit halber wandelt man die Dualzahl noch in das uns besser vertraute Dezimalsystem und erhält den Wert 254.

Genau umgekehrt verhält es sich, wenn ein Objekt von der Applikation aus den Status SELECTED erhalten soll. In diesem Fall muß dieses Bit - ungeachtet seines bisherigen Inhalts - auf den Wert 1 gesetzt werden. Mit einer Undierung gelangt man dabei jedoch nicht zum Ziel, vielmehr heißt es logisch Odieren.

```
Irgendetwas OR %00000001
```

**Vorsicht:** Während bei AND in der Maske nur Bit 0 nicht gesetzt sein durfte, würde dies bei OR zu einer mittelschweren Katastrophe führen, da alle Bits, nur nicht das gewünschte gesetzt würden. Also müssen die Bits umgedreht, d.h. negiert werden, ehe die Oder-Verknüpfung iher Pflicht tun darf.

Nach diesen Vorüberlegungen kann gezielt der Objektstatus eines Eintrages im Formular manipuliert werden. Und da dies häufiger geschehen muß, schreibt man am besten gleich eine kleine Procedure, die diese Aufgabe für uns erledigt. Über ein Flag als Parameter wird ihr mitgeteilt, ob das entsprechende Objekt selektiert oder deselektiert werden soll:

```
1 => Objekt selektieren
0 => Objekt deselektieren
```

Der Parameter Nummer% enthält wieder die Objektnummer des betreffenden Objekts in der Liste.

```
DEF PROC Select(Nummer%,Flag%)
LOCAL Adr%L=Tree%L+24*Nummer%
IF Flag%=1 THEN
   LOCAL Ob_State%L= WPEEK(Adr%L+10) OR 1
ELSE
   LOCAL Ob_State%L= WPEEK(Adr%L+10) AND 254
ENDIF
Objc_Change(Nummer%,Ob_State%L,Xobj%L,Yobj%L,Wobj%L,Hobj%L,Tree%L)
RETURN
```

Nachdem die Adresse des Objektstatus für das gewünschte Objekt berechnet ist, besorgt sich das Programm den Objektstatus mittels WPEEK() und kümmert sich um das Bit 0, indem es einmal gesetzt (OR) und einmal ausmaskiert wird (AND). Das Ergebnis wird in der lokalen Variablen Ob_State%L zwischengespeichert. Doch damit ist erst die halbe Arbeit getan. Schließlich muß der neue Status noch in die Objektstruktur eingetragen werden. Eine Funktion des GEM nimmt uns auch diese Arbeit ab. Sie lautet:

```
Objc_Change(Index,Neuer_Status,X_Obj,Y_Obj,W_Obj,H_Obj,Tree)
```

Wobei unter Index die Nummer des Objekts einzusetzen ist, dessen Status in Neuer_Status geändert werden soll. Die restlichen Parameter geben die Ausmaße des Begrenzungsrechtecks, innerhalb dessen eine Änderung durchgeführt werden soll, sowie das Wurzelobjekt Tree an. Es existiert noch eine zweite Variante dieser Funktion, die bei ihrem Aufruf keinerlei Begrenzungskoordinaten benötigt, wohl aber das Wurzelobjekt:

```
Objc_Change(Index,Neuer_Status,Tree)
```

ändert den Status des Objekts mit der Nummer Index in Neuen_Status um.

Anhand eines zweiten Beispiels möchte ich Ihnen noch das Enablen und Disablen eines Objekts zeigen. Unter diesen beiden Begriffen versteht man das Ver- und Entriegeln eines Objekts, das dann ab sofort nicht mehr mit der Maus selektiert werden kann. Gleichzeitig soll durch eine hellere Darstellung des Objektinhaltes kenntlich gemacht werden, daß dieses Objekt mit der Maus nicht (mehr) selektiert werden kann.

Für diese Operation sind sowohl der Objektstatus als auch die einzelnen Objektflags von Bedeutung. Der Status ist nämlich für das Flag Disabled (Bit 3) verantwortlich, während sich in den Objektflags die Bits befinden, die ein Anwählen und Verlassen eines Objekts gestatten.

```
DEF PROC Enable(Nu%,Flag%)
LOCAL Adr=Tree+24*Nu%
IF Flag%=1 THEN
   WPOKE Adr+8, WPEEK(Adr+8) OR 1
   Objc_Change(Nu%, WPEEK(Adr+10) AND
247,Xobj,Yobj,Wobj,Hobj,Tree)
ELSE
   WPOKE Adr+8, WPEEK(Adr+8) AND 254
   Objc_Change(Nu%, WPEEK(Adr+10) OR 8,Xobj,Yobj,Wobj,Hobj,Tree)
ENDIF
RETURN
```

Zuerst einmal zu den Objektflags. Bit 0 ist innerhalb der Flags dafür verantwortlich, daß ein Objekt mit der Maus selektiert werden kann. Ein Ausmaskieren dieses Bits bewirkt, daß dieses

ab sofort nicht mehr selektiert werden kann. Andererseits verhilft ein gesetztes Bit dem Objekt zur Selektierbarkeit mit dem Mauspfeil. Mit

```
WPOKE Adr+8, WPEEK(Adr+8) OR 1 [AND 254]
```

werden die Objektflags gelesen, logisch verknüpft und gleich wieder zurückgeschrieben. Der nächte Schritt hat dem Objektstatus zu gelten. Dort wird es noch einmal um eine Kleinigkeit komplizierter:

Das relevante Bit ist hier Bit 3. Ein gesetztes Bit 3 verkörpert den Wert 8 (2^3). Möchte man andererseits alle Bits bis auf das dritte Bit setzen, muß die Zahl 8 negiert werden. Diese Negation bewirkt ein Umdrehen der Bitwerte und somit die Zahl 247 (255-2^3). Den so erhaltenen neuen Objektstaus dürfen wir wieder getrost der Funktion OBJC_CHANGE anvertrauen, sie wird ein weiters Mal dafür Sorge tragen, daß der Objektstatus entsprechend geändert wird.

Ruft man also die Funktion Enable() auf, wird das angegebene Objekt je nach dem Zustand des Flags

```
0 => Disabled
1 => Enabled
```

Gleichzeitig bewirkt ein Aufruf dieser Funktion, daß der Text, den das Objekt beinhaltet, heller dargestellt wird, um optisch zu kennzeichnen, daß dieses Objekt nicht angewählt werden kann. Eine etwas abgewandelte Variante dieser Funktion werden wir auch in der eigenen File-Selektor-Box verwenden. Dort wird sie uns gute Dienste beim Verriegeln der Umschaltknöpfe von nicht angeschlossenen Laufwerken leisten. Dennoch müssen ein paar Kleinigkeiten geändert werden, um eine universelle Einsetzbarkeit für alle Bereiche der Selektor-Box zu gewährleisten.

## 6.3 Schieberegler (Slider)

GEM erlaubt die Benutzung von Schiebereglern, sogenannten Slidern. Ein solcher Slider besteht zunächst aus zwei Komponenten: einem übergeordneten Boxobjekt (Parent), innerhalb dessen sich ein kleineres Boxobjekt (Child) befindet, das dann innerhalb des übergeordneten Objekt, verschoben werden kann.

Der eigentliche Schieber muß dabei das Attribut TOUCHEXIT tragen, auch ein TOUCHEXIT für das Parent-Objekt ist mit Sicherheit nicht verkehrt. Würde dies nämlich unterlassen, ließe sich der Schieber nicht einen Deut bewegen. Sind beide Objekte - Vater und Kind - erst einmal konstruiert, stellt der Rest der Slider-Programmierung eine reine Formsache dar.

Die Funktion Graf_Slidebox, die im AES für unsere Zwecke bereitgestellt wird, gestattet das Verschieben eines Objekts innerhalb eines anderen. Die innere Box kann dabei solange verschoben werden, wie die Maustaste gedrückt bleibt. Als Parameter erwartet die Funktion:

```
Graf_Slidebox(Parent,Child,Richtung,Position)
```

| | |
|---|---|
| Parent: | Index des übergeordneten Objekts |
| Child: | Index des untergeordneten Objekts |
| Richtung: | Gibt die erlaubte Verschieberichtung an: |
| | 0 => waagrecht |
| | 1 => senkrecht |
| Position: | Rückantwort, gibt die Position des verschiebbaren Rechtecks relativ zum übergeordneten Objekt an. Ein Rückgabewert von 0 entspricht ganz links (bzw. oben) ein Wert von 1000 ganz rechts (unten). |

Da die Position relativ gemeldet wird, muß sie erst noch in das richtige Verhältnis umgerechnet werden. Dies wird bewerkstelligt, indem man zuerst einmal die Breite der inneren Box (des eigentlichen Schiebers) von der Breite der äußeren Box subtra-

hiert. Das so gewonnene Ergebnis muß noch mit dem Rückgabeparameter (Position) multipliziert und anschließend durch 1000 dividiert werden. Daraus läßt sich dann folgende Formel zusammenstellen:

```
Position = (Außenbreite-Innenbreite)*Position/1000
```

Der Schieberegler wird von GEM verwaltet. Worum wir uns jedoch selbst kümmern müssen, ist die richtige Größe des Reglers innerhalb der Leiste. Da die Reglergröße und damit die mögliche Strecke, um die der Regler bewegt werden können soll, stark davon abhängig ist, wie viele Einträge im Inhaltsverzeichnis einer Diskette vorhanden sind, müssen diese Werte als Grundlage für die Berechnung der Größe des Schiebereglers herangezogen werden:

- Können alle Einträge angezeigt werden, so soll der Regler die Größe des Vaterobjekts (Schiebestrecke) erhalten, um ein (sinnloses) Verschieben des Reglers zu verhindern.
- Je mehr Dateien auf der Diskette vorhanden sind, desto kleiner werden die Ausmaße des Sliders, damit er einen entsprechend größeren Aktionsradius eingeräumt bekommt.

Ferner muß noch die Breite des Reglers der Breite des Vater-Objekts angepaßt werden, da dies vom RCS aus nur bedingt möglich ist. Alle diese Daten finden sich im Eintrag des entsprechenden Objekts in der Objektliste und können manipuliert werden. Als praktisches Beispiel sei hier auf die eigene File-Selector-Box verwiesen, in der ein Slider nach obigen Überlegungen verwaltet wird.

## 6.4 Pull-Down-Menüs

Längst hat man sich an jene Menüleisten gewöhnt, die bei einer Berührung mit dem Mauspfeil herunterklappen und ihre verschiedenen Funktionen zur Auswahl stellen. Auch sie sind eine Errungenschaft des GEM und können somit mit einem Resource Construction Set erstellt werden.

Der erste Titel (von links), der üblicherweise als Desk bezeichnet wird, birgt die verschiedenen, beim Rechnerstart oder nach einem Reset gebooteten Accessories. Der erste Eintrag ist jedoch für die Infomeldung des gerade aktiven Programmes reserviert. Eine Menüleiste zu konstruieren ist relativ trivial, da sie aus nur wenigen Teilen zusammengebastelt wird:

| | |
|---|---|
| TITLE | Eintrag in der Menüleiste |
| ENTRY | Eintrag innerhalb des Pull-Down-Menüs |

Ferner steht noch eine getrichelte Linie, die zum Abtrennen der einzelnen Einträge untereinander dient und nicht angewählt werden kann, und eine Box, die als Platzhalter für einen späteren Eintrag (während des Programmlaufes) zur Verfügung steht. Um eine neue Zeile in das Menü einzufügen, muß zuerst einmal der Rahmen entsprechend vergrößert werden. In diesen werden dann die gewünschten Teile transportiert.

Ferner sollte - damit ein Häkchen vor einem Menü-Eintrag dargestellt werden kann - mindestens ein, besser (die gängige Methode) sind zwei Leerzeichen, vor jeden Eintrag gesetzt werden. Anhand eines keinen Demoprogrammes möchte ich Ihnen erklären, wie solche Menüleisten in eigenen Programmen eingesetzt werden können.

Laden Sie zunächst Ihr Resource Construction Set und bauen eine Menü-Baumstruktur nach folgender Rezeptur zusammen: Draggen Sie zuerst das Icon Menu in das Arbeitsfenster und nennen die neue Baumstruktur Leiste. Der erste Eintrag im ersten Menüpunkt unter Desk soll Info getauft werden. Klicken Sie dazu einmal auf DESK, worauf sich das dazugehörige Untermenü öffnen sollte. Sie finden darin die Einträge Your message here (Ihre Mitteilung hier) und Desk Accessory 1-6. DESK entspricht dabei der Objektart TITLE, während ein Eintrag (Your message here) die Objektart ENTRY benutzt.

Öffnen Sie nun das Objekt Your Message here mit einem Doppelklick und taufen es in Info um. Die ersten beiden Zeichen sollten wie gesagt stets aus Leerzeichen bestehen. Einen Namen müssen wir unserem Eintrag natürlich auch noch verpassen:

MINFO. Anschließend benennen Sie noch FILE nach der gleichen Methode in Datei um. Wir sind schließlich in Deutschland! Wenn Sie nun das Untermenü des neugeschaffenen Menüpunktes Datei ansehen, werden Sie feststellen, daß bisher lediglich der Eintrag Quit darin vorhanden ist.

Damit weitere Einträge in das Untermenü aufgenommen werden können, muß zuerst einmal die Box, die dafür viel zu klein ist, entsprechend vergrößert werden. Mit einem Mausklick auf die rechte untere Ecke müßte dies gleich bewerkstelligt sein. Von wegen! Sie erwischen nämlich bei der ganzen Aktion stets die rechte untere Ecke des Eintrages. Ein Taschenspieler-Trick hilft uns aus der Patsche: Verkleinern Sie zuerst die Box des Menü-Eintrages, dann kann die eigentliche Box problemlos vergrößert werden. Den bereits vorhandenen Eintrag taufen Sie nun in laden um, und geben ihm den Namen Mload. Anschließend draggen Sie die restlichen Untermenüs (Objekt ENTRY im Objektfenster verwenden) nach folgender Abbildung in das Untermenü. Fertig!

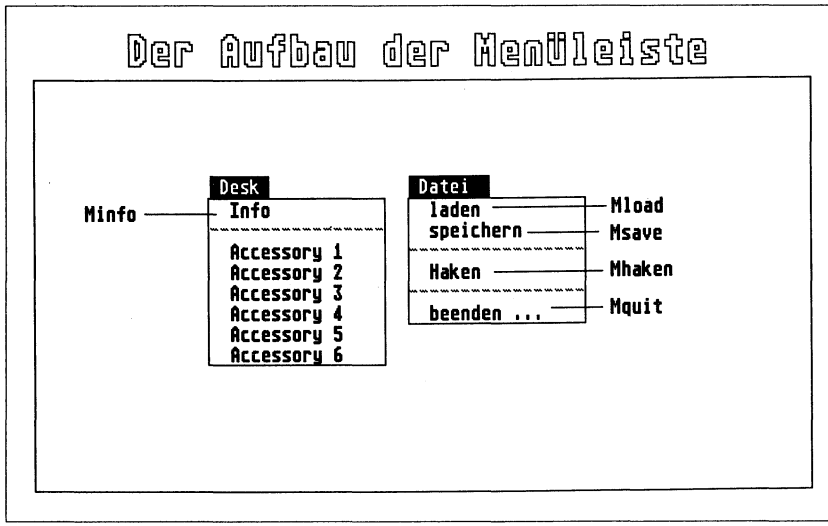



*Abb. 6.5: Die Menüleiste*

Sobald Sie eine komplette Menüleiste mit dem RCS erstellt, und sämtlichen Einträge (Entry) benannt haben, speichern Sie diese Datei auf Diskette ab. Von dort aus kann sie dann wieder in den Computer geladen werden. Zum Laden einer Resource-Datei verwendet man den altbekannten Befehl

```
Rsrc_Load(Datei$,Antwort)
```

Dieser darf jedoch erst angewendet werden, wenn die Application beim GEM mittels

```
Appl_Init
```

angemeldet wurde. Die fast schon zum Einleitungszerimoniell eines GEM-Programmes gehörende Abfrage, ob die RCS-Datei auch korrekt geladen werden konnte, sollte ebenfalls nicht vergessen werden. Jetzt erst kommen die eigentlichen Befehle zur Verwaltung der Menüleiste:

Damit eine Menüleiste dargestellt werden kann, benötigt die betreffende Funktion einen Zeiger auf eine Baumstruktur, das Wurzelobjekt der Menüleiste, das hier ausnahmsweise nicht Tree, sondern Menu_Adr tituliert werden soll. Die Funktion, die für uns diese Adresse besorgt, sollte Ihnen bereits aus dem zweiten Kapitel bestens vertraut sein:

```
Rsrc_Gaddr()
```

So, die Adresse der Baumstruktur hätten wir, was fehlt ist noch die Routine, die unsere Menüleiste auf den Monitor bringt. Sie lautet:

```
Menu_Bar(Menu_Adr)
```

Jetzt wird es ein wenig komplizierter. Zur Verwaltung eines Formulares sieht GEM die im AES implementierte Routine FORM_DO vor, zur Verwaltung von Menüleisten ist diese Funktion jedoch nicht zu gebrauchen. Vielmehr existieren im Betriebssystem - genauer im GEM - des Atari ST bestimmte Funktionen, die warten, bis ein bestimmtes Ereignis (Event) eingetreten ist. Sobald dies der Fall ist, wird der Application mit-

geteilt, welches Ereignis eingetreten ist, die dann daraus ihre Folgerungen zu ziehen hat. Wie gesagt, es stehen eine ganze Menge verschiedener Funktionen zur Verfügung, die auf das Eintreten bestimmter Ereignisse warten. Zur Verwaltung von Menüleisten wird eine Event-Funktion aus dieser Sammlung verwendet, die eine Nachricht in Stringform hinterläßt, welches Ereignis gerade eingetreten ist. Sie lautet:

Evnt_Mesag(Nachricht$)

Ist ein Ereignis aufgetreten, so hinterläßt das AES eine entsprechende Nachricht im Rückgabeparameter Nachricht$ (Message Pipe). Dieser hat bei einem Menü-Ereignis folgenden Aufbau:

| Byte | Inhalt |
|---|---|
| 1+2 | Wert '10' als Identifikationsnummer für ein Menü-Ereignis (Mn_Selected) |
| 7+8 | Objektnummer der Menütitels |
| 9+10 | Objektnummer des angewählten Eintrages |

Der Einfachheit halber wandeln wir nach dem Vorliegen des Ereignisses die Nachrichtenzeichenkette in ein Array, das die einzelnen Punkte der Message festhält. Das Umwandeln ist kein Problem, da das Format der Nachricht in einer wie für CVI() geschaffenen Form vorliegt. Mit einer FOR...NEXT-Schleife können wir die jeweils zwei Byte in Anspruch nehmenden Datenpakete in einen Integerwert wandeln:

```
FOR T%= 0 TO 4
   Ereignis(T%)= CVI( MID$(Nachricht$,T%*2+1,2))
NEXT T%
```

Jetzt geht es ans Vergleichen! Zuerst ist einmal zu überprüfen, ob überhaupt ein Menü-Ereignis vorliegt, da die Funktion auch noch andere Ereignisse auswerten kann. Als Identifikation eines solchen Events dient der Wert 10, der im ersten Eintrag (Index 0) des Ereignis-Arrys festgehalten wird:

```
IF Ereignis(0) = 10 THEN .... (Menüereignis!)
```

Steht zweifelsfrei fest, daß es sich um das gewünschte Ereignis handelt, können die einzelnen Menü-Einträge abgefragt werden. Die Objektnummer des angeklickten Eintrages steht ebenfalls im Ereignisarray, jedoch an Position 5 (Index 4). Für die einzelnen Vergleiche empfiehlt es sich wieder die vom RCS gelieferte Header-Datei in das Programm einzubinden und anschließend die Variablennamen zur Abfrage heranzuziehen:

```
IF Ereignis(0) = 10 THEN
   IF Ereignis(4) = Minfo THEN
      <Reaktion auf dieses Ereignis>
      <z.B. Sprung in ein Unterprogramm>
   ENDIF
   '
   IF Ereignis(4) = Mquit THEN
      .....
      .....
   ENDIF
ENDIF
```

Da der Eintrag in der Menüleiste nach einem erfolgten Menü-Aufruf selektiert bleibt, muß er von uns noch zurückgesetzt werden. Auch dazu existiert in ST-BASIC ein GEM-Aufruf:

```
Menu_Tnormal(Objektnummer,1)
```

versetzt den entsprechenden Titel wieder in einen nicht selektierten Zustand. Die Objektnummer des zum aufgerufenen Eintrag gehörenden Titels steht im Ereignisarray an vierter Position (Index 3):

```
Menu_Tnormal(Ereignis(3),1)
```

Die Verwaltung mitsamt der Menü-Auswahl wird wieder in eine Schleife eingebunden, damit auch mehrmalige Menü-Aufrufe möglich sind. Da es sich um eine nichtabweisende Schleife handeln muß (zumindest ein Menü-Aufruf muß ja gestattet sein), verwendet man dazu die REPEAT...UNTIL-Schleife, deren Abbruchbedingung dann erfüllt sein muß, wenn der Menü-Eintrag Beenden oder Programm verlassen aktiviert wurde:

```
Ende = 0
REPEAT
   ....
   < warten bis Nachricht im Buffer vorliegt)
   ....
   IF Ereignis(0) = 10'  Menü-Eintrag angewählt?
      ....
      ....
     ....
      IF Ereignis(4) = Mquit 'Schleife verlassen
         Ende = 1
      ENDIF
   ' Menütitel wieder in Normaldarstellung bringen
Menu_Tnormal(Ereignis(3),1)
   ENDIF
UNTIL Ende ' wiederhole bis Ende = '1'
```

Üblicherweise wird vorm endgültigen Programmabbruch noch eine Sicherheitsabfrage in das Programm eingebaut, damit das Programm nicht ungewollt verlassen wird und Daten verloren gehen können. Form_Alert liefert eine solche Sicherheitsabfrage:

```
Möchten Sie das Programm wirklich verlassen? (J/N)
```

In einer Variablen (z.B. Ret%) wird nach Aufruf der Alertbox zurückgegeben, welcher Button vom Benutzer angeklickt wurde. Demzufolge könnte die Abfrage dann lauten:

```
IF Ret% = 1 THEN
   Ende = 1
ELSE
   Ende = 0
ENDIF
```

Besser und kürzer ist aber in jedem Fall die folgende Abfrage:

```
Ende = Ret% = 1
```

Die liefert das gleiche Ergebnis, besteht jedoch aus nur einer Zeile! Wie funktioniert diese Zeile? Schauen wir uns dazu (zunächst) einmal die rechte Hälfte an: Das rechte Gleichheitszeichen wird als Vergleichsoperator eingesetzt. Und der als Ergebnis dieses Vergleichs (Ret% = 1?) resultierende Wahrheitswert

wird der Variablen Ende zugewiesen, also falsch wenn Ret% ungleich 1 und wahr wenn Ret% gleich 1 war. Dort ist er gut aufgehoben, da sein Inhalt die Bedingung für das Schleifenende darstellt. Tja, man kann eben so oder so an ein Problem herangehen! Aber eleganter als mit der zweiten Methode geht es bestimmt nicht mehr!

Ein Eintrag innerhalb des Pull-Down-Menüs kann auch mit einem vorangestellten Haken versehen werden. Auf diese Weise kann z.B. angezeigt werden, daß bestimmte Einstellungen (z.B. Einfügemodus in einem Texteditor) aktiv sind. Um einen Eintrag mit einem solchen schmückenden Beiwerk zu versehen, genügt der Aufruf der Funktion:

```
Menu_Icheck(Nummer%,Flag)
```

Nummer% bezeichnet die Nummer des Menü-Eintrages innerhalb der Baumstruktur, Flag kann zwei Werte annehmen:

0 Ein voranstehender Haken wird gelöscht.
1 Haken vor diesen Menü-Eintrag setzen

Ferner sollte es möglich sein, bestimmte Menüeinträge zu disablen, also vor einer Aktivierung zu bewahren. Es gäbe nämlich keinerlei Sinn, eine Datei bearbeiten zu wollen, die noch gar nicht geladen worden ist! Das übernimmt die Funktion

```
Menu_Ienable(Nummer%,Flag)
```

wobei Nummer% wieder den Eintrag innerhalb der Objektstruktur bezeichnet und Flag bestimmt, ob der betreffende Eintrag aktiviert oder deaktiviert werden soll:

0 Menü-Eintrag deaktivieren (helle Darstellung)
1 Menü-Eintrag aktivieren

Jetzt aber das Beispielprogramm, anhand dessen Sie die Verwaltung von Pull-Down-Menüs leicht nachvollziehen können. Es wird die Resourcedatei MENUE.RSC benötigt, in der die Baumstruktur des Menüs enthalten ist:

```
0  '*********************************************************************
1  '*                          MENUE.BAS                                *
2  '*-------------------------------------------------------------------*
3  '* Autor: Michael Maier    Version: 1.00     Datum: 16.09.1988 *
4  '*      Ein Programm aus dem 'GROSSEN ST-BASIC BUCH'                 *
5  '*        (C) 1988 by DATA BECKER GmbH Düsseldorf                    *
6  '*********************************************************************
7  '
8  '
9  ' Rumpfprogramm zur Demonstration der Menüleistenprogrammierung
10 '                          in ST-BASIC
11 '*********************************************************************
12 '
13 '
14 ' zuerst einmal die Variablen mit Werten versorgen ...
15 '
16 Leiste%L=0
17 Minfo%L=7'   STRING in tree LEISTE
18 Mload%L=16'  STRING in tree LEISTE
19 Msave%L=17'  STRING in tree LEISTE
20 Mhaken%L=19' STRING in tree LEISTE
21 Mquit%L=21'  STRING in tree LEISTE
22 '
23 ' Variable für Schleifenende auf den Wert Null setzen
24 ' und Array für Ereignis dimensionieren
25 '
26 Ende%L=0
27 DIM Ereignis%L(4)
28 '
29 ' dann die übliche GEM-Zeremonie
30 '
31 Appl_Init
32 Rsrc_Load("MENUE.RSC",Ret%)
33 IF Ret%=0 THEN
34    FORM_ALERT (1,"[3][Fatal Error!][ Abbruch ]")
35    Appl_Exit
36    END
37 ENDIF
38 '
39 ' Wurzeladresse des Objektbaumes (Menu) bestimmen
40 '
41 Rsrc_Gaddr(0,0,Menu_Adr%L)
42 '
43 ' und die Menüleiste darstellen
44 '
45 Menu_Bar(Menu_Adr%L)
```

```
46 Graf_Mouse(0)' Maus in Pfeilform (sicher ist sicher!)
47 '
48 ' Die nun folgende Schleife kümmert sich um die Verwaltung
49 ' der soeben dargestellten Menüleiste
50 '
51 REPEAT
52 '
53 ' warten bis ein Ereignis im sogenannten 'Message-Buffer'
54 ' vorliegt. Wurde die Menüleiste in irgendeiner Form an-
55 ' gewählt, so besitzt der resultierende String folgenden
56 ' Aufbau:
57 '         Wort 0 => '10' (im Format LOW-HIGHbyte)
58 '         Wort 3 => Nummer der Leiste, in der sich der an-
59 '                   gewählte Eintrag befindet
60 '         Wort 4 => Objektnummer des angewählten Eintrages
61 '
62   Evnt_Mesag(Me$)
63   '
64   ' Jetzt das aufgetretene Ereignis auswerten
65   '
66   FOR T%=0 TO 4
67      Ereignis%L(T%)= CVI( MID$(Me$,T%*2+1,2))
68   NEXT T%
69   '
70   ' Ereignis in der Menüleiste ?
71   '
72   IF Ereignis%L(0)=10 THEN
73      '
74      ' Die Objektnummer befindet sich dann in 'Ereignis(4)'
75      '
76      IF Ereignis%L(4)=Minfo%L THEN
77         FORM_ALERT (1,"[1][Ein  Demo  von¦Michael  Maier][ OK ]")
78      ENDIF
79      '
80      IF Ereignis%L(4)=Mload%L THEN
81         FORM_ALERT (1,"[1][Dann tun Sie's doch!][ OK ]")
82      ENDIF
83      '
84      IF Ereignis%L(4)=Msave%L THEN
85         FORM_ALERT (1,"[2][Welche Datei?][ Keine! ]")
86      ENDIF
87      '
88      IF Ereignis%L(4)=Mhaken%L THEN
89      '  Inhalt der Variablen 'umdrehen'
90         IF Haken%=1 THEN
91            Haken%=0
```

```
92          ELSE
93             Haken%=1
94          ENDIF
95          '
96          ' und den Haken entsprechend setzen bzw. löschen
97          '
98          Menu_Icheck(Mhaken%L,Haken%)
99          '
100       ENDIF
101       '
102       IF Ereignis%L(4)=Mquit%L THEN
103          FORM_ALERT (2,"[2][Programm beenden?][Ja|Nein]",Ret%)
104          ' ein bißchen zaubern ist erlaubt ...
105          ' zuerst wird nämlich einmal überprüft, ob in Ret% der
106          ' Wert '1' enthalten ist. Der Wahrheitswert dieser
107          ' Operation wird der Variablen 'Ende' zugewiesen, die
108          ' für den Schleifenabbruch verantwortlich ist.
109          Ende%L=Ret%=1
110       ENDIF
111       '
112       ' hier können weitere Abfragen nach angewählten
113       ' Menüeinträgen erfolgen .....
114       '
115       ' Menütitel wieder in Normaldarstellung bringen
116       '
117       Menu_Tnormal(Ereignis%L(3),1)
118       '
119    ENDIF
120 UNTIL Ende%L
121 '
122 ' GEM wieder abmelden und Programm beenden
123 '
124 Appl_Exit
125 END
126 '
127 '************************************************************
128 '*          hier folgt dann die Libery GEMLIB.BAS           *
129 '************************************************************
130 '
131 ' ..........
132 ' ..........
133 ' ..........
```

## 6.5 Eine eigene File-Selector-Box

Obwohl die GEM-eigene File-Selector-Box eine ganze Reihe von Möglichkeiten bietet, ist sie dennoch relativ umständlich zu bedienen. Möchten Sie nur das Laufwerk wechseln, ist dazu zuerst einmal das Abändern der Laufwerksbezeichnung im Pfad notwendig. Anschließend muß noch das Schließfeld mit der Maus angeklickt werden, damit die soeben durchgeführte Änderung auch übernommen wird. Ähnlich ergeht es jedem, der die angezeigten Dateien begrenzen will, indem er den Such-Extender ändert.

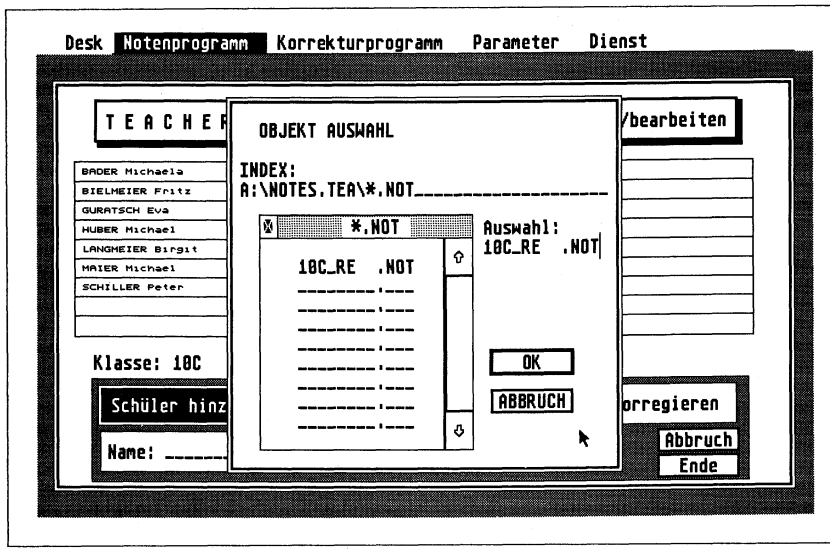



*Abb. 6.6: File-Selector-Box*

Ferner wäre auch eine Infozeile nicht zu mißachten, in der der Benutzer erfährt, wozu er überhaupt eine Datei auswählt: Datei laden ..., Datei speichern ... usw. machen ein Programm wesentlich anwendungsfreundlicher. All diese Dinge sollen bei der eigenen File-Selector-Box berücksichtigt werden. Beim Aufruf der Funktion sind folgende Parameter anzugeben:

- Pfadname mit vorangestellter Laufwerksbezeichnung,
- Dateiname, der als Voreinstellung erscheinen soll,
- Infozeile der Box (Datei laden ...).

Als Rückgabewerte liefert die Box:

- neuen Pfadnamen,
- ausgewähler Dateiname,
- Extender,
- gedrückte Taste (Abbruch oder OK).

Da das Listing hinreichend ausführlich dokumentiert ist, erübrigt sich eigentlich eine nochmalige Erklärung an dieser Stelle. Trotzdem möchte ich ihre Bedienung noch einmal kurz erläutern:

- Es können nur angeschlossene Laufwerke (Betriebssystemroutine!) angeklickt werden. Dann wird der Pfadname entsprechend korrigiert und neu gesetzt.

- Zum Ändern des Extenders kann einer, der acht zur Auswahl stehenden Boxen angeklickt werden. Der Inhalt im Fenster wird entsprechend korrigiert. Benötigen Sie einen nicht vorhandenen Extender, so ändern Sie bitte den Extender im Pfadnamen und klicken in das Feld oberhalb der Auswahlbox, in dem die aktuelle Extension angezeigt wird. Die neue Extension wird übernommen und gleichzeitig in den letzten der Extension-Buttons eingetragen. Dort steht er dann für weitere Aufrufe zur Verfügung. Dies funktioniert jedoch nur solange, wie die Länge der Extension 5 Zeichen nicht übersteigt: (*.BAK,*.DOC,*.H).

- Da ein Fehler im Betriebssystem vorliegt, sollte die Benutzung des Unterstrichs im Pfadnamen tunlichst vermieden werden, damit der Computer nicht abstürzt. Beim Blitter-TOS wurde dieser Fehler jedoch schon behoben!

- Um die ganze Sache nicht noch komplizierter zu machen, als sie ohnehin schon ist, werden in Unterverzeichnissen Ordner - ähnlich wie beim Befehl FILES des ST-BASICs - durch

ein, bzw. zwei Punkte statt des Ordnernamens dargestellt; sie können jedoch nicht ausgewählt werden, da Sie sich bereits in diesem Ordner befinden.

So, nachdem die Bedienung der Box jetzt geklärt sein dürfte, folgt das Listing. Die einzelnen Routinen wurden entweder schon einmal behandelt oder sie werden innerhalb des Listings - sofern Sie etwas komplizierter sind - kommentiert:

```
1  '*********************************************************************
2  '*                          FILSEL.BAS                               *
3  '*-------------------------------------------------------------------*
4  '* Autor: Michael Maier    Version: 1.00     Datum: 01.10.1988 *
5  '*      Ein Programm aus dem 'GROSSEN ST-BASIC BUCH'                 *
6  '*        (C) 1988 by DATA BECKER GmbH Düsseldorf                    *
7  '*********************************************************************
8  '
9  '  ** Dieses Programm verwaltet die eigene File-Selector-Box **
10 '  ***************************************************************
11 ' zuerst das Hauptprogramm ...
12 '
13 '
14 DIM Name$(130)' enthält die Directoryeinträge
15 '
16 ' die einzelnen Variablen des RSC initialisieren ...
17 '
18 Filsel%L=0
19 Fsinfo%L=1'    BOXTEXT in tree FILSEL
20 Fspfad%L=3'    FTEXT in tree FILSEL
21 Fscloser%L=5  BOXCHAR in tree FILSEL
22 Fsextend%L=6'  BOXTEXT in tree FILSEL
23 Fsup%L=7'      BOXCHAR in tree FILSEL
24 Fsdown%L=8'    BOXCHAR in tree FILSEL
25 Fschiene%L=9'  BOX in tree FILSEL
26 Fsslider%L=10' BOX in tree FILSEL
27 Fsdatei0%L=11' TEXT in tree FILSEL
28 Fsdatei1%L=12' TEXT in tree FILSEL
29 Fsdatei2%L=13' TEXT in tree FILSEL
30 Fsdatei3%L=14' TEXT in tree FILSEL
31 Fsdatei4%L=15' TEXT in tree FILSEL
32 Fsdatei5%L=16' TEXT in tree FILSEL
33 Fsdatei6%L=17' TEXT in tree FILSEL
34 Fsdatei7%L=18' TEXT in tree FILSEL
35 Fsdatei8%L=19' TEXT in tree FILSEL
36 Fsdatei9%L=20' TEXT in tree FILSEL
```

```
Fsname%L=22'   TEXT in tree FILSEL
Fsdrva%L=24'   BOXCHAR in tree FILSEL
Fsdrvb%L=25'   BOXCHAR in tree FILSEL
Fsdrvc%L=26'   BOXCHAR in tree FILSEL
Fsdrvd%L=27'   BOXCHAR in tree FILSEL
Fsdrve%L=28'   BOXCHAR in tree FILSEL
Fsdrvf%L=29'   BOXCHAR in tree FILSEL
Ext1%L=31'     BOXTEXT in tree FILSEL
Ext2%L=32'     BOXTEXT in tree FILSEL
Ext3%L=33'     BOXTEXT in tree FILSEL
Ext4%L=34'     BOXTEXT in tree FILSEL
Ext5%L=35'     BOXTEXT in tree FILSEL
Ext6%L=36'     BOXTEXT in tree FILSEL
Ext7%L=37'     BOXTEXT in tree FILSEL
Ext8%L=38'     BOXTEXT in tree FILSEL
Fsabbruc%L=44' BUTTON in tree FILSEL
Fsreturn%L=45' BUTTON in tree FILSEL
'
'
Appl_Init
PRINT @(0,30);"File-Selector-Box"
' Ressource laden
Rsrc_Load("FSELECT.RSC",Dummy%)
' alles in Ordnung ?
IF Dummy%=0 THEN
   FORM_ALERT (1,"[3][RSC konnte nicht¦geladen werden !!][Ende]")
   Appl_Exit
   END
ENDIF
' dann Adresse ermitteln
Rsrc_Gaddr(0,0,Tree%L)
'
Info$=" Datei laden ... "
Fil_Sel("C:\","TEST.PRG",Info$,Path$,Name$,Post$,Taste%)
'
Appl_Exit
END
'
'
DEF PROC Fil_Sel(Path$,Name$,Inf$,R Path$,R Na$,R Ext$,R Key%)
LOCAL T%,Anzahl%,Anker%,Ret%
Ext$="*.*"
'
Form_Center(Tree%L,Xobj%L,Yobj%L,Wobj%L,Hobj%L)
Form_Dial(0,Xobj%L,Yobj%L,Wobj%L,Hobj%L)
Form_Dial(0,Xobj%L,Yobj%L,Wobj%L,Hobj%L)
```

```
'
Put_Text(Fsinfo%L,Info$+ CHR$(0))
Put_Text(Fspfad%L,Path$+Ext$+ CHR$(0))
CHDIR Path$
Pos%= INSTR(Name$,".")
IF Pos%<>0 AND Pos%<9 THEN
   Name$= LEFT$(Name$,Pos%-1)+ SPACE$(9-Pos%)+ MID$(Name$,Pos%+1)
ENDIF
Put_Text(Fsname%L,Name$+ CHR$(0))
Put_Text(Fsextend%L, CHR$(32)+Ext$+ CHR$(32)+ CHR$(0))
'
FOR T%=0 TO 7
   IF FN Proof_Sel%L(Ext1%L+T%) THEN
     Get_Text(Ext1%L+T%,Ext$)
     Put_Text(Fspfad%L,Path$+Ext$+ CHR$(0))
     EXIT
   ENDIF
NEXT T%
'
' Directory einlesen und Filezahl zurückgeben
'
Anker%=1
Directory(Ext$,Anzahl%)
Inhalt(Anker%,Anzahl%)
'
Objc_Draw(0,8,Xobj%L,Yobj%L,Wobj%L,Hobj%L,Tree%L)
'
' Jetzt die nicht angeschlossenen Laufwerke disablen
'
BIOS (Dummy%,10)
FOR T%=0 TO 5
   ' Bit in Drive-Map gesetzt? nein => BIT(..) = 0
   IF BIT(T%,Dummy%)=0 THEN
      ' dann Box disablen
      Enable(Fsdrva%L+T%,0)
      Disable(Fsdrva%L+T%,0)
   ELSE
      ' enablen (sicher ist sicher)
      Enable(Fsdrva%L+T%,1)
      Disable(Fsdrva%L+T%,1)
   ENDIF
NEXT T%
' zuerst vorsichtshalber alle Laufwerke deselektieren
FOR T%=0 TO 5
  Select(Fsdrva%L+T%,0)
NEXT T%
```

```
' dann aktuelles Laufwerk laut Pfad selektieren
Select(Fsdrva%L+ ASC(Path$)-65,1)
CHDIR Path$
'
' das Laufwerk könnte auch mit dieser GEMDOS-Funktion
' umgestellt werden, aber der liebe Pfad ...
' GEMDOS (,14, ASC(Path$)-65)
'
REPEAT
  ' Kontrolle des Formulars an GEM übergeben
  Form_Do(Fsname%L,Tree%L,Ret%)
  ' Doppelklick verhindern
  Ret%=Ret% AND $7FFF
  '
  ' Datei angeklickt
  '
  IF Ret%>=Fsdatei0%L AND Ret%<=Fsdatei9%L THEN
     Get_Text(Ret%,Datei$)
     IF Datei$<>"" THEN
       IF ASC(Datei$)=7 THEN
         ' Ordner entsprechend behandeln
         Get_Text(Fspfad%L,Path$)
         Pos%= INSTR( MIRROR$(Path$),"\")
         Path$= LEFT$(Path$, LEN(Path$)-Pos%+1)
         Datei$= MID$(Datei$,3, INSTR(Datei$, CHR$(0))-3)
         ' Spaces aus dem Ordnernamen entfernen
         WHILE INSTR(Datei$, CHR$(32))
            Pos%= INSTR(Datei$, CHR$(32))
            Datei$= LEFT$(Datei$,Pos%-1)+ MID$(Datei$,Pos%+1)
         WEND
         ' korrekter Pfad schließt mit einem Backslash: '\'
         Path$=Path$+Datei$+"\"
         ' Pfad setzen und in Box eintragen
         CHDIR Path$
         Put_Text(Fspfad%L,Path$+Ext$+ CHR$(0))
         ' neues Inhaltsverzeichnis ermitteln
         Directory(Ext$,Anzahl%)
         Inhalt(Anker%,Anzahl%)
         Objc_Draw(0,8,Xobj%L,Yobj%L,Wobj%L,Hobj%L,Tree%L)
         '
       ELSE
         ' normale Datei => in Feld 'Dateiname' kopieren
         Pos%= INSTR(Datei$,".")
         ' der Punkt macht Schwierigkeiten, also weg damit!
         IF Pos%<>0 THEN
            Datei$= LEFT$(Datei$,Pos%-1)+ MID$(Datei$,Pos%+1)
```

```
175         ENDIF
176         Put_Text(Fsname%L, MID$(Datei$,3))
177         Objc_Draw(Fsname%L,0,Xobj%L,Yobj%L,Wobj%L,Hobj%L,Tree%L)
178       ENDIF
179     ENDIF
180   ENDIF
181   '
182   ' Laufwerk umstellen
183   '
184   IF Ret%>=Fsdrva%L AND Ret%<=Fsdrvf%L THEN
185     Get_Text(Fspfad%L,Path$)
186     Path$= LEFT$(Path$, LEN(Path$)- INSTR( MIRROR$(Path$),"\")+1)
187     Path$= CHR$(65+Ret%-Fsdrva%L)+ MID$(Path$,2)
188     Put_Text(Fspfad%L,Path$+Ext$+ CHR$(0))
189     CHDIR Path$
190     ' statt 'CHDIR Path$' kann auch eine GEMDOS-Funktion
191     ' aufgerufen werden: Dsetdrv. Als Argument muß ihr dann
192     ' die Nummer des neuen Laufwerks übergeben werden:
193     ' GEMDOS (,14, ASC(Path$)-65)
194     Directory(Ext$,Anzahl%)
195     Anker%=1
196     Inhalt(Anker%,Anzahl%)
197     Objc_Draw(0,8,Xobj%L,Yobj%L,Wobj%L,Hobj%L,Tree%L)
198   ENDIF
199   '
200   ' Neue Extension
201   '
202   IF Ret%>=Ext1%L AND Ret%<=Ext8%L THEN
203     Get_Text(Ret%,Ext$)
204     Ext$= LEFT$(Ext$, LEN(Ext$)-1)
205     Get_Text(Fspfad%L,Path$)
206     Path$= LEFT$(Path$, LEN(Path$)- INSTR( MIRROR$(Path$),"\")+1)
207     Put_Text(Fspfad%L,Path$+Ext$+ CHR$(0))
208     Put_Text(Fsextend%L, CHR$(32)+Ext$+ CHR$(32)+ CHR$(0))
209     CHDIR Path$
210     Directory(Ext$,Anzahl%)
211     Anker%=1
212     Inhalt(Anker%,Anzahl%)
213     Objc_Draw(0,8,Xobj%L,Yobj%L,Wobj%L,Hobj%L,Tree%L)
214   ENDIF
215   '
216   ' Slider angeklickt?
217   '
218   IF Ret%=Fsslider%L THEN
219     Graf_Slidebox(Fschiene%L,Fsslider%L,1,Tree%L,Pos%)
220     Pos%=Pos%+500\(Anzahl%-10)
```

```
221     Anker%=((Anzahl%-10)*Pos%\1000)+1
222     Inhalt(Anker%,Anzahl%)
223     Objc_Draw(0,8,Xobj%L,Yobj%L,Wobj%L,Hobj%L,Tree%L)
224   ENDIF
225   '
226   ' Directory scrollen (Pfeil nach oben)
227   '
228   IF Ret%=Fsup%L THEN
229     IF Anker%>1 THEN
230       Anker%=Anker%-1
231       Inhalt(Anker%,Anzahl%)
232       Objc_Draw(0,8,Xobj%L,Yobj%L,Wobj%L,Hobj%L,Tree%L)
233     ENDIF
234   ENDIF
235   '
236   ' Directory scrollen (Pfeil nach unten)
237   '
238   IF Ret%=Fsdown%L THEN
239     IF Anker%<Anzahl%-9 THEN
240       Anker%=Anker%+1
241       Inhalt(Anker%,Anzahl%)
242       Objc_Draw(0,8,Xobj%L,Yobj%L,Wobj%L,Hobj%L,Tree%L)
243     ENDIF
244   ENDIF
245   '
246   ' Schiene angeklickt => Slider verschieben
247   '
248   IF Ret%=Fschiene%L THEN
249     Graf_Mkstate(Dummy%,Pos%,Dummy%,Dummy%)
250     Objc_Offset(Fsslider%L,Tree%L,Ob_Ofx%L,Ob_Ofy%L)
251     IF Pos%>Ob_Ofy%L THEN
252       Anker%=Anker%+10
253       IF Anker%>Anzahl%-9 THEN
254         Anker%=Anzahl%-9
255       ENDIF
256     ELSE
257       Anker%=Anker%-10
258       IF Anker%<1 THEN
259         Anker%=1
260       ENDIF
261     ENDIF
262     Inhalt(Anker%,Anzahl%)
263     Objc_Draw(0,8,Xobj%L,Yobj%L,Wobj%L,Hobj%L,Tree%L)
264   ENDIF
265   '
266   ' Subdirectory verlassen (falls möglich)
```

```
'
IF Ret%=Fscloser%L THEN
   Get_Text(Fspfad%L,Path$)
   Pos%= INSTR( MIRROR$(Path$),"\")
   IF Pos%<>0 THEN
      Path$= LEFT$(Path$, LEN(Path$)-Pos%)
      Pos%= INSTR( MIRROR$(Path$),"\")
      IF Pos%<>0 AND Pos%<>3 THEN
         Path$= LEFT$(Path$, LEN(Path$)-Pos%)+"\"
         CHDIR Path$
         Put_Text(Fspfad%L,Path$+Ext$+ CHR$(0))
         Anker%=1
         Directory(Ext$,Anzahl%)
         Inhalt(Anker%,Anzahl%)
         Objc_Draw(0,8,Xobj%L,Yobj%L,Wobj%L,Hobj%L,Tree%L)
      ENDIF
   ENDIF
ENDIF
'
' neue Extension in letzte Extender-Box
'
IF Ret%=Fsextend%L THEN
   Get_Text(Fspfad%L,Path$)
   Path$= LEFT$(Path$, LEN(Path$)- INSTR( MIRROR$(Path$),CHR$(0)))
   Pos%= INSTR( MIRROR$(Path$),"\")
   ' nicht Backslash am Pfadende!
   IF Pos%< LEN(Path$) THEN
      Ext$= MID$(Path$, LEN(Path$)-Pos%+2)
      FOR T%=0 TO 7
         Select(Ext1%L+T%,0)
      NEXT T%
      IF LEN(Ext$)<6 THEN
         Put_Text(Fsextend%L, CHR$(32)+Ext$+ CHR$(32)+ CHR$(0))
         Put_Text(Ext8%L,Ext$+ CHR$(0))
         Select(Ext8%L,1)
         Directory(Ext$,Anzahl%)
         Anker%=1
         Inhalt(Anker%,Anzahl%)
      ELSE
         Ext$="*.*"
         Path$= LEFT$(Path$, LEN(Path$)-Pos%+1)
         Put_Text(Fspfad%L,Path$+Ext$+ CHR$(0))
         Put_Text(Fsextend%L, CHR$(32)+Ext$+ CHR$(32)+ CHR$(0))
         Select(Ext1%L,1)
      ENDIF
   ENDIF
```

```
313   Objc_Draw(0,8,Xobj%L,Yobj%L,Wobj%L,Hobj%L,Tree%L)
314   ENDIF
315   '
316 UNTIL Ret%=Fsabbruc%L OR Ret%=Fsreturn%L
317 '
318 Get_Text(Fspfad%L,Path$)
319 Path$= LEFT$(Path$, LEN(Path$)- INSTR( MIRROR$(Path$), CHR$(0)))
320 '
321 IF Ret%=Fsabbruc%L THEN
322    Na$=""
323    Key%=0
324 ELSE
325    Key%=1
326    Get_Text(Fsname%L,Na$)
327    Pos%= INSTR(Na$, CHR$(0))
328    ' falls kein Name vorhanden ist => Leerstring zurück
329    IF Pos%=1 THEN
330      Na$=""
331    ELSE
332      Na$= LEFT$(Na$, LEN(Na$)-1)
333      ' Ein Punkt als Extenderseperator darf nicht fehlen
334      IF LEN(Na$)>8 THEN
335         Na$= LEFT$(Na$,8)+"."+ MID$(Na$,9)
336      ENDIF
337      WHILE INSTR(Na$, CHR$(32))
338         Pos%= INSTR(Na$, CHR$(32))
339         Na$= LEFT$(Na$,Pos%-1)+ MID$(Na$,Pos%+1)
340      WEND
341    ENDIF
342 ENDIF
343 '
344 RETURN
345 '
346 '
347 DEF PROC Directory(Extender$,R Anzahl%)
348 LOCAL T%,Buffer$= SPACE$(44),Buf_Adr%L,Ext_Adr%L,Dummy%
349 LOCAL Pos%,Post$
350 FOR T%=0 TO 125
351   Name$(T%)=""
352 NEXT T%
353 T%=0
354 ' Adresse des DTA-Buffers
355 Buf_Adr%L= LPEEK( VARPTR(Buffer$))+ LPEEK( SEGPTR +28)
356 GEMDOS (,26, HIGH(Buf_Adr%L), LOW(Buf_Adr%L))
357 Ext_Adr%L= LPEEK( VARPTR(Extender$))+ LPEEK( SEGPTR +28)
358 '
```

```
' und jetzt den ersten Directoryeintrag holen
'
GEMDOS (Dummy%,78, HIGH(Ext_Adr%L), LOW(Ext_Adr%L),16)
IF Dummy%<0 THEN '  kein einziger Eintrag verhanden
   Anzahl%=0
   RETURN
ENDIF
' Dateityp feststellen und im Array festhalten
IF ASC( MID$(Buffer$,22,1))=16 THEN ' Ordner kennzeichnen!
   Name$(T%)= CHR$(7)+ CHR$(32)+ MID$(Buffer$,31,13)
ELSE
   Name$(T%)= CHR$(32)*2+ MID$(Buffer$,31,13)
ENDIF
' bis zum Nullyte übernehmen ... (für GEM !!!)
Name$(T%)= LEFT$(Name$(T%), INSTR(Name$(T%), CHR$(0)))
'
REPEAT
  T%=T%+1
  GEMDOS (Dummy%,79)
  ' erst wenn kein Eintrag mehr vorhanden => Schleife verlassen
  IF Dummy%<0 THEN
    EXIT
  ENDIF
  IF ASC( MID$(Buffer$,22,1))=16 THEN ' Ordner kennzeichnen!
     Name$(T%)= CHR$(7)+ CHR$(32)+ MID$(Buffer$,31,13)
  ELSE
     Name$(T%)= CHR$(32)*2+ MID$(Buffer$,31,13)
  ENDIF
  ' Name bis zum Nullyte übernehmen (für GEM !!!)
  Name$(T%)= LEFT$(Name$(T%), INSTR(Name$(T%), CHR$(0)))
UNTIL 0
Anzahl%=T%
' jetzt noch sämtliche Einträge auf die richtige Länge bringen
FOR T%=0 TO Anzahl%-1
   Pos%= INSTR(Name$(T%),".")
   ' Punkt vorhanden, aber Position nicht korrekt?
   IF Pos%<>0 AND Pos%<11 AND Pos%>3 THEN
      Post$= MID$(Name$(T%),Pos%)
      ' dann Leerzeichen zwischen Name und Extender einfügen
      Name$(T%)= LEFT$(Name$(T%),Pos%-1)+ SPACE$(10-Pos%+1)
      Name$(T%)=Name$(T%)+Post$
   ENDIF
NEXT T%
RETURN
'
'
```

```
405 DEF PROC Select(Nummer%,Flag%)
406 LOCAL Adr%L=Tree%L+24*Nummer%
407 IF Flag%=1 THEN
408    LOCAL Ob_State%L= WPEEK(Adr%L+10) OR 1
409 ELSE
410    LOCAL Ob_State%L= WPEEK(Adr%L+10) AND 254
411 ENDIF
412 Objc_Change(Nummer%,Ob_State%L,Xobj%L,Yobj%L,Wobj%L,Hobj%L,Tree%L)
413 RETURN
414 '
415 '
416 DEF PROC Enable(Nummer%,Flag%)
417 LOCAL Adr%L=Tree%L+24*Nummer%
418 IF Flag%=1 THEN
419    WPOKE Adr%L+8, WPEEK(Adr%L+8) OR 65
420 ELSE
421    WPOKE Adr%L+8, WPEEK(Adr%L+8) AND 190
422 ENDIF
423 RETURN
424 '
425 '
426 DEF PROC Disable(Nu%,Flag%)
427   LOCAL Adr%L=(Tree%L+24*Nu%)+10
428   IF Flag%=1 THEN
429     Objc_Change(Nu%, WPEEK(Adr%L) AND
247,Xobj%L,Yobj%L,Wobj%L,Hobj%L,Tree%L)
430   ELSE
431     Objc_Change(Nu%, WPEEK(Adr%L) OR
8,Xobj%L,Yobj%L,Wobj%L,Hobj%L,Tree%L)
432   ENDIF
433 RETURN
434 '
435 '
436 DEF PROC Put_Text(Nummer%,Text$)
437    ' zwerst einmal die Adresse des Textes ermitteln ...
438    LOCAL Adr%L=Tree%L+24*Nummer%
439    LOCAL Tedinfo%L= LPEEK(Adr%L+12)
440    LOCAL Text_Adr%L= LPEEK(Tedinfo%L)
441    LOCAL T%=0
442    ' und dann zeichenweise eintragen ...
443    WHILE T%< LEN(Text$)
444      POKE Text_Adr%L+T%, ASC( MID$(Text$,T%+1,1))
445      T%=T%+1
446    WEND
447 RETURN
448 '
```

```
449 '
450 DEF PROC Get_Text(Nummer%,R Text$)
451    LOCAL Adr%L=Tree%L+24*Nummer%
452    LOCAL Tedinfo%L= LPEEK(Adr%L+12)
453    LOCAL Text_Adr%L= LPEEK(Tedinfo%L)
454    LOCAL T%=-1
455    Text$=""
456    ' alle Zeichen bis zum Nullbyte abholen
457    ' und in String 'Text$' eintragen ...
458    REPEAT
459       T%=T%+1
460       Text$=Text$+ CHR$( PEEK(Text_Adr%L+T%))
461    UNTIL PEEK(Text_Adr%L+T%)=0
462    '
463 RETURN
464 '
465 '
466 DEF PROC Inhalt(Start%,Zahl%)
467   LOCAL T%,Note_1%L
468   IF Zahl%<=10 THEN
469      WPOKE (Tree%L+24*Fsslider%L)+18,0
470      WPOKE Tree%L+24*Fsslider%L+22, WPEEK(Tree%L+24*Fschiene%L+22)
471      FOR T%=0 TO Zahl%-1
472        Put_Text(FsdateiO%L+T%,Name$(T%))
473        IF MID$(Name$(T%),3,1)="." THEN
474           Enable(FsdateiO%L+T%,0)
475        ELSE
476           Enable(FsdateiO%L+T%,1)
477        ENDIF
478        Select(FsdateiO%L+T%,0)
479      NEXT T%
480      FOR T%=Zahl% TO 9
481        Put_Text(FsdateiO%L+T%, CHR$(0))
482        Enable(FsdateiO%L+T%,0)
483        Select(FsdateiO%L+T%,0)
484      NEXT T%
485   ELSE
486      Note_1%L= WPEEK(Tree%L+24*Fschiene%L+22)
487      WPOKE Tree%L+24*Fsslider%L+18,(Note_1%L*(Start%-1))/Zahl%
488      WPOKE Tree%L+24*Fsslider%L+22,(Note_1%L*10)/Zahl%
489      FOR T%=0 TO 9
490        IF Name$(Start%+T%)="" THEN
491           Put_Text(FsdateiO%L+T%, CHR$(0))
492           Enable(FsdateiO%L+T%,0)
493           Select(FsdateiO%L+T%,0)
494        ELSE
```

```
495           Put_Text(FsdateiO%L+T%,Name$(Start%+T%))
496           IF MID$(Name$(Start%+T%),3,1)="." THEN
497              Enable(FsdateiO%L+T%,0)
498           ELSE
499              Enable(FsdateiO%L+T%,1)
500           ENDIF
501           Select(FsdateiO%L+T%,0)
502        ENDIF
503     NEXT T%
504  ENDIF
505 RETURN
506 '
507 '
508 DEF FN Proof_Sel%L(Nummer%)
509 LOCAL Adr%L=Tree%L+24*Nummer%
510 IF WPEEK(Adr%L+10) AND 1=1 THEN
511    RETURN (-1)
512 ENDIF
513 RETURN (0)
514 '
```

## 6.6 Fenstertechnik

Eine der zweifellos überragensten Errungenschaften des GEM (also wird für die da kommenden Dinge die GEMLIB.BAS-Library benötigt!) ist die Möglichkeit zur Verwendung von Fenstern, die auch als Windows bezeichnet werden. Ob es sich um ein Textverarbeitungsprogramm oder eine Tabellenkalkulation handeln mag, stets können verschiedene Daten gleichzeitig im Speicher gehalten und auf dem Monitor dargestellt werden.

Da ein Fenster aus mehreren Elementen zusammengesetzt ist (zwei Slider für horizontale und vertikale Verschiebung, ein Schließ-, Maximalgrößen- und Größeneinstellfeld, die sich jeweils in einer Fensterecke finden lassen) muß ein Programm, das in irgendeiner Form mit Fenstern hantiert, in der Lage sein, verschiedene Ergeignisse - da ist dieses Wort schon wieder! - zu registrieren und entsprechend zu reagieren.

Zu diesem Zweck wird eine GEM-Routine eingesetzt, die schon einmal zu Ehren kam, als es darum ging, die Menüleiste zu verwalten:

```
Evnt_Mesag(Ereignis$)
```

Die auf den ersten Blick vielleicht unscheinbar anmutende Funktion hat es in sich: Sie ist nämlich (Gott sei Dank!) in der Lage, die unterschiedlichsten Ereignisse, die aufgetreten sind (Benutzer hat z.B. ein Fenster verschoben), an die Application zu melden. Doch ehe sie ihre Pflicht und Schuldigkeit tun darf, muß sich das Programm erst einmal um die Erstellung eines Fensters kümmern.

## Fensterkomponenten

Ein Fenster besteht aus den verschiedensten Komponenten, die alle in der folgenden Abbildung beschriftet sind:

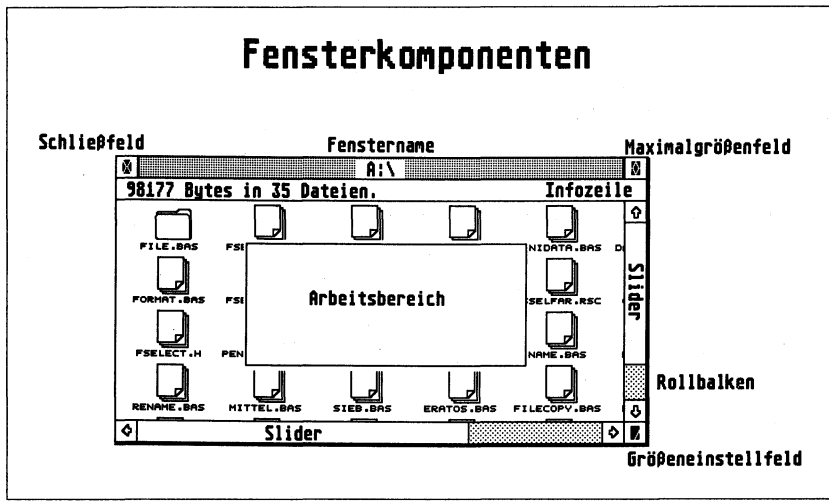



*Abb. 6.7: Bestandteile eines Fensters*

Da ein Fenster nicht unbedingt alle möglichen Bausteine besitzen muß (z.B. kein Schließ- oder Größeneinstellfeld), wird ein Fenster vor seiner ersten Darstellung auf dem Monitor erst einmal genauer spezifiziert. Dies geschieht über den Befehl

```
Wind_Create(kind,xmax,ymax,wmax,hmax,handle)
```

Kind trägt dabei die einzelnen Bestandteile des Fensters in sich, indem für jedes Element ein Bit reserviert ist. Je nachdem, ob dieses Bit gesetzt oder nicht gesetzt ist, wird das zugehörige Element im Window enthalten sein:

| Bit | verantwortlich für | Name | Wert |
|---|---|---|---|
| 0 | Titelbalken | Name | 1 |
| 1 | Löschfeld | Close | 2 |
| 2 | Maximalgrößenfeld | Full | 4 |
| 3 | Fenster verschiebbar | Move | 8 |
| 4 | Infozeile vorhanden | Info | 16 |
| 5 | Größeneinstellfeld | Size | 32 |
| 6 | Pfeil hoch | Uparrow | 64 |
| 7 | Pfeil runter | Dnarrow | 128 |
| 8 | Vertikaler Schieber | Vslide | 256 |
| 9 | Pfeil links | Ffarrow | 512 |
| 10 | feil rechts | Rtarrow | 1024 |
| 11 | horizontaler Schieber | Hslide | 2048 |

Die einzelnen Komponenten, aus denen sich das Window zusammensetzen soll, können laut obiger Tabelle einfach zusammengestellt werden, indem die einzelnen Werte der rechten Spalte addiert werden, und das resultierende Ergebnis für Kind eingesetzt wird. Möchten Sie einem Fenster alle Komponenten spendieren, erreichen Sie das auch mit der headezimalen Schreibweise für Kind

```
$FFF
```

wobei alle Bits (sogar noch ein paar mehr, aber das stört nicht!) gesetzt sind.

Die nächsten vier Parameter, nämlich xmax, ymax, hmax und wmax geben die maximale Größe an, die das Fenster annehmen kann. Diese sind abhängig von der Bildschirmauflösung. Aus

diesem Grund empfiehlt es sich, zuerst einmal die maximale Größe des Bildschirms abzufragen. Da der Monitor des Atari ST gleich einem Fenster behandelt wird, kann dies mit einen GEM-Befehl geschehen, der die gesuchten Werte liefert:

```
Wind_Get(Handle, Gfield, xmax, ymax, wmax, hmax)
```

Als Handle (darauf kommen wir später noch einmal zurück) muß der Wert 0 eingesetzt werden, damit die maximale Größe des Bildschirms (Fenster 0, deshalb auch dieser Wert als Handle) zurückgegeben wird. Zusätzlich können mittels dieser Funktion die verschiedensten Parameter abgefragt werden. Welchen Parameter Sie wünschen, muß in Gfield angegeben werden:

| Gfield | liefert als Ergebnis die Koordinaten |
|---|---|
| 4 | des Arbeitsbereichs auf dem Bildschirm |
| 5 | der Gesamtfenstergröße |
| 6 | des vorhergehenden Fensters |
| 7 | in der maximalen Fenstergröße |
| 11 | des ersten Rechtecks innerhalb des Fensters |
| 12 | des nächsten Rechtecks innerhalb des Fensters |

Für unsere Zecke möchten wir die Koordinaten des Arbeitsbereiches auf dem Monitor erfragen und geben deshalb als Parameter in Gfield den Wert 4 an, so daß der gesamte Funktionsaufruf nun lautet:

```
Wind_Get(0,4,xmax,ymax,wmax,hmax)
```

Nach dem Funktionsaufruf findet sich die maximal erlaubte Fenstergröße, die wie gesagt von der Bildschirmauflösung abhängig ist, in den letzten vier Variablen wieder. Diese setzt man jetzt beim Aufruf von Wind_Create() ein.

Der letzte Parameter ist das sogenannte Window-Handle oder einfach nur Handle. Was hat es damit auf sich? Jedes Fenster muß vom GEM eindeutig identifizierbar sein, da mehrere Fenster gleichzeitig geöffnet werden können. Zur Identifikation erhält nun jedes Fenster eine bestimmte Nummer, das Handle, anhand dessen dann ein bestimmtes Fenster angesprochen werden

kann. Dieses Handle teilt GEM der Application nach dem Aufruf des Create-Befehls mit und wird für so ziemlich alle folgenden Funktionsaufrufe immer wieder benötigt.

So, ein Fenster wäre damit - fast - fertig definiert. Eine Kleinigkeit fehlt noch: Jedes Fenster kann einen Namen und eine Infozeile tragen. Mit den bisherigen Befehlen ist es uns jedoch nicht möglich, diese einem Fenster zu verpassen. Dazu wird ein anderer Befehl benötigt:

```
Wind_Set(Handle,Sflied,Sline$,Adresse)
```

Unter Handle hat die Identifikationsnummer des Fensters zu stehen, dem die Daten draufgedrückt werden sollen. Sfield bestimmt den Paramter, der gesetzt werden soll:

| Sfield | setzt Parameter |
|---|---|
| 2 | Fenstername |
| 3 | Infozeile |

Der Inhalt, den die Infozeile oder der Fenstername bekommen soll, besteht aus einer Zeichenkette, die in Sline$ vom Computer erwartet wird. Da GEM seine Eigentümlichkeiten hat, wird zusätzlich noch eine Adresse benötigt, in der der Name des Fensters sowie die (neue) Infozeile abgelegt werden kann. Diese Adresse besorgt man sich am besten dadurch, daß man einen Speicherplatz mit dem Befehl

```
Adresse = MEMORY(Anzahl-Byte)
```

in gewünschter Menge reservieren läßt. Der so reservierte Platz - dessen kann man sich sicher sein - wird ab sofort nicht mehr vom Computer in irgendeiner Form angetastet. Gleichzeitig liefert die Funktion auch noch (wie schön!) die Adresse, ab der der reservierte Platz zu sinden ist. 70 Byte sollten für's erste einmal reichen:

```
Adresse = MEMORY(70)
Wind_Set(Handle,2,"Fenstername",Adresse)
```

Diese beiden Zeilen sorgen nun dafür, daß zuerst einmal Speicherplatz (in Hülle und Fülle) bereitgestellt und anschließend der Name dort eingetragen wird. Das Fenster hat somit einen Namen erhalten. Wind_Set existiert in ST-BASIC noch öfters, jedoch mit unterschiedlicher Parameterzahl:

```
Wind_Set(Handle,Sfield,Sw1)
```

legt die Fenster-Parameter fest:

| Sfield | ändert Parameter |
|---|---|
| 1 | legt die bei 'Wind_Create' angegebenen Komponenten neu fest |
| 8 | ändert die Position von Hslide |
| 9 | ändert die Position von Vslide |
| 15 | ändert die relative Größe von Hslide |
| 16 | ändert die relative Größe von Vslide |

Die benötigten Daten werden jeweils in der Variablen Sw1 übergeben. Da GEM nur jeweils ein Fenster in einem aktivierten Zustand verwalten kann, kann ein neues Fenster per

```
Wind_Set(Handle)
```

aktiviert werden. Das bisher aktive Fenster wird inaktiv.

### Ereignisse überprüfen

Nach all dieses Vorreden ist es soweit, der Befehl Evnt_Mesag() darf in Aktion treten. Auch diesmal wieder befindet er sich in einer Schleife:

```
REPEAT
   Evnt_Mesag(Ereignis$)
   .....
   .....
   < auf die verschiedensten Ereignisse überprüfen>
   ......
   ......
UNTIL <Abbruchbedingung erfüllt>
```

Nachdem GEM den Befehl Evnt_Mesage erhalten hat, wird es warten, bis irgendein Ereignis auftritt, und dieses dann in Ereignis$ melden. Da hier jedoch eine Vielzahl verschiedener Events auftreten können, reicht eine Zerlegung des Message-Buffers in 5 Ereignisindices nicht mehr aus. Der Buffer muß bis zum 8. Integer-Wort (Index 7) aufgespalten werden:

```
DIM Ereignis(7)
REPEAT
   Evnt_Mesag(Me$)
   FOR T%= 0 TO 7
      Ereignis(T%)= CVI( MID$(Me$,T%*2+1,2))
   NEXT T%
....
....
UNTIL <Abbruchbedingung erfüllt>
```

Wieder steht im ersten Eintrag (Index 0) der Identifikationscode des Ereignisses. Mögliche Events, die eingetreten sind, können sein:

| Code | Name | eingetretenes Ereignis ('Index' = >) |
|---|---|---|
| 10 | Mn_Selected | Menüeintrag angeklickt (alter Hut!) |
| 20 | Mw_Redraw | Bereich muß neu gezeichnet werden<br>3 => Window-Handle<br>4 => X-Koordinate des Bereichs<br>5 => Y-Koordinate des Bereichs<br>6 => Breite des Bereichs<br>7 => Höhe des Bereichs |
| 21 | Wm_Topped | Fenster soll aktiviert werden<br>3 => Handle des Fensters |
| 22 | Wm_Closed | Schließfeld wurde angeklickt<br>3 => Handle des Fensters |
| 23 | Wm_Fulled | Maximalgrößenfeld wurde angeklickt |
| 24 | Wm_Arrowed | Ein Pfeil wurde angeklickt<br>3 => Handle des Fensters<br>4 => angeklicktes Objekt:<br>0 = Seite nach oben<br>1 = Seite nach unten<br>2 = Zeile nach oben<br>3 = Zeile nach unten<br>4 = Seite nach links |

| Code | Name | eingetretenes Ereignis ('Index' = >) |
|---|---|---|
| | | 5 = Seite nach rechts<br>6 = Spalte nach links<br>7 = Spalte nach rechts |
| 25 | Wm_Hslid | Horizontaler Schieber verschoben<br>3 => Handle des Fensters<br>4 => relative Schieberposition:<br>0 = ganz links<br>1000 = ganz rechts |
| 26 | Wm_Vslid | Vertikaler Schieber verschoben<br>3 => Handle des Fensters<br>4 => relative Schieberposition<br>0 = ganz oben<br>1000 = ganz unten |
| 27 | Wm_Sized | Fenster in seiner Größe verändert<br>3 => Handle des Fensters<br>4 => neue X-Koordinate<br>5 => neue Y-Koordinate<br>6 => neue Fensterbreite<br>7 => neue Fensterhöhe |
| 28 | Wm_Moved | Fenster wurde verschoben<br>3 => Handle des Fensters<br>4 => neue X-Koordinate<br>5 => neue Y-Koordinate<br>6 => neue Fensterbreite<br>7 => neue Fensterhöhe |
| 29 | Wm_Topped | Ein Fenster wurde aktiviert<br>3 => Handle des Fensters |
| 30 | Ac_Open | Accessory wurde angeklickt<br>3 => Menüidentifikationsnummer |
| 31 | Ac_Close | Accessory soll gelöscht werden<br>3 => Menüidentifikationsnummer |

Ermittelt die Application beispielsweise, daß ein Fenster geöffnet werden soll (muß natürlich zuvor definiert worden sein, sonst klappt die Geschichte nicht!), so wird einfach die Procedure

```
Wind_Open(Handle,xmax,ymax,wmax,hmax)
```

aufgerufen, um das Fenster auf dem Bildschirm darzustellen. Ein kleiner Befehl (abfragen, ob das Schließfeld des aktiven Fensters angeklickt wurde!) genügt, um das Fenster wieder verschwinden zu lassen:

```
Wind_Close(Handle)
```

Dabei wird das Fenster zwar vom Monitor verbannt, bleibt aber gemäß seiner Definition auch noch weiterhin intern vorhanden. Zum endültigen Vernichten dient der Befehl

```
Wind_Delete(Handle)
```

Mit diesem Wissen ausgetattet, dürfte es im Prinzip keine Schwierigkeiten mehr mit der Fensterprogrammierung geben. Ein ausführliches Beispielprogramm finden Sie diesmal auf der ST-BASIC-Diskette in der Datei GEMDEMO.BAS, in dem alle hier dargestellten Grundlagen in der Praxis vorgeführt werden.

# 7. Multitasking

Multitasking - jene Errungenschaft, die es ermöglicht, mehrere Programme gleichzeitig im Speicher zu behalten und scheinbar gleichzeitig abarbeiten zu lassen - ist auf dem Atari nur bedingt möglich. Das liegt allerdings nicht an dem eingebauten Mikroprozessor, sondern vielmehr an dem Betriebssystem, das dieses Feature nicht vorsieht.

Wenn es nun so aussieht, daß ein Computer zwei Programme gleichzeitig abarbeiten kann, so ist dies nur deshalb möglich, weil der Computer geschickt zwischen beiden Programmen hin- und herwechselt. Kein Mensch - und auch kein Computer - ist in der Lage, zwei Dinge auf einmal zu erledigen. Arbeitet man aber jeweils nur einen Programmschritt von dem einen, dann einen vom dem anderen Programm ab, so entsteht - wenn man nur schnell genug zwischen beiden Programmen wechselt - tatsächlich der Eindruck, der Computer können zwei oder sogar mehrere Programme auf einmal abarbeiten.

Unter die bedingte Multitaskingfähigkeit des Atari ST fällt zum Beispiel der Event-Aufruf des GEM, mit dessen Hilfe auf verschiedene Ereignisse gewartet und entsprechend reagiert werden kann. Doch davon soll in diesem Kapitel nicht die Rede sein.

Vielmehr geht es hier um die eingebauten Befehle des ST-BASICs, mit deren Hilfe während des Programmablaufs ein bestimmtes Ereignis (bestimmte Taste gedrückt) gemeldet werden kann. Dabei arbeiten diese Befehle stets nach dem gleichen Schema:

- Am Programmbeginn wird der Befehl gegeben, daß ST-BASIC in ein bestimmtes Unterprogramm verzweigen soll, wenn das angegebene Ereignis eintritt
- ST-BASIC arbeitet das Programm nun ganz normal ab.

- Tritt das Ereignis ein, verzweigt ST-BASIC - wie vorgeschrieben - in das entsprechende Unterprogramm und arbeitet dieses ab. Anschließend stetzt es seine Arbeit im Hauptprogramm an der Stelle fort, an der der Einsprung in das Unterprogramm erfolgte.

Die verschiedenen Multitasking-Befehle des ST-BASIC lauten:

### ON KEY GOSUB <Ziel>

Springt in die Unterroutine <Ziel>, sobald eine Taste gedrückt wurde. Am Anfang des Unterprogramms muß die Unterbrechungsbedingung (Taste gedrückt) erst einmal mit

```
ON KEY GOSUB 0
```

zurückgesetzt werden, ehe das Unterprogramm abgearbeitet werden kann:

```
ON KEY GOSUB Eingabe
.....
.....
.....
-Eingabe
ON KEY GOSUB 0
.....
< hier folgt dann das Unterprogramm >
......
......
RETURN
```

Soll das Programmm unterbrochen werden, wenn eine Maustaste gedrückt wurde, muß der Befehl

```
ON MOUSEBUT GOSUB <Ziel>
```

verwendet werden. Auch dieser Befehl wird wieder ausgeschaltet, wenn als <Ziel> eine Null angegeben wird:

```
ON MOUSEBUT GOSUB 0
ON HELP GOSUB <Ziel>
```

springt in das Unterprogramm, sobald die Taste <HELP> (im Mittelfeld der Tastatur zu finden) gedrückt wurde. Auch hier wird der Befehl wieder mit

```
ON HELP GOSUB 0
```

beendet. Der Tastencode für <HELP> wird übrigens aus dem Tastaturbuffer entfernt.

### ON TIMER <Zeitspanne> GOSUB <Ziel>

Mit diesem Befehl wird immer dann in das Unterprogramm <Ziel> verzweigt, wenn die <Zeitspanne> (angegeben in Sekunden) verstrichen ist. Die Abstufung dieser Zeitspanne ist in Schritten zu 1/200 Sekunden möglich, wird jedoch bei Bedarf von ST-BASIC selbstständig gerundet. Um diesen Befehl auszuschalten, genügt:

```
ON TIMER GOSUB #0
```

# 8. Der Compiler

Bisher war in diesem Buch von Interpreter und bisweilen auch von Compiler die Rede. Diese beiden Worte habe ich in den Raum gestellt, ohne genauer auf sie einzugehen. Dafür wird es jetzt aber höchste Zeit!

Zunächst einmal zum Begriff Interpreter. Der Prozessor des Atari ST spricht seine eigene Sprache, die aus lauter Nullen und Einsen besteht. BASIC versteht er so wenig, wie Sie "Japanisch". Wenn es nun trotzdem möglich ist, einen Computer in BASIC zu programmieren, dann nur deshalb, weil es eine Instanz im Rechner gibt, die zwischen dem BASIC-Programm und dem Prozessor (Maschinensprache) dolmetscht. Diese Instanz heißt Interpreter, und ist das ST-BASIC-Programm, das vor Beginn jeder Sitzung erst einmal geladen werden muß, ehe man in BASIC programmieren kann.

Ein solcher Interpreter arbeitet - da wir gerade bei diesem Begriff angelangt sind - wie ein Simultandolmetscher, d.h. er holt sich einen BASIC-Befehl nach dem anderen aus dem Programmtext und übersetzt ihn für den Prozessor. Dies hat natürlich einen gewaltigen Nachteil: der Übersetzungsvorgang benötigt seine Zeit, und der übersetzte Code wird nie optimal sein (Haben Sie schon einmal die Übersetzung eines Simultandolmetschers im Fernsehen gehört?). Dies hat unweigerlich eine geringere Verarbeitungsgeschindigkeit des Programms zur Folge.

Ein Compiler arbeitet ebenfalls wie ein Dolmetscher, nur er legt seine Arbeit schriftlich nieder, in eine Datei, die später keinen Interpreter mehr benötigt. Es entsteht ein eigenes Programm. Da sich damit dann die Zwischenschaltung eines Interpreters erübrigt, laufen compilierte Programme wesentlich schneller ab, als wenn sie von einem Interpreter ausgeführt würden. Der Geschwindigkeitsvorteil beträgt im Falle des Omikron.-Compilers - darf man den Angaben des Hersteller glauben - Faktor 9,5.

## 8.1 Die Bedienung des Compilers

Die Bedienung des Compilers ist sehr einfach:

- Starten Sie den Compiler von Diskette, indem Sie die Datei Compiler.prg zweimal anklicken.

- Der Compiler meldet sich mit einer File-Selector-Box, in die der Name des Programms eingetragen werden muß, das Sie compiliren möchten. Dieses BASIC-Programm muß in kodierter Form (also nicht als ASCII-File mit SAVE,A oder <F8> gespeichert) auf Diskette vorliegen. Ansonsten müssen Sie erst den Editor starten und das Programm erneut im korrekten Format abspeichern.

- Der Compiler verrichtet seine Arbeit in drei Durchgängen, sogenannten Passes. Im zweiten und dritten Durchgang können Sie den Fortgang der Arbeit auf dem Bildschirm beobachten, da die gerade bearbeitete Zeilennummer ausgegeben wird.

- Hat der Compiler seine Arbeit getan, steht ein direkt ausführbares Programm auf der Diskette, das den gleichen Namen, jedoch die Extension ".Prg" trägt (Demo.bas => Demo.prg). Beachten Sie bitte, daß auf der Diskette noch genügend freier Speicherplatz vorhanden ist, da die compilierte Programmversion bis zu 2-mal so groß werden kann. Der Omikron.Compiler meldet sich erneut mit der File-Selector-Box, ein weiteres Programm könnte compiliert werden.

- Auf jeder Diskette, die compilierte Programme enthält, muß die Datei Baslib abgespeichert werden, da sie von den Compilaten nachgeladen wird (Ausnahme: Cutilib!).

- Das Programm kann ab sofort geladen und gestartet werden, ohne weiterhin den Interpreter bemühen zu müssen.

## 8.2 Compiler-Steuerworte

Der Compiler versteht gewisse Steuerworte, die er bei der Bearbeitung des Programms berücksichtigt. Damit der Interpreter nicht darüber "stolpert" wird das Steuerwort als Prozedurdefinition in den Programmtext eingeschmuggelt:

```
DEF PROC <Steuerwort>: RETURN
```

Somit ist gewährleistet, daß sich der Interpreter nicht an dem Steuerwort, das er nicht versteht, stört. Für ihn ist es nur eine Definition, der Compiler weiß aber sehr wohl etwas damit anzufangen. Die beiden folgenden TRACE-Befehle können auf in eine IF...THEN-Abfrage integriert werden, für die beiden folgenden Befehle gilt dies jedoch nicht:

```
MA   (MULTITASKING_ALWAYS)
MBS  (MULTITASKING_BETWEEN_STATEMENTS)
```

Ebenso können die TRACE-Statements von der Erfüllung einer Bedingung abhängig gemacht werden, bei den beiden Multitasking-Befehlen ist dies nicht möglich.

```
TRACE_ON
```

Dieser Befehl sorgt dafür, daß zwischen der Ausführung von Befehlen auf

- CTRL-C (Break)
- Interrupts
- die Tastatur

geachtet wird. Befehle wie ON_MOUSEBUT oder auch ON_KEY werden somit ausgeführt. Der für die Befehle ON ERROR und RESUM wichtige Stapelzeiger wird gespeichert.

```
TRACE_OFF
```

Eine Abfrage der Tasten

- CTRL-C (Break)
- Interrupts
- der Tastatur

wird unterbunden. Daher werden die Befehle zum Multitasking außer Gefecht gesetzt. Ferner erhält die Systemvariable ERL stets den Wert 0, RESUME ist nicht möglich!

```
MULTITASKING_ALWAYS (MA)
```

Selbst wenn kein Steuerwort TRACE_ON im Quelltext zu finden ist, prüft das compilierte Programm immer, ob CTRL-C gedrückt wurde oder eine Multitasking-Bedingung eingetreten ist. Jetzt ist es allerding auch möglich, mitten in einem Befehl die Programmausführung zu unterbrechen (z.B. SORT), solange der Computer nicht gerade mit der Ausführung einer Betriebssystemroutine beschäftigt ist. Die Routinen, die über Multitasking-Befehle aufgerufen wurden, dürfen keine Stringmanipulation ausführen.

```
MULTITASKING_BETWEEN_STATEMENTS
```

Die zu ST- und Omikron.BASIC kompatible Unterbrechungsform. Alle Unterbrechungen sind - falls überhaupt gestattet (Trace_On) - nur an Befehlsgrenzen möglich.

Diese beiden Befehle müssen nach einem CLEAR erneut aufgerufen werden, und dürfen wie gesagt, nicht innerhalb von IF-THEN-Abfragen benutzt werden. Des weiteren existieren noch Kombinationen der verschiedenen Steuerbefehle:

```
MA & TRACE_OFF
```

Dies ist die Standardeinstellung: RESUME arbeitet nicht, ERL enthält stets den Wert 0. In Multitasking-Routinen sind keine Befehle zum String-Handling, zur Dimensionierung eines Arrays (DIM) und Öffnen einer Datei (OPEN) erlaubt.

```
MBS & TRACE_OFF
```

RESUME funktioniert nicht, in ERL steht immer der Wert 0 und Multitasking gibt es auch nicht!

```
MA & TRACE_ON
```

In Multitasking-Routinen sind Befehle zur Stringverwaltung sowie OPEN und DIM nicht erlaubt.

```
MBS & TRACE_ON
```

Die volle Kompatibilität zum Interpreter bleibt gewahrt.

## 8.3 BASIC-Programme auf dem Compiler

Der Compiler ist in der Behandlung der Syntax im allgemeinen wesentlich pingeliger als der Interpreter . Deshalb lassen sich gewisse Anpassungen nicht vermeiden:

- Der Compiler dimensioniert - im Gegensatz zum Interpreter - nicht automatisch. Jedes Array muß am Programmbeginn und nach einem erfolgten CLEAR neu dimensioniert werden. Erfolgt dennoch ein Zugriff auf eine solche Variable, führt dies zu einem "? BUS-ERROR". Ein Dimensionieren in Prozeduren und Funktionen ist verboten, wenn diese Prozeduren lokale Variablen oder Parameter benutzen.
- Ein Variablenfeld kann vom Compiler verkleinert werden, wenn es redimensioniert wird. Der Interpreter strukturiert ein solches Feld nur um.
- Flags (%F) dürfen nicht lokal benutzt werden, sie können auch nicht als Rückgabeparameter in Funktionen und Prozeduren benutzt werden.
- Bei Verwendung lokaler Variabeln, die nicht im GARBAGE-Segment, sondern im String-Segment und dem Prozessorstapel gehalten werden, ist besondere Vorsicht angebracht. Nötigenfalls muß der Stack über "CLEAR, X" vergrößert werden.
- Funktionen sind so zu definieren, daß sie stets das Postfix

(Endung) der Variablen tragen, die sie zurückgeben. Gibt eine Funktion also eine Float-Variable zurück, muß sie als Endung ein "!" oder "#" besitzen, je nach verwendeter Genauigkeit. Mit CSNG und CINT können die Formate jedoch innerhalb der Funktion gewandelt werden.

- Stringvariablen, die in FIELD-Anweisungen stehen, dürfen nicht lokal benutzt werden, bis die FIELD-Anweisung durch die Zuweisung eines Wertes mit LET aufgehoben wurde. Ebenso wird die FIELD-Anweisung aufgehoben, wenn ein CLOSE des entsprechenden Kanals erfolgt.

- Werden zwei Integerzahlen dividiert, resultiert dabei immer ein Fließkommawert, ungeachtet, ob nun Nachkommastellen angefallen sind oder nicht. Deshalb sollte man besser für Integerdivisionen die Ganzzahldivision "\" benutzen.

- Der Variablentyp einer DATA-Zeile muß mit der entsprechenden READ-Anweisung übereinstimmen. Möchten Sie die Integerzahl "1" in die Float-Variable "G!" einlesen, ist dies nur möglich, wenn die Integerzahl als Fließkommazahl in die Data-Zeile eingesetzt wird. Dies geschieht, indem man einen Dezimalpunkt an die Zahl hängt: 1.

- Tauchen im Programmtext Befehle zum Multitasking auf, sollten die Compiler-Steuerworte TRACE_ON und MSB darin enthalten sein. Besonders der Aufruf von CALL verlangt nach einem MSB.* INPUT$ und INPUT USING werden sebst dann durch ON TIMER GOSUB unterbrochen, wenn als Befehlswort MSB in den Programmtext eingestreut ist.

- Durch den Einsprung in eine Fehlerroutine kann der Endwert einer Integer FOR-NEXT-Schleife verloren gehen. In diesem Fall muß die Fehlerroutine den Schleifenzähler wiederherstellen. Eine Idee zur Problemlösung stammt aus dem Handbuch zum Omikron.Compiler:

```
100 FOR M=0 TO Nummer
110   <hier steht der Schleifeninhalt>
120 NEXT M
....
1000 Fehlerroutine
1010 IF ERL=110 THEN FOR M=0 TO Nummer:RESUME NEXT: NEXT M
1020 RESUME NEXT
```

- Die ASC-Funktion ASC("") liefert den Wert 0.
- Bei Integer-Arithmetik wird nicht auf OVERFLOW geachtet, deshalb können falsche Ergebnisse entstehen, wenn die Zahl den eigentlichen Wert nicht mehr aufnehmen kann. Der Interpreter kann eine entsprechende Typentransformation durchführen, beim Compiler ist dieser Vorgang jedoch nicht möglich.
- EXIT arbeitet im Compiler nur ohne Zielangabe bzw. mit EXIT -1. EXIT TO darf nur verwendet werden, um eine Schleife zu verlassen.

## 8.4 Programme optimieren

Möchten Sie das Letzte aus ihren Programmen herausholen, so sollten Sie folgende Tips beherzigen:

- Da Integer-Operationen am effektivsten übersetzt werden können, sollten Sie alle in diesem Typus ausführbaren Berechnungen auch in Integer durchführen.
- Vermeiden Sie es, Zeichenketten zwischenzuspeichern, da Stringfunktionen an sich eine Menge Zeit erfordern.
- Je mehr Klammern in eine IF...THEN-Abfrage gepackt werden, desto langsamer wird das Compilat werden. Ebenso lassen sich normale IF...THEN-Ausdrücke recht gut übersetzen. Je mehr Sie diese allerdings verschachteln, desto mehr Zeit wird für ihre Bearbeitung benötigt.* Verwenden Sie bei ineinander geschachtelten Schleifen die am häufigsten durchlaufene Schleife nach Möglichkeit als innerste Schleife.

- Manche Berechnungen (Division, Wurzelberechnung, ...) erzwingen eine Berechnung im Fließkommaformat, obwohl nur Integer dazu benutzt werden. Werden diese Nachkommastellen anschließend nicht benötigt, sollten Sie entweder bei Division die Integerdivision (\) benutzen oder mit CINT eine Bearbeitung im Integerformat erzwingen.

## 8.5 Fehlermeldungen des Compilers

*Bad exit*
Der Compiler gestattet die Benutzung von EXIT nur als EXIT und EXIT -1. Zum Verlassen von Schleifen darf auch EXIT TO herangezogen werden.

*Out of memory*
Es ist nicht genug freier Speicherplatz vorhanden. Abhilfe: Verkeinern Sie die RAM-Disk oder zerstückeln Sie das Programm in mehrere kleine Teile, die dann nachgeladen werden können.

*Structure too long*
Eine in einer IF...THEN-Abfrage oder Schleife verwendete Struktur erweist sich als zu lang, d.h größer als 32 KByte Compilat. Bringen Sie deshalb den Inhalt der Schleife zumindest teilweise in einem Unterprogramm unter.

*Too many variables*
Der Speicherbedarf für alle Variablen außer Arrays darf die maximale Größe von 64 KByte nicht übersteigen.

*Type Mismatch*
Eine Variable des falschen Typs wird an eine Funktion oder Prozedur übergeben.

*Undefined statement(s) or DIM*
Dafür können folgende Ursachen verantwortlich sein:

1. Sprünge verweisen auf nicht (mehr) vorhandene Ziele.
2. Es wurde vergessen, ein Array zu dimensionieren.

*Warning: RETURN type mismatch*
Eine Funktion, die sich über mehrere Zeilen erstreckt, muß den Variablentyp zurückgeben, den sie im Namen trägt. Dies ist jedoch nicht der Fall!

*Warning: Unused Statement(s)*
Der Compiler ist während seiner Arbeit auf Label-Definitionen und Prozeduren gestoßen, die nicht benutzt werden. Diese können entfernt werden, wenn Sie Speicherplatz einsparen möchten.

## 8.6 Verarbeitungstypen der Funktionen

Die folgende Liste gibt an, welche Variablentypen von den einzelnen Funktionen zurückgegeben werden. Tritt ein Type mismatch Error auf, so lohnt es sich, einmal einen Blick in diese Übersicht zu werfen:

**Integer:**

AND, ASC, BIT, CINT, CINTL, CRSLIN, CVI, CVIL, EOS, EQV, ERL, ERR, FRE, HIGH, IMP, INSTR, LEN, LOC, LOF, LOW, LPEEK, LPOS, MEMORY, MOUSEBUT, MOUSEX, MOUSEY, NAND, NOR, NOT, OR, PEEK, POINT, POS, SEGPTR, SGN, SHL, SHR, TIMER, USR, VARPTR, WPEEK, XOR, =, >, >=, <=, <>

**Float (Single und Double):**

ARCCOS, ARCCOT, ARCCOTH, ARCSIN, ARCSINH, ATN, COS, COSEC, COSH, COT, COTH, DET, EXP, FACT, LN, LOG, SEC, SECH, SIN, SINH, SQR, TAN, TANH, ^, /

**Single-Float:**

CSNG, CVS, RND

**Double-Float:**

CDBL, CVS, PI, VAL

**String:**

BIN$, CHR$, DATE$, ERR$, HEX$, INKEY$, INPUT$, LEFT$, LOWER$, MID$, MIRROR$, MKD$, MKIL$, MKS$, OCT$, RIGHT$, SPACE$, SPC, STR$, STRING$, TIME$, UPPER$, @

**Vom Typ des Arguments abhängig:**

ABS, FIX, FRAC, INT, MAX, MIN, MOD, +,-, *, \, (), +1, -1, *2

## 8.7 Hilfsprogramme auf der Compilerdiskette

### CUTLIB.PRG

Das Programm CUTLIB.PRG wird verwendet, um die BASLIB-Library zu kürzen und an das Programm anzuhängen. Dadurch wird das Programm alleine lauffähig und nicht mehr benötigt.

Nach dem Programmstart erscheint eine File-Selector-Box, in der der Name des Programmes einzutragen ist, für das die Datei BASLIB gekürzt werden soll. Ferner können Sie folgende Optionen einstellen:

- CTRL-C erlauben (J/N)
- Zeicheneingabe unter Omikron.-BASIC (J/N)
- Zeichenausgabe unter Omikron.-BASIC (J/N)

Werden die beiden letzteren Fragen mit "J" beantwortet, verlängert sich zwar das Programm etwas, dafür stehen Ihnen die Omikron.-typischen Möglichkeiten bei der Ein- und Ausgabe von Zeichen zur Verfügung. Vom Programm benötigte Routinen werden aber auf alle Fälle eingebunden! CUTLIB ist auch im Batch-Betrieb ausführbar, die Syntax dafür lautet:

```
CUTLIB Name [,Name,Name ...] [-Antwort]
```

"Name" gibt das Programm an, das bearbeitet werden soll. Werden von diesem andere Programmteile nachgeladen, müssen diese durch Komma getrennt ebenfalls angegeben werden. Unter "Antwort" werden die drei Fragen von CUTLIB beantwortet.

Dabei steht ein großes J für ja, ein großes N für nein. Wahlweise kann auch 999 angegeben werden. Dann wird die ganze BASLIB in das Programm eingearbeitet.

## SHELL.PRG

Das Programm SHELL ist ein textorientierter Kommando-Interpreter, der folgende Befehle versteht:

| Befehl | bewirkt .... |
|---|---|
| DIR | Ausgabe des Inhaltsverzeichnisses |
| CHDIR Pfad | Wechsel des Inhaltsverzeichnisses |
| MKDIR Pfad | Anlegen eines neuen Ordners |
| RMDIR Pfad | LÖschen eines Ordners |
| COPY Quelle Ziel | Kopieren der Datei von 'Quelle' nach 'Ziel' |
| REN Alt Neu | Umbenennen einer Datei |
| DEL Name | Löschen der Datei |
| DATE Datum | Setzen des Datums |
| TIME Zeit | Einstellen der Uhrzeit |
| PAUSE | Wartet auf Tastendruck |
| REM | Anmerkung (REMark) |
| PROMPT Zeichenkette | Stellt das Prompt ein: |
| $p | aktuellen Pfad anzeigen |
| $d | Datum ausgeben |
| $g | >-Zeichen |
| $n | Aktuelles Laufwerk |
| TYPE Name | Gibt eine Datei auf dem Monitor aus |
| VER | Liefert Versionsnummer des TOS |
| Programmname [Parameter] | Startet das Programm |
| Batchname [Parameter] | Startet eine BATCH-Datei |
| ECHO ON | Bildschirmausgabe bei Batchdatei einschalten |
| ECHO OFF | Bildschirmausgabe bei Batchdatei ausschalten |
| EXIT | verläßt den Kommando-Interpreter |

Die Dateitypen .BAT und .PRG müssen bei Dateinamen nicht mit angegeben werden.

# Anhang

## Anhang A: ASCII-Tabelle

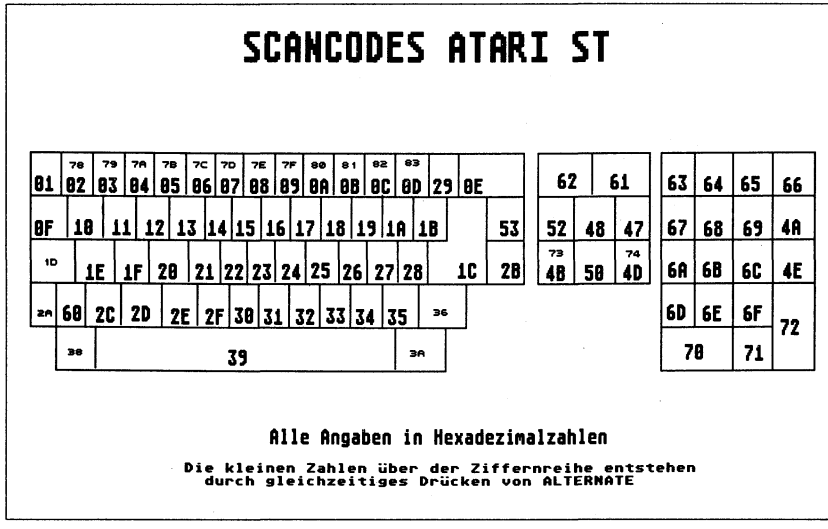
Der folgenden Tabelle können Sie den ASCII-Zeichensatz des Atari ST in Dezimal- und Hexadezimalnotation entnehmen:

| Dez. | Hex | Zeichen | Dez. | Hex | Zeichen | Dez. | Hex | Zeichen | Dez. | Hex | Zeichen |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 00 | | 32 | 20 | | 64 | 40 | @ | 96 | 60 | ` |
| 1 | 01 | ☺ | 33 | 21 | ! | 65 | 41 | A | 97 | 61 | a |
| 2 | 02 | ● | 34 | 22 | " | 66 | 42 | B | 98 | 62 | b |
| 3 | 03 | ♥ | 35 | 23 | # | 67 | 43 | C | 99 | 63 | c |
| 4 | 04 | ♦ | 36 | 24 | $ | 68 | 44 | D | 100 | 64 | d |
| 5 | 05 | ♣ | 37 | 25 | % | 69 | 45 | E | 101 | 65 | e |
| 6 | 06 | ♠ | 38 | 26 | & | 70 | 46 | F | 102 | 66 | f |
| 7 | 07 | • | 39 | 27 | ' | 71 | 47 | G | 103 | 67 | g |
| 8 | 08 | ◘ | 40 | 28 | ( | 72 | 48 | H | 104 | 68 | h |
| 9 | 09 | ○ | 41 | 29 | ) | 73 | 49 | I | 105 | 69 | i |
| 10 | 0A | ◙ | 42 | 2A | * | 74 | 4A | J | 106 | 6A | j |
| 11 | 0B | ♂ | 43 | 2B | + | 75 | 4B | K | 107 | 6B | k |
| 12 | 0C | ♀ | 44 | 2C | , | 76 | 4C | L | 108 | 6C | l |
| 13 | 0D | ♪ | 45 | 2D | - | 77 | 4D | M | 109 | 6D | m |
| 14 | 0E | ♫ | 46 | 2E | . | 78 | 4E | N | 110 | 6E | n |
| 15 | 0F | ☼ | 47 | 2F | / | 79 | 4F | O | 111 | 6F | o |
| 16 | 10 | ► | 48 | 30 | 0 | 80 | 50 | P | 112 | 70 | p |
| 17 | 11 | ◄ | 49 | 31 | 1 | 81 | 51 | Q | 113 | 71 | q |
| 18 | 12 | ↕ | 50 | 32 | 2 | 82 | 52 | R | 114 | 72 | r |
| 19 | 13 | ‼ | 51 | 33 | 3 | 83 | 53 | S | 115 | 73 | s |
| 20 | 14 | ¶ | 52 | 34 | 4 | 84 | 54 | T | 116 | 74 | t |
| 21 | 15 | § | 53 | 35 | 5 | 85 | 55 | U | 117 | 75 | u |
| 22 | 16 | ▬ | 54 | 36 | 6 | 86 | 56 | V | 118 | 76 | v |
| 23 | 17 | ↨ | 55 | 37 | 7 | 87 | 57 | W | 119 | 77 | w |
| 24 | 18 | ↑ | 56 | 38 | 8 | 88 | 58 | X | 120 | 78 | x |
| 25 | 19 | ↓ | 57 | 39 | 9 | 89 | 59 | Y | 121 | 79 | y |
| 26 | 1A | → | 58 | 3A | : | 90 | 5A | Z | 122 | 7A | z |
| 27 | 1B | ← | 59 | 3B | ; | 91 | 5B | [ | 123 | 7B | { |
| 28 | 1C | ∟ | 60 | 3C | < | 92 | 5C | \ | 124 | 7C | ¦ |
| 29 | 1D | ↔ | 61 | 3D | = | 93 | 5D | ] | 125 | 7D | } |
| 30 | 1E | ▲ | 62 | 3E | > | 94 | 5E | ^ | 126 | 7E | ~ |
| 31 | 1F | ▼ | 63 | 3F | ? | 95 | 5F | _ | 127 | 7F | ⌂ |

| Dez. | Hex | Zeichen | Dez. | Hex | Zeichen | Dez. | Hex | Zeichen | Dez. | Hex | Zeichen |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 128 | 80 | Ç | 160 | A0 | á | 192 | C0 | └ | 224 | E0 | α |
| 129 | 81 | ü | 161 | A1 | í | 193 | C1 | ┴ | 225 | E1 | ß |
| 130 | 82 | é | 162 | A2 | ó | 194 | C2 | ┬ | 226 | E2 | Γ |
| 131 | 83 | â | 163 | A3 | ú | 195 | C3 | ├ | 227 | E3 | π |
| 132 | 84 | ä | 164 | A4 | ñ | 196 | C4 | ─ | 228 | E4 | Σ |
| 133 | 85 | à | 165 | A5 | Ñ | 197 | C5 | ┼ | 229 | E5 | σ |
| 134 | 86 | å | 166 | A6 | ª | 198 | C6 | ╞ | 230 | E6 | µ |
| 135 | 87 | ç | 167 | A7 | º | 199 | C7 | ╟ | 231 | E7 | τ |
| 136 | 88 | ê | 168 | A8 | ¿ | 200 | C8 | ╚ | 232 | E8 | Φ |
| 137 | 89 | ë | 169 | A9 | ⌐ | 201 | C9 | ╔ | 233 | E9 | Θ |
| 138 | 8A | è | 170 | AA | ¬ | 202 | CA | ╩ | 234 | EA | Ω |
| 139 | 8B | ï | 171 | AB | ½ | 203 | CB | ╦ | 235 | EB | δ |
| 140 | 8C | î | 172 | AC | ¼ | 204 | CC | ╠ | 236 | EC | ∞ |
| 141 | 8D | ì | 173 | AD | ¡ | 205 | CD | ═ | 237 | ED | φ |
| 142 | 8E | Ä | 174 | AE | « | 206 | CE | ╬ | 238 | EE | ε |
| 143 | 8F | Å | 175 | AF | » | 207 | CF | ╧ | 239 | EF | ∩ |
| 144 | 90 | É | 176 | B0 | ░ | 208 | D0 | ╨ | 240 | F0 | ≡ |
| 145 | 91 | æ | 177 | B1 | ▒ | 209 | D1 | ╤ | 241 | F1 | ± |
| 146 | 92 | Æ | 178 | B2 | ▓ | 210 | D2 | ╥ | 242 | F2 | ≥ |
| 147 | 93 | ô | 179 | B3 | │ | 211 | D3 | ╙ | 243 | F3 | ≤ |
| 148 | 94 | ö | 180 | B4 | ┤ | 212 | D4 | ╘ | 244 | F4 | ⌠ |
| 149 | 95 | ò | 181 | B5 | ╡ | 213 | D5 | ╒ | 245 | F5 | ⌡ |
| 150 | 96 | û | 182 | B6 | ╢ | 214 | D6 | ╓ | 246 | F6 | ÷ |
| 151 | 97 | ù | 183 | B7 | ╖ | 215 | D7 | ╫ | 247 | F7 | ≈ |
| 152 | 98 | ÿ | 184 | B8 | ╕ | 216 | D8 | ╪ | 248 | F8 | ° |
| 153 | 99 | Ö | 185 | B9 | ╣ | 217 | D9 | ┘ | 249 | F9 | • |
| 154 | 9A | Ü | 186 | BA | ║ | 218 | DA | ┌ | 250 | FA | · |
| 155 | 9B | ¢ | 187 | BB | ╗ | 219 | DB | █ | 251 | FB | √ |
| 156 | 9C | £ | 188 | BC | ╝ | 220 | DC | ▄ | 252 | FC | η |
| 157 | 9D | ¥ | 189 | BD | ╜ | 221 | DD | ▌ | 253 | FD | ² |
| 158 | 9E | ₧ | 190 | BE | ╛ | 222 | DE | ▐ | 254 | FE | ■ |
| 159 | 9F | ƒ | 191 | BF | ┐ | 223 | DF | ▀ | 255 | FF | |

## Anhang B: Scancodes des Atari ST

Der folgenden Grafik können Sie die Scancodes der einzelnen Tasten entnehmen:



## Anhang C: Verzeichnis aller VT-52-Sequenzen

In der folgenden Tabelle wird CHR$(27) durch ESC ersetzt:

| Sequenz | bewirkt |
|---|---|
| ESC A | Cursor eine Zeile nach oben |
| ESC B | Cursor eine Zeile nach unten |
| ESC C | Cursor nach rechts |
| ESC D | Cursor nach links |
| ESC E | Löscht Bildschirm (CLS) |
| ESC H | Setzt Cursor in linke obere Bildschirmecke |
| ESC I | Cursor hoch |
| ESC J | Bis Bildschirmende alles löschen |
| ESC K | Ab Cursor bis Zeilenende alles löschen |
| ESC L | Leerzeile einfügen |
| ESC M | Zeile löschen, Rest rückt auf |
| ESC Y s z | PRINT @(Spalte,Zeile)<br>s = CHR$(Spalte+32)<br>z = CHR$(Zeile+32) |
| ESC b n | Wählt Schriftfarbe <n> |
| ESC c n | Wählt Schrifthintergrund <n> |
| ESC d | Löscht Bildschirm bis zum Cursor |
| ESC e | Cursor einschalten |
| ESC f | Cursor ausschalten |
| ESC j | Cursorposition speichern |
| ESC k | Setzt Cursor an gespeicherte Position |
| ESC l | Zeile löschen |
| ESC o | Zeile bis Cursor löschen |
| ESC p | Invers ein |
| ESC q | Invers aus |
| ESC v | Automatischer Zeilenüberlauf (NICHT SCREEN 0) |
| ESC w | Zeilenüberlauf ausschalten (NICHT SCREEN 0) |

## Anhang D: Fehlermeldungen des Interpreters

*1* *Structure too long*
Zwischen zwei Strukturwörtern (FOR...NEXT, WHILE ... WEND) befinden sich mehr als 64 KB Programmcode. Zur Abhilfe sollte der Inhalt zumindest teilweise in ein Unterprogramm transportiert werden.

*2* *Syntax Error*
ST-BASIC kann mit diesem Befehl nichts anfangen. Sehr wahrscheinlich liegt ein Tippfehler vor.

*3* *RETURN without GOSUB*
Der Interpreter ist auf ein RETURN gestoßen, ohne daß ein Einsprung in eine Unterroutine mit GOSUB erfolgte.

*4* *Out of DATA*
Der DATA-Zeiger verweist bereits hinter das letzte Element. Deshalb konnte ein READ keine Daten mehr lesen. Abhilfe: Anzahl der DATAs korrigieren.

*5* *Illegal function call*
Ein Befehl oder eine Funktion wurde in einer Art und Weise aufgerufen, in der er nicht verwendet werden darf.

*6* *Overflow*
Der Rechenbereich des Variablentyps wurde überschritten.

*7* *Out of memory*
Es ist entweder kein Speicherplatz für weitere Variablen vorhanden oder aber kein Platz mehr auf dem Prozessor-Stapel.

*8* *Undefined Statement*
Das Sprungziel existiert nicht.

*9* *Subscript out of range*
Der angesprochene Variablenindex ist größer als die im DIM festgelegte Dimension.

*10* *Duplicate definition*
Ein Funktions- oder Prozedurname wurde doppelt vergeben.

*11* *Division by zero*
Es wurde versucht, durch Null zu teilen.

*12* *Illegal direct*
Der Befehl ist im Direktmodus nicht erlaubt.

*13* *Type mismatch*
Eine Variable sollte mit einem falschen Inhalt versorgt werden.

*14* *RETURN without funktion*
Der Interpreter hat ein RETURN entdeckt, ohne daß zuvor eine Funktion aufgerufen worden wäre.

*15* *String too long*
Eine Zeichenkette darf maximal 32766 Zeichen enthalten.

*16* *Formula too complex*
Eine Berechnung ist zu sehr verschachtelt (haben Sie wieder einmal übertrieben?) und verbraucht zu viel Stack-Bereich.

*17* *Cant't continue*
Ein CONT ist nicht möglich.

*18* *Undefined user function*
Es existiert keine Funktion mit diesem Namen.

*19* *No RESUME*
In einer Fehlerroutine (ON ERROR) fehlt RESUME.

*20* *RESUME without error*
Der Interpreter soll ein RESUME ausführen, ohne daß er mit ON ERROR in die Fehlerroutine gesprungen ist.

*21* *Use EXIT*
Schleifen dürfen nur mit EXIT verlassen werden.

*22* *Missing operand*
Ein Operand wurde vergessen anzugeben.

*23* *Line buffer overflow*
Eine Programmzeile ist zu lang (bei der Eingabe: 255 Zeichen, beim Auflisten: 512 Zeichen erlaubt).

*24* *REPEAT without UNTIL*
Eine REPEAT-Schleife wurde geöffnet, ohne daß ein UNTIL vorhanden ist.

*25* *UNTIL without REPEAT*
Ein UNTIL wurde entdeckt, ohne daß ein REPEAT vorhanden ist.

*26* *FOR without NEXT*
Eine FOR-Schleife wurde geöffnet. Im Programmtext kann aber kein NEXT gefunden werden.

*27* *NEXT without FOR*
Der Interpreter ist auf ein NEXT gestoßen, ohen daß ein FOR dafür vorhanden ist.

*28* *IF without THEN or ENDIF*
Entweder Sie haben bei einem IF-Befehl das THEN vergessen, oder aber es fehlt ein ENDIF (bzw. ist überflüssig).

*29* *WHILE wihout WEND*
Zu WHILE existiert kein WEND.

*30* *WEND without WHILE*
Zu WEND existiert kein WHILE.

*31* *THEN, ELSE or ENDIF without THEN*
Zu einem THEN, ELSE oder ENDIF fehlt das zugehörige IF bzw. für ein ELSE oder ein ENDIF ist ein THEN zuwenig vorhanden.

*33* *Reset*
Sie haben den Reset-Knopf gedrückt, alle Variableninhalte sind verloren, das Programm ist nicht gelöscht.

*34* *Bus Error*
Ein BUS-Fehler ist aufgetreten (verursacht durch ein von CALL oder USR aufgerufenes Programm).

*35* *Adress error*
Ein Adreßfehler ist aufgetreten.

*36* *Unknown opcode*
Ein unbekannter Maschinenbefehl steht in einem mit CALL oder USR aufgerufenen Programm.

*45* *EXIT without Structure*
EXIT wurde entdeckt, obwohl kein Unterprogramm oder Schleife vorhanden ist, das mit EXIT verlassen werden könnte.

*46* *USE EXIT TO in functions*
In einer mehrzeiligen Funktion ist nur EXIT TO erlaubt.

*47* *Not regular Matrix*
Zu dieser Matrix existiert keine Inverse (Gegenprobe: Wenn die Determinate Null ist => nicht invertierbar).

*50* *Field overflow*
In einer FIELD-Anweisung wurden zu viele Daten angegeben.

*52* *Bad file number*
Es sind nur die Kanalnummern von 1 bis 16 erlaubt.

*53* *File not found*
Die Datei ist auf der Diskette nicht vorhanden.

*54* *Bad file mode*
Eine nicht erlaubte Operation sollte mit der Datei ausgeführt werden.

*55* *File already open*
Die Datei ist bereits einmal an anderer Stelle geöffnet worden. Abhilfe: andere Kanalnummer verwenden oder die offene Datei zuerst schließen.

*56* *File not open*
Sie haben vergessen die Datei vor dem Zugriff zu öffnen.

*57* *TOS error #XX*
Ein TOS-Fehler ist aufgetreten (siehe Anhang E).

*61* *Disk full*
Es ist auf der Diskette kein Speicherplatz mehr vorhanden, der belegt werden könnte.

*62* *Input past end*
Es wurde versucht in einer sequentiellen Datei über das Datei-Ende hinauszulesen. Fragen Sie mit EOF das Datei-Ende ab.

*63* *Bad record number*
Es wurde versucht in einer relativen Datei auf einen Datensatz zuzugreifen, der nicht existiert.

*64* *Bad file name*
Der Dateiname enthält unerlaubte Zeichen: Punkt, Komma, Doppelpunkt, Semikolon usw.

*65* *Direct statement in file*
Ein Programm, das geladen werden soll, beseitzt keine Zeilennummern. Abhilfe: 'LOAD BLOCK *.*' benutzen.

*66* *Too many files*
Das Inhaltsverzeichnis kann keine weiteren Dateien mehr aufnehmen. Abhilfe: Mit KILL Programme von der Diskette entfernen.

## Anhang E: TOS Fehlermeldungen:

| Fehlernummer | Beschreibung des Fehlers |
|---|---|
| 1 | Allgemeiner Fehler |
| 2 | Laufwerk nicht bereit |
| 3 | Unbekannter Befehl |
| 4 | Prüfsummen-Fehler |
| 5 | Falsche Rückmeldung (Befehl ungültig) |
| 6 | Sektor nicht gefunden |
| 7 | Fehler im Bootsektor |
| 8 | Sektor nicht gefunden |
| 9 | Druckerfehler (Paper out) |
| 10 | Fehler beim Schreiben |
| 11 | Fehler beim Lesen |
| 12 | Allgemeiner Fehler |
| 13 | Diskette schreibgeschützt |
| 14 | Diskette wurde gewechselt |
| 15 | Gerät unbekannt |
| 16 | Prüffehler |
| 17 | Keine Diskette vorhanden |
| 32 | Ungültige Funktionsnummer |
| 33 | Datei nicht gefunden |
| 34 | Pfad nicht gefunden |
| 35 | Zu viele Dateien geöffnet |
| 36 | Zugriff nicht möglich |
| 37 | Ungültige Handle-Nummer |
| 39 | Zu wenig freier Speicherplatz |
| 40 | Speicherblockadresse ungültig |
| 46 | Laufwerksbezeichnung ungültig |
| 49 | Keine weiteren Dateien vorhanden |

## Anhang F: GfA-BASIC-Programme umschreiben.

Zuerst sollte Sie das GFA-BASIC-Programm einmal als ASCII-File auf Diskette abspeichern. Laden Sie dann den ST-BASIC-Interpreter und laden das soeben abgespeicherte Programm mit LOAD BLOCK *.* ein. Bitte nicht mit LOAD, da andernfalls eine Fehlermeldung ausgegeben würde. Jetzt müssen Sie noch Zeile für Zeile durchgehen, und die Befehle, die sich unterscheiden, entsprechend in eine für ST-BASIC verständliche Syntax wandelt.

Vergessen Sie bitte nicht, daß in GfA-BASIC alle nicht anderweitig deklarierten Variablen FLOAT-Variablen sind, während ST-BASIC diese als Long-Integers interpretiert. Abhilfe kann hier der Befehl DEFSNG A-Z in der ersten Programmzeile schaffen.

Beim Umschreiben des Programms können Ihnen auch die beiden DATA BECKER Führer zu GfA- und Omikron.BASIC eine große Hilfe sein, mit denen Sie gezielt die Syntax eines Befehls nachschlagen können.

# Index

Accessory ..... 31
AND ..... 149, 158
Array ..... 90
ASC ..... 149
ASCII ..... 48
ASCII-Code ..... 154, 232
ASCII-Datei ..... 37
ASCII-Tabelle ..... 69

Backslash ..... 213
BACKUP ..... 208
BAK ..... 56, 207
Baslib ..... 372
Bconin ..... 257
Bconout ..... 257
Bconstat ..... 257
Bcostat ..... 259
Benutzeroberfläche ..... 247
Betriebssystem ..... 68
Bildschirm laden ..... 285
Bildschirm-Editor ..... 16
Bildschirm-Aufteilung ..... 38
BIN$ ..... 89
Binärzahl ..... 43
BIOS ..... 247
Bioskey ..... 267
Bit ..... 43
BITBLT ..... 283
BLOAD ..... 286
Block ..... 25
Blockoperation ..... 35
BOX ..... 281
BSAVE ..... 286
Byte ..... 43

Carriage Return ..... 70
Cauxin ..... 249
Cauxis ..... 250
Cauxos ..... 250
Cauxout ..... 249
Cconos ..... 250
Cconout ..... 249
Cconrs ..... 249
Change Size ..... 28
Char ..... 46
Character ..... 46
CHDIR ..... 213
CHECKED ..... 314
CINT ..... 376
CLEAR ..... 96, 205, 374, 375
Clipboard ..... 294
CLOSE ..... 188
CLS ..... 52
Cluster ..... 251
CMD ..... 238
Compiler ..... 371
Compiler.prg ..... 372
COPY ..... 191, 208
Cprnos ..... 250
Cprnout ..... 249
CROSSED ..... 314
CSNG ..... 376
CSRLIN ..... 245
CTRL-C ..... 373
Cutilib ..... 372
CUTLIB.PRG ..... 380

DATA ..... 140
DATE$ ..... 245
Datei-Auswahlbox ..... 196
Date-Averwaltung ..... 181
Datensatz ..... 182
Dcreate ..... 252
Ddelete ..... 252
DEFAULT ..... 312

Dezimalkomma ..... 51
Dezimalsystems ..... 43
Dfree ..... 251
Dgetdrv ..... 250
Dgetpath ..... 256
DIM ..... 91
Dimension ..... 91
Directory ..... 22
Direktmodus ..... 19, 207
DISABLED ..... 314
Diskette formatieren ..... 270
DO_FILE ..... 200
DOODLE ..... 286
Dosound ..... 268
Double Precision ..... 45
Draggen ..... 294
DRAW ..... 281
Drucker ..... 236
Drvmap ..... 259
Dsetdrv ..... 250
Dsetpath ..... 253
Dualsystem ..... 43, 88

EDIT ..... 32
EDITABLE ..... 312
Editor ..... 15
Einfügemodus ..... 27
Einzeilige Funktion ..... 135
ELSE ..... 101
Enable() ..... 331
ENDIF ..... 103
Endlosschleife ..... 116, 119
EOF ..... 189
ERL ..... 205
ERR ..... 205
ERR$ ..... 205
ERROR ..... 207
Exec *.Prg ..... 31
EXIT ..... 117, 312
Extender ..... 198, 345

FACT ..... 139
Fattrib ..... 255
Fclose ..... 253
Fcreate ..... 253
Fdattime ..... 256
Fdelete ..... 254
Fdup ..... 255
Fehlermeldung ..... 387
Fensterkomponente ..... 358
Fenstertechnik ..... 357
Ffcorce ..... 255
Ffopfmt ..... 262
Fgetdta ..... 251
FIELD ..... 214
File ..... 21
File-Selector-Box ..... 195, 344
FILES ..... 213
Files kopieren ..... 191
FILESELECT ..... 199
FILL ..... 282
FIND ..... 23
FIND ERROR ..... 25, 30
FIND NEXT ..... 24
FIND TOKEN ..... 23, 25
FIX ..... 64
Flags ..... 48
Fließkommavariable ..... 45
Fließkommazahl ..... 44
Float ..... 45
Floprd ..... 262
Flopver ..... 267
Flopwr ..... 262
Fopen ..... 253
FOR..NEXT ..... 110
Form_Center ..... 325
Form_Dial ..... 326
Form_Do ..... 327
Formatierte Eingabe ..... 152
Formulars ..... 292

FRAC ..... 64
Fread ..... 254
Frename ..... 256
Fseek ..... 254
Fsetdta ..... 251
Fsfirst ..... 256
Fsnext ..... 256
Full-Screen-Editor ..... 20
Funktionen ..... 134
Funktionstastenbelegung ..... 92
Füllbyte ..... 249
Fwrite ..... 254

Garbage Collection ..... 271
GARBAGE-Segment ..... 375
GEM ..... 196, 246, 291
GEM-VDI ..... 247
GEMDOS ..... 247
GET ..... 216
Getbpb ..... 258
Getdsb ..... 263
Getmpb ..... 257
Getrez ..... 261
Gettime ..... 267
Giaccess ..... 268
Globale Variable ..... 126
GO ..... 29
GOTO ..... 106

Handle ..... 361
Hauptdirectory ..... 197
Header ..... 52
HEADER-File ..... 296
HEX$ ..... 88
Hexadezimalsystem ..... 87
Hide ..... 27
HIDETREE ..... 313
HIRES ..... 280

Icons ..... 294
IF ..... 101
Ikbytedws ..... 268
Index ..... 91, 196
Indexdatei ..... 183
Indexsequentielle Datei ..... 181
INDIRECT ..... 313
Initmous ..... 261
INKEY$ ..... 147
INPUT ..... 58
INPUT USING ..... 152
INPUT# ..... 189
Insert ..... 26, 27
Insert$ ..... 140
Insert-Modus ..... 18
INT ..... 64
Integer-Operationen ..... 377
Integerdivision ..... 378
Intel ..... 274
Interaktive Programmierung ..... 58
Interpreter ..... 371
Interrupts ..... 373
Iorec ..... 263
ISAM ..... 181
ISAM-Dateien ..... 183
Iterativ ..... 134

Jenabit ..... 268
Jidisint ..... 268
Joker ..... 198

K-Resource ..... 301
Kbdvbase ..... 269
Kbrate ..... 269
Kbshift ..... 260
KEY ..... 92
KEY LIST ..... 93
Keytbl ..... 265
Kill ..... 26, 208
Kontrollstruktur ..... 138

Label ..... 94, 108
LASTOB ..... 313
LDUMP ..... 96
Leerzeichen ..... 72
LEFT$ ..... 75
LEN ..... 78, 87
LET ..... 52
Library ..... 231
LINE COLOR ..... 281
LINE INPUT ..... 61
LINE INPUT# ..... 189
Line Numbers ..... 28
LINE STYLE ..... 281
Line to Bottom ..... 30
Line To Top ..... 30
Linien ..... 281
Link-Zeilen ..... 19
LIST TOKEN ..... 25
Listing ..... 51
LLIST ..... 95, 238
LOAD ..... 57
Load *.* ..... 22
Load Block ..... 22
Load Block *.* ..... 27
LOF ..... 194
Logbase ..... 261
Logische Verknüpfung ..... 158
Lokale Variable ..... 124
Long ..... 44
Longinteger ..... 44
LOWER$ ..... 81
LPEEK ..... 310
LPOKE ..... 310
LPRINT ..... 237
LSET ..... 215

Malloc ..... 256
Mark Bl. End ..... 27
Mark Bl. Start ..... 26
Marken ..... 29, 94, 108
Mediach ..... 259
MEGA-ST ..... 294
Mehrzeilige Funktion ..... 137
Menüzeile ..... 21
MERGE ..... 232
Mfree ..... 256
MID$ ..... 76, 77
Midi ..... 263
Midiws ..... 263
MIRROR$ ..... 80
MKDIR ..... 211
MKI$() ..... 272
MOD ..... 54
Mode ..... 27, 81, 281
MODE LPRINT ..... 237
Modulo-Funktion ..... 54
MOUSEBUT ..... 242
MOUSEX ..... 241
MOUSEY ..... 242
Move ..... 26
MS-DOS ..... 273
Multitasking ..... 367

NAME AS ..... 208
New ..... 23, 56
NEXT ..... 112
NOT ..... 161, 190
Nullbyte ..... 202

Objektbaum ..... 293
Objektflag ..... 311
Objektspezifikation ..... 315
Objektstatus ..... 313
OCT$ ..... 89
Offgibit ..... 268
Oktalsystem ..... 88

OM_BASIC ..... 15
Omikron.BASIC ..... 15, 16
ON ERROR GOTO ..... 204
ON HELP GOSUB ..... 368
ON KEY GOSUB ..... 368
ON MOUSEBUT GOSUB ..... 368
On TIMER GOSUB ..... 369
ON TRON GOSUB ..... 98
Ongibit ..... 268
OR ..... 160
Ordner ..... 196, 252
OUTLINED ..... 314

PBOX ..... 282
Peek ..... 309
Pfadname ..... 196
Physbase ..... 261
PI ..... 245
Pokeq ..... 309
Postfixe ..... 42, 44
PRINT ..... 39, 42, 186
PRINT @ ..... 239
PRINT AT ..... 146
Print Block ..... 27
PRINT# ..... 186
Programm laden ..... 37
Programm speichern ..... 37
Programmierhilfen ..... 92
Protobt ..... 265, 272
Prozeduren ..... 123
Prtblk ..... 269
Pull-Down-Menüs ..... 333
Punkte ..... 281
Puntaes ..... 269
PUT ..... 216

Query Replace ..... 25
Quicksort ..... 129
Quit Edit ..... 23

RADIO-BUTTON ..... 312
Rahmen ..... 20
RAM-Disk ..... 191
Random ..... 265
Random-access-Dateien ..... 182
RCS ..... 292
READ ..... 140
Rechtecke ..... 281
Record ..... 214
Rekursion ..... 124
Relative Dateie ..... 181
REM ..... 51
Rename Token ..... 25
RENUM ..... 233
RENUMBER ..... 98
Repeat ..... 37
Repeat..Until ..... 115
Replace all... ..... 25
RESTORE ..... 140
RESUME ..... 206
RESUME NEXT ..... 206
RIGHT$ ..... 76
RMDIR ..... 212
RND ..... 137
Rsconf ..... 264
RSET ..... 215
Rsrc_Gaddr ..... 325
Rsrc_Load ..... 324
RUN ..... 30, 57
Run *.Bas ..... 31
Rwabs ..... 258

SAVE ..... 55
SAVE & Compile ..... 31
SAVE & RUN ..... 30
SAVE *.* ..... 21
SAVE Block ..... 22
SAVE Block *.* ..... 27
SAVE Settings ..... 28
Scancode ..... 154

Schieberegler ..... 332
Schleifen ..... 110
Schleifenzähler ..... 113
Scrdmp ..... 267
Scrollen ..... 94
Sektor ..... 251
SELECTABLE ..... 312
SELECTED ..... 314
Sequentielle Datei ..... 181
Set Mark #X ..... 30
Setcolor ..... 262
Setexc ..... 258
Setpalette ..... 261
Setprt ..... 268
Setscreen ..... 261
Settime ..... 267
SHADOWED ..... 315
SHELL.PRG ..... 381
Show Errors ..... 28
Single Precision ..... 45
Slider ..... 332
Space ..... 53, 72
SPACE$ ..... 79
SPC ..... 79
Speicheroperation ..... 309
Split Screen ..... 27, 28
Sprungstelle ..... 108
Ssbrk ..... 261
Stack ..... 134
STEP ..... 114
Str$ ..... 74, 87
String ..... 46
STRING$ ..... 73, 79
Strukturierte Programmierung ..... 99
Subdirectory ..... 211, 252
Subroutine ..... 121
Such-String ..... 82
Suchen ..... 23
Supexec ..... 269
Sversion ..... 251

SWAP ..... 131
Switch Screen ..... 27
SYSTEM ..... 38
Systemvariable ..... 241

Tastatur ..... 373
Tastaturprozessor ..... 67
Tastaturpuffer ..... 20
Te_color ..... 317
Te_font ..... 317
Te_just ..... 317
Te_ptext ..... 316
Te_ptmplt ..... 316
Te_pvalid ..... 316
Te_thickness ..... 318
Te_tmplen ..... 319
Te_txtlen ..... 319
TEDINFO ..... 315
THEN ..... 101
Tickcal ..... 258
TIME$ ..... 244
TIMER ..... 243
To last Mark ..... 29
To Line... ..... 29
To Mark #X ..... 30
Tokens ..... 23
TOS ..... 246
TOUCHEXIT ..... 313
Trace ..... 96
TRACE_OFF ..... 373
TRACE_ON ..... 373
Trash ..... 294
Tron & Run ..... 30

Unterinhaltsverzeichnis ..... 211
UPPER$ ..... 80
USING ..... 166

Val ..... 74
VAL() ..... 149
Variable ..... 39
Variablenfelder ..... 90
VDI ..... 291
Vsync ..... 269
VT52-Emulator ..... 19

WHILE..WEND ..... 119
Wildcards ..... 198
Wind_Close ..... 366
Wind_Delete ..... 366
Wind_Open ..... 365
Word ..... 44
WPOKE ..... 309
WRITE# ..... 187

X-Koordinate ..... 241
XBIOS ..... 247
XOR ..... 161

Y-Koordinate ..... 242

Zahlensysteme ..... 87
Zeichen ..... 46
Zeichen löschen ..... 17
Zeichenkette ..... 46
Zeile einfügen und löschen ..... 18
Zeilennummern ..... 51
Zufallsgenerator ..... 137
Zuweisungsoperator ..... 40

Nach den ersten gescheiterten Versuchen mit ATARI-BASIC haben Sie sich bestimmt nach einer besseren VVrsion umgeschaut. Das GFA-BASIc ist sicherlich eine der leistungsstärksten Programmiersprachen auf dem ST. Gerade dadurch ist es aber für den Anfänger meist nicht leicht zu verstehen. Genau hier setzt das Buch an. Von der Installation einer Arbeitskopie bis hin zu einfachen GEM-Funktionen wird der Grundwortschatz von GFA-V3.0 und V2.0 leichtverständlich an vielen Beispielen erklärt.


Aus dem Inhalt:

- Einführung in die Bedienung des GFA-Editors
- Zuweisungen und Variablendeklaration
- Ein- und Ausgaben
- Schleifenprogrammierung an Beispielen
- strukturierte Programmierung
- Grafikprogrammierung mit Minigrafikprogramm
- Diskettenoperationen
- kleines Datenbankprogramm
- Pannenhilfe – Vermeidung von Fehlern
- Konvertierungshinweise GFA V2.0 nach GFA V3.0
- ausführliche Funktionsübersicht zum schnellen Nachblättern

**Schumann**
**GFA-BASIC für Einsteiger**
**247 Seiten, DM 29,–**
***ISBN 3-89011-248-X***

## Bücher zum Basic 3.0

Endlich gibt es GFA V3.0! Zu diesem umfangreichen BASIC gehört auch ein umfangreiches Buch, in dem detailliert jeder Befehl behandelt wird. Dabei liefert es keine nackte Befehlsübersicht, sondern wirklich brauchbares Material in Hülle und Fülle. Anhand zahlreicher Beispielprogramme lernen Sie dieses leistungsfähige BASIC spielend zu beherrschen.


Aus dem Inhalt:

- Das Editor-Menü
- Variablentypen und -organisation
- Diskettenoperationen
- Strukturierte Programmierung
- Mausabfrage in eigenen Programmen
- Sound-Programmierung
- Beschreibung des Resource-Construction-Set
- Verwendung von Multitasking-Befehlen
- Programmieren von Pull-Down-Menüs
- Abfrage von Ereignissen (Events)
- Window-Programmierung
- Zugriff auf GEMDOS, BIOS und XBIOS
- Komplette AES- und VDI-Library-Beschreibung
- Verwenden eigener Fonts mit GDOS
- Eine komplette Adressenverwaltung als RAM-Kartei
- Komplette Befehlsübersicht über alle Befehle des GFA-BASIC

**Litzkendorf**
**Das große GFA BASIC 3.0 Buch**
**828 Seiten, DM 49,–**
***ISBN 3-89011-222-6***

# Bücher zum Atari St

Die besten Tips und Tricks zum ATARI ST sind eine wahre Fundgrube für jeden ATARI-ST-Besitzer. Anhand vieler nützlicher Routinen werden die fantastischen Möglichkeiten dieses Rechners ausführlich erklärt. Programmier- und Hardwaretips vermitteln zusätzlich eine Menge über die Rechnerstruktur und dessen Programmierung in GFA-Basis, C und Assembler.


Aus dem Inhalt:

- GEM-Starter
- Uhrzeit resetfest
- echtes Multitasking
- Sprite-Programmierung
- schnelle Grafikroutinen
- Dia-Show
- Sound-Programmierung
- Konvertierungsprogramme
- Floppy-Speeder
- Short-Cuts für beliebige Programme
- Accessory-Aufbau
- Filesearch für Festplatte
- Bildschirmschoner
- Ordner umbennenen
- Einschaltverzögerung für Festplatte

**Pauly, Schepers, Schulz**
**Atari ST**
**Die besten Tips & Tricks**
**428 Seiten, DM 59,–**
***ISBN 3-89011-210-2***

## Bücher zum Atari St

Bei der Arbeit mit dem Atari ST tauchen immer wieder Probleme auf. Genau dort setzt dieses Buch an. Es beantwortet alle Fragen zu den Bereichen Desktop, Massenspeicher, Drucker, Schnittstellen ... kurz zu allem, was zum Atari ST gehört. Der gut strukturierte Aufbau des Buches mit zusätzlichen Symbolen zum schnellen Finden der Lösungen in lexikonähnlicher Form macht dieses Buch zum unentbehrlichen Nachschlagewerk, das das nötige Hintergrundwissen liefert.


Aus dem Inhalt:

- Desktop: Umgang mit Icons, Menüleiste und Windows, Bedienung von Objektauswahlboxen
- Massenspeicher: Schnelles Kopieren mit einem Laufwerk, Umbenennen von Ordnern, Tips zu Festplatten
- Drucker: Zeichensätze, DIP-Schalter, Druckeranpassung
- Schnittstellen: Centronics, RS232, DMA Datenübertragung
- Computerwissen: Aufbau des Computers, Zahlensysteme
- Software: Tips & Tricks zu Standardsoftware
- Pflege und Wartung: Einbau von TOS-Roms, kleine Reparaturen selbst gemacht
- Glossar

**Liesert**
**Das große Atari ST Handbuch**
**370 Seiten, DM 49,–**
***ISBN 3-89011-273-0***

## DAS STEHT DRIN:

Wer bisher glaubte, professionelle Programme könne man nur in C oder gar Assembler entwickeln, wird mit diesem Buch eines Besseren belehrt. Nach einem ausführlichen Basic-Grundkurs erfahren Sie alles über die Dateiverwaltung, die Nutzung von Betriebssystemroutinen, die Grafikprogrammierung, oder die Programmierung unter GEM. Nützliche Tools und Libaries runden das Buch ab.

Aus dem Inhalt:

- Variablentypen in ST-Basic
- Strings und Stringmanipulation
- Strukturierte Programmierung
- Rekursionen
- Formatierte Ein- und Ausgabe
- sequentielle und relative Dateiverwaltung
- logische Verknüpfungen
- Betriebssystemprogrammierung
- GEM-Programmierung
- eigene File-Selector-Box
- Multitasking
- Omikron.Compiler
- nützliche Libaries

## UND GESCHRIEBEN HAT DIESES BUCH:

Michael Maier, Student der Betriebswirtschaftslehre und Informatik, kann als Computerfreak der ersten Stunde auf eine langjährige Erfahrung in der Programmierung zahlreicher Rechner zurückblicken. In diesem Buch zeigt er, wie man in ST-Basic professionelle Programme schreibt.

ISB N 3-89011-283-8 DM +049.00

DM 49,–
ÖS 382,–
sFr 47,–

DATA BECKER

9 783890 112831 04900