Winfried Baum

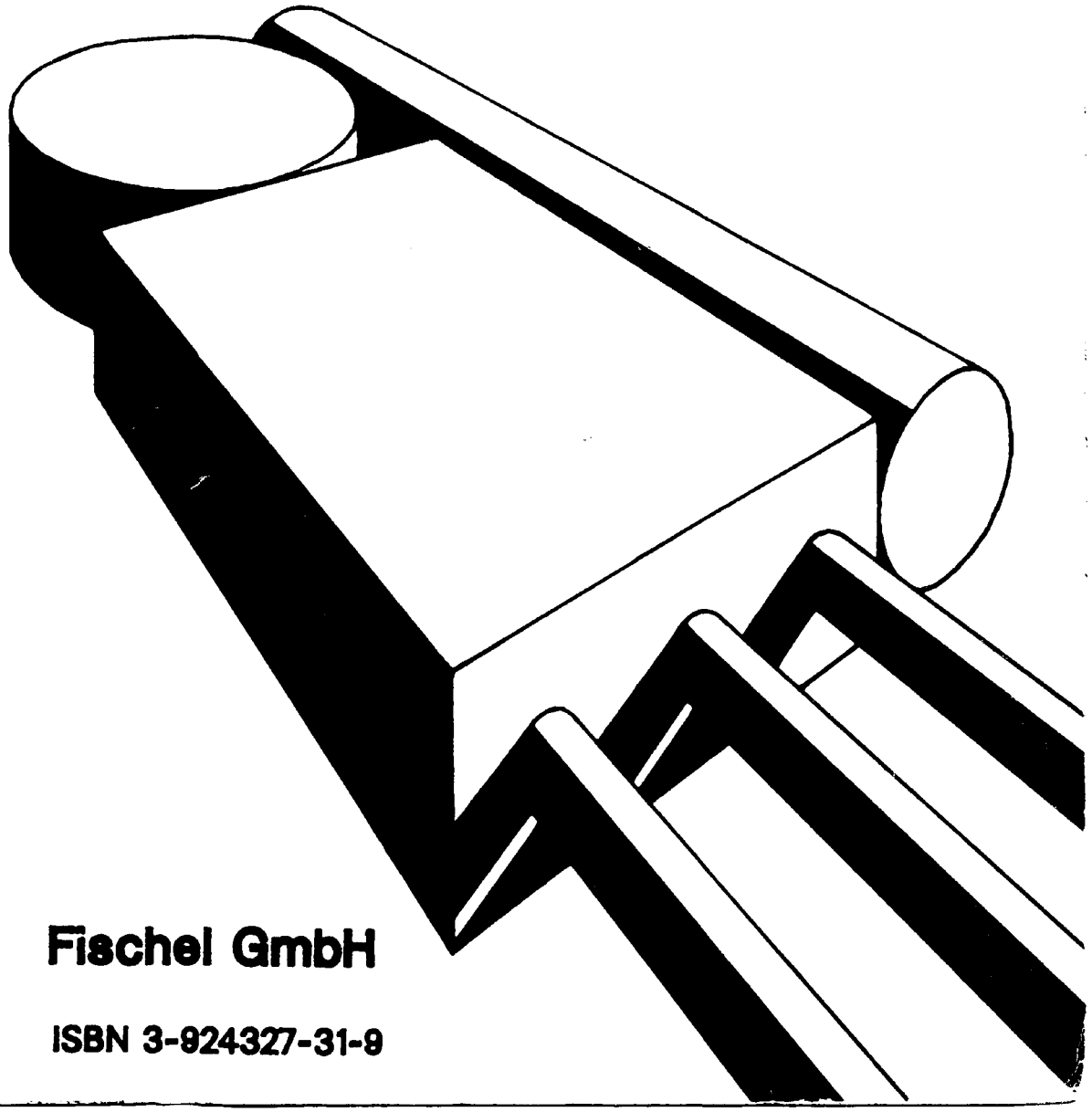
SHARP

PC-1600

Taschen-Computer

DER FORTSCHRITTLICHE TASCHEN-COMPUTER

Systemhandbuch



Fischel GmbH

ISBN 3-924327-31-9

INHALT

ARBEITSWEISE DES BASIC-INTERPRETERS

Interne Darstellung von Programmen 5

Interne Darstellung von Zahlenwerten 9

Interne Darstellung von Zeichenketten 10

Interne Darstellung von Variablen 12

Abarbeitung von Programmen 15

SPEICHERPLATZOPTIMIERUNG BEI BASIC PROGRAMMEN 19
Allgemeines 19
Komprimierung von Programmen 20
Einsparen von Variablenspeicher 23
Vergrößerung des zur Verfügung stehenden Speicherplatzes 24

LAUFZEITOPTIMIERUNG VON BASIC-PROGRAMMEN 27
Allgemeines 27
Arithmetik 28
Zeichenkettenverabeitung 31
Sprunganweisungen 32
Schleifen 33
IF-Anweisungen 35
Variablenzugriffe 37
Sperren von Interrupts 41

DER TASTATURPUFR 42
Funktionsweise 42
Zugriff auf den Tastaturpuffer 45
DER RESERVESPEICHER 50
DER LAUTSPRECHER 51
DER PLOTTER 54
Funktionsweise 54
Ein-/Ausgabeadressen **55**
DAS DISKETTENLAUFWERK 56

Funktionsweise 56
Ein-/Ausgabeadressen 57

SERIELLE DATENÜBERTRAGUNG 58
Funktionsweise 58
XON/XOFF- und RTS/CTS-Protokoll 59
SHIFT IN SHIFT OUT -Protokoll 60

INTERRUPTS 62
Vorgänge bei der Interruptverarbeitung auf Prozessorebene 62
Interrupts in BASIC 64
Interruptquellen 64
Initialisierung von Interruptquellen 65
ON/STOP **66**
Speicherung von Interruptanforderungen 67
Entfernen von Interrupteinträgen 69
Ablauf eines Interrupts in BASIC **70**

DAS DISPLAY 72
Zugriff auf den Displayspeicher **72**
Darstellungsweise des Displayspeicherinhalts auf der Anzeige 77
Verschieben des Displayinhalts 79
Ein- und Ausschalten der Anzeige 81
Hinweise für Assemblerprogrammierer 82

DER ANALOGEINGANG 84

DER CASSETTENRECORDERANSCHLUß 86
Aufzeichnungsverfahren 86
Lesen von Cassette 87
Schreiben auf Cassette 87
Der Remote-Ausgang 88

DIE TASTATUR 89
Abfrage durch Interruptroutine 89
Direkte Abfrage der Tastaturmatrix 89
Die ON-Taste 91

ANHANG A: EIN-/AUSGABE-ADRESSEN 93

ANHANG RAM-ADRESSEN **99**

ANHANG C: ADRESSRAUMBELEGUNG **103**

ANHang D: RAM-BELEGUNG **104**

ANHANG E: TABELLE DER BASIC-TOKENS **105**

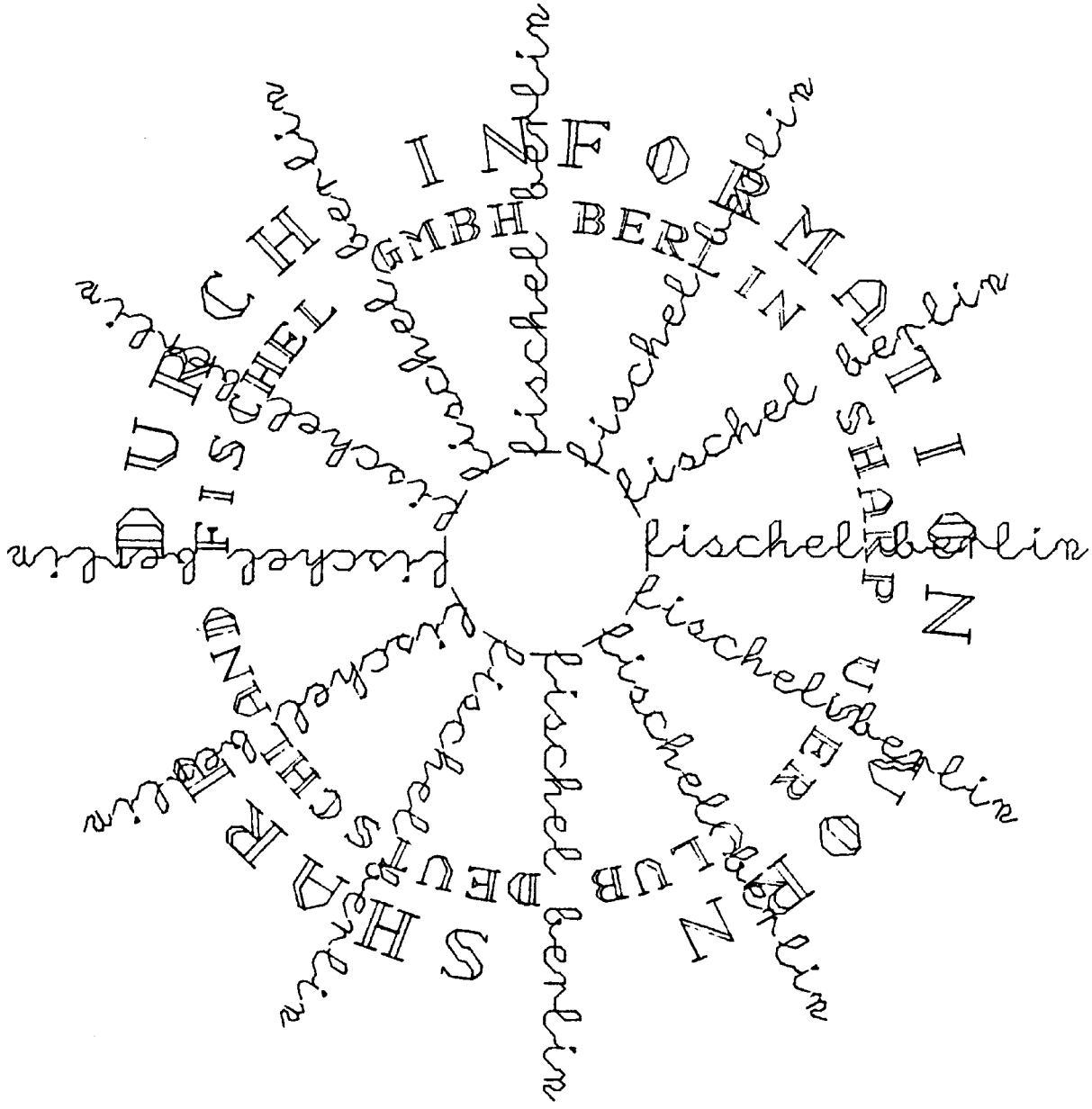
ANHANG F: BEISPIELPROGRAMME **108**
Schnelle Potenzrechenroutine 108
Sieb des Eratosthenes 109
Zeichnen eines Punktes ohne PSET 111
Softscrolling 113
Bewegte Darstellung dreidimensionaler Funktionsgraphen 114
Adressierung der Sondersymbole 115
Anzeige der Belegung der Funktionstasten 116
Neubelegung der Funktionstasten durch ein Programm 117
Einfaches Multitasking 118
Abfrage von mehreren gleichzeitig gedrückten Tasten 119
Dampflok: Rauschgenerator und Bewegung 120
KBUFF$ ohne Löschen 123

LITERATUR 124

SHARP

SHARP

SHARP



ARBEITSWEISE DES BASIC-INTERPRETERS

Es ist immer vorteilhaft, ungefähr zu wissen, was man mit Programmen, die in einer höheren Programmiersprache erstellt sind, einem Rechner Überhaupt zumutet. Dieses Wissen ist vor allem dann wichtig, wenn man besonders effiziente Programme schreiben möchte.

In diesem Kapitel wird deshalb in groben-Zügen beschrieben, wie der Interpreter des PC-1600 BASIC-Programme verarbeitet. Die im nächsten Kapitel vorgestellten Methoden zur Effizienzsteigerung von Programmen beruhen größtenteils auf der Nutzung spezieller Eigenschaften desselben. Um nicht nur das Wie sondern auch das Warum zu verstehen, sollte man nach Möglichkeit zuerst dieses Kapitel gelesen haben.

Interne Darstellung von Programmen

Anweisungen

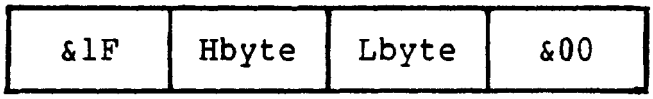
Um Speicherplatz zu sparen, sind Befehlsnamen nicht Buchstabe fÜr Buchstabe gespeichert. Vielmehr werden sie schon bei der Eingabe der Programmzeilen in sogenannte **Tokens** umgewandelt. Ein Token besteht beim PC-1600 aus 2 Bytes, von denen das erste im Bereich von &E0 bis &FF liegt. Das zweite kann jeden Wert zwischen 0 und &FF annehmen. So belegt z.B. der Befehl **PRINT** statt 5 nur 2 Bytes, nämlich &F0 &97. Eine Tabelle der Tokens aller Befehle ist im Anhang zu finden.

Alle Zeichen, die nicht Bestandteile eines Befehlsnamens sind (Ziffern, Klammern, Stringkonstanten usw.), werden durch ihren ASCII-Code dargestellt.

Eine Ausnahme bildet die Angabe der Zeilennummer bei **GOTO und GOSUB.** Für sie gibt es zweierlei Darstellungen:

1. Ziffernweise als ASCII-Zeichen:
Diese Darstellungsart wird nur noch gepflegt, um die Kompatibilität zum PC-1500 aufrecht zu erhalten.

2. Als Binärzahl:
Diese Darstellungsart wird immer dann verwendet, wenn ein Programm neu eingegeben wird. Sie hat gegenüber der beim PC-1500 verwendeten den Vorteil, daß während eines Programmlaufs nicht jedesmal von dezimal nach binär umgewandelt werden muß Um die Binär- von der ASCII-Darstellung unterscheiden zu können, wird das folgende Format verwendet:



Die Anweisung **GOTO 12345** würde somit folgendermaßen dargestellt:

ASCII: &F1 &92 &31 &32 &33 &34 &35

binär: &F1 &92 &1F &30 &39 &00

Programmzeilen

Eine Programmzeile besteht aus

- zwei Bytes, die die Nummer der Zeile bilden,
- einem Byte, das die Länge der Zeile angibt,
- bis zu 80 Bytes, die die einzelnen Anweisungen, wie im vorigen Abschnitt beschrieben, enthalten und
- einem Byte mit dem Wert &OD, das das Ende der Zeile markiert.

Beispiel:

```
500         PRINT SIN A : A = 1   2   3
01 F4   0C   F0 97 F1 7D 41 3A 41 3D 31 32 33 0D
Zeilen Länge                                    End-
nummer                                          marke
```

Programme

Ein Programm ist eine Abfolge von Zeilen, wie sie im vorigen Abschnitt beschrieben sind, gefolgt von einem Byte mit dem Inhalt &FF, das das Ende des Programms anzeigt. Der Beginn des Programms liegt bei der Grundversion der PC-1600, wenn nicht mit **NEW "SO:", n** anders definiert, an der Adresse &C0C5.

Anwendungsbeispiel: selbstmodifizierendes Programm

Das folgende Programm ersetzt auf Wunsch alle in sich selbst enthaltenen PRINT- durch-LPRINT-Befehle und umgekehrt.

```
10 "A"
15 DEGREE
20 PRINT "Ausgabe auf A(nzeige"
30 PRINT "       oder D(rucker"
40 INPUT "--> ";A$
50 GOSUB "ERSETZ"
60 PRINT "Sinustabelle": PRINT "---------------"
70 PRINT "  x    sin x"
80 FOR X=0 TO 359
90 PRINT USING "####"; X; "  ";
100 PRINT USING "##.###";SIN X
110 NEXT I
115 END
120 "ERSETZ"
130 'Programmstartadresse:
140 A = &8000 + 256 * PEEK &F865 + PEEK &F866
150 'Muss Überhaupt etwas geändert werden?
160 'Letzte Einstellung (A oder D) steht in
170 'der ersten Kommentarzeile
180 IF CHR$ PEEK (A+4) = A$ RETURN
190 'Bereitstellen der beiden Tokens (das er-
200 'ste Byte (&F0) ist beiden gemeinsam)
210 IF A$ = "D" LET V = &97 : N = &B9
            ELSE LET V = &B9 : N = &97
230 IF PEEK A &FF RETURN
240 A = A+3
250 B = PEEKA A=A+1
260 IF B = &0D GOTO 230
270 IF B <> &F0 GOTO 250
280 3 = PEEKA : A=A+1
290 IF B <> V GOTO 250
300 POKE A-1, N : GOTO 250
```

Interne Darstellung von Zahlenwerten

Fließkommazahlen

Der PC-1600 rechnet meist mit Fließkommazahlen, die aus folgenden Komponenten zusammengesetzt sind:

1. Exponent: 1 Byte im Zweierkomplement, d.h. negative Zahlen werden als -1=&FF, -2=&FE USW. dargestellt.

2. Vorzeichen: 1 Byte enthält den Wert 0 (positiv) oder &80 (negativ).

3. Mantisse: 5 Bytes enthalten im gepackten BCD-Format zehn Dezimalziffern (jede Ziffer 4 Bit).

4. Ein verbleibendes Byte enthält in Variablen immer den Wert 0. Dieses Byte kann bei Zwischenergebnissen innerhalb arithmetischer Operationen weitere zwei Ziffern aufnehmen, um Rundungsfehler klein zu halten.

Beispiele:

	hexadezimal
5678 :	03 00 56 78 00 00 00 00
-3.778993648E45 :	2D 80 37 78 99 36 48 00
0.0000035267:	FA 00 35 26 70 00 00 00

Binärzahlen

Es gibt auch eine Binärdarstellung von Zahlen, die allerdings nur fÜr interne Zwecke benutzt wird. Es werden ebenfalls 8 Bytes verwendet, aber nur 3 davon sind von Bedeutung:

- 5. Byte: Enthält als Erkennungsmerkmal immer den Wert &B2

- 6. Byte: Höherwertiges Byte einer 16-Bit-Zahl im Zweierkomplement

- 7. Byte: Niederwertiges Byte der Zahl

Die restlichen 5 Bytes bleiben unbenutzt und kÖnnen beliebige Werte enthalten.

Beispiele:

```
 1234 : xx xx xx xx B2 04 D2 xx
-4567 : xx xx xx xx B2 EE 29 xx
```

Diese Darstellungsart ist gut dazu geeignet, von Assemblerprogrammen ganzzahlige Werte an BASIC-Variablen zurückzugeben, da man damit geschickt die Umrechnung ins BCD-Format umgehen kann.

Interne Darstellung von Zeichenketten

Strings sind Folgen von ASCII-Zeichen und werden durch deren Codes dargestellt, immer ein Byte je Zeichen. Ein bestimmter String belegt immer gleichviel Speicherplatz, unabhängig von der durch **LEN** angezeigten Länge. Dieser Speicherbedarf beträgt bei Standardstrings 16 Byte, bei Nichtstandardstrings 1 bis 80 Bytes je nach Definition mit

DIM. Ist die tasächliche Länge kleiner als die maximal erlaubte, so wird das Ende durch ein &00-Byte angezeigt.

Die **LEN**-Funktion zählt alle Zeichen, bis sie entweder eine 0 oder die maximale erlaubte Länge erreicht. Aus diesem Grund kann ein String selbst keine CHR$0-Zeichen enthalten. Im Gegensatz zu anderen Rechnern sind

"" und CHR$0

genau dasselbe.

Normalerweise bildet eine Stringvariable und ihr Inhalt eine feste Einheit. Es gibt jedoch auch Fälle, wo nicht die ASCII-Folge direkt , sondern eine Referenz auf sie verwendet wird. Ein Beispiel ist die Übergabe eines Strings an ein Assemblerunterprogramm. Nicht einzelne Zeichen werden Übergeben, sondern die Anfangsadresse und die Länge des Strings. Eine Entsprechende Darstellung gibt es auch im Zusammenhang mit numerischen Variablen.

Hat das fünfte der 8 Bytes eines Zahlenwertes den Wert &D0, so werden die restlichen 3 Bytes als Stringreferenz aufgefaßt:

xx xx xx xx D0 C8 10 05

verweist auf einen String, der an der Adresse **&C810** beginnt und die Länge 5 hat. Das folgende Beispiel demonstriert diese Möglichkeit. Die Variable B erhält als Inhalt eine Referenz auf A$. Man beobachte, was passiert:

```
10 POKE &F90C,&D0,&F8,&C0,&10
20 A$="Hoppla"
30 PRINT B
```

Leider können die meisten Stringverarbeitungsbefehle nicht auf diese zweckentfremdeten Zahlenwerte angewandt werden. Der eine oder andere Leser wird aber sicher interessante Anwendungsmöglichkeiten finden.

Interne Darstellung von Variablen

Standardvariablen

Die Standardvariablen befinden sich an fester Stelle im Systembereich des RAMS, und zwar

- die Numerischen Variablen A bis Z in den Adressen von &F900 bis &F9CF, jede belegt 8 Bytes.

- die Stringvariablen in den Adressen von &F8C0 bis &F8FF (A$ bis D$), von &F650 bis &F6FF (E$ bis O$) und von &F750 bis &F7FF (P$ bis Z$).

Nichtstandardvariablen

Die Nichtstandardvariablen werden dynamisch verwaltet, d.h. sie können während des Programmlaufs erzeugt und auch wieder entfernt werden. Ihre Position im Speicher hängt davon ab, wieviele andere Variablen bereits vor ihnen erzeugt und welche von diesen bereits wieder gelöscht worden sind.

Deshalb muß mit jeder Variablen außer ihrem Wert noch zusätzliche Information abgespeichert werden,

um dem Interpreter die Suche nach ihnen zu erleichtern.

Diese zusätzliche Information umfaßt fÜr alle Variablentypen 7 Bytes:

- Ein Byte für das erste Zeichen des Variablennamens
- Ein Byte fÜr das zweite Zeichen des Namens und den Variablentyp, dabei- werden vom ASCII-Code des Zeichens nur die Bits 6, 4, 3, 2, 1 und 0 benutzt. Da nur Buchstaben und Ziffern erlaubt sind, reichen diese 6 Bits zur eindeutigen Identifikation des Zeichens aus. Bit 7 und 5 ergeben sich folgendermaßen:

Wert	Bit 7	Bit 5
0	einfache Var.	numerische Var.
1	Arrayvar.	Stringvar.

- 2 Bytes geben die Adressdistanzzur nächsten Variablen an. Sie errechnet sich aus dem Speicherbedarf des Variableninhalts plus 3 und wird bei der Suche nach Variablen verwendet. Bei einer einfachen numerischen Variablen ist dieser Wert 8+3=11.

- 2 Bytes enthalten die maximal erlaubten Werte für die Indizes der zweiten und der ersten Dimension (in dieser Reihenfolge).

- 1 Byte fÜr die maximal erlaubte Länge vonStrings. Bei numerischen Variablen steht hier der Wert &88.

Hierauf folgt der Inhalt der Variablen. Bei eindimensionalen Arrays folgen die Inhalte einander in der Folge

```
A(0), A(l), A(2),...
```

Zweidimensional Arrays (Matrizen) werden zeilenweise abgelegt:

```
A(0,0), A(0,1), A(0,2), ... A(1,0), A(1,1)
```

In diesem Zusammenhang ist noch erwähnenswert, daß auf zweidimensionale Arrays auch mit nur einem Index zugegriffen werden kann. Dadurch lassen sich die Probleme, die entstehen, wenn man ein eindimensionales Array mit mehr als 256 Elementen benötigt, leicht in den Griff bekommen. Statt

```
DIM A(799)
```

was zu einer Fehlermeldung führen würde, schreibt man

```
DIM A(19,39)
```

Dies definiert ebenfalls 800 Elemente, und man kann auf sie mit

```
A(0), A(1), A(2), ... A(799)
```

zugreifen. Eine andere Anwendung desselben Sachverhalts ist die Belegung einer ganzen Matrix mit Werten ohne die Verwendung ineinanderverschachtelter Schleifen, wie dies z.B. in der Sprache FORTRAN Üblich ist. Statt

```
10 DIM A(9,9)
20 FOR I=0 TO 9 : FOR J=0 TO 9
30 A(I,J) = 1
40 NEXT J: NEXT I
```

schreibt man einfacher so:

```
10 DIM A(9,9)
20 FOR I=0 TO 99
30 A(I) = 1
40 NEXT I
```

Abarbeitung von Programmen

Nachdem ein Programm gestartet ist, wird vom Interpreter Zeile fÜr Zeile abgearbeitet. Dabei muß er folgende Dinge berücksichtigen:

1. Zuerst muß er die Zeilennummer und das Byte mit der Zeilenlänge Überlesen.

2. Meist trifft er dann auf ein BASIC-Token. Folgende Ausnahmen sind möglich:

   - Die Anweisung ist eine Zuweisung ohne **LET**
   - Es handelt sich um einen Kommentar mit '.
   - Die Zeile beginnt mit einer Sprungmarke

Alle vier möglichen Fälle sind eindeutig durch das erste Zeichen unterscheidbar. Sprungmarken werden überlesen, Apostrophe werden wie REMs und Zuweisungen ohne LET wie solche mit LET behandelt.

Liegt ein Token vor, so wird dieses in einer Tabelle im ROM gesucht. Diese Tabelle beginnt an der Adresse &B717 in Bank 6. Zu (fast) jedem Token gibt es im ROM ein Unterprogramm, das aufgerufen wird, sobald der Befehl gefunden ist. Dieses liest dann gegebenenfalls Arg-

umente oder Parameter aus dem Programm und zwar solange, bis es auf einen Doppelpunkt oder das Zeilenende (&0D) stößt.

3. Ist das Zeilenende erreicht, wird die nächste Zeile begonnen. Ist keine weitere Zeile vorhanden, so steht dort ein &FF (Programmende) und der Interpreter kehrt in den Eingabemodus zurück. Die Programmendemarke steht an derselben Stelle wie sonst eine Zeilennummer. Um keine Verwechslungen aufkommen zu lassen, darf eine Zeilennummer niemals mit &FF beginnen. Dies erklärt die obere Grenze von 65279.

Auf die Befehle **GOTO, GOSUB und FOR ... NEXT** soll im folgenden etwas näher eingegangen werden, da ihre Funktionsweise auch in den folgenden Kapiteln zur Sprache kommen wird.

GOTO

Ein BASIC-Programm stellt eine einfach verkettete Liste dar, jede Programmzeile entält mit der Angabe ihre Länge im dritten Byte einen Verweis auf den Anfang der nächste Zeile. Dies macht sich der GOTO-Befehl zunutze, um möglichst schnell eine bestimmte Zeilennummer zu finden. Er schaut sich immer nur die Zeilenanfänge an und läßt den Rest der Zeilen unberücksichtigt.

Hat das Sprungziel eine höhere Zeilennummer als die momentan bearbeitete Zeile, so wird dies erkannt, und die Suche beginnt nicht am Programmanfang, sondern in der folgenden Zeile. Bei einem Sprung nach Marken hingegen muß immer das gesamte Programm durchsucht werden, da nicht entschieden werden kann ob das Sprungziel weiter vorn oder hinten liegt.

GOSUB

Unterprogrammaufrufe funktionieren gleich wie gewöhnliche Sprünge, mit dem Unterschied, daß die momentan bearbeitete Position des Programms festgehalten werden muß, um später wieder dorthin zurückkehren zu können.

Dies geschieht dadurch, daß die Zeilennummer und die Speicheradresse der nachfolgenden Anweisung auf einen RETURN-Stapel-gelegt wird. Dieser belegt zusammen mit dem FOR-NEXT-Stapel, auf den noch eingegangen wird, die Adressen &FA38 bis &FAFF. Der Zeiger auf die letzte abgelegte Rücksprungadresse liegt in &F891. Er wird bei jedem Unterprogrammaufruf um 6 erniedrigt (es werden immer 6 Bytes an Information abgelegt) und bei einer **RETURN**-Anweisung wieder um 6 erhöht.

FOR ... NEXT

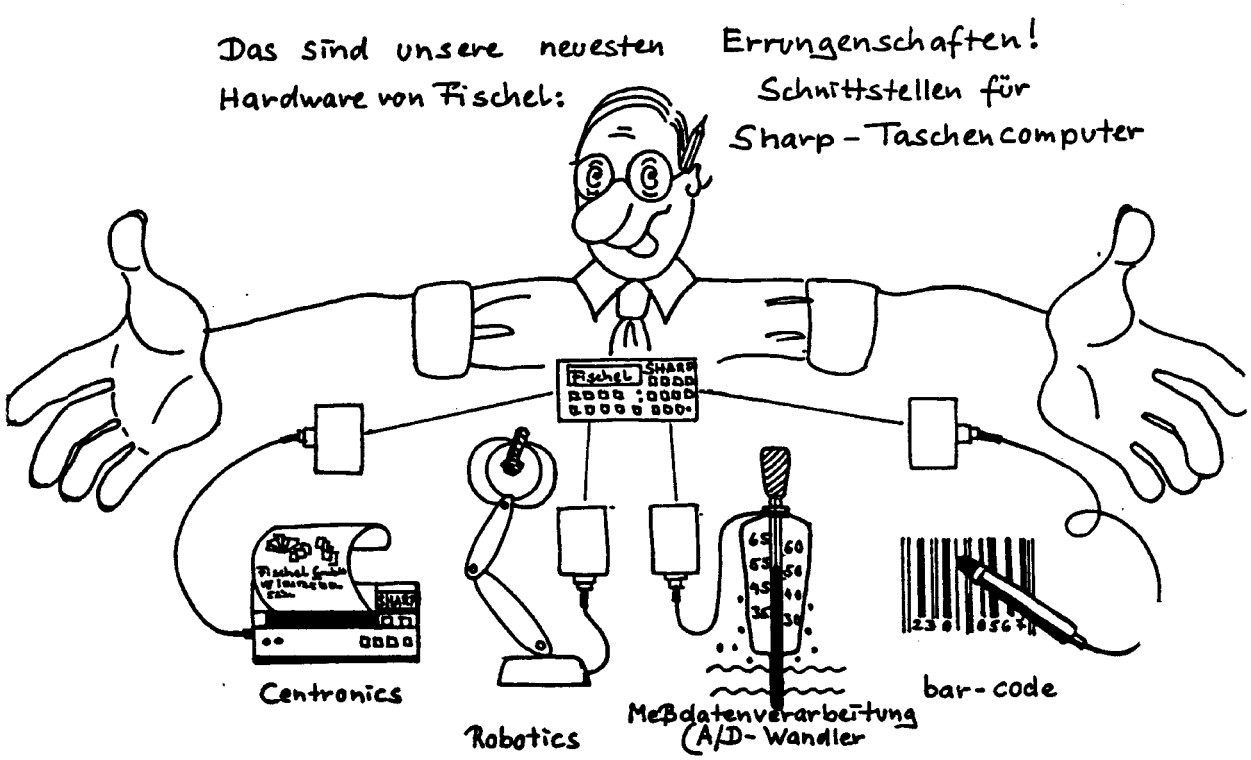
Trifft der Interpreter auf die FOR-Anweisung, so legt er

- die Position an der gerade gearbeitet wird,
- den mit **STEP** definierten Inkrement
- und die obere Zählgrenze

auf den FOR-NEXT-Stapel. Er befindet sich, wie der RETURN-Stapel im Bereich &FA38 bis &FAFF, wächst aber von unten nach oben. Der zugehörige Stapelzeiger liegt in &F890.

Wichtig zu wissen ist, daß beim Zurückspringen an den Anfang der Schleife keine Zeilennummer gesucht

werden muß, da der Interpreter direkt Über die auf dem Stapel liegende Speicheradresse dorthin gelangt. Deshalb laufen **FOR**-Schleifen im allgemeinen schneller als entsprechende Schleifen mit **GOTO**



SPEICHERPLATZOPTIMIERUNG BEI BASIC-PROGRAMMEN

Allgemeines

Der vom Benutzer verfügbare Speicher des PC-1600 ist mit bis zu 77 KB für Pocketcomputerverhältnisse geradezu riesig. Trotzdem kann es schon vorkommen daß ein Programm mangels einiger weniger Bytes nicht fertiggestellt werden kann. Die folgenden Abschnitte geben Hinweise, wie an der einen oder anderen Stelle etwas Speicherplatz eingespart werden kann. Dabei bieten sich prinzipiell drei Möglichkeiten an:

1. **Komprimierung des Programms**
   Dies bringt normalerweise eine Einsparung von ca. 10 bis 30% des durch das Programm belegten Speichers.

2. **Bessere Ausnutzung des Variablenspeichers**
   Auch hier können je nach Anwendung etwa 10 bis 80% eingespart werden.

3. **Vergrößerung des zur Verfügung stehenden Speicherplatzes**
   Hier wird nicht das Programm selbst, sondern die Programmumgebung optimiert. Bei nahezu allen Anwendungen kann etwa 1.5 KByte zusätzlicher Speicherplatz gewonnen werden.

Komprimierung von Programmen

Weglassen von Kommentaren

Dies ist wahrscheinlich die am einfachsten realisierbare Methode. Ein gutes Programm besteht etwa zur Hälfte aus Kommentaren. Da eine REM-Zeile mehr Speicher belegt als eine gleichlange ausführbare Zeile, könnte schon allein durch das Entfernen der REMs über 50% gespart werden.

Verwendung kurzer Variablennamen

Da die Variablennamen in Programmen durch Zeichen fÜr Zeichen als ASCII-Code dargestellt wird, sollten sie möglichst kurz sein. Es sind ohnehin immer nur die ersten beiden Zeichen signifikant, so daß ohne viel Mühe ein bestehendes Programm dadurch vereinfacht werden kann, daß alle Variablennamen auf zwei Zeichen gekürzt werden.

Noch besser ist natürlich die Verwendung von einzelnen Buchstaben als Namen, leider gibt es nur 26 davon.

Zusammenfassen von Programmzeilen

Jede Zeile enthält 4 Bytes an Informationen fÜr den Interpreter (siehe Kapitel "Arbeitsweise des BASIC-Interpreters"). Werden zwei Zeilen zu einer zusammen gefaßt, so werden zunächst 4 Bytes gespart, eines davon geht allerdings durch die Not-

wendigkeitdes Doppeltpunktes als Trennzeichenwieder verloren.

Da der Anweisungsteil einer durchschnittlichen BASIC-Zeile selten mehr als 20 Bytes umfaßt, aber maximal 80 Bytes enthalten darf, kann die Gesamtzahl der Programmzeilen auf ein Viertel verringert werden. Daraus ergibt sich eine Ersparnis von 9%.

Natürlich muß beachtet werden, daß eine Zeilennummer, die Sprungziel eine GOTO-Anweisung ist, als solche bestehen bleibt. Ebenfalls Vorsicht geboten ist bei Zeilen mit IF-Anweisungen.

Beispiel: Zeichnen eines Kreises um (X,Y) mit dem Radius R

```
10 INPUT "X";A
20 INPUT "Y";B
30 INPUT "R";R
40 D=1/R
50 X=R
60 Y=0
70 FOR I=0 TO 6.3*R
80 PSET (A+X,B+Y)
90 X=X-D*Y
100 Y=Y+D*X
110 NEXT I
```

oder aber:

```
10    INPUT "X";A,"Y";B,"R";R: D=1/R: X=R:
      Y=0:
      FOR I=0 TO 6.3*R: PSET (A+X,B+Y): X=X-D*Y
      : Y=Y+D*X: NEXT I
```

Die Speicherplatzersparnis beträgt in diesem Beispiel Über 30%.

IF-Anweisungen

Diese lassen sich umständlich

```
80 IF A>B THEN GOTO 200 : ELSE GOTO 300
```

oder einfach

```
80 IF A>B GOTO 200 ELSE GOTO 300
```

schreiben (3 Bytes gespart).

INPUT

Statt

```
10 INPUT "X-Wert: ";X
20 INPUT "Y-Wert: ";Y
30 INPUT "Z-Wert: ";Z
```

schreibt man besser

```
10 INPUT "X-Wert: ";X, "Y-Wert: ";Y,
"Z-Wert: ";Z
```

Man spart in diesem Fall 10 Bytes.

String-Konstanten

Auch wenn es unschön aussieht: Man kann bei String-Konstanten am Ende einer Zeile die abschließenden "GänsefÜßchen" weglassen:

```
50 IF A=0 LET M$="So nicht
```

Sonstiges

Folgende Beispiele sprechen fÜr sich:

	1.0		1
	0.1		.1
nicht	**1000**	sondern	1E3
	Pi		]
	SQR		[

Einsparen von Variablenspeicher

Auch bei den Variablen kann durch die Ausnutzung spezieller Eigenschaften der Variablenverwaltung Speicherplatz gespart werden.

Verwendung von Standardvariablen

Wo immer möglich, sollten die 2*26 Standardvariablen verwendet werden, da diese, egal ob benutzt oder unbenutzt, immer denselben Speicherbereich belegen.

Speicherung kleiner Zahlenwerte

Ein Array mit 5000 Elementen belegt 7+5000*8=40007 Bytes. Sind die Elemente nur kleine Zahlen (< 256), so ist es vorteilhaft, sie als Stringarray darzustellen:

```
DIM A$(24,199)*1
```

Mit Z = ASC A$(I) und A$(I)CHR$ Z

kann auf die einzelnen Elemente zugegriffen werden. Diese Methode ist zwar langsamer, benötigt aber nur 7+5000*1 = 5007 Bytes Variablenspeicher. Damit sind über 87% gespart.

Zugriff auf einzelne Bits

Manchmal werden große Arrays von boolschen Werten (z.B. nur 0 und 1) benötigt. Dank der eingebauten **AND-** und **OR-**Operatoren kann elegant auf einzelne Bits in Zahlenwerten zugeriffen werden.
Im Anhang steht ein Programm zur Berechnung von Primzahlen nach der Methode von Eratosthenes, in dem jede Zahl durch ein einzelnes Bit repräsentiert-wird, das für jede zusammengesetzte Zahl eine 1 zugewiesen bekommt. Die nahezu optimale Ausnutzung des Speichers ermöglicht die Berechnung aller Primzahlen bis Über 150000 ohne Verwendung einer Speichereweiterung. Werden die Bitflags durch ein Array von Fließkommazahlen ersetzt, sinkt diese Zahl auf etwa 2500.

VergrÖßerung des zur Verfügung stehenden Speicherplatzes

Der PC-1600 besitzt in der Grundversion 16 KByte RAM, von denen aber dem Benutzer nur 11834 Bytes, bei Verwendung eines Diskettenlaufwerks sogar nur noch 10810 Bytes zur Verfügung stehen. Der Rest wird für den Reservespeicher, Ein-/Ausgabepuffer

Systemvariablen usw. benötigt. Ist man bereit, auf einige Funktionen zu verzichten, kann der davon belegte Speicherplatz u.U. BASIC-Programmen zugänglich gemacht werden.

Verzicht auf die Funktionstasten

Der Reservespeicher belegt (in der Grundversion) den Bereich von &C008 bis &COC4. Da der Bereich von &COOO bis &C007 ohnehin ohne Funktion ist, kann man 197 Bytes hinzu gewinnen, wenn man den Zeig-er auf den Programmanfang auf &COOO setzt. Dies geschieht folgendermaßen:

```
POKE &F865, &40, &00
NEW
```

Nun dürfen weder die Funktionstasten belegt noch ein **ALARM$** gesetzt werden, da das im Speicher befindliche Programm dadurch beschädigt würde.

Verzicht auf Diskettenzugriffe und serielle Ein-/Ausgabe

Die Puffer für Diskette, RS-232C und SIO stehen am oberen Ende des Anwenderbereichs. Mit

```
POKE &F864, &F0 : CLEAR
```

gewinnt man bis zu 1280 Bytes hinzu.

Nun muß natürlich auf alle Diskettenzugriffe verzichtet werden. Nicht einmal **FILES** ist mehr erlaubt. Da der hinzugewonnene Bereich meist nur von Variablen genutzt wird, ist es sinnvoll, am Ende

eines Programms die Adresse &F864 wieder mit ihrem alten Wert zu versehen. Damit werden zwar die Variablen gelöscht, dafür kann man das Diskettenlaufwerk wieder benutzen.

Insgesamt laßt sich der BASIC-Speicherbereich also um über 13% erweitern.

LAUFZEITOPTIMIERUNG VON BASIC-PROGRAMMEN

Allgemeines

Immer wieder kommt es vor, daß ein sehr rechenintensiver Algorithmus programmiert werden soll und das Programm aus einem der folgenden Gründe nicht schnell genug läuft:

- Der Programmlauf dauert Stunden, Tage oder gar Wochen und man möchte nicht gern so lange auf das Ergebnis warten.

- Das Programm soll zur Steuerung von Peripherie eingesetzt werden, die nicht auf den Rechner warten kann, soll oder will.

Eine naheliegende Lösung des Problems besteht darin, eine andere Programmiersprache, z.B. Assembler zu wählen. Im ersten Fall ist das auch meist die einzig praktikable, da eine Erhöhung der Geschwindigkeit um eine ganze Größenordnung, wenn überhaupt, nur auf diese Weise möglich ist.

Ist jedoch schon eine Geschwindigkeitssteigerung von vielleicht 30% ausreichend, sollte man zuerst versuchen, diese durch geschicktere Formulierung des Problems in der bereits verwendeten Sprache, in unserem Fall BASIC, herbeizuführen. Im folgenden werden dafür einige Methoden und Tricks vorgstellt, mit deren Verwendung durchschnittliche BASIC-Programme um 10 bis 50%, einige sogar um weit Über 100% schneller gemacht werden können.

Arithmetik

Grundrechenarten

Bei den vier Grundrechenarten kann Zeit gespart werden, wenn Multiplikationen durch Additionen und Divisionen durch Multiplikationen ersetzt werden:

```
A = 3 * B  --> A = B + B + B
A = B / 4  --> A = A * .25
```

Die Ganzzahldivision und die Restbildung **MOD** sind langsamer als ihre mit **INT** gebildeten Äquivalente:

```
A = B / C          A = INT (B / C)
A = B MOD C        A = A - C * INT (B / C)
```

Ist bei der **MOD**ulo-Funktion der zweite Operand eine Potenz von 2, so so kann sie durch die logische Operation **AND** ersetzt werden:

```
A = B MOD C   -->    A = B AND (C-1)
```

```
(C = 1, 2, 4, 8, ... )
```

Potenzen

Das Potenzieren mit ^ sollte nur dann geschehen, wenn der Exponent gebrochen ist. Da dies in der Praxis äußerst selten der Fall ist, man also fast immer mit ganzzahligen Exponenten zu tun hat, sollte man sich die folgenden Methoden näher anschauen:

Ist der Exponent positiv und kleiner als 80, so verläuft die Potenzierung schneller, wenn sie durch fortgesetzte Multiplikation ersetzt wird:

```
A = B^5      -->   A = B*B*B*B*B
```

Bei negativen Exponenten wird anschließend einfach der Kehrwert gebildet:

```
A = B^(-3)    -->   A = B*B*B : A = 1/A
```

Ist die Basis nicht eine einzelne Variable (in unserem Fall B), sondern ein Ausdruck, so ist es meist zweckmäßig, diesen nur einmal auszurechnen:

```
S = (SIN (4*X))^4        S = SIN(4*X)
                         S = S*S*S*S
```

Noch effizienter, vor allem bei relativ großen Exponenten, ist das folgende Verfahren, das auch noch die Zahl der benötigten Multiplikationen reduziert:

Beispielsweise soll **A = B^13** berechnet werden.
Die erste Anweisung lautet

```
A = 1
```

Sodann ermittelt man die Dualdarstellung des Exponenten (1101). Diese arbeitet man, von links beginnend, ab, indem man fÜr jede vorkommende 1

```
A = A*A*B
```

und für jede 0

```
A = A*A
```

ins Programm schreibt. In userem Fall sieht dies folgendermaßen aus:

```
A = 1 : A = A*A*B : A = A*A*B
A = A*A : A = A*A*B
```

Die ersten drei Anweisungen können noch geschickt zusammengefaßtwerden, so daß man schließlich erhält:

```
A = B*B*B : A = A*A : A = A*A*B
```

Somit hat man die Zahl der benötigten Multiplikationen, von 12 auf 5 reduziert und man überzeugt sich leicht, daß damit tatsächlich die 13. Potenz berechnet wird.

Das Verfahren- läßt sich leicht in ein kleines Unterprogramm umsetzen so daß auch variable Exponenten zugelassen sind (siehe Anhang).

Obwohl die Berechnung einige Schleifendurchläufe benötigt, ist das Verfahren immer noch schneller als das Original, da dieses die Potenzen über Logarithmus, Multiplikation und Exponentiation bestimmt.

Trigonometrische und zyklometrische Funtionen

Bei den trigonometrischen und zyklometrischen Funktionen ist zu berücksichtigen, daß intern immer im Gradmaß gerechnet wird, unabhängig davon welches Winkelmaß (deg, rad oder grad) gerade eingestellt ist. Deshalb sollte man nur dann z.B. ins Bogenmaß schalten, wenn die Problemstellung dies erfordert. Dadurch wird dem Rechner die Umrechnung erspart.

Es sind vor allem dort Geschwindkeitsvorteile erreichbar, wo ein String in einer Schleife Zeichen für Zeichen abgearbeitet wird. DafÜr gibt es mehrere Verfahren, die anhand eines kleinen Beispiels erläutert werden:

Es sollen in der Variable A$ alle Kommata durch %-Zeichen ersetzt werden. Dies ist z.B. dann nötig, wenn der String in eine Textdatei geschrieben werden soll (Zeilen mit Kommata werden nicht vollständig wieder eingelesen).

1. Möglichkeit:
Zugriff auf die einzelnen Zeichen mit MID$

```
10 FOR I=1 TO LEN A$
20 IF MID$(A$,I,1) = "," LET
   A$ = LEFT$(A$,I-1) + "%" +
   MID$(A$,I+1,16)
30 NEXT I
```

2. Möglichkeit:
Zugriff auf die einzelnen Zeichen Über KBUFF$ und INKEY$(1)

```
10 L=LEN A$ : KBUFF$ = A$ : A$ = ""
20 FOR I=1 TO L
30 Z$ = INKEY$(1)
40 IF Z$ = "," LET A$ = A$+"%"
   ELSE LET A$ = A$+Z$
50 NEXT I
```

3. Möglichkeit:
Direkter Zugriff auf die Zeichen mittels PEEK und POKE

```
10 A = &F8BF
20 FOR I=1 TO LEN A$
30 IF PEEK(A+I) = 44 POKE A+I,37
40 NEXT I
```

In diesem Fall ist die dritte Möglichkeit die schnellste, bei anderen Anwendungen muß erst entschieden werden, welcher der drei Alternativen man den Vorzug geben will.

Sprunganweisungen

In kleinen Programmen werden Sprünge recht schnell ausgeführt, so daß nicht viel optimiert werden braucht. Ganz anders verhält es sich, wenn ein Programm einige hundert oder gar tausend Zeilen lang ist, was bei bis zu 80 KByte Speicher nichts außergewöhnliches sein dürfte. Ein Sprung von der 2001. in die 2000. Zeile dauert immerhin schon 110 ms. Dies hat seine Ursache in der Art und Weise, wie der BASIC-Interpreter nach Zeilennummern sucht:

Beginn der Suche nach Zeilennummern bei **GOTO**, **GOSUB** etc.:

	Sprung vorwärts	Sprung rückwärts
GOTO 1234	nächste Zeile	erste Zeile
GOTO "LABEL"	erste Zeile	erste Zeile

Aus der Tabelle ist zu entnehmen, daß vor allem die folgenden Fälle einer Optimierung bedürfen:

1. Vorwärtssprünge nach Zeilennummern, wenn zwischen der aktuellen Programmzeile und dem Sprungziel viele weitere Zeilen vorhanden sind

2. Rückwärtssprünge nach Zeilennummern, wenn dem Sprungziel viele weitere Zeilen vorangehen (Ein Sprung in die selbe Zeile gehört auch hierher.)

3. Alle Sprünge nach Marken, wenn dem Sprungziel viele weitere Zeilen vorangehen

Vorwärtssprünge gehen normalerweise nur Über eine geringe Distanz, es sei denn, sie dienen einem Unterprogrammaufruf. Stellt man kleine, häufig benötigte Unterprogramme nicht ans Ende des Programms, wie dies von vielen Programmierern getan wird, sondern ganz an den Anfang, so ist dieses Problem weitgehend behoben.

Rückwärtssprünge haben als Sprungziel meist eine Zeile, die früher schon einmal durchlaufen wurde. Damit sind sie Bestandteil von Schleifen, deren Optimierung im folgenden Abschnitt behandelt wird.

Es bringt nie *einen* Geschwindigkeitsnachteil (höchstens einen Vorteil) mit sich, wenn Sprungmarken durch Zeilennummern ersetzt werden. Für diese gilt dann das in den beiden vorangegangenen Absätzen Gesagte.

Schleifen

Schleifen sind nichts anderes als Programmabschnitte, bei denen am Ende ein (ggf. bedingter) Sprung zum Anfang erfolgt.

Eine besondere Form, die Zählschleife, ist als FOR ... **NEXT** bereits in der BASIC-Sprache vorhanden. Ein Leerschleifendurchlauf dauert, unabhängig von der Stellung der Schleif innerhalb eines Programms etwa 3.5 ms; das ist so wenig, daß hier nicht mehr viel verbessert werden kann.

Anders verhält es sich mit Schleifen, bei denen mit **GOTO** an den Anfang zurückgekehrt wird. Für ihr Zei.tverhalten gilt dasselbe wie für alle Rückwärtssprünge. Steht sie weit vom Programmanfang entfernt, so ist es zweckmäßig, sie durch eine FOR-Schleife zu auszudrücken. Dies kann nach einem der folgenden beiden Schemata geschehen:

Unbedingter Sprung zum Schleifenanfang:

```
        .                                        .
        .                                        .
        .                                        .
                                        FOR I=0 TO 1
50 <Anweisung>                          50 <Anweisung>
        .                                        .
        .              --->                      .
        .                                        .
     GOTO 50                            I=0 : NEXT I
```

Bedingter Sprung zum Schleifenanfang:

```
        .                          .
        .                          .
        .                          .
                              FOR I=1 TO 0 STEP -1
50 <Anweisung>                  50 <Anweisung>
        .          --->            .
        .                          .
        .                          .
IF <Bed> GOTO 50               I = <Bed> : NEXT I
```

IF-Anweisunge

Folgendes ist bei der Verwendung von IFs zu beachten:

- Das Suchen nach einem **ELSE** oder nach der nächsten Zeile kostet Zeit. Deshalb sollten in

```
    IF <Bed> THEN <Anwsg 1>
und IF <Bed> THEN <Anwsg 1> ELSE <Anwsg 2>
```

  die Anweisungen <Anwsg 1> und <Anwsg 2> nicht Übermäßig lang sein.

- Die selbe Anweisung(sfolge) wird im **ELSE**-Zweig etwas schneller abgearbeitet alsim **THEN**-Zweig. Deshalb sollte diejenige Aktion, die wahrscheinlicher ist, als ELSE-Zweig geführt werden. Gegebenenfalls ist die **IF**-Bedingung zu negieren.

```
IF RND 0 > .1 OR B<>3 LET C=C+1
```

```
ELSE LET C=C-1
```

würde umgeschrieben in

```
IF RND 0 <= 1 AND 3=3 LET C=C-1
ELSE LET C=C+1
```

NatÜrlich sollte nicht krampfhaft versucht werden, eine Bedingung zu negieren, die dadurch viel komplizierter würde, im Gegenteil:

. Hat die negierte Bedingung eine einfachere Form, so sollte man dieser den Vorzug geben und und die Anweisungen hinter dem **THEN** mit denen hinter dem **ELSE** vertauschen. Exisiert kein **ELSE**-Zweig, spielt das keine Rolle. Auch die Form

```
IF <bed> ELSE <anw>
```

ist zulässig.

. In fast allen Fällen kann das THEN entfallen. Eine der wenigen Ausnahmen zeigt das folgende Beispiel:

```
IF A>G THEN PRINT A
```

würde ohne THEN vom Interpreter aufgefaßt als

```
IF A> GPRINT A
```

NatÜrlich kann auch dieses umgangen werden, wenn man statt A>G einfach G<A schreibt.

. Verknüpfungen von Bedingungen mittels AND sollten durch mehrere ineinanderverschachtelte IFs ersetzt werden:

nicht `IF A>64 AND A<91 AND A<>Z LET A=A+32`

sondern IF A<91 IF A>64 IF A<>Z LET A=A+32

Dies bringt gleich zwei Vorteile. Erstens ist ein IF schneller wie ein **AND**. Zweitens brauchen, wenn bereits die erste Bedingung nicht erfüllt ist, die restlichen Bedingungen gar nicht erst ausgewertet zu werden. Deshalb sollten die Bedingungen nach der Wahrscheinlichkeiten, mit der sie erfüllt werden, geordnet werden.

. Verknüpfungen mit **OR** können auch als Additionen dargestellt werden (Eine Addition ist schneller als eine logische Verknüpfung.):

nicht IF A>B OR A<>C GOTO 100

sondern IF (A>B) + (A<>C) GOTO 100

Noch bessere Ergebnisse erzielt man, wenn die **OR**s durch **AND**s darstellt und diese dann durch verschachtelte **IF**s ausdrückt. Im obigen Beispiel sieht das dann so aus:

```
      IF A<=B AND A=C ELSE 100

--->  IF A>B GOTO 100 ELSE IF A<>C GOTO 100
```

Variablenzugriffe

Der Zugriff auf Variablen ist eine der am häufigsten vorkommenden Oprationen des BASIC-Interpreters. Z.B. muß bei AusfÜhrung der Anweisung

```
Y = F1 * A(X,Y) + F2 * B(X,Y)
```

neunmal ein Variablenname erkannt, die zugehörige Adresse im Speicher bestimmt und der 8 Byte lange Inhalt gelesen bzw. geschrieben werden.

Erkennen eines Variablennamens

Ein Name wird natürlich am schnellsten als solcher erkannt, wenn er möglichst kurz ist. Deshalb sollte man von allzu langen Bezeichnungen wie z.B. HYPOTHENUSE absehen und dafür treffende AbkÜrzungen verwenden.

Lokalisieren der Speicheradresse

Standardvariablen befinden sich immer an der selben Stelle im Speicher und können deshalb am schnellsten lokalisiert werden.

Nichtstandardvariablen hingegen werden dynamisch verwaltet und müssen daher immer erst gesucht werden. Diese Suche beginnt immer bei den zuletzt definierten Variablen, so daß sich fÜr diese die kürzesten Zugriffszeiten ergeben.

Deshalb sollte man die am seltesten benötigten Variablen gleich zu Anfang definieren, um die 'guten Plätze' aufzuheben.

Ist eine Variable trotz aller BemÜhungen auf einen 'schlechten Platz' gerutscht, so kann mit **ERASE** bewirkt werden, daß sie beim nächsten Zugriff neudefiniert wird. Den Unterschied zeigt das folgende Programm:

```
10 ' Stellvertretend fÜr ein langes Programm
```

```
11 ’  werden hier 100 Variablen folgendermaßen
12 ’  mit Werten vorbelegt:
13 ’
14 ’  A0=1 A1=1 A2=1 ... A9=1
15 ’  B0=1 B1=1 B2=1 ... B9=1
16 ’  C0=1 C1=l C2=1 ... C9=1
17 ’
18 ’
19 ’
20 ‘  J0=1 J1=1 J2=1 ... J9=1
21 ‘
40 I=ASC "A" : J=ASC "0"
50 KBUFF$ = CHR$ I + CHR$ J"=1" + CHR$ 13
    + "GOTO 70" + CHR$ 13
60 END
70 J = J+1 : IF CHR$ J <= "9" GOTO 50
80 J = ASC "0" : I = I+1
90 IF CHR$ I <= 'litt GOTO 50
100 ‘
110 PRINT "Ohne ERASE"
120 GOSUB 200
130 PRINT "Mit ERASE"
140 ERASE A0, A1, A2, A3
150 GOSUB 200
160 ‘
200 PRINT "Anfang" : BEEP 1
210 FOR A0=1 TO 2000
220 A1 = A0 : A2 = A0 : A3 = A0
230 NEXT A0
240 BEEP 1   : PRINT "Ende"
250 RETURN
```

Noch aufwendiger ist die Lokalisierung von Arrayelementen. Wie bei den einfachen Nichstandardvariablen muß auch hier zuerst der Name gesucht. Darüberhinaus ist der relative Abstand des gesuchten Elements zur Basisadresse zu bestimmen. Dies erfordert bis zu zwei Multiplikationen. Wird auf ein einzelnes Arrayelement sehr oft in einer Schleife zugegriffen, so ist zu prüfen, ob vielleicht eine

zeitweilige Übertragung in eine einfache Variable Vorteile bringt:

```
10 N = 99
20 DIM A(N,7)
30 S = 3
35 X = RND -1234
40 FOR I=0 TO N
50 A(I,S)=RND 1000
60 NEXT I
70 BEEP 1:GOSUB 100:BEEP 1
80 END
90 ‘
100 ‘Sortieren der S-ten Spalte einer Matrix
110 FOR I=0 TO N-1
120 FOR J=I+1 TO N
130 IF A(J,S)<A(I,S) LET H=A(J,S): A(J,S)=A(I,S): A(I,S)=H
140 NEXT J:NEXT I
150 RETURN
```

Anmerkung: Die Zeile 35 (RND mit negativem Argument) bewirkt, daß bei jedem Programmlauf genau dieselbe Folge von Zufallszahlen erzeugt wird. Nur so ist ein gerechter Vergleich beider Alternativen möglich.

Das Programm sortiert eine einzelne Spalte in einem zweidimensionalen Array (alle Elemente, für die der zweite Index konstant bleibt). Auffallend ist, daß in der inneren Schleife wiederholt auf A(I,S) zugegriffen wird, obwohl I und S ihren Wert dabei nicht verändern. Deshalb ist es hier zweckmäßig, A(I,S) für die Dauer dieser Schleife durch eine Standardvariable A zu ersetzen. Folgende Änderungen sind vorzunehmen:

```
115   A = A(I,S)
130   IF A(J,S) < A LET H=A(J,S) :
      A(J,S)= A : A = H
140   NEXT J: A(I,S)=A : NEXT I
```

Sperren von Interrupts

Der Prozessor erhält alle 1/64 sec und alle 0,5 sec einen Interrupt. Dabei unterbricht er das laufende Programm und führt verschiedene Aktionen durch, die ständig wiederholt werden müssen, z.B. die Abfrage der Tastatur, des CI-Eingangs, des Timers usw. (näheres dazu siehe Kapitel "Interrupts").

Die Abarbeitung der zugehörigen Interruptroutinen nimmt etwa 6% der Gesamtrechenzeit in Anspruch. Eine Geschwindigkeitssteigerung ist daher durch Sperren der Interrupts möglich:

```
OUT &35, &1F    1/2s-Interrupt sperren
OUT &35, &4F    1/64s-Interrupt sperren
OUT &35, &0F    beide Interrupts sperren

OUT &35, &5F   Alle Interrupts wieder
               freigeben, Normaleinstellung
```

Achtung: Wird ein Programm aufgrund eines Fehlers oder durch DrÜcken der ON-Taste abgebrochen oder wird per **INPUT** eine Eingabe erwartet, während der 1/64s-Interrupt gesperrt ist, gerät der Rechner außer Kontrolle, da keine Tastatureingaben mehr möglich sind. Der einziger Ausweg ist Reset.

Bevor mit dem Sperren der Interrupts gearbeitet wird, sollte also sichergestellt sein, daß das Programm wirklich fehlerfrei läuft. Desweiteren sollte ein unbeabsichtes Unterbrechen per ON-Taste durch **BREAK OFF** verhindert werden.

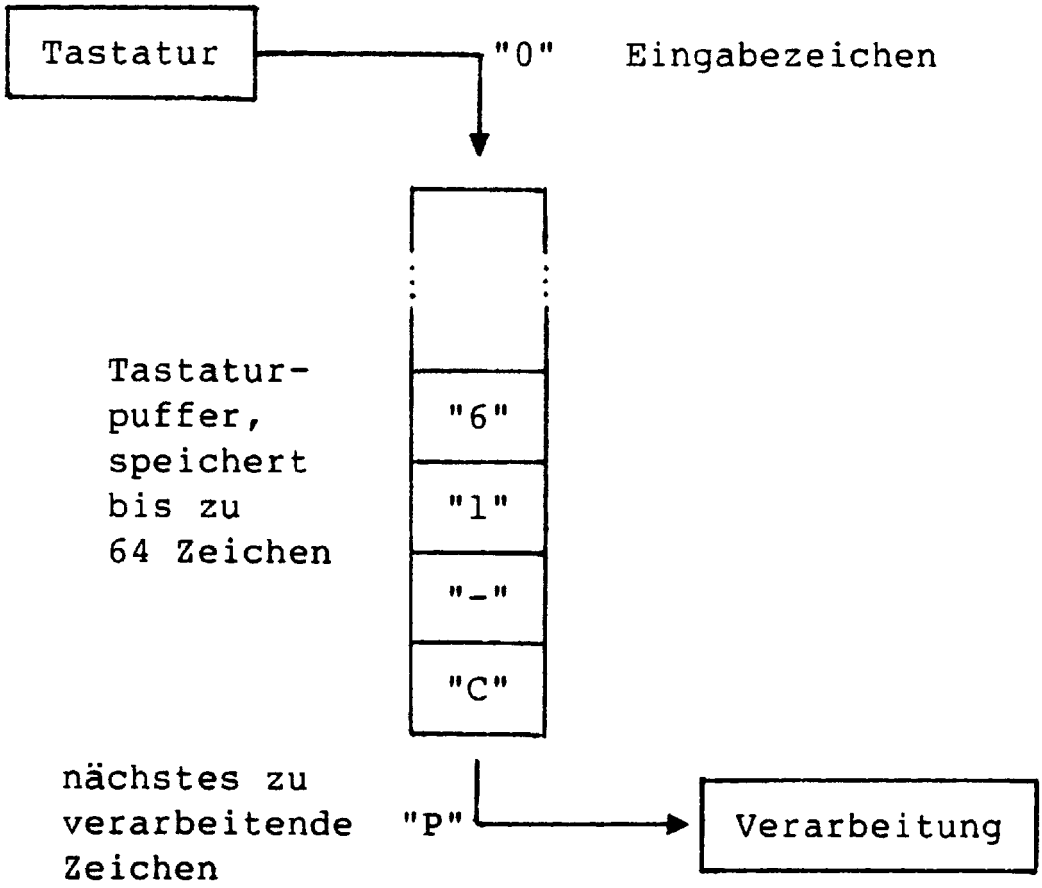
DER TASTATURPUFFER

Der PC-1600 erlaubt es, im Gegensatz zu seinen Vorgängern, über die Tastatur auch dann Befehle, Daten etc. einzugeben, wenn er sie nicht sofort verarbeiten kann, z.B. während der Ausführung eines langen Programms. Beispielsweise kann beim folgenden Programm der Wert fÜr A schon eingegeben werden, bevor die Schleife beendet ist.

```
10 FOR I=1 TO 3000 : NEXT I
20 INPUT "A ="; A
```

Funktionsweise

Jeder Tastendruck wird zunächst in einen Puffer geschrieben. Erwartet der Rechner eine Eingabe, so liest er die Zeichen in derselben Reihenfolge, wie sie eingegeben worden sind, aus diesem Puffer wieder aus. Ist der Puffer leer, so wird auf den nächsten Tastendruck gewartet. Folgendes Diagramm verdeutlicht dies:



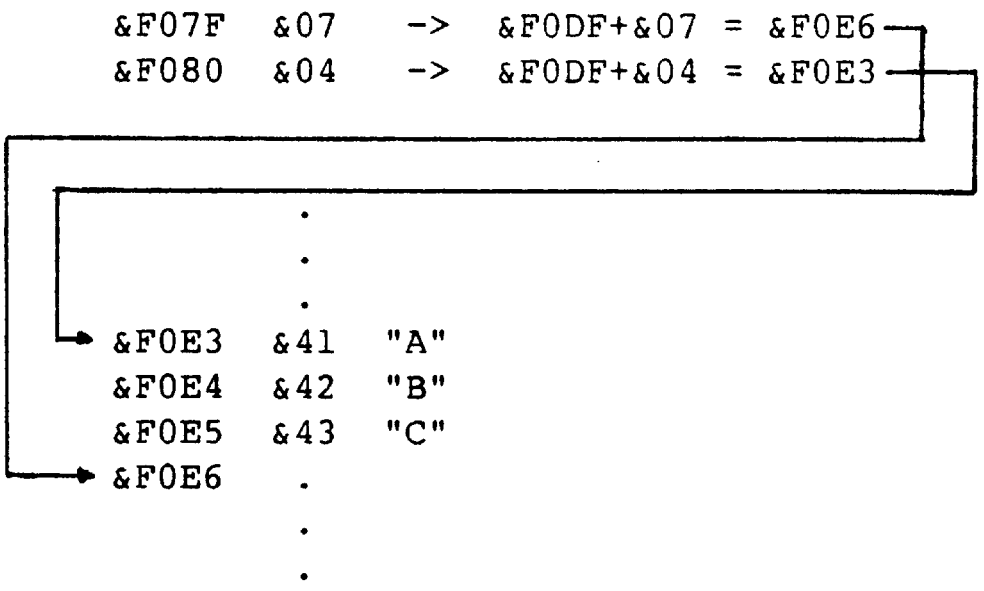
Die eingegebenen Zeichen (in der Skizze die "0") wandern am einen Ende in den Puffer hinein und können am anderen Ende wieder abgeholt werden (das "P"). Man erkennt deutlich, daß der Benutzer das Wort "PC-1600 " schneller eingibt, als es verarbeitet wird.

Der Tastaturpuffer wurde softwaremäßig realisiert, d.h. die Zeichen werden vom Prozessor selbst hineingeschrieben. Dazu erhält er alle 1/64 Sekunden einen Interrupt, der ihn auffordert, die Tastatur abzufragen.

Für den benötigten Speicher ist ein Bereich im System-RAM reserviert und belegt dort die Adressen &F0DF bis &F11E. Zwei Pointer (Adressen &F07F und &F080) zeigen auf den Kopf bzw. den Schwanz der Eingabeschlange. Sie zeigen an, an wievielter Stelle das nächste Zeichen eingetragen bzw. herausgelesen wird. Die zugehörige absolute Adresse

ergibtsich durch Addition des Pufferanfangs (&F0DF).

Beispiel:



Im Puffer stehen die 3 Zeichen A, B, C. Wird nun z.B. ein D eingegeben, so wird &44 (=ASC("D")) in die Adresse &F0E6 geschrieben und anschließend der Kopfzeiger um eins erhöht (&07 --> &08). Ist dieser bei 64 angelangt, so wird er wieder auf 0 gesetzt. Wird der Puffer mit genau 64 Zeichen gefüllt, so beißt der Kopf in den Schwanz, d.h. die Adressen &F07F und &F080 enthalten denselben Wert. Um den vollen Puffer vom leeren zu unterscheiden, bei dem ja ebenfalls Kopf=Schwanz ist, wird bei ersterem Bit 7 in &F07F auf 1 gesetzt. Ist der Puffer voll, werden solange keine weiteren Eingaben angenommen, bis ein Teil der gespeicherten Zeichen verarbeitet worden ist.

Wird vom Rechner eine Eingabe angefordert, so wird zuerst ein Zeichen aus dem Puffer gelesen und dann der Schwanz um eins erhöht. Ist bereits Kopf=Schwanz, dann bedeutet dies, daß der Puffer leer ist.

Zugriff auf den Tastaturpuffer

INKEY$(1), INPUT

INKEY$(1) liest genau ein Zeichen aus dem Tastaturpuffer. Ist dieser leer, so wird der Leerstring "" zurückgegeben und das Programm fortgesetzt.

Zum Vergleich: Die Funktionen **INKEY$** bzw. **INKEY$(0)** haben mit dem Tastaturpuffer überhaupt nichts zu tun. Mit ihnen wird geprüft, welche Taste im Augenblick gerade gedrückt ist.

Mit **INPUT** wird immer eine ganze Zeile aus dem Tastaturpuffer gelesen. Das Ende derselben ist durch den Code &0D (ENTER-Taste) markiert. Ist der Puffer leer, so wird, im Gegensatz zu **INKEY$(1)**, solange gewartet, bis weitere Eingaben von der Tastatur gemacht werden.

KBUFF$

Das einzige Mittel, in den Puffer hineinzuschreiben, ist - abgesehen von POKE und der Eingabe per Tastatur - der Befehl **KBUFF$.**

Mit **KBUFF$** wird ein beliebiger String (max. 64 Zeichen lang) in den Puffer geschrieben. Dabei geht dessen alter Inhalt verloren. Mit Hilfe der **CHR$**-Funktion ist es möglich, alle Zeichen zu verwenden, die auch von der Tastatur eingegeben werden können, also auch alle Sondertasten, wie z.B.
ENTER, MODE, die Cursortasten usw. Dies läßt eine

Vielzahl von Anwendungen zu, die in anderen BASIC-Dialekten nur mit viel Mühe realisierbar sind:

Beispiele:

Vereinfachte Wiederholung von Eingaben

Das folgende Programm berechnet aus Höhe, Radius und Dichte die Masse eines Kreiszylinders. Bei Wiederholung der Berechnung mit ähnlichen Werten, brauchen diese nicht neu eingegeben zu werden, da sie aus denn vorher verwendeten mit den Editiertasten erzeugt werden können.

```
  5 CLS
 10 PRINT "Höhe:   ";
 15 W=H : GOSUB "EINGABE" : H=W
 20 PRINT "Radius: ";
 25 W=R : GOSUB "EINGABE" : R=W
 30 PRINT "Dichte: ";
 35 W=D : GOSUB "EINGABE" : D=W
 40 V=R*R*PI*H
 50 M=V*D
 60 PAUSE "Masse:  ";
 70 GOTO 5
100 "EINGABE"
105 W$ = STR$ W
110 FOR I=1 TO LEN W$
115 'CHR$ 8 = Cursor nach links
120 W$ = W$ + CHR$ 8 : NEXT I
130 KBUFF$ = W$ : INPUT " "; W
140 RETURN
```

Interaktive Eingabe von Funktionstermen

Möchte man ein Funktionsplotprogramm o.ä schreiben, so stellt sich jedesmal die Frage, auf welche Art und Weise Funktionsterme eingegeben werden sollen. Meist wird das Problem dadurch umgangen, daß man den Anwender vor dem eigentlichen Pro-

grammlauf im PRO-Modus eine Programmzeile ähnlich der folgenden

```
480 Y = EXP(X) * SIN(X)
```

eingeben läßt. Abgesehen davon, daß diese Methode nicht besonders benutzerfreundlich ist, ergeben sich Gefahren fÜr das Programm, wenn statt 480 versehentlich eine andere Zeilennummer eingegeben wird. Die folgende Lösung nimmt dem Anwender das Ändern des Programms ab, er muß lediglich noch den Term selber angeben.

```
10 ' Interaktive Eingabe
20 ' von Funktionstermen
30 '
40 INPUT "F(X) = ";F$
50 'CHR$31 ist die MODE-Taste
60 KBUFF$ = CHR$31 + "120 Y=" + F$
   + CHR$13 + CHR$31 + "GOTO 8011 +CHR$13
70 END
80 'Fortsetzung des Programms
         .
         .
         .
120 'In diese Zeile wird der Funktions-
130 'term geschrieben.
         .
         .
         .
```

Direkter Zugriff

Nachdem bekannt ist, wo der Tastaturpuffer liegt und wie er organisiert ist, kann versucht werden, ihn auf direktem Wege, d.h. mit Hilfe von **PEEK** und **POKE**, zu bearbeiten. Dazu sind im folgenden einige Beispiele angegeben:

Zerstörungsfreie Abfrage

Es kann ermittelt werden,

- wieviele Zeichen momentan gepuffert sind:

```
F=PEEK &F07F - PEEK &F080
IF F<0 LET F=F+64
IF F>64 LET F=64
```

- oder ob das nächste Zeichen eine Ziffer ist

```
IF PEEK &F07F <> PEEK &F080
  LET A$=CHR$ PEEK (&F0DF+PEEK &F080)
  ELSE LET A$=""
IF A$>="0" IF A$<="9"
```

und das alles, ohne am Inhalt des Puffers etwas zu ändern, wie dies bei der Anwendung von INKEY$(1) oder INPUT zweifellos der Fall sein würde.

Einfügen eines Strings vor den schon bestehenden Pufferinhalt

Das Beschreiben des Puffers mit KBUFF$ hat den Nachteil, daß ein schon existierender Inhalt Überschrieben wird. Ist z.B. die Routine zur Funktionseingabe im vorigen Abschnitt Bestandteil eines langen Programms, so gehen bei ihrem Aufruf alle TastendrÜcke verloren die der Benutzer zuvor gemacht hat, da die KBUFF$-Anweisung verwendet wird. Dies kann verhindert werden, wenn das Programm folgender maßen abgeändert wird:

```
55 DIM K$(0)*80
60 K$(0) = CHR$(1) ...
65 GOSUB 500

500 ' KBUFF$ ohne Löschen
510      .
```

Das Originallisting des Unterprogramms ist im Anhang abgedruckt.

Nach dem Programmende in Zeile 65 werden zuerst die mit Hilfe des Unterprogramms neu in den Puffer eingetragenen Befehle bearbeitet, genauso, wie es auch in der ersten Version mit KBUFF$ der Fall war. Der alte Inhalt bleibt aber nach wie vor stehen, so daß er nach dem RÜcksprung ins Programm mit GOTO 70 abgefragt werden kann, als ob nie etwas geändert worden wäre.

DER RESERVESPEICHER

Der Reservespeicher reicht von &C000 bis &C0C4 und ist folgendermaßen unterteilt:

&C000 bis &C007	unbenutzt
&C008 bis &C021	Funktionstastenbeschriftung für Ebene 1
&C022 bis &C03B	Funktionstastenbeschriftung fÜr Ebene 2
&C03C bis &C355	Funktionstastenbeschriftung für Ebene 3
&C056 bis &C0C4	Tastenbelegung

Die einzelnen Tastenbelegungen können verschieden lang sein, und werden in der Reihenfolge abgespeichert, in der sie eingegeben werden. Am Anfang jeder Belegung steht ein Byte, das angibt, um welche Taste und welche Ebene es sich handelt:

	Ebene		
	I	II	III
F1	&01	&11	&09
F2	&02	&12	&0A
F3	&03	&13	&0B
F4	&04	&14	&0C
F5	&05	&15	&0D
F6	&06	&16	&0E

Das Ende des genutzten Reservespeicherbereichs wird durch ein &00-Byte angezeigt.

Im Anhang sind Programme zur Belegung der Funktionstasten und zum Anzeigen der Belegung (ohne SHIFT MODE) zu finden.

DER LAUTSPRECHER

Der **PC-1600** hat einen kleinen Piezo-Lautsprecher eingebaut der auf verschiedenste Art und Weise in Aktion versetzt werden kann:

- Durch den Nebenprozessor **LU57813** (Tastaturclick, ALARM$ und WAKE$)
- Beim Aufzeichnen und Einlesen von Daten über den Cassettenrecorderanschluß

- Softwaremäßig über einen der Ein-/Ausgabeports (Von dieser Möglichkeit macht auch die BEEPAnweisung Gebrauch.)

Besonders die softwaremäßige Ansteuerung des Lautsprechers ist gut dazu geeignet, eigene Ideen zu verwirklichen.

Mit den Befehlen

OUT &18,&40 und OUT &18,&C0

kann der Lautsprecher zu einer Schwingung in die eine oder in die andere Richtung angeregt werden. Werden beide Befehle abwechselnd und sehr schnell hintereinander ausgeführt, so entsteht ein Ton.

```
10 ‘ Erzeugung eines Tons ohne BEEP
20 BREAK OFF : OUT &35,&0F
30 FOR I=1 TO 500
40 OUT &18,&40 : OUT &18,&C0
50 NEXT I
60 OUT &35,&5F : BREAK ON
```

Da ein Schleifendurchlauf in diesem Programm immerhin etwa 10 ms benötigt, ist eher ein Brummen als ein Ton zu hören. Die Zeile sperrt bestimmte

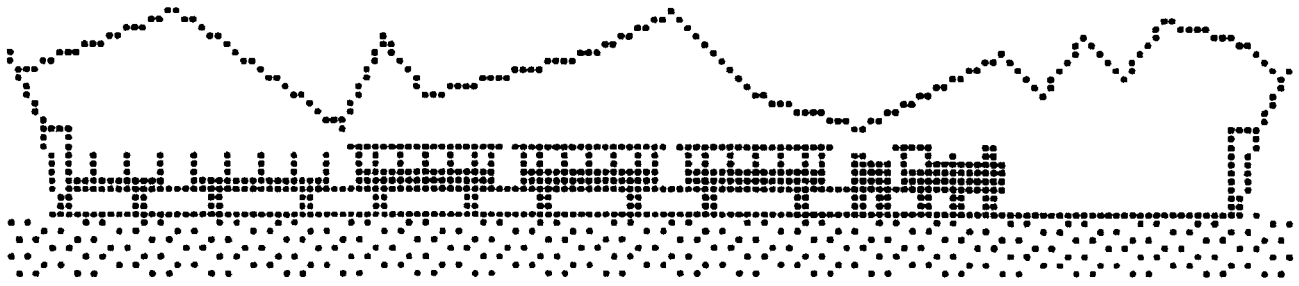
Interrupts des Prozessors, die regelmaßig zu kleinen Verzögerungen im Programmablauf führen. Laßt man Zeile 20 weg, macht sich dies in einem unregelmäßigen Klangbild bemerkbar.

Schönere und höhere Töne zu erzeugen, ist in BASIC leider nicht möglich. Man muß auf die Maschinensprache zurückgreifen. Es würde den Rahmen dieses Buches sprengen, alle Instruktionen des Prozessors (oder gar der beiden Prozessoren) zu erklären.

Ein Beispiel soll genügen: Im Anhang steht ein Programm, das eine Dampflokomotive fauchend und stöhnend Über das Display zischen läßt. Der dazu verwendete Rauschgenerator kann auch in anderen Programmen Anwendung finden. Die einzelnen Bytes in den Datazeilen können in beliebige freie Speicheradressen gePOKEt, da keine absolute Adressierung verwendet wurde. Aufgerufen wird das Unterprogramm mit

```
P=256*F+D
IF P>32767 LET P=P-65536
CALL <Anfangsadr>, P
```

D gibt die Dauer, F die Farbe (hell --> dumpf) des Rauschens an. Im Gegensatz zum BEEP-Befehl, wo die Tondauer auch von der TonhÖhe abhängt, sind in dieser Routine entsprechende Voraussetzungen getroffen worden, damit dies nicht der Fall ist.



Es besteht noch eine weitere Möglichkeit zur Tonerzeugung: Mit

OUT &17,&41

wird hardwaremäßig ein Dauerton mit fester Frequenz erzeugt, der solange anhält, bis er mit

OUT &17,0

wieder ausgeschaltet wird. Der Vorteil dieses Verfahrens liegt darin, daß während der Tonerzeugung das Programm weiterläuft. Leider ist der Ton dabei in seiner Höhe nicht variabel.

DER PLOTTER

Funktionsweise

Der CE-1600P ist ein sogenannter Trommelplotter, d.h. das Papier wird auf einer Walze vor- und zurück bewegt, während ein Schreibstift in horizontaler Richtung hin und herläuft.

Beide Bewegungen werden durch Schrittmotoren ausgeführt. Ein dritter Schrittmotor hat gleich zwei Aufgaben:
Zum einen besorgt er den Farbwechsel, indem er Über einen ausgeklügelten Mechanismus den Stiftträger um 90 Grad dreht, zum anderen bewegt er den Schreibstift in seiner Halterung vor und zurück.

Die zum Betrieb der Schrittmotoren erforderlichen vier Rechtecksignale werden fÜr alle drei Funktionen durch die Software im ROM des PC-1600 erzeugt. Dies ist auch der Grund dafür, daß während des Zeichnens einer Linie alle Interrupts gesperrt werden, so daß in dieser Zeit auch die Tastatur nicht abgefragt wird.
Sind die Motoren in Ruhe, so wird ihnen auch kein Strom zugeführt. Dies hat den Vorteil, daß der Stromverbrauch im Vergleich zu anderen Geräten, bei denen die Motoren dauernd unter Strom stehen, sehr gering ist.

Die Auflösung in x- und y-Richtung ist beim CE- 1600P doppelt so gut wie z.B. beim CE-150, nämlich 0,1 mm. Dies äußert sich vor allem darin, daß die beim Zeichnen von schrägen Linien immer entstehenden 'Treppenstufen' etwas unterdrückt werden.

Leider können die Endkoordinaten von Linien nur im 0,2-mm-Raster angegeben werden. Der Grund dafür liegt darin, daß die Motoren im Halbschrittverfahren betrieben werden (deshalb auch die doppelte Auflösung). Da sie in ruhendem Zustand stromlos sind, behalten sie ihre Stellung nur dann bei, wenn sie zwischen zwei Vollschritten angehalten werden.

Ein-/Ausgabeadressen

Der Plotter belegt im I/O-Bereich die zieldressen von &80 bis &8F. Über diese Adressen lassen sich die Schrittmotoren und die drei Tasten für den Papiertransport steuern bzw. abfragen. Im folgenden sind einige Funktionen angegeben:

				Abfrage
INP	&81	AND	&01	Farbwechseltaste
INP	&81	AND	&02	Papiertransporttaste vor
INP	&81	AND	&04	Papiertransporttaste zurück
INP	&81	AND	&10	PRINT-Schalter

Diese Adressen können nur dann mit Erfog abgefragt werden, wenn vorher mit

```
OUT &35,&1D
```

der Peripherieinterrupt gesperrt wird. Weitere Adressen (Ansteuerung der Schrittmotoren, Interruptmasken usw.) sind im Anhang beschrieben.

DAS DISKETTENLAUFWERK

Funktionsweise

Eine Diskette des PC-1600 hat zwei Seiten. Jede Seite enthält 16 konzentrische Spuren, die jeweils wieder in 8 Sektoren unterteilt sind, von denen jeder 512 Bytes umfaßt. Die Spuren werden dabei, beginnend mit der äußersten, von 0 bis 15 numeriert.

Ähnlich wie der Plotter ist auch das CE-1600F verblüffend einfach aufgebaut. Ein einziger Motor treibt die Diskette an und bewegt gleichzeitig den Schreiblesekopf von Spur zu Spur. Ein Elektromagnet besorgt die Koppelung des Kopfträgers mit dem Antriebsmotor.

Da die Drehrichtung des Antriebsmotors nicht geändert werden kann bewegt sich der Kopf normalerweise nach innen (zur Antriebsachse hin). An jeder der 16 Spuren kann er angehalten werden. Befindet er sich auf Spur 15, so führt die nächste Bewegung wieder ganz nach außen auf Spur 0. Auf dieser Strecke kann nicht angehalten werden.

Dies ist der Grunde dafür, daß eine Bewegung z.B. von Spur 5 nach Spur 6 relativ schnell ausgeführt wird, da dei Spur 6 in 'Fahrtrichtung' liegt. Soll der Kopf jedoch auf Spur 4 positioniert werden, muß er erst nacheinander die Spuren 6, 7, ... 15, 0, 1, 2 und 3 überstreichen, was länger dauert und sich auch in einem entsprechenden Geräusch bemerkbar macht.

Desweiteren enthält das Laufwerk zwei Kontakte, von denen einer auf den Schreibschutz, der andere

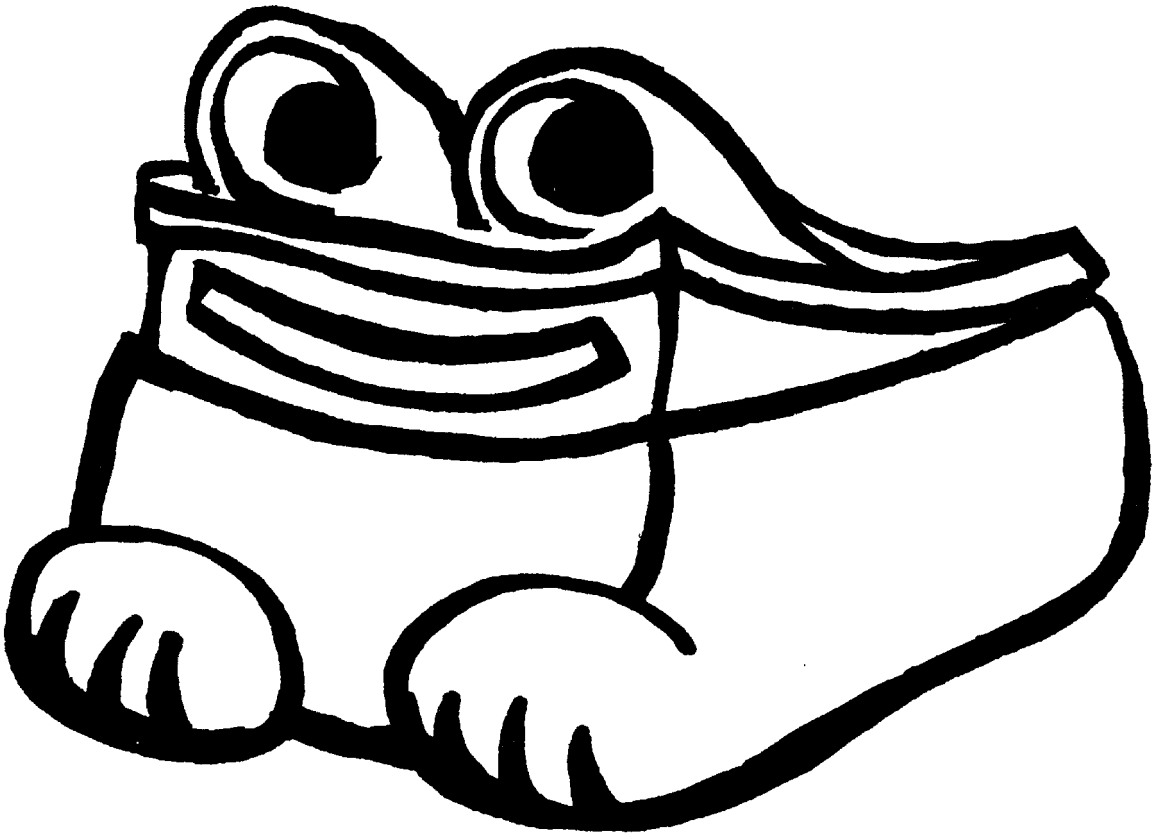
auf eine eingelegte Diskette reagiert. Beide können durch Programme abgefragt werden.

Ein-/Ausgabeadressen

Folgende Steuerbefehle und Abfragen sind beim **CE-1600F** möglich:

```
INP &78 AND &08 PrÜfen ob Diskette eingelegt
INP &78 AND &40 Schreibschutz abfragen
                (nur bei laufendem Motor)
OUT &7A,&80     Antriebsmotor einschalten
OUT &7A,0       Antriebsmotor ausschalten
```

Weitere Adressen sind im Anhang zu finden. Im Kapitel "Das Display" ist steht kleines Beispiel, daß u.a. auch von den Laufwerksfunktionen Gebrauch macht.



SERIELLE DATENÜBERTRAGUNG

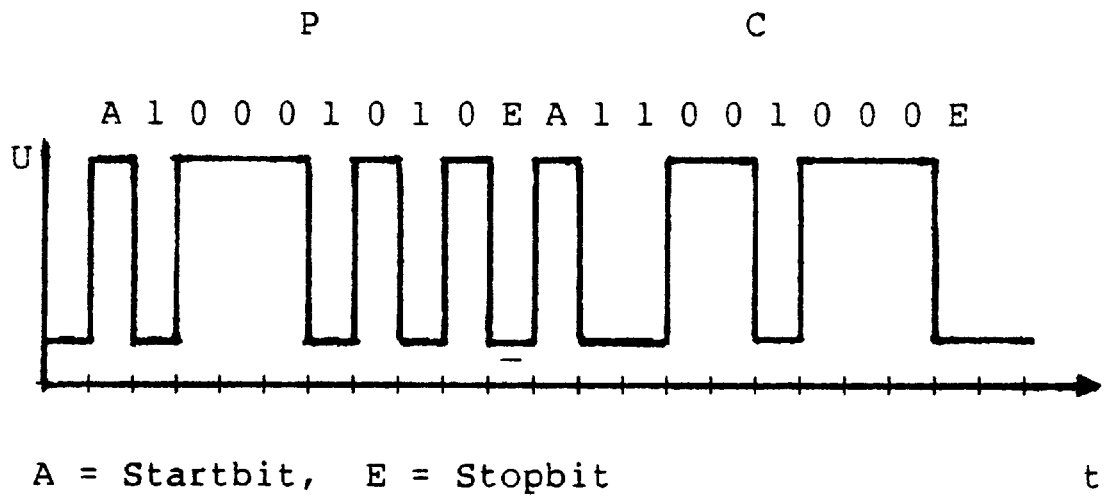
Funktionsweise

Der PC-1600 besitzt zur Kommunikation mit anderen Rechnern bzw. Peripheriegeräten zwei serielle Schnittstellen. Beide werden durch dieselben BASIC-Befehle angsprochen und durch denselben Baustein (UART) gesteuert, jedoch gibt es zwischen ihnen einige Unterschiede, die, je nach Einsatzbereich, die eine oder die andere Schnittstelle vorteilhafter machen.

Die serielle Datenübertragung unterscheidet sich von der parallelen dadurch, daß die einzelnen Bits eines Bytes nicht gleichzeitig sondern zeitlich nacheinander über ein Kabel geleitet werden. Der Zeitabstand zwischen zwei solcher Bits wird einmal festgelegt und muß sowohl dem Sender und dem Empfänger bekannt sein. Desweiteren geht jedem Byte ein sogenanntes Startbit (logisch 0) voraus, sein Ende wird durch ein oder zwei Stoppbits (logisch 1) angezeigt.

Ein 1-Bit entspricht einem negativen, ein 0-Bit einem positiven Spannungspegel. Die Anzahl der Bits je gesendetes Byte kann variieren. Für Textdaten sind 7 Bits ausreichend, bei Binärdaten oder der Verwendung von Sonderzeichen, werden 8 Bits benötigt. Die einzelnen Bits werden immer mit dem niederwertigsten beginnend Übertragen.

Beispiel: Übertragung der Buchstaben P und C mit 8 Bit und einem Stopbit.



Parität

Um die Übertragungssicherheit zu erhöhen, kann jedem gesendeten Byte ein weiteres Bit hinzugefügt werden, das die Anzahl der 1-Bits auf eine gerade (gerade Parität) oder eine ungerade (ungerade Parität) ergänzt.

Wird durch eine Störung ein einzelnes Bit auf dem Übertragungsweg verändert, so kann der Empfänger dies einfach dadurch feststellen, daß er die 1-Bits zählt. In diesem Fall muß er den Sender auffordern, dasselbe Byte noch einmal zu senden.

XON/XOFF- und RTS/CTS-Protokoll

Ein Problem bei der Datenüertragung besteht darin, daß der Empfänger die Daten u.U. nicht so schnell verarbeiten kann, wie er sie erhält. Es muß daher eine Möglchkeit vorhanden sein, dem Sender mitzuteilen, daß dieser die Übertragung für kurze Zeit

einstellen soll. Man nennt diesen Vorgang **Handshaking.**

Hierfürsind beim PC-1600 zwei Möglichkeiten vorgesehen:

- **RTS/CTS-Protokoll:** Eine spezielle Leitung führt vom RTS-Ausgang des Empfängers zum CTSEingang des Senders. Ist der Empfänger gerade beschäftigt, so kann er dies dem Sender über diese Leitung mitteilen.
- **XON/XOFF-Protokoll:** Die Empfangsbereitschaft wird dem Sender Übermittelt, indem eines von zwei reservierten Zeichen Über die Datenleitung geschickt wird. Erhält der Sender ein XOFF, so stellt er die Übertragung ein, erhält er ein XON, setzt er sie fort. Dieses Verfahren hat den Vorteil, daß bei einer bidirektionalen Datenübertragung nur drei Leitungen (zwei Daten- und eine Masseleitung) benötigt werden. Beim RTS/CTS-Protokoll sind es mindestens fünf (nämlich zusätzlich noch je eine Handshakeleitung in jede Richtung.

SHIFT IN / SHIFT OUT -Protokoll

Bei Verwendung des vollen Zeichensatzes (einschließlich der Graphikzeichen) zu Datenübertragung sind 8 Bits erforderlich. Kann aus irgendeinem Grund nur eine 7-Bit-Übertragung stattfinden, braucht der Zeichensatz deswegen nicht einzuschränkt zu werden. Hier hilft das SI/SO-Protokoll weiter.

SI und SO sind zwei ASCII-Zeichen mit

```
SI = CHR$ 15
```

SO = CHR$ 14

SI bedeutet: Die folgenden Zeichen sind gewöhnliche ASCII-7-Zeichen (Code kleiner als 128).

SO bedeutet: Die folgenden Zeichen haben einen ASCII-Code größer oder gleich 128.

Jeder Code hat somit zwei Bedeutungen, je nach dem, ob zuvor ein SI oder ein SO gesendet wurde.

Beispiel: Um die Formel

$V = r^2\pi * h$

zu übertragen, müssen die folgenden 7-Bit-Zeichen gesendet werden:

SI	"v"	"="	"r"	SO	"²"	"π"	"*"	SI	"h"
&0F	&56	&3D	&72	&0E	&7D	&63	&79	&0F	&68

INTERRUPTS

Ein Interrupt ist eine Unterbrechung eines laufen den Programms, um zwischendurch andere Aufgaben erledigen zu können. Er wird meist durch die Hardware (z.B. von Ein-/Ausgabebausteinen etc.) ausgelöst.

Vorgänge bei der Interruptverarbeitung auf Prozessorebene

Folgendes spielt sich der Reihe nach ab:

1. Irgendeine **Interruptquelle** möchte das laufende Programm unterbrechen, da gerade wichtigere Dinge zu tun sind.

2. Sie fordert Über einen **Interrupteingang** beim Prozessor einen Interrupt an (engl.: interrupt request, **IRQ**).

3. Der Prozessor kann diese Anforderung normalerweise nicht sofort entgegennehmen. Die Interruptquelle wird mindestens so lange warten müssen, bis er den geradd bearbeiteten Befehl zu Ende geführt hat.

4. Hat der Prozessor nun endlich Zeit, so 'merkt' er sich die Stelle im Programm, an der er gerade arbeitet, indem er die zugehörige Speicheradresse auf einen Stapel legt. Dies ist wichtig, wenn er das Programm später fortsetzen soll.

5. Nun wird das Programm in eine **Interrupt(bedien)routine** verzweigt. In ihr

stehen die Anweisungen, die die Interruptquelle gerne ausgeführt haben möchte. Bei mehreren in Frage kommenden interuptquellen muß in der Interruptroutine zuerst geprüft werden, welche von ihnen die Anforderung gestelllt hat.

6. Beim Abarbeiten der Interruptroutine kann es vorkommen, daß dieselbe oder eine andere Quelle erneut einen Interrupt anfordert. In diesem Fall, gibt es zwei-Möglichkeiten:

   . Die Anforderung wird gar nicht angenommen. Die anfordernde Quelle muß entweder warten, bis die laufende Interruptroutine zu Ende ist, oder ihre Anforderung zurücknehmen und es vielleicht später noch einmal versuchen.

   . Die zweite Möglichkeit besteht darin, daß der Prozessor die laufende Routine tatsächlich unterbricht und bei Punkt 4 weitermacht. Die Interruptroutinen sind in diesem Fall verschachtellt

7. Ist die Interruptroutine zu Ende geführt, so wird normalerweise an die Stelle im Programmspeicher zurückgesprungen, deren Adresse oben auf dem Stapel liegt, und damit das unterbrochene Programm fortgesetzt.

Bei verschachtelten Interrupts wird die zuletzt aufgerufene Routine als erste zu Ende geführt, ähnlich, wie dies auch bei verschachtelten Unterprogrammen der Fall ist.

Eine Interruptroutine kann mit dem unterbrochenen Programm Teile gemeinsam haben, z.B. ein häufig benötigtes Unterprogramm. Wird das Programmlauf nun gerade während der Bearbeitung dieses Unterprogramms unterbrochen, so kann es passieren, daß durch die Interruptroutine wichtige Zwischenergeb-

nisse zerstört werden, da sie ja ihrerseits dasselbe Unterprogramm aufruft. Dies kann dadurch verhindert werden, daß gemeinsam verwendete Unterprogramme gleich bei ihrem Aufruf alle benützten Variablen auf einen Stapel retten. Vor der RÜckkehr in den aufrufenden Programmteil werden dann die alten Werte wiederhergestellt. Man nennt solche Unterprogramme, die direkt oder indirekt durch sich selbst aufgerufen werden können, reentrant.

Interrupts in BASIC

Der PC-1600 bietet im Gegensatz zu den meisten anderen Pocket- und Personalcomputern die Möglichkeit, Interruptverarbeitung auch von BASIC aus zu betreiben. Es sind im wesentlichen alle Befehle vorhanden, die auch auf der Prozessorebene zu diesem Zweck zur VerfÜgung stehen.

Interruptquellen

Es sind 11 verschiedene Interruptquellen ansprechbar:

Kennzahl	Bezeichnung	Auslösung durch
0	PHONE	High-Pegel am CI-eingang der RS-232C Schnittstelle
1	COM1	ankommende Daten am RS-232C
2	COM2	ankommende Daten am SIO
7	ADIN	Bereichsüber-schreitung am Ana-

		logeingang
8	KEY(1)	Funktionstaste 1
9	KEY(2)	Funktionstaste 2
10	KEY(3)	Funktionstaste 3
11	KEY(4)	Funktionstaste 4
12	KEY(5)	Funktionstaste 5
13	KEY(6)	Funktionstaste 6
14	TIME$	Erreichen einer eingestellten Uhrzeit

Initialisierung von Interruptquellen

Die Adressen &F32D bis &F334 enthalten die Kennzahlen aller Interruptquellen, die durch die Anweisung **ON ... GOSUB** initialisiert worden sind. Ist deren Anzahl kleiner als 8, so werden die restlichen Bytes mit &FF (=255) belegt.

Das Programmstück

```
10 ON PHONE GOSUB 100
20 ON TIME$ ="06/12/05/30" GOSUB 400
30 ON KEY GOSUB 200, 300
```

belegt die o.g. Speicherstellen wie folgt:

Adresse	F32D	F32E	F32F	F330	F331	F332	F333	F334
Inhalt	0	14	8	9	255	255	255	255

Jede initialisierte Interruptquelle wird darüberhinaus durch ein 1-Bit innerhalb des Bytepaars &F1CF/D0 angezeigt:

	b7	b6	b5	b4	b3	b2	b1	b0
&F1CF :	7					2	1	0
&F1D0 :		14	13	12	11	10	9	8

Das obige Programmstück würde die beiden Bytes folgendermaßen belegen:

	binär	dezimal
&F1CF	00000001	1
&F1D0	01000011	67

ON/STOP

Eine Interruptanforderung wird nur dann angenommen, wenn sie mit **ON** freigegeben wird. Alle auf **ON** gesetzten Quellen sind in dem Bytepaar &F1D1/2 als 1-Bit repräsentiert. Das Darstellungsformat ist dasselbe wie in &F1CF/D0.

Das Gegenstück zu ON ist **STOP.** Mit **PHONE STOP,**
KEY(1) STOP usw. werden die entsprechenden Bits in &F1D1/2 wieder auf 0 zurückgesetzt.
Dieser Sachverhalt kann dazu genutzt werden, Interruptquellen per **POKE**-Befehl direkt auf **ON bzw. STOP** zu setzen.

Beispiel 1:

Um alle 6 Funktionstasteninterrupts freizugeben werden normalerweise 6 Anweisungen benötigt:

```
40 KEY(1) ON : KEY(2) ON : KEY(3) ON
41 KEY(4) ON : KEY(5) ON : KEY(6) ON
```

Einfacher geht's so:

```
40 POKE &F1D2, &3F
```

Beispiel 2:

Die Anweisung KEY(N) ON ist unzulässig, da das Argument eine Konstante sein muß. Möchte man trotzdem die Funktionstasten in Abhängigkeit von einem zuvor errechneten Wert N freigeben, so schreibe man dafür

```
POKE &F1D2, (2^(N-1)) OR PEEK &F1D2
```

Die OR-Verknüpfung mit dem alten Wert bewirkt, daß schon bestehende Einstellungen nicht gelöscht werden.

Speicherung von Interruptanforderungen

Ankommende Interruptanforderungen können im allgemeinen nicht sofort angenommen werden. Wie der Prozessor muß auch der BASIC-Interpreter zuerst eine gerade bearbeitete Anweisung zu Ende führen. Dies kann in manchen Fällen einige Zeit dauern, wie das folgende Beispiel zeigt:

```
10 PRINT ATN(1); ATN(2); ATN(3); ATN(4);.....
```

Dies ist eine einzige Anweisung!

Deshalb werden alle Interruptanforderungen zwischengespeichert, und zwar in den Speicherstellen &F1D3/4. Wie in den beiden vorangehenden Abschnitten bereits erläutert, steht wieder jeder Quelle ein bestimmtes Bit zu. Nach jeder vollendeten An-

weisung wird durch Abfragen dieser Bytes geprüft, ob ein zwischenzeitlich ein Interrupt vorlag; ist dies der Fall, so wird in die entsprechende Interruptroutine verzweigt.

Auch die Interrupts von Quellen, die auf **STOP** gesetzt sind, werden gespeichert und werden sofort nach der Ausführung einer entsprechenden **ON**-Anweisung verarbeitet. Möchte man dies vermeiden, muß ein eventuell gespeicherter Interrupt kurz vor der Freigabe mit **ON** gelöscht werden.

Beispiel:

```
10 ON KEY GOSUB "INT"
20 KEY(1) ON
30 PRINT "INT freigegeben!"
40 FOR I=1 TO 2000 : NEXT I
50 KEY(1) STOP
60 PRINT "INT gesperrt!"
70 FOR I=1 TO 2000 : NEXT I
80 POKE &F1D4,0
90 GOTO 20
100 "INT"
110 BEEP 3 : RETI
```

Zeile 80 löscht einen bis dahin aufgetretenen Interrupt. Somit bleibt eine Betätigung der Funktionstaste 1 während der zweiten Warteschleife im Programm ohne Wirkung.

Interruptanforderungen können natürlich nicht nur gelöscht, sondern auch 'künstlich' erzeugt werden. Wird im vorangegangenen Beispiel die Zeile

```
45 POKE &F1D4,1
```

eingefügt, so wird jedesmal nach AusfÜhrung dieser Zeile in die Interruptroutine gesprungen. Folgende Anwendungen sind denkbar:

- Hilfe beim Austesten von Programmen und bei der Fehlersuche
- Behandlung einer Interruptroutine als gewöhnliches Unterprogramm

Entfernen von Interrupteinträgen

Eine mit **ON ... GOSUB** initialisierte Interruptquelle kann mit der **OFF**-Anweisung wieder entfernt werden. Dabei wird einfach das entsprechende Bit in &F3CF/D0 zurückgesetzt. Zu beachten ist je' doch, daß der Eintrag der Kennzahl in den Adressen &F32D bis &F334 bestehen bleibt. Es ist daher ratsam, diesen mit einem gezielten **POKE** ebenfalls zu löschen.

Dasfolgende Beispiel zeigt den Anfang eines Programms, daßauf die Interrupts von **TIME$, KEY(2) bis KEY(6), PHONE und ADIN** reagieren. Obwohl sich zusammen gerade die maximale Anzahl von 8 Quellen ergibt, treten Probleme auf, weil die Funktionstasten mit **ON KEY GOSUB** immer bei KEY(1) beginnend initialisiert werden, so daß man ungewollt 9 Interruptquellen erhält. **KEY(1) OFF** alleine hilft aus dem oben genannten Grund recht wenig.

```
10 ON TIME$ = "??/??/20/15" GOSUB 100
20 ON KEY GOSUB 200,200,300,400,500,600
30 KEY(1) OFF : POKE &F32E, 255
40 ON PHONE GOSUB 700
50 ON ADIN (45, 55) GOSUB 800
```

In Zeile 30 wird der nicht benötigte KEY(1)-Interrupt entfernt. Da er an zeiter Stelle (nach TIME$) initialisiert wurde, liegt der Eintrag seiner Kennzahl in der Adresse &F32E. Nachdem dorthinein 255 geschrieben wird, kann der dadurch frei-

werdende Platz in der darauffolgenden Zeile durch **PHONE** belegt werden.

Ablauf eines Interrupts in BASIC

Beim Sprung in eine Interruptroutine geschieht nun folgendes:

- Die Position, an der das laufende Programm unterbrochen wurde, wird genau wie bei einer **GOSUB** auf den RETURN-Stapel gelegt (s. Kapitel "Arbeitsweise des BASIC-Interpreters")

- Die Interruptfreigabemaske in &F1D1/2 wird auf einen zweiten Stapel gelegt. Dieser läuft von der Adresse &F32C an abwärts und umfaßt maximal 16 Bytes. Da er bei jedem Eintritt in eine Interruptroutine um 2 Bytes wächst, können die Interrupts maximal 8-fach verschachtelt werden. Dies stellt in den meisten Fällen keine Einschränkung dar, da sowieso nur 8 Quellen initialisiert werden können.

Der Kopf dieses Stapels wird durch den Inhalt von &F335 angezeigt. Die Adresse, an der die zuletzt abgelegte Freigabemaske liegt, errechnet sich zu

```
&F32C - PEEK &F335
```

Bei leerem Stapel, d.h. wenn keine Unterbrechungen erfolgt sind, enthält &F335 den Wert &FF.

- Diejenige Quelle, die den Interrupt ausgelöst hat, wird auf **STOP** gesetzt, um eine Verschachtelung ein und derselben Interruptroutine zu vermeiden.

. Der Tastaturpuffer wird gelöscht.

. Erst, wenn all dies geschehen ist, wird mit der Abarbeitung der Interruptroutine begonnen.

Beim Rücksprung mit **RETI** wird gerade umgekehrt verfahren:

. Die Freigabemaske erhält wieder ihren ursprünglichen Wert, der zu diesem Zweck auf den Stapel gelegt wurde. Die auslösende Interruptquelle-ist nun wieder **ON**.

. Die RÜcksprungadresse wird vom RETURN-Stapel genommen und das Programm an der Stelle fortgesetzt, an der es unterbrochen wurde.

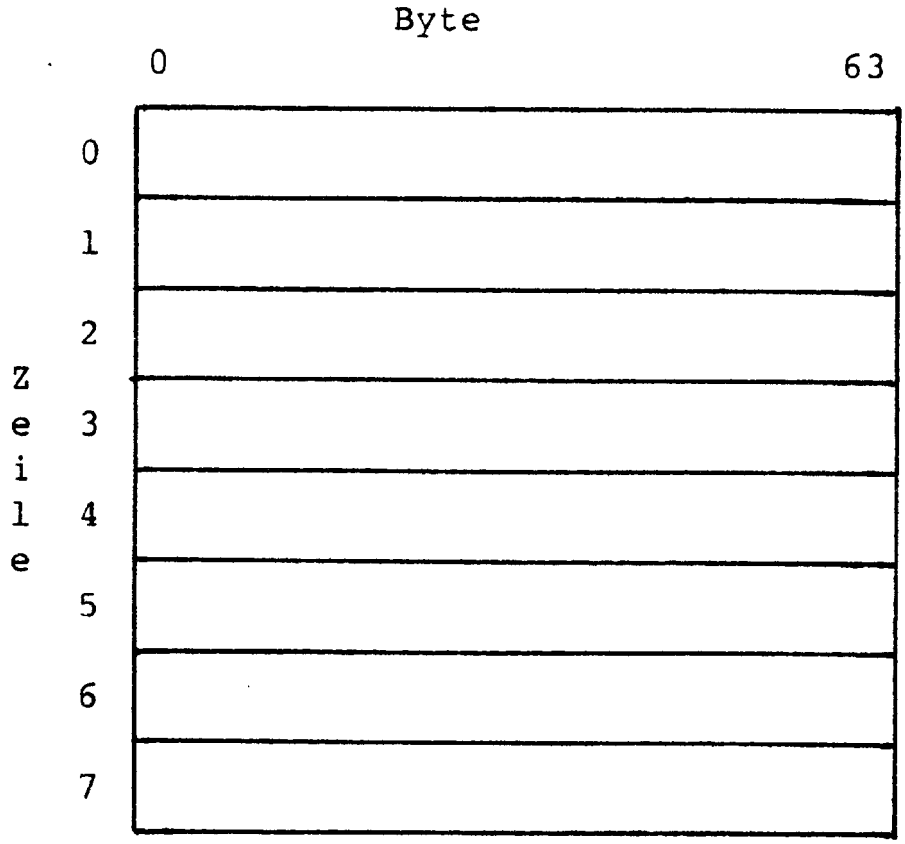
Das Lesen und Verändern von Stapelzeigern und inhalten macht einige interessante Anwendungen möglich. Der Anhang enthält ein Programm für ein einfaches Multitasking.

DAS DISPLAY

Der PC-1600 besitzt ein LC-Display mit 156 x 32 adressierbaren Bildpunkten und 18 Sondersymbolen. Es wird angesteuert durch zwei Displayspeicher/treiber und einen Multiplexerbaustein. Dies stellt einen wichtigen Unterschied zu anderen Pocketcomputermodellen dar, deren Displayinhalt einen mehr oder weniger großen Teil des RAMs belegt.

Zugriff auf den Displayspeicher

Jeder der beiden Displayspeicher umfaßt 512 Bytes. Man kann sie sich vorstellen als Matrix von 8 Zeilen zu je 64 Bytes:



Zusätzlich verfügt jeder Speicher noch Über ein Kontroll- und ein Statusregister.

Man kann jedes einzelne Byte lesen oder beschreiben. Vorher muß aber angegeben werden, um welches der 512 Bytes es sich handeln soll, es muß adressiert werden.

Adressierung

Die Adressierung eines Bytes geschieht folgendermaßen (fÜr Speicher 1):

1. Zur Nummer der Zeile, in der das Byte steht (0 bis 7), wird &B8 addiert und das Ergebnis in das Kontrollregister (E/A-Adresse &54) geschrieben:

```
OUT &54, &B8+ZEILE
```

2. Zur Position des Bytes in dieser Zeile (0 bis 63), wird &40 addiert und das Ergebnis wieder in das Kontrollregister geschrieben:

```
OUT &54, &40+POS
```

Nun kann der Inhalt des adressierten Bytes gelesen oder verändert werden.

Schreiben

Über die Adresse &56 kann der Speicher beschrieben werden:

```
OUT &56, WERT
```

Sollen mehrere aufeinanderfolgende Bytes in einer Zeile beschrieben werden, so braucht nur das erste adressiert zu werden. Das Beispiel zeigt, wie in Zeile 3 allen Bytes von Position 10 bis 20 der Wert &88 zugewiesen wird:

```
10 'Zeilenadressierung:
20 OUT &54, &B8+3
30 'Spaltenadressierung:
40 OUT &54, &40+10
50 'Schreiben der 11 Bytes:
60 FOR I=1 TO 11
70 OUT &56, &88
80 NEXT I
```

Lesen

Wie beim Schreiben wird auch hier zuerst das gewünschte Byte adressiert, sodann kann es Über die Adresse &57 gelesen werden.

Zu beachten ist, daß nicht direkt aus dem Speicher gelesen werden kann. Das gewünschte Byte wird in einem sogenannten **Latch** bereitgestellt. Vom Prozessor aus kann nur auf diesen Latch zugegriffen werden. Sofort nach einem Lesezugriff wird der Latch mit dem nächsten Byte derselben Zeile aufgefüllt. Bei einer Neuadressierung ist also immer zu berücksichtigen, daß der Latch noch einen alten Wert enthält. Dieser muß erst ausgelesen werden, um an das gewünschte Byte heranzukommen.

Folgendes Beispiel zeigt, wie zu verfahren ist:

```
20 OUT &54,&B9 : OUT &54,&47
```

```
30 'Auslesen des alten Wertes:
40 A = INP &57
50 'Nun erst steht der richtige im Latch:
60 INHALT = INP &57
```

Auch das Auslesen mehrerer aufeinanderfolgender Bytes einer Zeile ist möglich. Dazu wird z.B. die Zeile 60 des vorigen Programmstücks abgeändert:

```
60 FOR I=0 TO 15
70 A(I) = INP &57
80 NEXT I
```

Zugriff auf Speicher 2

Zum Lesen und Beschreiben des zweiten Speicherbausteins wird in derselben Art und Weise verfahren. Lediglich die E/A-Adressen sind anders. Folgende Tabelle stellt alle benötigten Adressen zusammen:

Speicherbaustein		1	2	1 und 2
Daten	schreiben	&56	&5A	&52
	lesen	&57	&5B	-
Kontroll-register	schreiben	&54	&58	&50
Status-register	lesen	&55	&59	-

Über die Adressen &50 und &52 kann in beide Bausteine zugleich dasselbe Byte geschrieben werden. Theoretisch ist auch ein gleichzeitiges Lesen über

die Adressen &51 und &53 zwar denkbar, jedoch wenigsinnvoll und sollte nicht durchgeführt werden.

Wichtige Anmerkungen

Wer die oben angegebenen Beispiele eingegeben und gestartet hat, wird festgestellt haben, daß sich auf der Anzeige, wenn Überhaupt, seltsame und unvorhersehbare Dinge ereignen. Dies hat folgende Gründe:

. Alle 0,5 sec greift der Prozessor durch einen Interrupt auf die Displayspeicher zu und kann dadurch die mit **OUT &54** und **OUT &58** gemachten Adresseinstellungen Überschreiben, so daß die Daten an anderer Stelle als vorgesehen im Speicher abgelegt werden.

Abhilfe schafft hier die Anweisung

```
OUT &35, &1F
(Sperren des 0,5s-Interrupts)
```

Nach Beendigung eines Displayzugriffs sollte mit

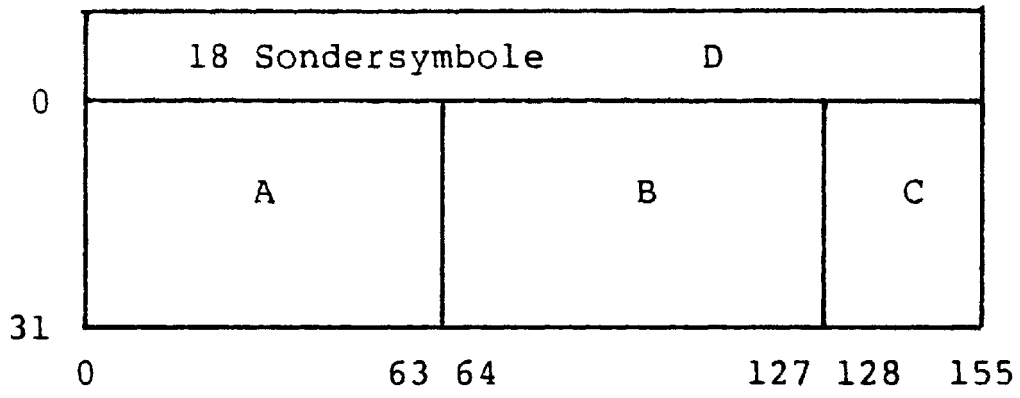
```
OUT &35, &5F
```

wieder der standardmäßige Wert eingestellt werden.

. Daß die geschriebenen Bytes einmal hier und einmal dort auf der Anzeige erscheinen, hat seinen Grund in der Art und Weise, wie der Speicherinhalt ins Display umgesetzt wird. Dies wird im folgenden Unterkapitel näher untersucht.

Darstellungsweise des Displayspeicherinhalts auf der Anzeige

Von der Anzeige aus gesehen sind die Displayspeicher als 64 Zeilen (0 bis 63) zu je 64 Bits organisiert. Das Display selber kann man sich folgendermaßen aufgebaut denken:



Nach AusfÜhrung der Anweisung

OUT &50, &C0 (Beschreibung weiter unten)

sieht die Abbildung des Speicherinhalts auf die Anzeige wie folgt aus:

Speicher Nr.	Speicher-zeile	Display-bereich	Display-zeile
1	0	A	0
|	|	|	|
1	31	A	31
1	32	C	0
|	|	|	|
1	63	C	31
2	0	B	0
|	|	|	|
2	31	B	31
2	32	D	
|	|	|	(Symbole)
2	63	D	

Nun kann bereits begonnen werden, das Display gezielt anzusprechen. Das folgende Programm fragt nach einem Bereich, einer Zeile und einer Spalte in diesem Bereich. Sodann wird an der dadurch gegebenen Position ein Punkt gezeichnet.

```
 10 INPUT "Bereich (A/B/C) ";A$
 20 INPUT "Zeile   (0-31)  ";Z
 30 INPUT "Spalte  (0-63)  ";S
 40 'E/A-Adressen für Kontrollregister
 41 'und Schreibdaten
 50 IF A$="B" LET K=&54 : D=&56
        ELSE LET K=&58 : D=&5A
 55 IF A$="C" LET Z=Z+32
 60 'Interrupt sperren
 70 OUT &35, &1F
 80 CLS : OUT &50, &CO
 90 OUT K, &B8+INT(Z/8)
100 OUT K, &40+S
110 OUT D, 2^(Z MOD 8)
120 'Interrupt wieder freigeben
130 OUT &35, &5F
140 IF INKEY$(1)="" GOTO 140
```

Sollte nach Beendigung des Programms die Anzeige etwas durcheinander geraten sein, kann dies durch mehrmaliges Drücken der ENTER-Taste wieder behoben werden.

Verschieben des Displayinhalts

Im vorangegangenen Unterkapitel wurde durch den Befehl **OUT &50,&C0** erreicht, daß Zeile 0 des Speichers als oberste Displayzeile angezeigt wird. Durch Setzen eines Zeigers, der der Einfachkeit halber mit Z0 bezeichnet wird, kann jede der 64 Speicherzeilen zur obersten Displayzeile gemacht werden:

```
OUT K, &C0+Z0     (K = Adresse des
                  Kontrollregisters)
```

Z0 liegt dabei im Bereich von 0 bis 63. Die Erhöhung von Z0 um 1 bewirkt, daß die Anzeige in den dem angesprochenen Speicherbaustein zugeordneten Bereichen zyklisch um eine Zeile nach oben rotiert. Dieser Sachverhalt wird im PC-1600 fÜr das Scrollen der Anzeige genutzt. Da die Zeichenhöhe 8 Zeilen beträgt, wird einfach entweder 8 oder -8 zu Z0 addiert.

Möchte man Daten gezielt in eine bestimmte Displayzeile schreiben, muß man natürlich den gerade eingestellten Wert von Z0 kennen, um daraus die entsprechende Zeile im Speicher errechnen zu können. Aus dem Speicherbaustein direkt, kann Z0 nicht gelesen werden, jedoch ist der Wert Z0/8 in der RAM-Adresse &F05C zu finden. Da die Anzeige normalerweise immer um ein ganzes Zeichen gescrollt wird, ist Z0 stets ein Vielfaches von 8, nämlich

```
Z0 = 8 * PEEK &F05C
```

Möchte man den Z0-Zeiger durch eigene Programme modifizieren, muß man dies mit dem Inhalt von &F05C ebfalls tun, um zu vermeiden, daß bei der nächsten **PRINT**-Anweisung oder nach Beendigung eines Programms die Anzeige in Unordnung gerät.

Im Anhang stehen zwei Unterprogramme, die einen Softscroll (auf- und abwärts) um eine Zeichenhöhe ausführen. Sie zeigen eine der Stärken von speziellen Displaytreibern auf. Wäre der Displayinhalt, wie bei den meisten anderen Pocketcomputern, im RAM abgelegt, so müßte bei einem Softscroll um 8 Zeilen jedes der 156*32/8 = 624 Bytes achtmal um ein Bit verschoben werden, was in BASIC programmiert sicher einige Minuten dauern würde.

Desweiteren gibt es dort ein Programm, das zeigt, wie ohne **PSET**-Befehl ein Punkt mit den Koordinaten X und Y auf die Anzeige zu zeichnen ist. Es hat zwar wenig praktischen Nutzen, jedoch werden die Verfahren der Byteadressierung, des Lesens und des Schreibens noch einmal gezeigt.

Zu guter letzt sind noch zwei Programme vorhanden, in denen von Assemblerroutinen auf das Display zugegriffen wird.

Im einen, dem schon erwähnten Lokomotivprogramm, wird ein Teil des Anzeigeninhalts pixelweise in horizontaler Richtung verschoben.

Das andere ist ein Programm zur bewegten Darstellung dreidimensionaler Funktionsgraphen. Dort wird ein Assemblerunterprogramm dazu verwendet, den Anzeigeninhalt schnell in einen RAM-Bereich zu schreiben und ihn von dort wieder auf das Display zurück zu holen. Diese Routine kann auch in anderen Programmen benutzt werden. Sie ist frei im

Speicher verschiebbar und wird wie folgt aufgerufen:

```
IF A>32767 LET A=A-65536
CALL <Anfangsadr>, A
```

Dabei zeigt A auf den Anfang des zu Übertragenden Speicherbereichs, der 624 Bytes umfaßt. Die Übertragungsrichtung

```
Display         -->   Speicher
oder Speicher   -->   Display
```

wird durch Veränderung von 3 Bytes im Programm festgelegt.

Ein-und Ausschalten der Anzeige

Die Displaytreiber bieten noch eine weitere Funktion: Mit

```
OUT K, &3E
```

kanndie Anzeige ausgeschaltet werden. K ist hierbei wieder die Adresse des Kontrollregisters (&54 oder&58). Bit 5 des Statusregisters (&55 bzw. &59) wird auf 1 gesetzt. Wenngleich keine Anzeige des Speicherinhalts erfolgt, kann dieser nach wie vor verändert und gelesen werden. Durch

```
OUT K, &3F
```

wird die Anzeige wieder eingeschaltet und Statusbit 5 wieder gelöscht.

Die Ein-/Ausschaltfunktion kann z.B. dazu benutzt werden, bei leerem Display irgendwelche Figuren zu

zeichnen,die dann beim Einschalten auf einen Schlag angezeigt werden.

Ein wiederholtes Aus- und Einschalten bewirkt ein Blinken des gesamten Displays. Damit kann der Benutzer z.B. auf irgendwelche Gefahren aufmerksam gemacht werden.

```
100 IF INP &78 AND 8 GOTO 210
110 CLS
120 PRINT "**************************"
130 PRINT "***** Bitte Diskette *****"
140 PRINT "*****  einlegen !!!  *****"
150 PRINT "**************************"
160 BEEP 1,50,100
170 OUT &50, &3E
180 FOR I=1 TO 100 : NEXT I
190 OUT &50, &3F
200 IF (INP &78 AND 8)=0 GOTO 160
210 'und weiter geht's
```

Hinweise für Assemblerprogrammierer

Da die Displaytreiber nicht nur vom Prozessor angsprochen werden, sondern darüberhinaus auch noch die Anzeige selbst versorgen müssen, kann auf sie nicht mit beliebiger Geschwindigkeit zugegriffen werden. Die maximale Zugriffszeit ist wesentlich kürzer als die Dauer eines **INP**- oder **OUT**-Befehls in BASIC. In Assemblerprogrammen hingegen kann es passieren, daß beim schnellen Schreiben oder Lesen Daten verschluckt werden.

Deshalb ist im Statusregister ein Flag (Bit 7) vorgesehen, das immer dann auf 1 gesetzt wird, wenn der Treiber gerade anderweitig beschäftigt ist. In diesem Fall muß das Programm so lange war-

ten, bis das Flag wieder 0 ist. Dies kann z.B. durch folgende drei Assemblerzeilen geschehen:

```
WARTEN    IN  A,55H       ;Status von Treiber 1
                          ;lesen
          AND A           ;Prozessorflags
                          ;entsprechend setzen
          JP  M,WARTEN    ;wiederholen, solange
                          ;Bit 7 gesetzt ist
```

Entsprechend verläuft die Abfrage für Treiber 2 (59H statt 55H).

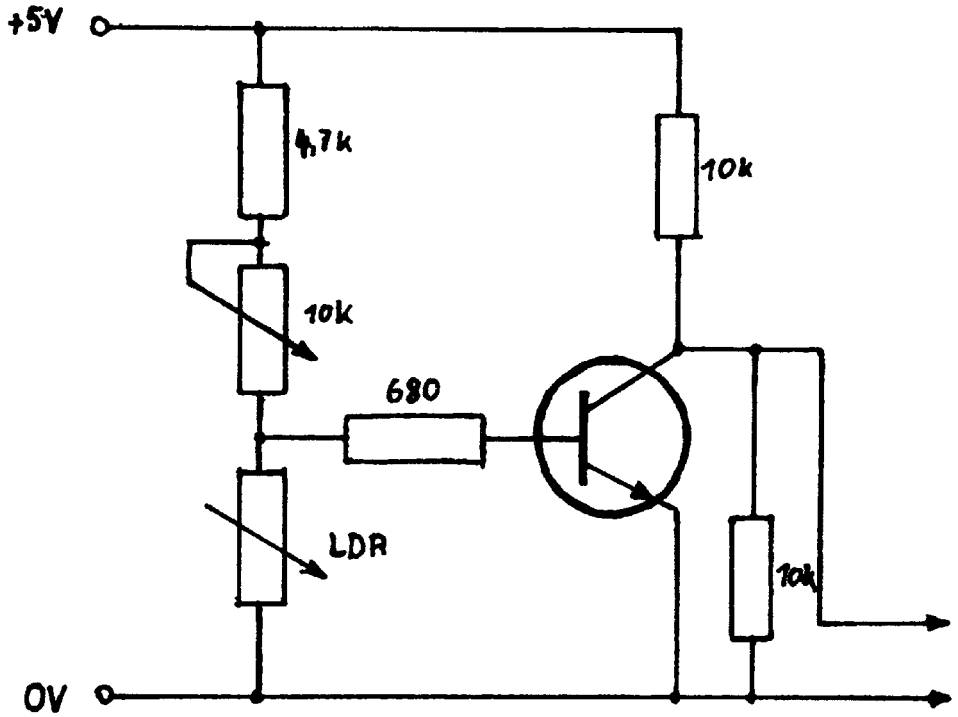
Im ROM befindet sich ein Unterprogramm, das denselben Zweck erfüllt und sogar gleich beide Treiber abfragt:

CALL 8EAH (ROM V1.3)

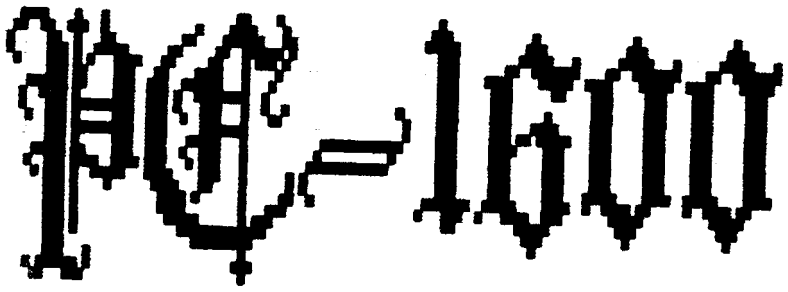
DER ANALOGEINGANG

Der Nebenprozessor **LU57813** enthält drei Analog-Digital-Umsetzer. Zwei davon werden zur Überwachung der Versorgungsspannungendes Rechners und des Plotters eingesetzt, der dritte ist frei für den Benutzer verfügbar. Zum Anschluß wird ein einfacher Stereoklinkenstecker verwendet.

Im Prinzip kann jeder Sensor (für Licht, Wärme, Feuchtigkeit usw.) angeschlossen werden, es muß nur dafür gesorgt werden, daß sich die am Analogeingang anliegende Spannung im Bereich von 0 bis 2,5 V bewegt.
Die folgende einfache Schaltung, zeigt, wie ein lichtempfindlicher Widerstand (LDR) angepaßt werden könnte:



Als Transistor kann jeder Universaltyp, z.B. BC238, verwendet werden. Mit dem Potentiometer kann die Empfindlichkeit eingestellt werden. Bevor man den PC-1600 anschließt, sollte man sich vergewissern, daß das Maximum der Ausgangsspannung tatsächlich bei 2,5 V liegt. Dazu wird der LDR mit einem kurzen Draht kurzgeschlossen und mit dem Voltmeter der Ausgang gemessen. Leicht abgewandelt kann die Schaltung auch fÜr andere veränderliche Widerstände wie z.B. Heiß- und Kaltleiter verwendet werden.



DER CASSETTENRECORDERANSCHLUß

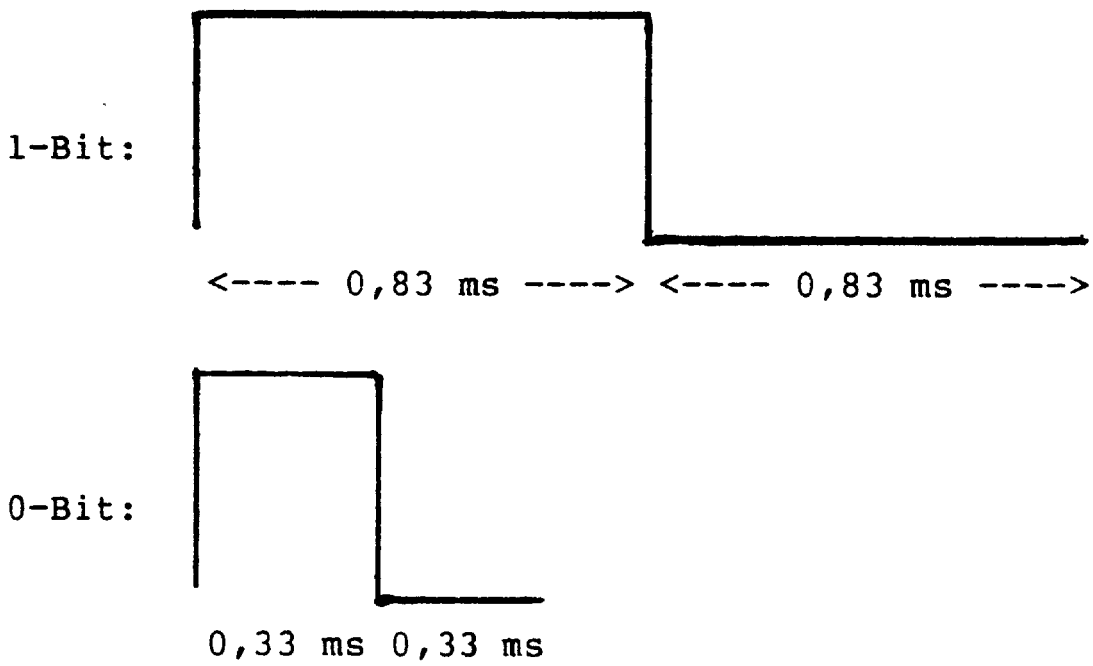
Aufzeichnungsverfahren

Jede Übertragung beginnt mit einem Dauerton der Frequenz 2,54 kHz. Er kann mit Hilfe von

OUT &17,&41 und OUT &17,0

ein- bzw. ausgeschaltet werden.

Danach folgen die Datenbytes. Jedem Byte geht zu Sychronisationszwecken ein Startbit (logisch 1) voraus. Die restlichen 8 Bits werden, beginnend mit dem höchtwertigen, auf folgende Art und Weise am Cassettenausgang ausgegeben:



Lesen von Cassette

Zuerst muß durch Setzen von Bit 7 in der E/A Adresse &82 der Cassetteneingang freigegeben werden:

```
OUT &82, INP &82 OR &80
```

Sodann kann der momentane Pegel des Cassettensignals am Bit 7 der Adresse &81 abgelesen werden

```
Z = INP &81 AND &80
```

Leider ist die Sprache BASIC zu langsam, um mit diesen beiden Befehlen eine eigene CLOAD-Routine zu schreiben. Jedoch stehen dem Assemblerprogrammierer viele Möglichkeiten offen. Denkbar wäre z.B. eine Routine, die Programme und Daten liest, die von anderen Rechnertypen auf Cassette geschrieben worden sind.

Natürlich kann der Cassetteneingang auch zum Empfang anderer Signalquellen dienen, wenn deren Spannungspegel dem eines Kopfhörerausgangs ähnlich ist. Zu beachten ist jedoch, daß im CE-1600P der Gleichstromnteil des Signals durch einen Kondensator unterdrückt wird.

Schreiben auf Cassette

Der schon erwähnte Dauerton wird mit

```
OUT &17,&41
```

aktiviert. Mit

```
OUT &18, INP &18 OR &80 bzw.
```

```
OUT &18, INP &18 AND &7F
```

kann der Cassettenausgang gezielt auf high oder low gesetzt werden. Die Verknüpfung mit **AND** bzw. **OR**, bewirkt, daß die restlichen Bits (Bit 6: BEEP ON/OFF) unverändert bleiben.

Auch beim Ausgang ist ein Kondensator in Reihe geschaltet, so daß nur kurze Impulse gesendet werden können.

Der Remote-Ausgang

Der Remote-Schaltung besteht im wesentlichen aus einem polarisierten Relais, zu dessen Kontakt der Remote-Schalter parallelgeschaltet ist. Durch **RMT OFF** wird der Relaiskontakt geschlossen, durch **RMT ON** wieder geöffnet. Die beiden Wicklungen können auch Über die Bits 5 und 6 der E/A-Adresse &82 direkt angesprochen werden. Zu beachten ist, daß ein kurzer Impuls auf die eine oder andere Wicklung genügt.

```
Schließen:   OUT &82,&10 : OUT &82,0
Öffnen:      OUT &82,&20 : OUT &82,0
```

Die Impulsbreite, die durch die Verarbeitungsdauer des OUT-Befehls bestimmt wird, ist genügend groß, um das Relais umspringen zulassen. Die Bits 0 bis 3 der Adresse &82 sollten auf jeden Fall auf 0 gehalten werden, da diese einen der drei Schrittmotoren des Plotters ansteuern.

DIE TASTATUR

Normalerweise wird die Tastatur durch ein ROM-Programm abgefragt und Über die Funktion **INKEY$** das Zeichen als fertiger String geliefert. In diesem Kapitel sollen die Vorgänge, die sich bei der Tastaturabfrage intern abspielen, etwas näher beleuchtet werden.

Abfrage durch Interruptroutine

Durch den 1/64s-Interrupt wird die Tastatur ständig abgefragt und alle Eingaben im Tastaturpuffer festgehalten. Desweiteren wird der Code der momentan gedrückten Taste in die RAM-Adresse &F081 geschrieben. Dadurch ist Abfrage in Assemblerprogrammen genauso einfach wie in BASIC mit **INKEY$.**

Direkte Abfrage der Tastaturmatrix

Die Tastatur ist kann gedacht werden als System von 8 horizontalen und 9 vertikalen Leitungen, an deren Kreuzungspunkten die einzelnen Tasten angebracht sind.

	PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0	PB6
KIN0	3	▶	=	+	)	1	.	2	CTRL
KIN1	6	MODE	◀	*	L	4	-	5	KEYII
KIN2	9	CL	P	/	O	7	OFF	8	BS
KIN3	G	A	F	D	K	J	S	H	
KIN4	S	DEF	#	"	&	%	!	SHIFT	
KIN5	T	Q	R	E	I	U	W	Y	
KIN6	B	Z	V	C	(	M	X	N	
KIN7	▼	SML	SPACE	RCL	ENTER	0	▲▼	▲	

Um nun zu prüfen, ob eine bestimmte Taste gedrückt ist, schickt man Über einen der PA/PB-Ausgänge ein Signal in die betreffende Spalte der Matrix (active low, es muß also eine 0 ausgegeben werden). Ist der Kontakt geschlossen, so wird dises Signal in der entsprechenden Zeile weitergeleitet und kann Über die KIN-Eingänge gelesen werden. Dieser Vorgang kann nacheinander für alle Spalten wiederholt werden, bis alle Tasten abgefragt sind.

Zum Ansteuern und Abfragen der Matrix sind die E/A-Adressen &1C, &1D, &1E (PA), &1F (PB) und &37 (KIN) vorgesehen. In den Adresse &1C und &1D wird die Übertragungsrichtung der bidirektionalen Ports PA und PB festgelegt. Ein 1-Bit bewirkt, daß der betreffende Anschluß als Ausgang geschaltet wird. Es sollte immer nur ein einziger Ausgang aktiviert akiviert werden, um Konflikte zwischen den Spaltenleitungen bei meheren gleichzeitig gedrückten Tasten zu vermeiden.

Jeder Zugriff auf die Tastatur wird mißlingen, solange der 1/64s-Interrupt freigegeben ist, da in

der Interruptroutine ja ebenfalls die Ports verändert werden. So sollte vor jeder Abfrage die Anweisung

```
10 OUT &35,&4F
```

stehen. Sodann kann eine Spaltenleitung (im Beispiel die vierte Spalte von links) als Ausgang geschaltet und ein Low-Pegel augegeben werden:

```
20 OUT &1C,&10 : OUT &1E,&EF
```

Mit

```
30 Z = INP &37
```

werden die Zeilenleitungen gelesen. Wurde die Taste 'E' gedrückt, so enthält Z nun den Wert &DF (Bit5=0). Sollen keine weiteren Abfragen mehr gemacht werden, wird mit

```
40 OUT &35,&5F
```

der 1/64s-Interrupt wieder freigegeben.

Bemerkenswert ist, daß auf dieses Verfahren auch dann funktioniert, wenn beliebig viele Tasten gleichzeitig betätigt werden. Die **INKEY$**-Funktion versagt in solchen Fällen, da sie immer nur ein einziges Zeichen liefern kann.

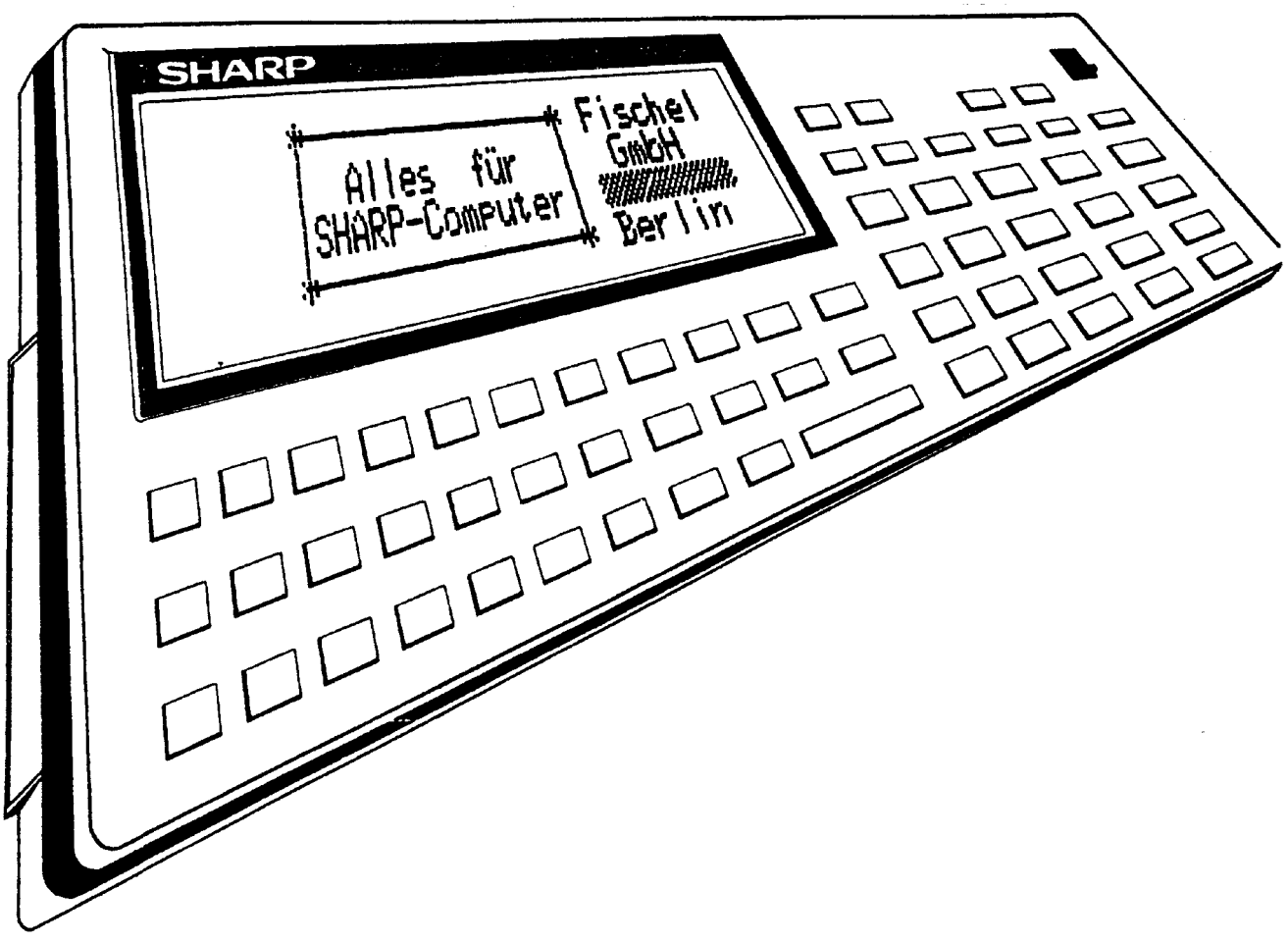
Die ON-Taste

Dem aufmerksamen Leser ist vielleicht aufgefallen, daß in der Tastaturmatrix eine Taste fehlt, nämlich die ON-Taste. Ihr kommt eine besondere Funktion zu: Sie muß jedesmal beim Einschalten des Rechners einen Interrupt an den Nebenprozessor

liefern, der dann seinerseits die beiden anderen CPUs und die restliche Hardware in Gang setzt.

Dies ist aber kein Grund die Taste nicht auch softwaremäßig abfragen zu können. Es gibt dafür sogar zwei Möglichkeiten: Sowohl Bit 5 der E/A Adresse &1A als auch Bit 7 in &1F (PB7) werden bei gedrückter ON-Taste gesetzt.

```
INP &1A AND &20    oder    INP &1F AND &80
```



ANHANG A: EIN-/AUSGABEADRESSEN

Über diese Adressen kann mit Peripheriebausteinen kommuniziert werden. Dies geschieht mit folgenden Befehlen:

	BASIC	Assembler
Daten lesen	Z=INP <adr>	IN A,<adr>
Daten schreiben	OUT <adr>,Z	OUT <adr>,A

<adr> ist eine E/A-Adresse im Bereich von 0 bis 255.

Parallelport

```
&17         2639 Hz Dauerton, Synchronisationssig-
            nal für Cassettenaufnahmen.
            Einschalten: OUT &17,65
            Ausschalten: OUT &17, 0

&18         Port PC
    Bit 6   0 : BEEP OFF
            1 : BEEP ON
    Bit 7   Cassettenausgang und Lautsprecher
            0 : low
            1 : high

&1A Bit 5   0 : ON-Taste nicht gedrückt
            1 : ON-Taste gedrückt

&1C         Übertragungsrichtung fÜr Port PA jedes Bit steht
            für einen Ein-/
```

Ausgang.
0 : Eingang
1 : Ausgang

&1D Übertragungsrichtung für Port PB
jedes Bit steht für einen Ein-/
Ausgang.
0 : Eingang
1 : Ausgang

&1E Port A
Spaltensignal für die Tastaturabfrage
(vorher Übertragungsrichtung mit
OUT &1C,xx einstellen)
0 : low
1 : high

&1F Port PB
Lesen:
Bit 5 1/64 sec Takt
Bit 7 ON-Taste (gleicher Wert wie &1A Bit 5)
Schreiben:
Bit 6 Spaltensignal fÜr die Tastaturabfrage
(vorher Übertragungsrichtung mit
OUT &1D,&40 einstellen)

Serielle Schnittstellen

&20 Schreiben:
Ein Byte über serielle Schnittstelle
(RS-232C oder SIO) senden.
Lesen:
Ein Byte empfangen

&22 schreiben:
Kontrollregister für serielle Über-
tragung
lesen:

Speicherverwaltung, Interrupts

&31 Auswahl der Speicherbank:

	Adressbereich	Adressierbare Bänke
Bit 0	&0000 bis &3FFF	0 und 1
Bit 1 Bit 2 Bit 3	&4000 bis &7FFF	0 bis 7
Bit 4 Bit 5 Bit 6	&8000 bis &BFFF	0 bis 7
Bit 7	&C000 bis &FFFF	0 und 1

&32 Interruptzustand:
Hier kann festgestellt werden, welche quelle gerade einen Interrupt anfordern.

Bit 0 Empfang von seriellen Daten
Bit 1 Peripherie (Drucker und Floppy Disk)
Bit 4 1/64 sec Timer
Bit 6 1/2 sec Timer

&35 Interruptmaske:
Durch setzen eines Bits auf 0 wird die entsprechende Interruptquelle gesperrt. Die Belegung der einzelnen Bits ist dieselbe wie in &32.

Display

&50	Kontrollregister beider Display-speicher
&52	Schreiben von Daten in beide Dis-playspeicher
&54	Kontrollregister von Displaytreiber 1
&55	Statusregister von Displaytreiber 1
Bit 5	0 : Anzeige ist eingeschaltet
	1 : Anzeige ist ausgeschaltet
Bit 7	0 : bereit zum Lesen und Schreiben
	1 : nicht bereit
&56	Schreiben in Displayspeicher 1
&57	Lesen aus Displayspeicher 1
&58	Kontrollregister von Displaytreiber 2
&59	Statusregister von Displaytreiber 2
Bit 5	0 : Anzeige ist eingeschaltet
	1 : Anzeige ist ausgeschaltet
Bit 7	0 : bereit zum Lesen und Schreiben
	1 : nicht bereit
&5A	Schreiben in Displayspeicher 2
&5B	Lesen aus Displayspeicher 2

Diskettenlaufwerk

&78 Bit 3	0 : Laufwerk ist leer
	1 : Laufwerk enthält Diskette
Bit 6	0 : Laufwerk steht oder der Schreib-

schutz ist aktiv
1 :Laufwerk läuft und Schreibschutz
ist inaktiv
Bit 7 0 : Laufwerk läuft
1 : Laufwerk steht

&79 Bit 0 Die drei Bits bilden zusammen die
Bit 1 Nummer der momentan eingestellten Bit 2 Spur

&7A Bit 7 schreiben:
0 : Laufwerksmotor ausschalten
1 : Laufwerksmotor **einschalten**

Drucker

&80 Interruptmaske fÜr die Peripherie:
Bit 0 Taste für Farbwechsel
Bit 1 Taste für Papiertransport vorwärts
Bit 2 Taste für Papiertransport rückwärts
Bit 3 Diskettenlaufwerk
Bit 4 PRINT-Schalter
Bit 5 Sensor für den Wagenrücklauf

&81 Interruptzustand, kann verwendet
werden, um verschiedene Tasten und
Schalter abzufragen.
Bit 0
bis Wie in Adresse &80
Bit 5
Bit 7 Cassetteneingang
0 : low
1 : high

&82 Weitere Druckersignale:
Bit 0 A
Bit 1 c Schrittmotor
Bit 2 B fÜr Farbwechsel

```
    Bit 3   D
    Bit 4   Remote ON, kurzer 1-Impuls genügt:
            Z=INP &82: OUT &82,Z OR 16: OUT &82,Z
    Bit 5   Remote OFF, kurzer 1-Impuls genügt:
            Z=INP &82: OUT &82,Z OR 32: OUT &82,Z
    Bit 7   Freigabe des Cassetteneingangs


&83         X-/Y-Schrittmotoren
    Bit 0   A
    Bit 1   C Schrittmotor
    Bit 2   B für X-Richtung
    Bit 3   D
    Bit 4   A
    Bit 5   C Schrittmotor
    Bit 6   B für Y-Richtung
    Bit 7   D
```

ANHANG B: RAM-ADRESSEN

Im folgenden werden einige nützliche Adressen des Systemteils des RAMs beschrieben. Mit den folgenden Befehlen wird auf ihre Inhalte zugegriffen:

	BASIC	Assembler
Daten lesen	Z=PEEK <adr>	LD A,(<adr>)
Daten schreiben	POKE <adr>,Z	LD (<adr>),A

<adr> liegt im Bereich *von 0 bis 65535.*

&F05F Vertikale Cursorposition für Textausgabe (CURSOR)

&F060 Horizontale Cursorposition für Textausgabe (CURSOR)

&F07F Zeiger auf den Kopf der Eingabewarteschlange im Tastaturpuffer
Bit 7 = 1 : Tastaturpuffer ist voll

&F080 Zeiger auf den Schwanz der Eingabeschlange

&F081 Code der momentan gedrückten Taste (ähnlich wie INKEY$)
Der Wert ist nur bei freigegebenem 1/64s-Interrupt gültig.

&F082 Autorepeatzähler, wird bei jedem Tastendruck neu gestartet. Erreicht er die 0, so beginnt der Autorepeat einzusetzen. Mit ihm kann festgestellt werden, wie lange der Benutzer an einer Taste 'geklebt' hat.

&F099 Horizontale Cursorposition für Graphik

&F09A ausgabe, 16 Bit im Zweierkomplement
       (GCURSOR)

&F098 Vertikale Cursorposition fÜr Graphikaus-
&F09C gabe, 16 Bit im Zweierkomplement
       (GCURSOR)

&F0B8 1/2 sec Zähler, zählt in Zweierschritten
       abwärts von 14 bis 0
       1/2s-Interrupt muß freigegeben sein.

&F0DF Tastaturpuffer, puffert maximal 64 Eingabe
 bis   zeichen bis zu ihrer Weiterverarbeitung
&F11E durch ein Programm

&F21D Eingabepuffer, nimmt eine Eingabezeile be-
 bis   stehend aus maximal 256 Zeichen aus dem
&F31C Tastaturpuffer auf. Solange die Zeile im
       Eingabepuffer steht, kann sie editiert werden. Erst
       bei einem ENTER wird sie weiterverarbeitet.

&F31D Stapel fÜr Interruptmasken, faßt bis
 bis   zu 8 Bytepaare
&F32C (näheres siehe Kapitel "Interrupts")

&F32D Nummern aller initialisierten Interrupt
 bis   quellen (maximal 8)
&F334

&F335 Stapelzeiger für Interruptmaskenstapel

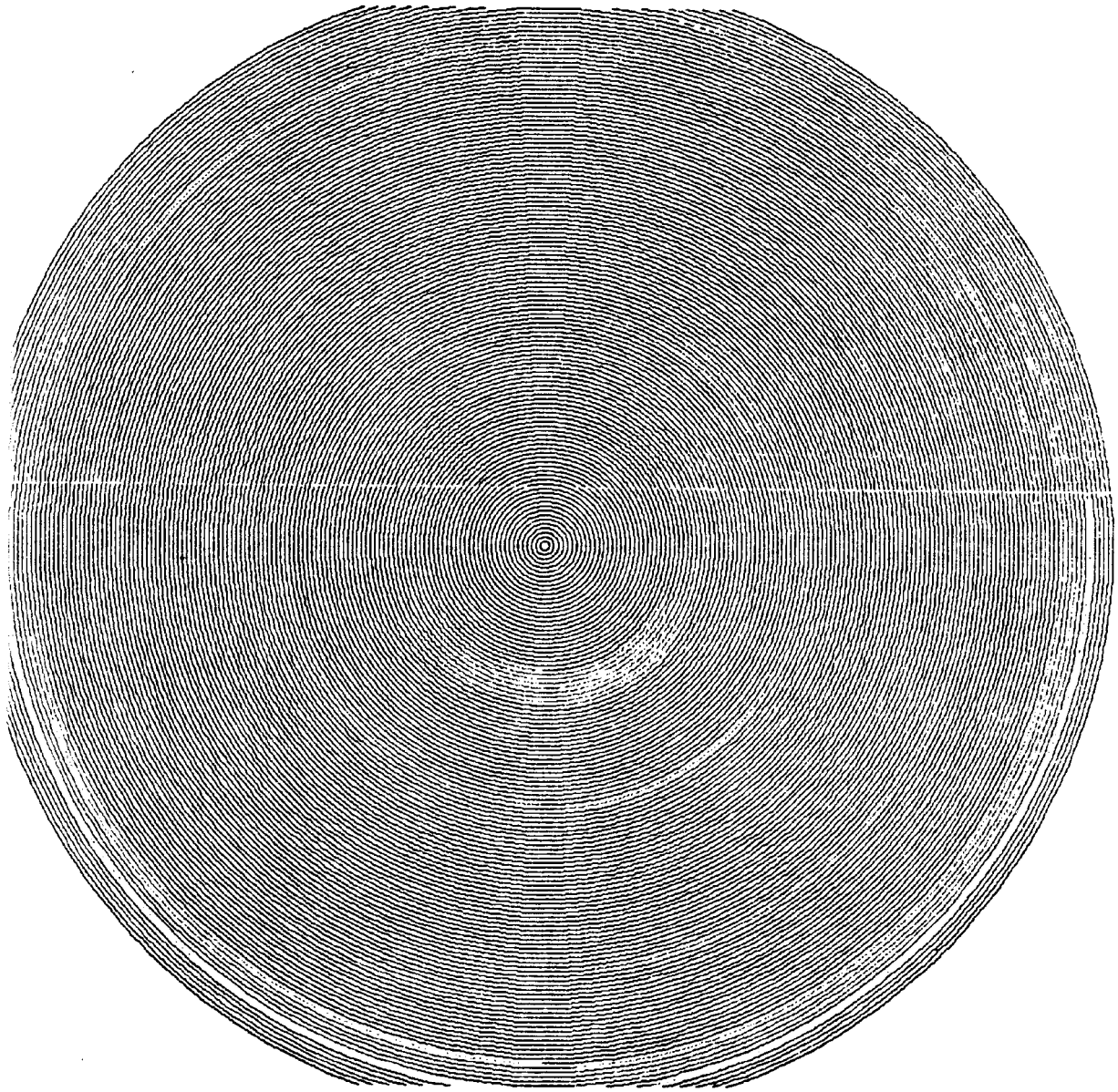
&F600 Displaypuffer
 bis   (nur im PC-1500-Modus von Bedeutung)
&F64F

&F64E Displaysymbole (RUN, PRO usw.)

&F64F Auch im PC-1600-Modus wird auf sie zugegriffen.

```
&F650  Standardvariablen
bis    E$ bis O$
&F6FF  (jeweils 16 Bytes)


&F700  Displaypuffer
 bis   (nur im PC-1500-Modus von Bedeutung)
&F74F


&F750 Standardvariablen
 bis   P$ bis Z$
&F7FF (jeweils 16 Bytes)


&F864  oberste frei verfügbare RAM-Adresse,
       nur höherwertiges Byte, Bit 7 ist 0
       Bei CLEAR wird das obere Ende des Variablenbereichs
       auf diesen Wert gesetzt.


&F865  Zeiger auf den Programmanfang, wird verwen-
&F866  det bei NEW   ohne Parameter
       Bit 7 ist 0 (LH5803-Adresse).

&F867  Zeiger auf das Programmende, Bit 7 ist 0
&F868  Er zeigt immer auf den &FF-Code


&F86B  Flags für BEEP ON/OFF und RMT ONIOFF

&F88D  TRON/TROFF-Flag


&F8BE  DATA-Zeiger, kann mit Hilfe von RESTORE n

&F8BF  zur Auffindung der Adresslage von BASIC-
       zeilen verwendet werden
&F8C0  Standardvariablen
bis    A$ bis D$
&F8FF  (jeweils 16 Bytes)

&F900  Standardvariablen
bis    A bis Z
&F9CF  (jeweils 8 Bytes)
```

&FA38 Stapel für FOR-Schleifen und GOSUBs
bis FOR-Parameter werden von &FA38 an aufwärts,
&FAFF GOSUB-Parameter von &FAFF an abwärts abgelegt.



ANHANG C: ADRESSRAUMBELEGUNG

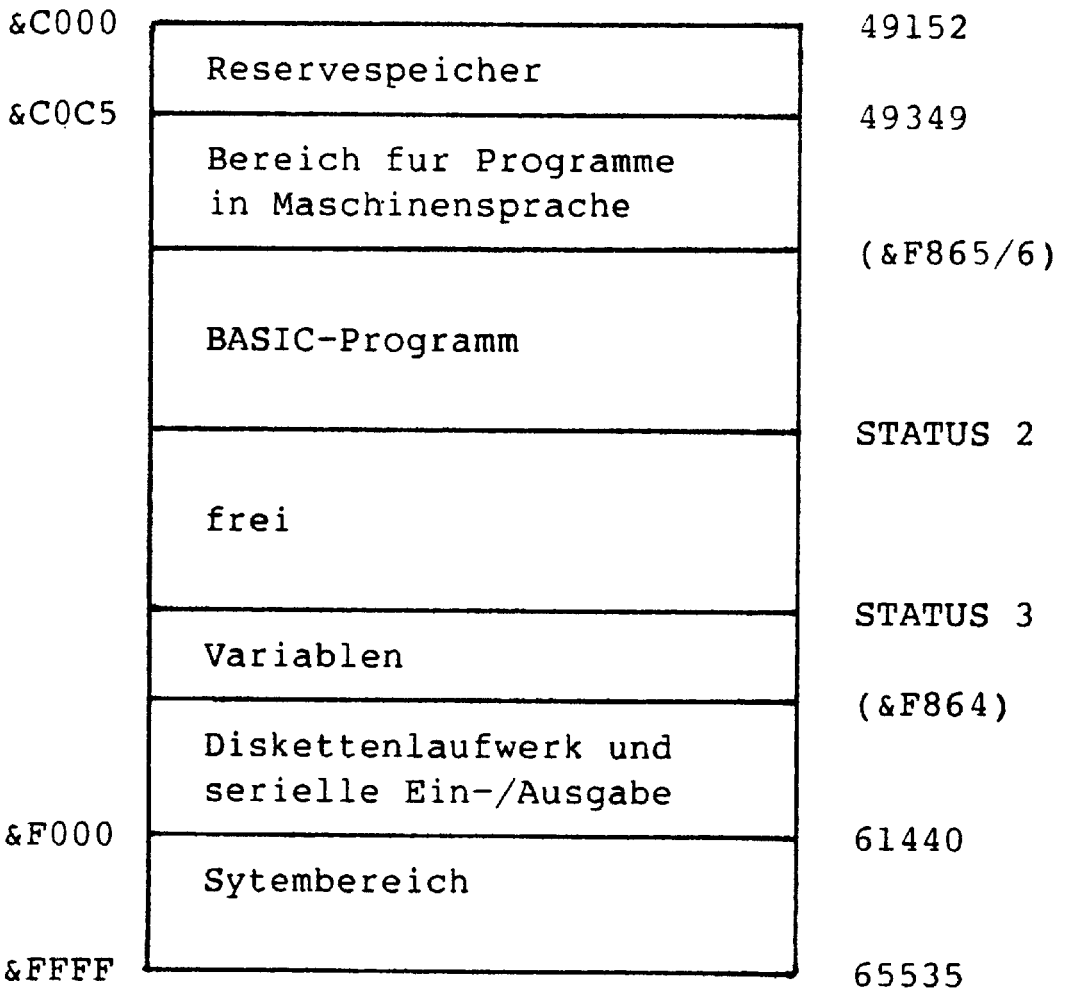
Es können bis zu 320 KByte in 8 Speicherbänken adressiert werden. Teile davon, die noch nicht genutzt sind, können von Sytemerweiterungen verwendet werden.

Bank Adressbereich

Bank	0000-3FFF	4000-7FFF	8000-BFFF	C000-FFFF
0	PC-1600 ROM	PC-1600 ROM	RAM Slot 1	RAM Grundv.
1	un-genutzt	RAM Slot 2		un-genutzt
2	nicht vergeben	un-genutzt	RAM Slot 2	nicht vergeben
3		PC-1600 ROM	Diskette	
4		Drucker ROM	un-genutzt	
5		Floppy ROM		
6			PC-1600 ROM	
7		un-genutzt		

ANHANG D: RAM-BELEGUNG

Das RAM im Bereich von &COOO bis &FFFF ist wie folgt unterteilt:



E: TABELLE DER BASIC-TOKENS

F16D	NOT	F2A4	RCVSTAT
F297	NAME	E85A	RINKEY$
F19B	NEW	F1AB	REM
F192	GOTO	F180	AREAD
F194	GOSUB	F150	AND
F09F	GPRINT	F170	Aas
F093	CCURSOR	F175	ATN
F186	GRAD	F173	ASN
E681	GRAPH	F160	ASC
E682	GLCURSOR	F174	ACS
F091	INPUT	F25A	AIN
F196	IF	F181	ARUN
F266	INP	F280	ADIN
F294	INIT	F2B6	AUTO
F267	INSTR	F2BD	AS
E859	INSTAT	F25C	ALARM$
F171	INT	F2BF	APPEND
F15C	INKEY$	F2BC	AOFF
F1A4	RUN	F182	BEEP
F199	RETURN	F290	BLOAD
F1A6	READ	F291	BSAUE
F2B5	RENUM	F0B3	BREAK
F1A7	RESTORE	F183	CONT
F28D	RESUME	F17E	COS
F28E	RETI	F0B5	COLOR
F17C	RND	F293	COPY
F256	RXD$	F0B1	CONSOLE
F1A8	RANDOM	E858	COM$
F172	RIGHT$	F2A3	COM
F1AA	RADIAN	F084	CURSOR
F0BA	RLINE	F163	CHR$
E7A9	RMT	F0B2	CHPIN
E685	ROTATE	F187	CLEAR

```
F088 CLS            F099 LIN
F089 CLOPD          F177 LOG
F292 CLOSE          F1B5 LOCK
F095 CSAVE          F295 LOAD
E680 CSIZE          F272 LOC
F282 CALL           F273 LOF
F18B DIM            F176 LN
F18C DEGREE         F198 LET
F2B9 DELETE         F164 LEN
F165 DEG            F17A LEFT$
E857 DEV$           F0A0 LFILES
F166 DMS            F0B6 LF
E884 DTE            F0B9 LPRINT
F18D DATA           F0B8 LLIST
F274 DSKF           F0B7 LLINE
F257 DATE$          F0A5 LCURSOR
F18E ENG            F158 MEM
F178 EXP            F08F MERGE
F271 EOF            F17B MID$
F283 ELSE           F288 MAXFILES
F1B4 ERROR          F2B3 MOOE
F053 ERL            F250 MOD
F052 ERN            F19C ON
F2B7 ERASE          F151 OR
F1A5 FOR            F296 OPEN
F098 FILES          F19D OPN
F0B0 FEED           F19E OFF
F265 HEX$           E880 OUTSTAT
F287 KILL           F2BE OUTPUT
F286 KEYSTPT        F28A OUT
F285 KEY            F097 PRINT
F284 KBUFF$         F09A PRESET
F090 LIST           F26E PEEK#
```

```
F26D PEEK
F0A4 PITCH
F15D PI
F28C POKE
F28B POWER
F168 POINT
F1A2 PAUSE
F2B8 PASS
E381 PAPER
F2B4 PZONE
F09B PSET
F2B1 PCONSOLE
F2A0 PHONE
F299 SAVE
F1AC STOP
F16B SQR
F17D SIN
F179 SGN
F161 STR$
F167 STATUS
F1AD STEP
E886 SETDEV
E882 SETCOM
F298 SET
F2A2 SNDSTAT
F2A1 SNDBRK
F061 SPACE$
E684 SORGN
F1AE THEN
F17F TAN
F258 TIME$
F15B TIME
F0BC TEST
E883 TERMINAL
E686 TEXT
F1AF TRON
F1B0 TROFF
E885 TRANSMIT
F1B1 TO
F0BB TAB
F2BA TITLE
F085 USING
F1B6 UNLOCK
F162 UAL
F1B3 WAIT
F261 WAKE$
F18A XCALL
F16E XPEEK#
F251 XOR
F16F XPEEK
F1A0 XPOKE#
F1A1 XPOKE
F0B4 ZONE
```

ANHANG F:
BEISPIELPROGRAMME

Schnelle Potenzrechenroutine

```
10:G=20
20:INPUT "BASIS";A
30:INPUT "EXPONENT";N
40:GOSUB "NORMAL"
50:GOSUB "SCHNELL"
6O:GOTO 20
20:"NORMAL"
80:BEEP 1
30:FOR I=1TO G
100:X=A^N
110:NEXT I
120:BEEP 1
13O:PRINT X
140:RETURN
150:"SCHNELL"
150:BEEP 1
120:FOR I=1TO G
180:T=A:X=1
190:FOR P=1T0 N+1
200:IF N AND P LET X=X*T
210:T=T*T: P=P+P-1
220:NEXT P
230:NEXT I
243:BEEP 1
250:PRINT X
250:RETURN
```

Sieb des Eratosthenes

```
10:DIM P(7)
20:'Belegung des verbleibenden Speicher-
30:'platzes mit einem 255*X-Arrag für
40:'für möglichst grosses X
50:F=INT ((STATUS 258-7)/255)
60:DIM A$(254,F-1)*1
70:'Im folgenden wird A$ nur, noch ein-
80:'dimensional benutzt. Neue Grenze:
90:F=F*255-1: F=10: G=16*F+15
100:PRINT "Es wer-den alle Zahlen"
110:PRINT "bis";G; " untersucht."
120:'Potenzen von 2 (zum Setzen und
130:'Ausblenden einzelner Bits)
140:P=1:FOR I=0TO 7:P(I)=P:P=P*2:NEXT I
150:'2 ist Prim, 1 nicht.
160:N=1:PRINT USING "#######";N;".";2
170:A$(0)=CHR$ &01
180:'Um Divisionen durch 8 zu sparen,
190:'werden einige Werte durch zwei
200:'Variablen ausgednückt: S=8*T+U+1
210:'Inkrement=B*D+E, Bitnr. =8*A+B
220:T=0:U=0
230:'Alle Primzahlen bis zur Wurzel:
240:FOR Z=1TO SQR GSTEP 2
250:IF ASC A$(T)AND P(U)GOTO 320
260:D=2*T:E=2*U+1
270:IF E>7LET E=E-8:D=D+1
```

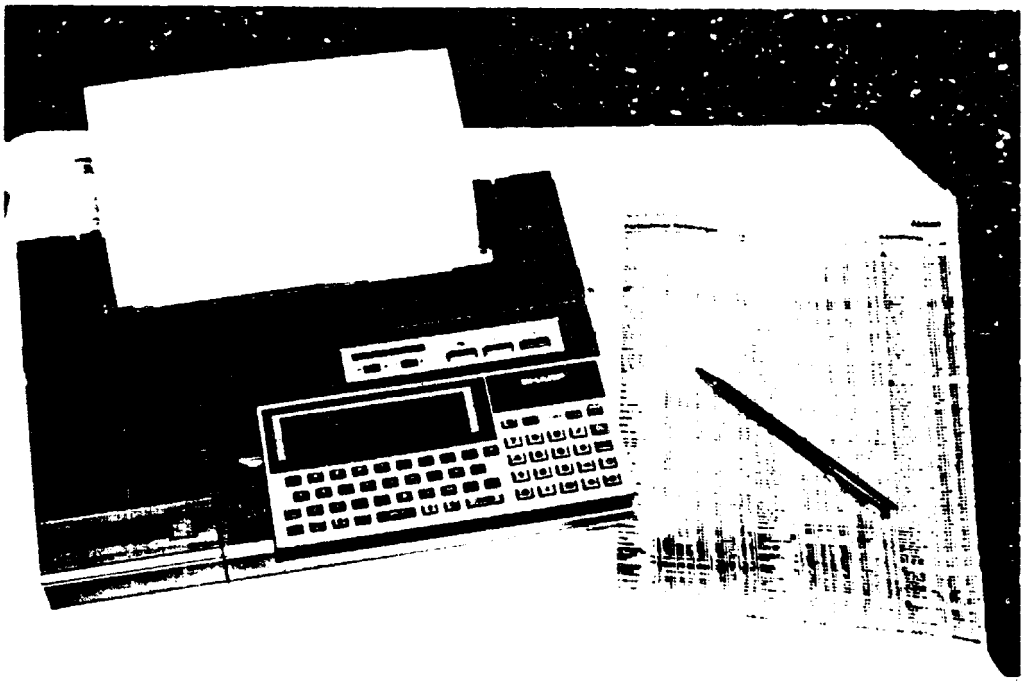
```
280:A=T:B=U:N=N+1:PRINT N;".";Z
290:A=A+D:B=B+E
300:IF B>7LET B=B-8: A=A+1: IF A>FGOTO 320
310:A$(A)=CHR$ (ASC A$(A)OR P(B)):GOTO 290
320:U=U+1:IF U>7LET U=0:T=T+1
330:NEXT Z
340:'Anzeigen der restlizhen Primzahlen
350:FOR Z=ZTO GSTEP 2
360:IF (ASC A$(T)AND P(U))=0LET N=N+1:PRINT N;
   "."; Z
370:U=U+1:IF U>7LET U=0:T=T+1
380:NEXT Z
```

Zeichnen eines Punktes ohne PSET

```
10:CURSOR 0,0:INPUT X,Y:GOSUB 100
20:GOTO 10
100:'PLOT (X, Y)
110:'Welcher, Speichenbaustein soll
120:'angesprochen werden?
130:IF (XAND 127)<64LET C=&58ELSE LET C=&54
140:'Bytenr. (0-63) aus X errechnen
150:D=C+2: A=XAND 63
160:'Speicherzeile
170:B=Y\8: IF X>127LET B=B+4
180:'ZO-Zeigen berücksichtigen
190:H=(PEEK &F05C+B)AND 7
200:'zu schreibendes Byte
210:P=2^(YAND 7)
220:'0,5s-Interrupt sperren
230:M = INP 53
240:OUT 53,MAND &BF
250:'Adressierung des Displaspeichers
260:OUT C,&B8+H:.OUT C,&40+P
270:'Zweimal lesen, um den alten Wert
280:'zu übengehen
290:L=INP (D+1):L=INP (D+1)
300:OUT C,&40+A
310:'OR bewirkt, dass der bisherige
320:'Anzeigeninhalt bestehen bleibt.
330:OUT D,LOR P
340:'Wiederherstellung der Interruptmaske
350:OUT 53,M
360:RETURN
```

Wertpapierverwaltung mit SHARP-Taschencomputern

Börse PC-1600



Fischel GmbH H.J.Neumann

ISBN: 3-924327-64-5

Softscrolling

```
10:'SCROLL AUF
20:M=PEEK &F05C:H=&C1+8*M
30:FOR I=0TO 7
40:OUT &50,(H+I)AND &FF
50:'Löschen der aus dem Display
60:'hinauslaufenden Zeile
70:LINE (0,I)-(155,I),R
80:NEXT I
90:'gespeicherter Y0-Zeiger korrigieren
100:POKE &F05C,(M+1) AND 7
110:RETURN
```

```
200:'SCROLL AB
210:M=(PEEK &F05C-1)AND 7
220:'Löschen der, Symbole, damit sie
230:'nicht in die Anzeige wandern
240:OUT &35,&1F
250:OUT &54,&B8+M:OUT &54,&7F
250:OUT &56,&0:OUT &35,&5F
270:H=&A8+8*M
280:FOR I=31T0 24STEP -1
290:OUT &50,(H+I)AND &FF
300:LINE (0,I)-(155,I),R
310:NEXT I
320:POKE &F05C, M
330:RETURN
```

Bewegte Darstellung dreidimensionaler Funktionsgraphen

```
10:'Erste freie Speicheradresse:
20:F=STATUS 2+&8001:A=F:S=0
30:'Einlesen eines Untenprogramms
40:'in Z8O-Code
50:READ B:POKE A,B:S=S+B
60:IF B<>&C9LET A=A+1:GOTO 50
70:'Prüfsumme ermitteln
80:IF S<>7478PRINT "Fehler!!!":END
90:'Nächste freie Adresse (im 2-Komplement)
100:G=A-65535:N=16
110:'Zeichnen und Speichern von N Bildern
120:R=360/128:S=360/32
130:FOR  I=0TO N-1
140:T=I*360/N:CLS
150:FOR Y=0TO 31
160:FOR X=0TO 126STEP 2
170:Z=7*(SIN (X*R+T)+COS (Y*S+T))
180:PSET (X+Y,Y-Z)
130:NEXT X:NEXT Y
200:'Ablegen an der Adresse A
210:'Jedes Bild belegt 624 Bytes
220:A=G+624*I:CALL F,A
230:NEXT I
240:'Modifizieren des Codepnogramms
250:POKE F+43,0:POKE F+49,0:POKE F+55,&A3
260:'Nun schreibt das Programm die Bilder
270:'wieder auf die Anzeige zurück
280:P=20
290:FOR I=0TO N-1
300:A=G+624*I:CALL F,A
310:FOR J=0TO P:NEXT J
320:NEXT I
300:GOT0 230
340:'Unteprogramm zu-,n Speichern
350:'Wiederherstellen von Bildern, als
```

```
360:'Parameter muss eine RAM-Adresse
370:'angegeben werden
380:DATA &F3,&62,&6B,&16,&0B,&7A,&0E,&58,&E6,&08
390:DATA &28,&02,&0E,&54,&7A,&06,&40,&E6,&04,&28
400:DATA &02,&06,&1C,&3A,&5C,&F0,&82,&E6,&07,&F6
410:DATA &B8,&CD,&EA,&08,&ED,&79,&3E,&40,&CD,&EA
420:DATA &08,&ED,&79,&0C,&0C,&0C,&CD,&EA,&08,&ED
430:DATA &78,&CD,&EA,&08,&ED,&A2,&20,&F9,&15,&7A
440:DATA &3C,&20,&C6,&FB,&37,&3F,&C9
```

Adressierung der Sondersymbole

```
10:CLS
20:INPUT "Speicherzeile (4-7): ";Z
30:INPUT "Byte (0-255): ";B
40:OUT &35,&1F:GOSUB 100
50:FOR I=1T0 800:NEXT I:OUT &35,&5F
60:GOTO 10
100:'SYMBOLE
110:'Speicher,zeile 2 adressieren
120:OUT &54,&B8+((PEEK &F05C+Z)AND 7)
130:'Die Sgmbole stehen im letzten Byte
140:'der, Zeile
150:OUT &54,&7F:OUT &56,B
160:RETURN
```

Anzeige der Belegung der Funktionstasten

```
10:'Belegung der Funktionstasten
20:DIM A$(2)*3
30:A$(0)= "  I":A$(1)="III":A$(2)=" II"
40:A=&C056
50:P=PEEK A
60:'Am Ende steht immer ein 0-Byte.
20:IF P=0END
80:'Berechnung der Ebene und der
90:'Tastennummen
100:E=(PAND &18)/8
110:N=PAND 7
120:PRINT A$(E);" F";STR$ (N);": ";
130:A=A+1: P=PEEK A
140:'Ist es ein BASIC-Token?
150:IF P>=&E0GOTO 210
160:'Ist es ein sichtbanes ASCII-Zeichen?
170:IF P>31PRINT CHR$ P;:GOTO 130
180:PRINT :GOTO 70
190:'Suchen des Tokens in der Token-
200:'Tabelle im ROM
210:A=A+1:Q=PEEK A:D=&B717
220:IF PEEK D<>QGOTO 280
230:IF PFEK (D+1)<>PGOTO 280
240:D=D+3
250:FOR I=1TO PEEK D
260:PRINT CHR$ PEEK (D+I);
270:NEXT I:GOTO 130
280:D=D+PEEK (D+3)+7
290:GOTO 220
```

Neubelegung der Funktionstasten durch ein Programm

```
10:E=3:'Ebene
20:N=2:'Tastennummer
25:'gewünschter Inhalt:
30:A$="PRINT SIN5@"
40:GOSUB 100:END
100:'Unterprogramm zun Belegung der
110:'Funktionstasten von Programmen aus
120:'Bgrechnung des Codes für Ebene
130:'und Tastennummen
140:G=8*((E>1)+(E=2))+N
150:'Beginn des Reservespeichers
160:D=&C055
120:'Suchen nach einem evtl. schon
180:'vorhandenen Eintrag für diese Taste
130:D=D+1: P=PEEK D
200:IF P=0 GOTO 320
210:IF P<>G GOTO 130
220:A=D
230:A=A+1: P=PEEK A
240:IF P>31GOTO 230
250:IF P=0GOTO 320
250:'Löschen des Eintrags,falls vorhanden
220:'Nachfolgende Einträge werden
280:'entsprechend verschoben.
290:POKE D,P:D=D+1
300:A=A+1 P=PEEK A
310:GOTO 250
320:POKE D,G
330:'Neueintrag
340:FOR I=1TO LEN A$
350:POKE D+I,ASC MID$ (A$,I,1)
360:NEXT I
370:'Endemarke setzen
380:POKE D+I,0:RETURN
```

Einfaches Multitasking

```
 10:'N=Anzahl der Tasks minus 1
 20:N=2:DIM B(N,5)
 30:'ADIN erzeugt alle 0,5s einen Interrupt
 40:'wenn nichts angeschlossen ist
 50:ON ADIN (255,255)GOSUB 160
 60:'Ermittlung der Startadresse jeder Task
 70:S=PEEK &F891:A=&F9FB+S
 80:FOR P=0TO N
 90:POKE &F891,S
100:ON P+1GOTO "A","B","C"
110:FOR I=0TO 5:B(P,I)=PEEK (A+I):NEXT I
120:NEXT P
130:'RETURN springt in die letzte Task (C)
140:P=N:ADIN ON :RETURN
150:'Merken den letzten bearbeiteten Position
160:FOR I=0TO 5:B(P,I)=PEEK (A+I):NEXT I
170:P=P+1:IF P>NLET P=0
180:'Aufruf der nächsten Task vorbereiten
190:FOR I=0TO 5:POKE A+I,B(P,I):NEXT I
200:POKE &F1D3,0:RETI
210:"A"GOSUB 110
220:'Task A zeichnen Sinuskurven
230:CLS :F=1
240:X=0
250:Y=16+15*SIN (X*3):PSET (X,Y)
260:X=X+1:IF X<156GOTO 250
270:F=F+1:GOTO 240
280:"B"GOSUB 110
290:'Task B zählt von B abwärts und gibt die
300:'Kontrolle nach jedem Schritt von sich aus
310:'weiter.
320:U=10
330:U=U-1:IF U>0POKE &F1D3,&80:GOTO 330
340:BEEP 1,50:GOTO 320
350:"C":GOSUB 110
360:'Task C knarrt und tut sonst nichts
370:BEEP 1,3,10:GOTO 370
```

Abfrage von mehreren gleichzeitig gedrückten Tasten

```
10:'A: Spaltenauswahl
20:'B: Zeilenauswahl
30:'Die angegeben Werte sind für die Tasten
40:' 2 4 6 und 8
50:A=1:B=1:GOSUB 220
60:IF FBEEP 1,17,60
70:A=4:B=2:GOSUB 220
80:IF FBEEP 1,20,50
90:A=128:B=2:GOSUB 220
100:IF FBEEP 1,25,40
110:A=1:B=4:GOSUB 220
120:IF FBEEP 1,30,30
130:GOTO 50
200:'Das Unterprogramm setzt F=1 wenn die
210:'betreffende Taste gedrückt ist.
220:BREAK OFF :OUT &35,&4F
230:OUT &1C,A:OUT &1E,255-A
240:F=(BAND INP &37)=0
250:OUT &35,&5F
260:BREAK ON :RETURN
```

Dampflok: Rauschgenerator und Bewegung

```
10:CLS :P=0
20:'Einlesen der Maschinenprogramme
30:FOR I=0TO 117:READ B
40:POKE &D000+I,B:P=P+B
50:NEXT 1
60:IF P=12329GOTO 110
70:PRINT "Fehler, in den DATA-Zeilen"
80:PRINT "für Anzeige oder Rausch-"
90:PRINT "generator!!!"
100:END
110:'Zeichnen der Landschaft
120:LINE (0,24)-(155,31),,1290,BF
130:LINE (0,5)-(4,14):LINE -(6,24)
140:LINE (155,7)-(151,14):LINE -(149,24)
150:LINE (2,7)-(20,0):LINE -(40,14)
160:LINE -(45,3):LINE -(50,10)
170:LINE -(70,5):LINE -(80,0)
180:LINE -(90,10):LINE -(103,14):LINE -(120,5)
190:LINE -(125-,10):LINE -(130,3):LINE -(135,8)
200:LINE -(140,1):LINE -(150,4):LINE -(153,7)
210:LINE (5,14)-(7,14):LINE —(7,24)
220:LINE -(148,24):LINE -(148,14):LINE -(150,14)
230:CURSOR 7,2:PRINT "Bitte warten"
240:D=&D100
250:A=D:G=170
260:FOR I=1TO G:POKE A,0:A=A+1:NEXT I
270:'Zusammenstellung des Zuges
280:RESTORE "W1":GOSUB "WAGGON"
290:RESTORE "W1":GOSUB "WAGGON"
300:RESTORE "W3":GOSUB "WAGGON"
310:RESTORE "W3":GOSUB "WAGGON"
320:RESTORE "W4":GOSUB "WAGGON"
330:RESTORE "W2":GOSUB "WAGGON"
340:RESTORE "W2":GOSUB "WAGGON"
```

```
350:RESTORE "W0":GOSUB "WAGGON"
360:RESTORE "W0":GOSUB "WAGGON"
370:RESTORE "W0":GOSUB "WAGGON"
380:RESTORE "LOK":GOSUB "WAGGON"
390:FOR I=1TO 140:POKE A,0:A=A+1:NEXT I
400:OUT 53,&1F
410:A=D+G-65536
420:P=25:Q=65:B=1536
430:'Und los geht's ...
440:FOR I=1TO G
450:IF I>POUT 23,Q:Q=65-Q:P=P+12+Q*.3
460:IF I<6LET B=B-256
470:IF I>141LET B=B+256:IF B>2048LET B=2048
480:T=511/(I+1):IF T<7LET T=7
490:R=T+B
500:CALL &D000,A:CALL &D045,R
510:FOR J=1T0 T-6:NEXT J
520:A=A-1:NEXT I
530:FOR I=1TO 999:NEXT I
540:OUT 53,&5F:CLS :END
550:"WAGGON"
560:READ N
570:FOR I=1TO N:READ B
580:POKE A,B:A=A+1:NEXT I
590:G=G+N:RETURN
600:'Anzeige
610:DATA &62,&6B,&3E,&48,&06,&38,&0E,&58,&16,&02
620:DATA &CD,&26,&D0,&3E,&40,&06,&40,&0E,&54,&16
630:DATA &02,&CD,&26,&D0,&3E,&40,&06,&14,&0E,&58
640:DPTA &16,&06,&CD,&26,&D0,&37,&3F,&C9,&ED,&79
650:DATA &0C,&ED,&78,&20,&FC,&0D,&3A,&5C,&F0,&82
660:DATA &E6,&07,&F6,&B8,&ED,&79,&0C,&0C,&0D,&ED
670:DATA &78,&20,&FC,&0C,&ED,&A3,&20,&F6,&C9
```

```
680:'Rauschgenerartor
690:DATA &F3,&1F,&CB,&10,&CB,&11,&CB,&15,&17,&A9
700:DATA &E6,&01,&C6,&FF,&D3,&18,&C5,&42,&25,&20
710:DATA &03,&1D,&28,&10,&05,&28,&0A,&E3,&E3,&0E
720:DATA &00,&00,&00,&E3,&E3,&18,&ED,&C1,&18,&D9
730:DATA &C1,&FB,&3E,&C0,&D3,&18,&37,&3F,&C9
740:'Bitmuster für verschiedene Waggons
750:"W0"DATA 20
760:DATA &21,&3F,&39,&FF,&F9,&39,&3F,&39,&39,&3F
770:DATA &39,&39,&3F,&39,&F9,&FF,&39,&3F,&21,&20
780:"W1"DATA 20
790:DATA &20,&38,&38,&F8,&F8,&38,&38,&38,&38,&38
800:DATA &38,&38,&38,&38,&F8,&F8,&38,&38,&20,&20
810:"W2"DATA 20
820:DATA &20,&3E,&30,&F0,&F0,&3E,&30,&30,&30,&3E
830:OPTP &30,&30,&30,&3E,&F0,&F0,&30,&3E,&20,&20
840:"W3"DATA 20
850:DATA &20,&2E,&3F,&FF,&FF,&2E,&20,&2E,&3F,&3F
860:DPTP &3F,&2E,&20,&2E,&FF,&FF,&3F,&2E,&20,&20
870:"W4"DATA 28
880:DATA &20,&FF,&FF,&3F,&21,&F9,&FF,&21,&2D,&7F
890:DATA &77,&7F,&7B,&61,&7F,&61,&67,&7F,&21,&21
900:DATA &FF,&E1,&21,&3F,&FF,&FF,&20,&20
910:"LOK"DATA 20
920:DATA &20,&FE,&FE,&3C,&FC,&FC,&21,&3F,&39,&F9
930:DATA &FF,&3C,&3C,&FE,&FC,&3C,&3C,&FF,&FF,&3C
```

KBUFF$ ohne Löschen

```
500:'KBUFF$ ohne Löschen des bisherigen
510:'T-Pufferinhalts
520:L=LEN K$(0)
530:'wieviele Bytes im Tastaturpuffer,
540:'sind noch frei?
550:S=PEEK &F080:K=PEEK &F07F
560:F=S-K:IF F<=0LET F=F+64
570:IF F<LOR F=128PRINT "Puffer voll!":GOTO 680
580:'Anfügen an das linke Ende des Puffers
590:FOR I=LTO 1STEP -1
600:S=S-1AND 63
610:POKE &F0DF+S,ASC MID$ (K$(0),I,1)
620:NEXT 1
630'wenn der Schwanz auf den Kopf trifft,d.h.
640:'wenn genau 54 Zeichen im Puffer stehen,
650:'wird Bit 7 des Kopfes auf 1 gesetzt.
660:IF S=KPOKE &F07F,S+128
670:POKE &F080,S
680:RETURN
```

LITERATUR

1. **PC-1600** Bedienungsanleitung

2. **PC-1600** Servicemanual