DULLIN · RETZLAFF · SCHNEIDER
STRASSENBURG

CPC
TIPS & TRICKS

EINE FUNDGRUBE FÜR DEN
CPC 464, 664 & 6128

Band 2

EIN DATA BECKER BUCH

DULLIN · RETZLAFF · SCHNEIDER
STRASSENBURG

CPC
TIPS & TRICKS

EINE FUNDGRUBE FÜR DEN
CPC 464, 664 & 6128

EIN DATA BECKER BUCH

ISBN 3-89011-131-9

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

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

Inhaltsverzeichnis

Einleitung

Die Gliederung dieses Buches

Das vorliegende Buch ist nach übergreifenden Themenbereichen in drei Teile gegliedert:

Der erste Teil beschäftigt sich ausschließlich mit dem BASIC der Schneider-Rechner. Dort werden interessante und nützliche BASIC-Programme und -Routinen vorgestellt, sowie einige nützliche Tips im Umgang mit dem BASIC, die beispielsweise den Schutz eigener Programme oder die Beschleunigung von BASIC-Programmen betreffen.
Am Ende dieses Teils finden Sie eine genaue Beschreibung von Schneider-spezifischen BASIC-Befehlen mit Angaben über das Einsatzgebiet dieser Befehle.

Der zweite Teil enthält Maschinenroutinen, die sofort eingesetzt werden können und sinnvolle Ergänzungen der Möglichkeiten des BASIC-Interpreters bilden.
Obwohl die in diesem Teil vorgestellten Programme durchweg in Maschinensprache geschrieben sind, haben sie, als sofort betriebsbereite Programme, auch für diejenigen Leser einen Wert, die mit Maschinensprache 'nichts am Hut haben'.
Obwohl dieser Teil in erster Linie fertige Programme zum Abtippen bietet, finden Sie auch dort sicherlich nützliche Hintergrundinformationen.

Der dritte Teil dieses Buches beschäftigt sich mit der Maschinensprache des Schneider-Rechners. Er enthält eine recht umfangreiche Liste interessanter Routinen des Betriebssystems und des BASIC-Interpreters. Weiterhin finden sich dort einige Tips und Tricks für die effektive Programmierung in Maschinensprache (Zeitoptimierung), sowie für die Verwendung bestimmter Mechanismen die Schneider-Betriebssystems, wie zum Beispiel die Befehlserweiterung mittels RSX.
Das Kapitel 23 in diesem Teil beschreibt eine nach unseren Informationen bislang unveröffentlichte Methode für die Relokalisation (Verschiebung mit Anpassung absoluter Adressen)

von Maschinenprogrammen. Dieses Kapitel wurde besonders ausführlich gestaltet, um die dort benutzten Techniken der Änderung eines Programmes durch ein Programm und einer trickreichen Benutzung des Stacks offenzulegen. Aus diesem Kapitel können sicherlich auch etwas fortgeschrittenere Maschinenprogrammierer noch etwas über 'Maschinenprogrammierung mit Stil' lernen.
Obwohl sich der dritte Teil praktisch ausschließlich mit Maschinensprache beschäftigt, dürfte er auch von Interesse für die Leser sein, die bislang noch keinen näheren Kontakt mit der Maschinensprache hatten. Besonders die ersten beiden Kapitel dieses Teils sind mit Rücksicht auf diese Lesergruppe mit einführendem Charakter geschrieben worden und enthalten ausgesprochen ausführlich erklärte und aufgrund ihres geringen Umfangs sehr übersichtliche Beispielprogramme.

Die Notation von hexadezimalen Zahlen

Hexadezimale Zahlen sind in diesem Buch grundsätzlich durch ein vorangestelltes '&' gekennzeichnet. Dies ist die Notation, die auch vom Schneider-BASIC verwendet wird; wir haben sie aus Gründen der Standardisierung übernommen. Einige Assembler verwenden andere Notationen (beispielsweise ein vorangestelltes '#' oder ein nachgestelltes 'H'). In diesem Fall ist natürlich beim Übernehmen der Source-Listings auf eine entsprechende Anpassung zu achten.

Sonderzeichen in Listings

In diesem Buch werden Sie des öfteren dem Zeichen '^' begegnen. Verzweifeln Sie nicht bei der Suche dieses Zeichens auf der Tastatur Ihres Rechners! Es entspricht dem Pfeil nach oben, der im BASIC als Potenzierungsoperator verwendet wird. Der Hochpfeil wird üblicherweise als '^' von einem Drucker ausgegeben.

Die Tastatur der CPCs

Unglücklicherweise sind die Bezeichnungen bestimmter Tasten der drei CPCs nicht einheitlich. Wann immer in diesem Buch von der Taste RETURN gesprochen wird (bezieht sich auf den 6128), so ist auf dem 664 und dem 464 die große ENTER-Taste im Haupt-Tastaturblock zu benutzen.

TEIL 1

Tips & Tricks zu BASIC

1. Sortierverfahren

Warum beschäftigen wir uns mit diesem Thema?

Handelt es sich dabei doch um Programme, die relativ unanschaulich sind, und deren Nutzen fragwürdig ist.

Besonders interessant ist die Untersuchung von Sortieralgorithmen in bezug auf die dabei erreichten Geschwindigkeiten. Ein Großteil der von Computern verbrauchten Rechenzeiten wird auf das Sortieren verwendet. Kaum ein Anwenderprogramm kommt, auch schon im kleineren Rahmen, ohne eine Sortierfunktion aus. Nicht grundlos ist man ständig auf der Suche nach besseren, d.h. schnelleren Sortierverfahren.

Ursprünglich sollte dieses Kapitel recht kurz werden. Bei der intensiveren Beschäftigung mit dem Thema erwies es sich jedoch als so interessant, daß wir etwas ausführlicher darauf eingehen werden.

Besonders Sortieralgorithmen sind oft trickreiche Verfahren, deren Programmierung nur wenige Zeilen in Anspruch nimmt. Gerade bei den einfacheren Programmen von weniger als 10 Zeilen ist erstaunlich, welch große Leistungsunterschiede auftreten können. Die Schwierigkeit ist also nicht die, ein Programm in irgendeiner Sprache zu erstellen, sondern liegt vielmehr darin, ein möglichst effektives Verfahren zum Sortieren von Daten zu entwickeln.

Beim Sortieren bestimmen zwei Faktoren maßgeblich die Rechenzeit.

1. Die Anzahl der durchschnittlich auszuführenden Vergleiche
2. Die Anzahl der durchschnittlich durchzuführenden Datenverschiebungen bzw. Datenvertauschungen.

Diese beiden Faktoren, besonders die Anzahl der Vergleiche, gilt es möglichst klein zu halten.

Die Qualität eines Algorithmus bestimmt sich aus diesen Größen. Das Problem ist, daß die Sortierzeit nicht proportional zur Menge der zu sortierenden Daten ist. Das bedeutet:

Wenn es eine Sekunde dauert, 10 Namen zu sortieren, verdoppelt sich die Zeit beim Sortieren von 20 Namen nicht notwendigerweise. Vielmehr steigt die Zeit bei schlechten Algorithmen quadratisch an. Bei der doppelten Menge an Daten benötigt man die vierfache Zeit ($4=2^2$), bei der 10fachen Menge die 100fache Zeit ($100=10^2$).

Machen wir uns das am einfachsten aller Sortierverfahren, dem sogenannten "Bubble Sort" klar. Bubble Sort erhält seinen Namen aufgrund der Eigenschaft, daß die größeren Elemente im Laufe des Sortierens, Luftblasen im Wasser gleich, nach oben steigen. Die größten steigen dabei am weitesten. Am Ende erhält man auf diese Weise ein sortiertes Feld.

Dazu werden je zwei nebeneinanderliegende Elemente verglichen. Ist das Element mit dem kleineren Index größer, werden die beiden vertauscht, das größere Element steigt wie eine Luftblase nach oben. Dann wird derselbe Schritt mit den nächsten beiden Elementen durchgeführt usw.. Dieses Verfahren endet erst dann, wenn das ganze Feld abgearbeitet ist. Nach diesem Durchgang befindet sich also das größte Element ganz oben, dort wo es hingehört.

Hat unser Feld n Elemente, so wurden in diesem Durchgang n-1 Vergleiche durchgeführt.

Da sich jetzt das größte Element auf seinem Platz befindet, braucht es nicht mehr berücksichtigt zu werden. Wir tun also so, als ob das zu sortierende Feld nur noch n-1 Elemente hätte und beginnen von vorn. Nach dem zweiten Durchgang ist auch das zweitgrößte Element auf seinem Platz. Wir haben dafür n-2, also insgesamt (n-1)+(n-2) Vergleiche gebraucht. Die Größe des zu sortierenden Feldes wird wieder um 1 erniedrigt und von vorn begonnen, bis schließlich alles sortiert ist. Insgesamt haben wir damit

$$(n-1)+(n-2)+(n-3)+...+2+2=n*(K-1)/2$$

Vergleiche durchführen müssen. Für n=10 benötigt man 45 Vergleiche, für n=100 (das 10fache) schon 4950, also mehr als das 100fache. Die Sortierzeit steigt quadratisch!
Bei allen folgenden Programmen bestehen folgende Vorgaben:

1. Das zu sortierende Feld befindet sich in a(anz).

2. Der maximale Feldindex ist in "anz" gespeichert. Damit ist die Anzahl der zu sortierenden Elemente anz+1, da auch unter Index 0 ein Element gespeichert ist.

3. Nach dem Sortiervorgang befindet sich das sortierte Feld wieder in a(anz).

4. Alle Variablen, die nicht durch "$" oder "!" gekennzeichnet sind, sind Integervariablen. Das geschieht aus Zeitgründen.

Hier folgt nun das Grundprogramm, das die nötigen Vorbereitungen trifft.

```
10 ' Sortieren
20 DEFINT a-z:DEFREAL t
30 RANDOMIZE TIME
40 aus=0:'Flag fuer Ausgabe des Sortiervorgangs
50 auga=0:'Kanalnummer fuer Ausgabe
60 anz=20:anz=anz-1:'Groesse des Feldes
70 DIM a(anz+1),b(anz),l(20),r(20)
80 FOR i=0 TO anz:a(i)=INT(100*RND):b(i)=a(i):NEXT:'zu sorti
erendes Feld erzeugen
90 READ j$:IF j$="#" THEN END ELSE j=VAL(j$):PRINT j,:IF aus
 THEN PRINT:'Programmnummern lesen
100 FOR i=0 TO anz:a(i)=b(i):NEXT:t=TIME
110 ON j GOSUB 200,300,400,500,600,700,800,900,1050,1200
120 PRINT#auga,(TIME-t)/300
130 GOTO 90:' Endekennzeichen
140 DATA 1,2,3,4,5,6,7,8,9,10:'Auszufuerhrende Programme
150 DATA #
160 REM SUB fuer Ausgabe des Sortiervorgangs
170 FOR n=0 TO anz:PRINT #auga,USING "##";a(n);:PRINT #auga,
"  ";:NEXT
180 PRINT#auga
190 RETURN
```

Nun zu Bubble Sort:

```
200 REM Bubblesort
210 FOR i=anz TO 1 STEP -1
220 FOR j=1 TO i
230 IF a(j-1)>a(j) THEN s=a(j-1):a(j-1)=a(j):a(j)=s
240 IF aus THEN GOSUB 170
250 NEXT j,i
260 RETURN
```

Die äußere Schleife verkleinert das zu sortierende Feld bei jedem Durchlauf um eins. In der inneren Schleife werden alle Elemente des Restfeldes paarweise miteinander verglichen und evtl. vertauscht.

Um richtige Zeitwerte zu erhalten, muß noch Zeile 240 aus dem Programm entfernt werden. Sie dient zur Ausgabe des Feldes nach jedem Vergleich. Die Ausgabe erfolgt, wenn Sie die Variable auf -1 setzen. So können Sie genau die Abläufe beim Sortieren verfolgen.

Beschäftigen Sie sich noch mit folgenden Erweiterungen:

Bauen Sie einen Zähler für die Anzahl der Vergleiche und einen zweiten für die Anzahl der Vertauschungen ein. Betrachten Sie die erhaltenen Werte im Zusammenhang mit der Zeit und mit den theoretischen Werten. Schätzen Sie ab, wie groß der Anteil der Vertauschungen am Gesamtzeitverbrauch ist (siehe auch Kapitel: Beschleunigen von BASIC-Programmen).

Bubble Sort ist ein schlechtes Sortierverfahren!

Doch gerade an diesem Beispiel läßt sich zeigen, durch welch kleine Änderungen ein Algorithmus oft bedeutend verbessert werden kann.

Eine Schwäche von Bubble Sort ist, daß es auch noch sortiert, wenn alles schon am richtigen Platz ist. Das können wir aber einfach feststellen. Findet eine Vertauschung statt, setzen wir ein Flag. Wenn das Flag am Ende eines Durchgangs noch nicht gesetzt ist, also keine Vertauschung stattgefunden hat, sind wir fertig.

Prüfen Sie anhand von Zeitmessungen, Vergleichs- und Vertauschungszählungen, wieviel diese Verbesserung durchschnittlich beträgt.

Die Idee mit dem Flag läßt sich noch weiter ausbauen. Anstatt das Flag einfach zu setzen, speichern wir den Index der Vertauschung ab.
Nach einem Durchlauf enthält das Flag dann den Index der letzten Vertauschung. Alle Elemente mit höherem Index sind dann bereits richtig sortiert, wir brauchen also nur noch bis zu diesem Index das Feld zu durchlaufen.

```
300 REM erweitertes Bubblesort
310 FOR i=anz TO 1 STEP -1
320 maxj=0
330 FOR j=1 TO i
340 IF a(j-1)>a(j) THEN s=a(j-1):a(j-1)=a(j):a(j)=s:maxj=j
350 IF aus THEN GOSUB 170
360 NEXT j
370 IF maxj=0 THEN RETURN ELSE i=maxj
380 NEXT i
390 RETURN
```

Stellen Sie auch hier wieder die Verbesserungen mit den erwähnten Methoden gegenüber den alten Verfahren fest.

Versuchen Sie noch eine "Bubbleversion" zu schreiben, in der abwechselnd "nach oben gebubbelt wird", d.h. daß das größte Element nach oben steigt, und "nach unten gebubbelt wird", also das kleinste Element nach unten fällt. Benutzen Sie zwei Zeiger,

die das noch nicht sortierte Feld jeweils begrenzen. Untersuchen Sie dieses Verfahren auf seine Leistungsfähigkeit.

Damit genug zu Bubble Sort. Nun folgen noch einige einfache Sortieralgorithmen, die sich jedoch teilweise beträchtlich unterscheiden. Führen Sie vergleichende Tests der einzelnen Algorithmen durch.

Um am Anfang das Verfahren besser verstehen zu können, ist in jedem Programm wieder die Ausgabeoption vorhanden.
Für die Durchführung der Geschwindigkeitsuntersuchungen sollten Sie jeweils diese Zeile löschen.

Maxsort

Maxsort durchsucht jeweils das noch nicht sortierte Feld nach dem größten Element. Wird ein Element, das größer ist, also das am Feldende gespeicherte, gefunden, werden die beiden vertauscht. Nach jedem Durchgang wird das Restfeld um eins verkleinert.

```
400 REM Maxsort
410 FOR i=anz TO 1 STEP -1
420 FOR j=0 TO i-1
430 IF a(j)>a(i) THEN s=a(j):a(j)=a(i):a(i)=s
440 IF aus THEN GOSUB 170
450 NEXT j,i
460 RETURN
```

Straightsort oder das Sortieren nach Auswahl

Dieses Verfahren ist prinzipiell ein verbessertes Maxsortverfahren. Die Verbesserung dabei ist, daß nicht automatisch vertauscht wird, wenn ein größeres Element gefunden wurde. Statt-

dessen wird erst im Restfeld das Maximum gefunden und dann vertauscht. Die Anzahl der Vergleiche ist gleich, die der Vertauschungen geringer.

```
500 REM Straight
510 FOR i=anz TO 1 STEP -1:m=0
520 FOR j=0 TO i
530 IF a(j)>m THEN m=a(j):k=j
540 NEXT
550 a=a(i):a(i)=a(k):a(k)=a
560 IF aus THEN GOSUB 170
570 NEXT
580 RETURN
```

Insertsort oder Sortieren durch Einfügen

Insertsort beruht auf einer Methode, die jeder von uns kennt. Beim Kartenspielen sortieren wir jede neue Karte ein, in dem wir sie zwischen die bereits vorhandenen einfügen. Genauso funktioniert das Insertsortverfahren:
Die neue Karte wird als größte Karte vorläufig einsortiert. Dann wird sie mit ihren Nachbarn verglichen und, wenn sie kleiner ist, vertauscht. Es wird also mit 2 Elementen angefangen, ein immer größeres sortiertes Feld aufzubauen.

```
600 REM Insertsort
610 FOR i=1 TO anz
620 FOR j=i TO 1 STEP -1
630 IF a(j)>=a(j-1) THEN 670
640 s=a(j-1):a(j-1)=a(j):a(j)=s
650 IF aus THEN GOSUB 170
660 NEXT j
670 NEXT i
680 RETURN
```

Da eine Verschiebung schneller ist als eine Vertauschung, kann Insertsort ähnlich wie Maxsort verbessert werden:
Zuerst wird die Stelle ermittelt, an die das neu einzusortierende Element gehört. Dann werden alle größeren Elemente verschoben und schließlich das Element auf seinen Platz gesetzt.

```
700 REM verbessertes Insertsort
710 FOR i=1 TO anz
720 a=a(i)
730 FOR j=i-1 TO 0 STEP -1
740 IF a<a(j) THEN NEXT j
750 FOR k=i TO j+2 STEP -1:a(k)=a(k-1):NEXT k
760 a(j+1)=a
770 IF aus THEN GOSUB 170
780 NEXT i
790 RETURN
```

Doch auch diese Version kann noch verändert werden. Das Verschieben des Feldes kann bei kleinen Feldern noch schneller ausgeführt werden als bei der letzten Version durch eine Extraschleife.

```
800 REM 3. Version Insertsort
810 FOR i=1 TO anz
820 a=a(i):j=i-1
830 IF a>=a(j) THEN 860
840 a(j+1)=a(j):j=j-1
850 IF j>=0 THEN 830
860 a(j+1)=a
870 IF aus THEN GOSUB 170
880 NEXT
890 RETURN
```

Binary Sort = Binary Surch mit Insert Sort

Beim letzten Programm wurde ein Großteil der Zeit darauf verwandt, den Platz eines neuen Elementes in einer schon sortierten Liste zu finden. Der Zeitaufwand für diese Suche kann mit Hilfe von Binary Surch beträchtlich verringert werden. Beim einfachen Insertsort wurde schrittweise mit jedem Element verglichen. Binary Sort dagegen ist ein "intelligentes" Suchverfahren.
Nehmen Sie an, Sie sollen durch Fragen eine Zahl zwischen 1 und 128 erfahren. Nach der "Insertsort Methode" würde man bei diesem Beispiel so fragen:

Ist die Zahl 128?
Nein!
Ist die Zahl 127?
Nein!
Ist die Zahl 126?
.
.
.

Effektiver währe es zu fragen:

Ist die Zahl größer, kleiner oder gleich 64?
Größer als 64!

Die nächste Frage lautet dann:

Ist die Zahl größer, kleiner oder gleich 96?
Kleiner als 96!

Dann:

Ist die Zahl größer, kleiner oder gleich 84?
Kleiner als 84!

Das Prinzip dieser Abfragen ist gewissermaßen ein "Umzingeln" bzw. "Einkreisen" der gesuchten Zahl. Durch das systematische Fragen wird die Anzahl der möglichen Antworten begrenzt.

Eine ähnliche Methode des Fragens wird bei einer bekannten Fernsehsendung oft beispielhaft vorgeführt.

> Frage: "Ist der Kandidat weiblichen oder männlichen Geschlechts?"
>
> Mögliche Antworten im Normalfall: männlich bzw. weiblich.

Hierbei hat man also nur zwei mögliche Antworten. Wichtiger aber ist, daß der zu erfragende Möglichkeitsbereich sich durch die Anwort auf diese Frage stark reduziert.

Das Intervall, in dem sich die Zahl befinden kann, wird durch jede Frage halbiert. Die zu findende Zahl sei 30. Die Intervalle nach jeder Frage sind dann:

Frage	Antwort	Intervall
>, < oder =64	<64	1-63
>, < oder =32	<32	1-31
>, < oder =16	>16	17-31
>, < oder =24	>24	25-31
>, < oder =28	>28	
>, < oder =30	=30	gefunden!!!

6 Fragen (Vergleiche) waren nötig, um die Zahl zu finden. Nach der alten Methode wären 98 Vergleiche notwendig gewesen. Die maximale Zahl der nötigen Vergleiche bei n Elementen mit Binary Surch ist proportional zu ln_2 2 (ln_2=Logarithmus zur Basis 2).

Der Name des Verfahrens ergibt sich aus der Tatsache, daß das in Frage kommende Intervall jedesmal halbiert wird. Aus diesem Grund wird auch der Logarithmus zur Basis 2 genommen. Die Ersparnisse sind bei großen Listen enorm.
Die Position eines Elementes, das in ein Feld mit 60 Millionen Elementen (Einwohnerzahl der BRD) einsortiert werden soll, kann mit 26 Vergleichen ermittelt werden. Schauen Sie sich das Binary Sort Programm genau an, und simulieren Sie es im

Zweifelsfalle einmal per Hand (spielen Sie Computer), damit Ihnen die Arbeitsweise des Algorithmus klar wird.

Aufgrund der etwas komplizierten Struktur des Programms, macht sich der Geschwindigkeitsvorteil gegen über Insertsort erst bei Feldgrößen von ca. 100 und mehr Elementen bemerkbar.

```
900 REM Binary Sort
910 FOR i=1 TO anz
920 IF a(i)>a(i-1) THEN 1000
930 l=-1:r=i+1
940 h=INT((l+r)/2)
950 IF a(i)>a(h) THEN l=h ELSE r=h
960 IF r>l+1 THEN 940
970 a=a(i):FOR j=i TO r+1 STEP -1:a(j)=a(j-1):NEXT
980 a(r)=a
990 IF aus THEN GOSUB 170
1000 NEXT
1010 RETURN
```

Shellsort

Der Shellsort Algorithmus trägt den Namen seines Erfinders. D. L. Shell, der den Algorithmus in den 50er Jahren entwickelte. Interessanterweise handelt es sich bei diesem Verfahren prinzipiell um Bubblesort mit dem leichten, aber wichtigen Unterschied, daß das zu sortierende Feld in kleinere Unterfelder aufgespalten wird, die sortiert werden und nach und nach schließlich das gesamte Feld sortieren.

Die Idee des Verfahrens ist, daß das Feld zunächst grob sortiert wird. Nehmen wir als Beispiel ein Feld mit 16 Elementen. Für die erste Grobsortierung werden jeweils das 1te und das 9te, das 2te und 10te, 3te und 11te usw. Element verglichen und bei Bedarf vertauscht. Damit ist die erste Grobsortierung erreicht.

Bei diesem Durchlauf wurden Elemente mit dem Abstand 8 verglichen.

Beim nächsten Durchlauf wird der Vergleichsabstand auf 4 halbiert. Nun werden die Unterfelder sortiert, die sich aus den Elementen mit dem Abstand vier ergeben, also das Unterfeld aus dem 1ten, 5ten, 9ten und 13ten Element, das aus dem 2ten, 6ten, 10ten und 14ten Element usw.. Diese neuen Unterfelder werden jetzt sortiert, wobei prinzipiell das Insertsortverfahren angewendet wird. Es wird also das 1te mit dem 5ten verglichen und evtl. vertauscht, dann das 9te in das 1te und 5te einsortiert, und schließlich das 13te zwischen 1tem, 5tem und 9tem einsortiert. Genauso wird mit den anderen Unterfeldern verfahren. Nun wird wieder der Vergleichsabstand halbiert, und der Vorgang beginnt von neuem.

Der Vergleichsabstand wird solange halbiert, bis er eins beträgt. Dann wird der letzte Sortierdurchlauf mit dem gesamten Feld durchgeführt.

Das Shellsortverfahren ist ein sehr interessanter Algorithmus, der allen bisher besprochenen weit überlegen ist, wenn Felder mit mehr als 20 Elementen sortiert werden.

```
1050 REM Shellsort
1060 FOR m=anz-1 TO 1 STEP -1
1070 m=INT((m+1)/2)
1080 FOR j=0 TO anz-m
1090 i=j
1100 IF a(i)<=a(i+m) THEN 1150
1110 s=a(i):a(i)=a(i+m):a(i+m)=s
1120 IF aus THEN GOSUB 170
1130 i=i-m
1140 IF i>=0 THEN 1100 ELSE 1150
1150 NEXT j,m
1160 RETURN
```

Quicksort

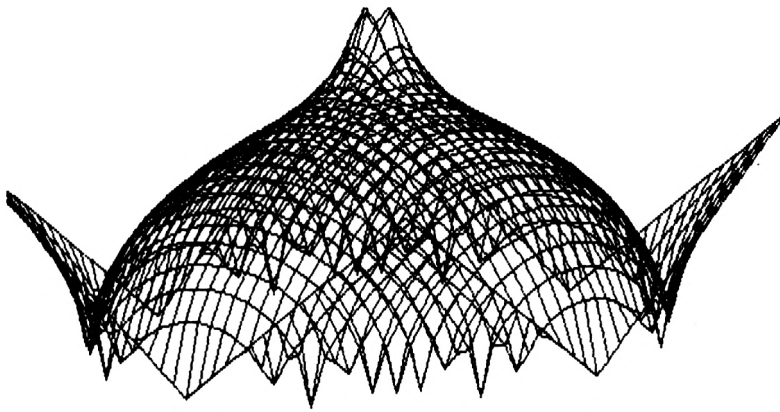
Eine der besten bekannten Sortierverfahren ist Quicksort. Für Felder mit mehr als 50 Elementen ist es den bisher vorgestellten Verfahren überlegen. die Idee für das Quicksortverfahren ist folgende:

Das zu sortierende Feld wird in zwei Hälften geteilt. Das Element zwischen beiden Hälften dient als Vergleichselement. Nun werden aus der unteren Hälfte alle Elemente, die größer sind als das Vergleichselement, vertauscht. Unter Umständen wird auch mit dem Vergleichselement selbst vertauscht. Diese erste Grobsortierung ist ähnlich einem erweiterten Shellsortverfahren.

Nach dem ersten Durchlauf wird nun mit jeder der beiden Hälften das Verfahren von neuem begonnen. D.h., daß das Verfahren rekursiv ist, also sich selbst wieder aufruft. Um diese rekursive Struktur auch im BASIC zu ermöglichen, müssen die wichtigsten Variablen vor dem Wiederaufruf abgespeichert werden, da sie sonst verloren gehen würden. Dazu dienen die Felder l(sp) und r(sp).

```
1200 REM Quicksort
1210 sp=0:l(sp)=0:r(sp)=anz
1220 l=l(sp):r=r(sp):sp=sp-1
1230 i=l:j=r:vv=a(INT((l+r)/2))
1240 WHILE a(i)<vv :i=i+1:WEND
1250 WHILE a(j)>vv :j=j-1:WEND
1260 IF i>j THEN 1300
1270 s=a(i):a(i)=a(j):a(j)=s:IF aus THEN GOSUB 170
1280 i=i+1:j=j-1
1290 IF i<=j THEN 1240
1300 IF i<r THEN sp=sp+1:l(sp)=i:r(sp)=r
1310 r=j:IF l<r THEN 1230
1320 IF sp>=0 THEN 1220
1330 RETURN
```

2. 3-D-Grafik



```
40 DEF FNf(x)=SQR(ABS((16-x*x-z*z)))+1/SQR(x*x+z*z+0.1)
50 ap=34
80 DATA -4,4,-2,6,-4,4
```

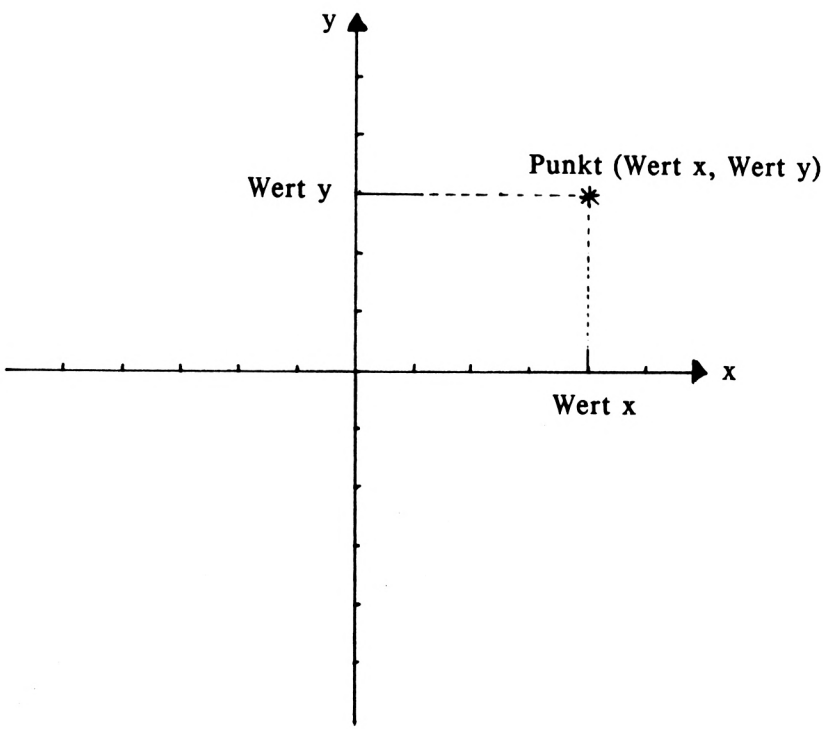
Eine der schönsten und interessantesten Anwendungen der Computergrafik bzw. des Computers überhaupt ist das Erstellen von dreidimensionalen Bildern. Die komplette Theorie der 3-D-Grafik ist sehr komplex und in den letzten Jahren im Zuge der Entwicklung von CAD Systemen (Computer Aided Design) weit vorangetrieben worden. Mit Hilfe der letzten Rechnergenerationen ist es sogar möglich, Filme, z.B besondere Sience Fiction-Szenen, vollständig von Computern erzeugen zu lassen.

In solchen Dimensionen können wir uns hier natÜrlich nicht bewegen. Die Schneider Rechner bieten allerdings in ihrer Klasse unerreichte Grafikfähigkeiten. Die im MODE 2 erreichte Auflösung von 640*200 Punkten bietet ideale Voraussetzungen für die Darstellung dreidimensionaler Funktionen.

Beschäftigen wir uns zunächst ein wenig mit der Theorie, d.h., mit dem Begriff der Funktion.

Eine Funktion ist eine Abbildung. Dabei wird, zumindest im Zweidimensionalen, jeder Wert aus einer bestimmten Zahlenmenge (dem Definitionsbereich) auf einen - und nur einen - Wert aus einer zweiten Zahlenmenge (dem Wertebereich) abgebildet, den man als Funktionswert bezeichnet. Den Zusammenhang dieser beiden Mengen bezeichnet man als die Funktion.

Man kann also eine Funktion als eine Wertetabelle auffassen, in der jedem Wert der Definitionsmenge ein Funktionswert zugeordnet ist. Die so erhaltenen Zahlenpaare kann man grafisch darstellen, indem man die Werte als Abschnitte auf zwei senkrecht zueinanderstehenden Achsen deutet, und den Schnittpunkt der horizontalen bzw. der vertikalen Linie durch die auf den Achsen abgetragenen Strecke als Punkt einzeichnet. Die beiden Wertepaare bezeichnet man dann als die Koordinaten des Punktes. Betrachten wir ein Beispiel folgender Funktion:



Jede beliebigen Kommazahl wird auf den Wert, den man erhält, indem man die Kommastellen der Zahl wegläßt, abgebildet.

Einige Wertepaare dieser Funktionen sind:

X	Y
1,5	1
3,7593	3
1000,379	1000
2	2
-1,5	1

Wir wollen diese Funktionen durch den Rechner zeichnen lassen.

Zunächst stellt sich damit die Aufgabe, die Abbildungsvorschrift in eine für den Recher "verständliche", d.h. mathematische Form umzuwandeln. Bei unserem Beispiel ist dies einfach. Die obengenannte Funktion wird durch die INT-Funktion des Rechners dargestellt.

Die Variable für die Werte des Definitionsbereiches ist X. Die Variable für die des Wertebereiches jedoch Y. Die Funktion soll für den Bereich von -5 bis 15 dargestellt werden. Ein erster Versuch könnte so aussehen:

```
10 MODE 2
20 FOR X=-5 TO 15
30 Y=INT(X)
40 PLOT X,4
50 NEXT
```

Der gewünschte Effekt kommt jedoch nicht zustande. Wir müssen unser Programm noch an das Bildschirmformat des Rechners anpassen. Die Breite des Bildschirmes (=X-Richtung) beträgt 640 Punkte. Die Höhe (=Y-Richtung) 200 Punkte, die jedoch mit den Koordinaten von 0 bis 400 bezeichnet werden und erst intern halbiert werden.

Das eben erhaltene Bild ist eine Miniaturdarstellung des gewünschten Bildes. Es muß noch entsprechend in die X- und Y-Richtung gestreckt werden.

Es werden also zwei Maßstabsfaktoren eingeführt, die die realen Werte von x und y so stauchen oder strecken, daß sie nach Möglichkeit den gesamten Bildschirm ausfüllen.

Die Funktion soll im Bereich von XOBG=15 bis XUNTG=-5 dargestellt werden. Der gesamte darzustellende Bereich hat dann die Größe XOBG-XUNTG=15-(-5)=20. Da 640 Punkte zur Verfügung stehen, muß die 20 auf 640 gestreckt werden, d.h., daß die X-Werte mit 640/20=32 multipliziert werden müssen.

```
10 MODE 2
20 XUNTG=-5:XOBG=15
30 MASX=640/(XOBG-XUNTG)
40 FOR X=XUNTG TO XOBG STEP 1/MASX
50 Y=INT(X)
60 PLOT X*MASX,Y
70 NEXT X
```

Die STEP-Anweisung bewirkt, daß auch für jeden der 640 zeichenbaren X-Werte ein Punkt gezeichnet wird.

Das erhaltene Bild ist noch ein wenig "platt", weil wir die Y-Achsenwerte noch nicht gestreckt haben.

Wie bei der X-Achse müssen wir beim Strecken der Y-Achse wissen, zwischen welcher Ober- und Untergrenze sie dargestellt werden soll. Dies ist oft nicht von vornherein feststellbar, wenn man eine unbekannte Funktion zeichnen läßt. Entweder man probiert ein bißchen aus, oder man macht zuerst einen Testlauf, bei dem der maximale und der minimale Y-Wert ermittelt werden. Dann nimmt man diese Werte als Grenzen und zeichnet jetzt erst den Graphen.

Aufgrund der Eigenschaften unserer Funktion ist jedoch einfach zu erkennen, daß Y sich zwischen -5 und 15 bewegen wird.

Fügen Sie also noch folgende Zeilen hinzu bzw. ändern Sie die alten.

```
21 YUNTG=-5:YOBG=15
31 MASY=400/(YOBG-YUNTG)
```

Zeile 60 ändern Sie zu PLOT X*MASX,Y*MASY

Trotz der Einführung der Maßstabsfaktoren (MAS) haben wir noch nicht erreicht, daß der gesamte Bildschirm von der Funktion "benutzt" wird. Es ist notwendig, anhand der X- und Y-Ober- bzw. Untergrenzen den Bildschirmnullpunkt entsprechend zu verlegen. Normalerweise liegt der Bildschirmnullpunkt in der unteren linken Ecke. Prinzipiell könnten wir den Nullpunkt (engl. Origin) mit dem dafür vorgesehenen ORIGIN-Befehl verlegen. In unserem Fall wäre dies mit Zeile

```
35 ORIGIN 160,100
```

möglich. Dieses Verfahren, das natürlich das einfachste ist, funktioniert allerdings nur unter der Voraussetzung, daß sich der Nullpunkt auch tatsächlich auf bzw. zumindest in der Nähe des realen Bildschirmes befindet. Sollte aber z.B. nur der Abschnitt der Funktion von 2000 bis 2040 dargestellt werden, versagt diese Methode, da ein ORIGIN -32000,-20000 einen Overflow ergibt. Die entsprechenden Berechnungen müssen also vom Programm durchgeführt werden. Wir müssen also den Bildschirm um den Wert von XUNTG in X-Richtung und von ZUNTG in Y-Richtung "verschieben". Die folgenden Änderungen der Zeile 60 bewirken dies (löschen Sie wieder Zeile 35 und geben Sie einmal ORIGIN 0,0 ein):

```
60 PLOT (X-XUNT)*MASX,(Y-YUNTG)*MASY
```

Fügen wir noch folgende Zeilen hinzu, so erhalten wir außerdem noch die Koordinatenachsen auf dem Bildschirm, falls diese auf ihm liegen.

```
35 IF XUNTG*XOBG>0 THEN 37
36 MOVE (X-XUNTG)*MASX,0:DRAW (X-XUNTG)*MASX,400
37 IF YUNTG*YOBG>0 THEN 40
38 MOVE 0,(Y-YUNTG)*MASY:DRAW 640,(Y-YUNTG)*MASY
```

Die eben betrachtete Funktion INT(X) hat eine "unschöne" Eigenschaft, die in der Mathematik als "Unstetigkeit" bezeichnet wird. Das bedeutet vereinfacht gesagt, daß im Verlauf des Graphen Sprünge sind oder andersherum gesagt, er nicht glatt verläuft. Im folgenden werden wir nur noch stetige Funktionen betrachten, da sie einige für die Programmierung hervorragende Eigenschaften in sich vereinigen. Eine der einfachsten Funktionen ist die, die den X-Wert auf sich selbst abbildet. Wie man leicht nachprüfen kann, ergibt diese Funktion als Bild eine Gerade, die eine Diagonale zu den Achsen bildet. Als Beispiel für eine einfache stetige Funktion betrachten wir nun eine quadratische, deren einfachste Form $y=x^2$ ist.
Zunächst definieren wir die Funktion der Einfachheit halber mit dem Befehl DEF FN:

```
15 DEF FNY(X)=X^2
```

Änderung:

```
50 Y=FNY(X)
```

Da eine stetige Funktion immer recht "gemütlich", d.h., mit nur langsamen Veränderungen in einem ausreichend kurzen Stück der X-Achse verläuft, ist es möglich, auf das Zeichnen jedes einzelnen Punktes zu verzichten. Vielmehr können in gewissen Abständen Punkte gezeichnet und diese dann durch Geraden verbunden werden. Je nach Abstand der zu verbindenden Punkte, folgt dadurch ein Zeitgewinn vom 10fachen zum 50fachen. Gerade diese Tatsache ist es, die das am Ende dieses Kapitels stehende 3-D-Netzgrafikprogramm den herkömmlichen

so überlegen macht, die alle Punkte berechnen. Folgende Änderungen sind daher im Programm notwendig:

```
16 PUAB=10:REM Punktabstand
```

Zeile 40 in Zeile 42 umnumerieren.

```
40 MOVE 0,(FNY(XUNTG)-YUNTG)*MASY
42 FOR X=XUNTG TO XOBG STEP 1/MASX*PUAB
```

Änderung von Zeile 60 zu:

```
DRAW (X-XUNTG)*MASX,(Y-YUNTG)+MASY
```

Außerdem sollen die Y-Grenzen wegen der nun neuen Funktion zu -10 und 250 (o.ä.) geändert werden.

Probieren Sie verschiedene andere quadratische Funktionen aus:

Funktion	XUNTG	XOBG	YUNTG	YOBG
x^2-50	-10	15	-150	200
x^2+20	-5	10	-10	90
$(X-5)^2$	-5	15	-10	100
$(X+5)^2$	-10	10	-10	250
$(X-5)^2$-50	-5	15	-60	60

Wie wir gesehen haben, ist die Darstellung zweidimensionaler Funktionen relativ einfach möglich. Der Graph einer zweidimensionalen Funktion liegt in einer Ebene und ist daher auch leicht auf die Bildschirmebene zu übertragen. Bei einer dreidimensionalen Funktion ist diese direkte Übertragung nicht möglich, da auf der Bildschirmebene nur zwei Dimensionen zur Verfügung stehen.

Bei einer dreidimensionalen Abbildung wird jedem Wertepaar des Definitionsbereiches ein Wert des Wertebereiches zugeordnet. Man könnte sich z.B. vorstellen, daß die Definitionsebene ein Tisch ist, dem zu jedem Punkt des Tisches eine bestimmte Höhe von der Tischplatte aus zugeordnet wird. Um ein wirklich dreidimensionales Bild der Funktion zu erhalten, müßte aus einem

formbaren Material, wie z.B. Knetgummi, die Fläche, die aus allen Punkten gebildet wird, nachgebaut werden.

Die Aufgabe für den Programmierer ist es, diese dreidimensionale Figur in einer Ebene, also mit zwei Dimensionen, abzubilden. Eine wirklichkeitsnahe Abbildung müßte dabei darauf achten, daß weiter entfernt liegende Strecken kleiner erscheinen und umgekehrt, dem Betrachter nahe liegende größer abgebildet werden. Durch diesen Effekt würde der Eindruck einer wirklichen Perspektive im Bild entstehen.

Für unsere Zwecke, also der Darstellung von Funktionen, können wir jedoch diesen Effekt vernachlässigen. Stellen Sie sich vor, Sie würden zunächst in unserem Knetgummimodell, paralel zur vorderen Tischkante, mit einem Messer in jeweils gleichen Abständen Linien in die Oberfläche des Knetgummis hineinritzen. Aufgabe ist es, die erhaltenen Linien nun mit dem Computer zu zeichnen. Die erste Linie, die über der vorderen Tischkante liegt, haben wir bereits mit dem alten Programm gezeichnet.

Betrachten wir diese Funktion:

```
DEF FNY(X)=X^2+Z^2
```

wobei Z den Werten auf der dritten Achse entspricht. In unserem Modell ist sie die seitlich verlaufende Tischkante.

Nehmen wir an, daß die zweite eingeritzte Linie durch den Wert -1 auf der Z-Achse verläuft oder anders ausgedrückt, den Abstand 1 nach hinten von der ersten Linie hat. Diese Linie wird nun gezeichnet, indem Z auf -1 gesetzt wird und dann die normale Schleife zum Zeichnen durchlaufen wird. Danach wird Z auf den Wert der nächsten Schnittlinie gesetzt usw.

Zuerst numerieren Sie Zeile 40 in 41 um, die Zeile 15 enthält obige Funktionsdefinition.

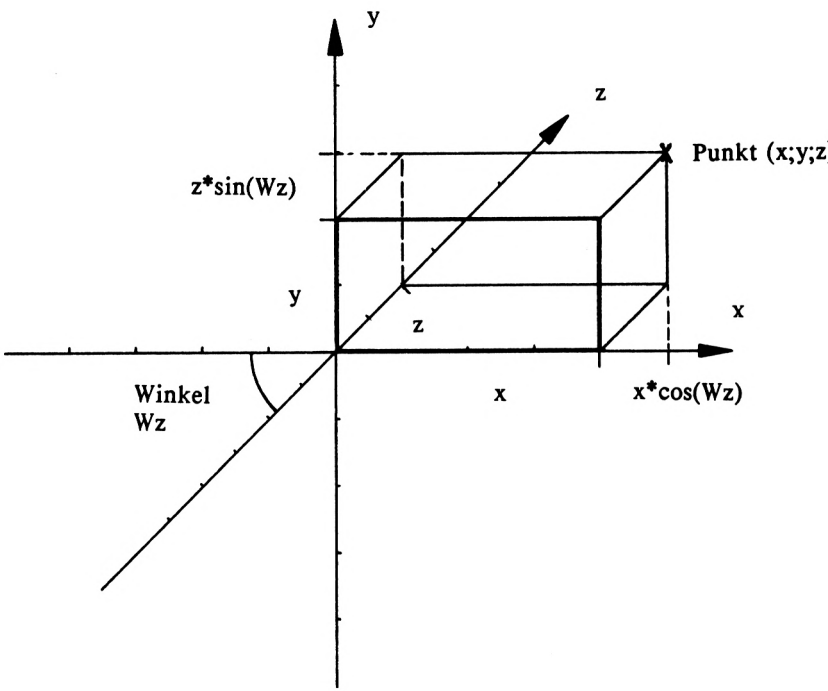
```
22 ZUNTG=-10:ZOBG=0
40 FOR Z=ZOBG TO ZUNTG STEP-1
```

Ändern Sie Zeile 41 zu:

```
41 MOVE 0,(FNY(XUNTG)-YUNTG)*MASY-Z*MASZ*MAZY
70 NEXT
```

Das so erhaltene Bild entspricht unserem Modell, wenn wir es frontal mit Augenhöhe=Tischplatte ansehen würden. Wir wollen jedoch einen beliebigen Blickwinkel wählen können.

Bei der Standardperspektive betrachtet man unser Modell von rechts oben. Bei dieser Betrachtung verläuft die Z-Achse fast diagonal nach hinten rechts. Ein Punkt, der weit hinten auf der Z-Achse liegt, würde in der zweidimensionalen Darstellung in der rechten oberen Bildschirmecke abgebildet sein. Das bedeutet, daß die in der Realität weiter hinten gelegenen Punkte in einer zweidimensionalen Abbildung auf dem Bildschirm auf der X- bzw. Y-Bildschirmkoordinatenleiste nach rechts bzw. oben verschoben erscheinen. Je weiter der Punkt auf der Z-Achse entfernt ist, desto weiter ist auch die Verschiebung auf der Bildschirmkoordinatenachse.



Im Programm berücksichtigen wir dies so:

Vorher muß noch Zeile 41 zu

```
41 MOVE -Z*MASZ*MAZX,(FNY(XUNTG)-YUNTG)*MASY-Z*MASZ*MASY
```

geändert werden.

```
6 DEG
25 WZ=45:'Winkel der Z-Achse zur X-Achse
26 MAZX=COS(WZ):MAZY=SIN(WZ)
32 MASZ=200/(ZOBG-ZUNTG)
60 DRAW (X-XUNTG)*MASX-Z*MASZ**MASX,(Y-YUNTG)*MASY-Z*MASZ*MAZY
17 LINAB=10
```

Änderung:

```
40 FOR Z=ZOBG TO ZUNTG STEP 1/MASZ*LINAB
```

Ändern wir noch einige Zeilen für eine neue Funktion und numerieren das Programm neu, so erhalten wir:

```
10 DEG
20 MODE 2
30 DEF FNy(x)=x/SQR((1-x*x)^2+(2*z*x/300)^2+0.001)
40 puab=20: REM Punktabstand
50 linab=20
60 xuntg=-2:xobg=2
70 yuntg=-7:yobg=7
80 zuntg=-300:zobg=300
90 wz=45:'Winkel der Z-Achse zur X-Achse
100 mazx=COS(wz):mazy=SIN(wz)
110 masx=640/(xobg-xuntg)
120 masy=400/(yobg-yuntg)
130 masz=400/(zobg-zuntg)
140 IF xuntg*xobg>0 THEN 160
150 MOVE (-xuntg)*masx,0:DRAW (-xuntg)*masx,400
160 IF yuntg*yobg>0 THEN 180
170 MOVE 0,(y-yuntg)*masy:DRAW 640,(y-yuntg)*masy
180 FOR z=zobg TO zuntg STEP-1/masz*linab
190 MOVE -z*masz*mazx,(FNy(xuntg)-yuntg)*masy-z*masz*mazy
200 FOR x=xuntg TO xobg STEP 1/masx*puab
210 y=FNy(x)
220 DRAW (x-xuntg)*masx-z*masz*mazx,(y-yuntg)*masy-z*masz*mazy
230 NEXT x,z
```

Dieses Programm erzeugt nun also 3-D-Liniengrafiken. Oftmals liefert diese Methode jedoch noch keine ausreichenden Bilder. Stellen Sie sich also noch einmal unser Knetgummimodell vor und ritzen Sie noch eine Reihe von Linien zusätzlich parallel zur seitlichen Tischkante (Z-Achse) ein. Damit ist die Oberfläche der Funktion mit einem Netz von Linien überzogen.

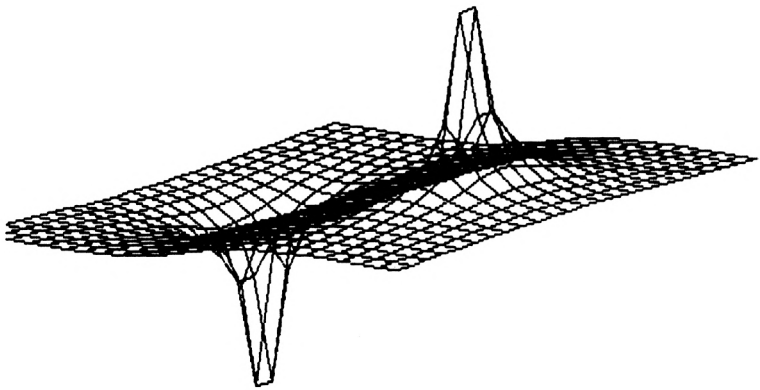
Um dies auch zweidimensional abzubilden, könnte man einfach das Programm an das alte in fast gleicher Form anhängen, mit dem Unterschied, daß die Z-Schleife mit der X-Schleife vertauscht wird. (Probieren Sie es aus!)

Dabei würde aber jeder Netzpunkt zweimal berechnet werden, was natürlich auch entsprechend mehr Zeit benötigt. Das folgende Programm erledigt beides auf einmal. Dazu werden die als letztes gezeichneten Netzpunkte abgespeichert und dann beim Zeichnen der nächsten Linien mit den dazugehörigen Punkten verbunden. Daraufhin werden die gerade berechneten Punkte anstelle der alten gespeichert. Auf diese Weise wird das komplette Netz gezeichnet. Im folgenden Abschnitt bringen wir das Programm und einige damit erzeugte Bilder.

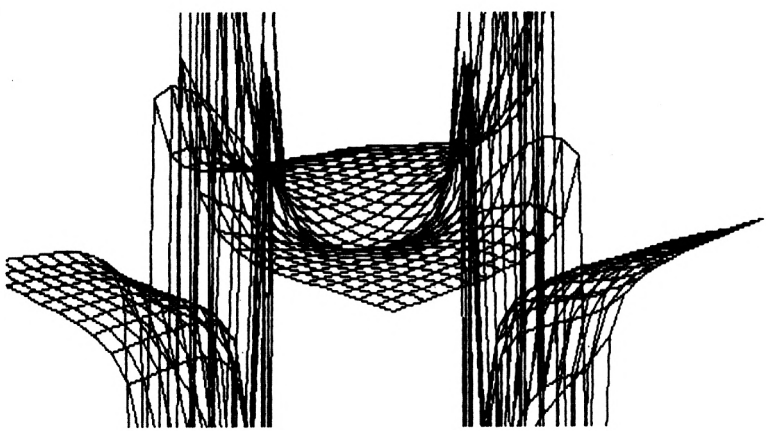
```
10 MEMORY &9FFF
20 MODE 2
40 DEF FNf(x)=x/SQR((1-x*x)^2+(2*z*x/300)^2+0.001)
50 ap=15
60 DIM x(ap+1),y(ap+1)
70 READ xa,xe,ya,ye,za,ze
80 DATA -2,2,-7,7,-300,300
90 mx=400/(xe-xa)
100 my=400/(ye-ya)
110 mz=400/(ze-za)
120 DEG
130 al=42-22/2
140 as=22
150 zx=COS(al)^2
160 xx=COS(as)^2
170 zy=SIN(al)^2
180 xy=SIN(as)^2
190 j=0
200 FOR z=ze TO za STEP -(ze-za)/ap
210 i=0
220 FOR x=xe TO xa STEP -(xe-xa)/ap
230 y=FNf(x)
240 xk=120+xx*mx*(x-xa)-zx*mz*(z)
250 yk=my*(y-ya)-zy*mz*(z)-xy*mx*(x-xa)
260 IF i=0 THEN 280
270 MOVE xk,yk:DRAW x(i),y(i)
280 i=i+1
290 IF j=0 THEN 310
300 MOVE xk,yk:DRAW x(i),y(i)
310 x(i)=xk
320 y(i)=yk
330 NEXT x
340 j=j+1
350 NEXT z
360 IF INKEY$="" THEN 360
```

Variablenliste:

As:	Winkel zwischen Waagerechte und X-Achse
AL:	Winkel zwischen Waagerechte und Z-Achse
AP:	Anzahl der Punkte auf der Netzlinie
I:	Zähler
J:	Zähler
MZ:	Maßstab Z
MY:	Maßstab Y
MX:	Maßstab X
XK:	X-Koordinaten Bildschirm
XY:	Projektionsfaktor X auf Y-Achse
XX:	Projektionsfaktor X auf X-Achse
XE:	X-Ende
XA:	X-Anfang
X:	X-Wert
YK:	Y-Koordinate
YE:	Y-Ende
YA:	Y-Anfang
Y:	Y-Funktionswert
ZY:	Projektionsfaktor Z auf Y-Achse
ZX:	Projektionsfaktor Z auf X-Achse
ZE:	Z-Ende
ZA:	Z-Anfang
Z:	Z-Wert



```
40 DEF FNf(x)=x/SQR((1-x*x)^2+(2*z*x/300)^2+0.001)
50 ap=23
60 DIM x(ap+1),y(ap+1)
70 READ xa,xe,ya,ye,za,ze
80 DATA -2,2,-9,7,-300,300
```



```
40 DEF FNf(x)=COS(4*x*x+z*z)*((x*x+z*z)/(x*z+1.1))+ABS(SIN(1
0*SQR(x*x)))
50 ap=24
80 DATA -4,4,-12,12,-4,4
```

3. BASIC-Programme benutzerfreundlich gestatlet

3.1 Menuegenerator

In vielen größeren Programmen kann dasselbe vom Benutzer mit Hilfe eines Menues gesteuert werden. In letzter Zeit zeichnet sich der Trend zu benutzerfreundlichen Programmen ab. Mit Sicherheit wird dieser Trend in Zukunft beibehalten, wenn nicht sogar verstärkt werden.

Eine komfortable Menuesteuerung ist ein wichtiger Beitrag zur Benutzerfreundlichkeit. Anhand eines Menues kann der Anwender, wie im Restaurant auf der Speisekarte (menue: engl. Karte) auswählen, was er gerne essen möchte. In unserem Fall also das Auswählen einer Funktion des Programms. Da die Auswahl bei einer Karte mit zu vielen verschiedenen Gerichten äußerst schwerfallen kann, auf jeden Fall aber viel Zeit benötigt, sind gute Menuekarten übersichtlich und inhaltlich gegliedert.

Ein Hauptmenue führt ins Untermenue, welches eventuell noch weiter aufgegliedert ist, bis man schließlich an dem gewünschten Punkt angelangt ist. Außerdem sollte ein gutes Menue alle nicht eindeutigen Bedienungsfunktionen mit anzeigen, z.B., wie das letzte oder nächste Menue erreicht werden kann.

Da das Programmieren eines Menues immer notwendig sein wird, haben wir einen Menuegenerator geschrieben, der eine erhebliche Zeitersparnis mit sich bringt.

Mit einem minimalen Aufwand, nämlich durch das Ändern der DATA Zeilen, in denen die Menuepunkte stehen, kann das Programm universell eingesetzt werden.

Das Programm selbst ist natürlich ohne das dazugehörige Hauptprogramm, welches vom Menue aus gesteuert wird, wertlos. Das Hauptprogramm ist jeweils Ihr eigenes, mit dem Menue auszustattendes Programm.

Die Technik des Menuegenerators zeigt eine sehr interessante Einsatzmöglichkeit der Feldvariablen des BASIC. Ohne diese wäre es gar nicht möglich gewesen, ein solches Programm zu schreiben. Das Ziel war es, jedes Menue durch eine einzige Routine anzeigen zu lassen. Nachdem der Benutzer seine Wahl getroffen hat, soll die Routine automatisch den Weg ins neue Menue finden.

Alle möglichen Menuepunkte sind dazu in einem Feld (m$) hintereinander gespeichert. Um innerhalb dieses Feldes die einzelnen Menuepunkte in der notwendigen Reihenfolge festzuhalten, gibt es ein zweites Feld (md für Menue-DATA), in dem zu jedem Menue folgende Informationen gespeichert sind:

- Die Nummer des nächsten Menues
- Die Nummer des letzen Menues
- Die Größe (Anzahl der Punkte) eines Menues
- Die zuletzt beim jeweiligen Menue getroffene Wahl

Die Variable mepo enthält die Nummer des aktuellen Menues. Damit enthält dann gleichzeitig m$(mepo) den Namen des Programmteils, in dem man sich gerade befindet.

md(mepo,gros) enthält die Anzahl der Punkte im aktuellen Menue, md(mepo,dflt) enthält den Punkt des Menues, der zuletzt ausgewählt wurde.

In md(mepo,ruck) ist die Nummer des Menues gespeichert, das durch Drücken der DEL-Taste erreicht werden kann (also das jeweils letzte Menue).

Schließlich ist noch die Nummer des jeweils nächsten Menues zum ersten Menuepunkt eines jeden Menues in md(mepo,nxt) gespeichert. md(mepo,nxt) hat aufgrund der entwickelten Datenstruktur noch eine andere Funktion. Einerseits kann damit, wie beschrieben, die Nummer des nächsten Menues gefunden werden, andererseits ist dieser Wert gleichzeitig die Nummer des ersten Menuepunktes des aktuellen Menues (siehe Zeilen 520-560).

Der wohl komplizierteste Teil des Programms ist die Schleife, die aus den DATA-Zeilen, in denen die Menuepunkte stehen, die oben beschriebenen Zeiger ermittelt und setzt. Die DATA Zeilen müssen dazu die einzelnen Menuepunkte enthalten, wobei jedes Menue vom anderen durch eine fortlaufende Nummer getrennt ist. Dabei wird nach der 0 das Hauptmenue angegeben, dann zu jedem Hauptmenuepunkt das dazugehörige Untermenue in der entsprechenden Reihenfolge.

Die Untermenues zu den ersten Untermenues werden auf dieselbe Weise angeführt. Existiert nicht zu allen Auswahlpunkten eines Menues ein Untermenue, so muß das fehlende in der Numerierung übersprungen werden.

Die Initialisierungsschleife (Zeilen 10100-10130) liest zunächst den nächsten Menuepunkt. Für jeden gelesenen Punkt wird der Rücksprungzeiger für das durch die Auswahl dieses Punktes erreichte Menue gesetzt, indem unter md(i,ruck) die Nummer des aktuellen Menues n gespeichert wird. Der Defaultwert (dflt), der den zuletzt gewählten Punkt des Menues enthält, wird hier erst einmal auf 1, also auf den jeweiligen ersten Punkt gesetzt. Dann wird festgestellt, ob alle Punkte eingelesen wurden. Am Ende der DATA Zeilen muß dafür ein "!" stehen.

Sind alle Punkte eingelesen, so wird noch die Größe des letzten Menues ermittelt und die Initialisierungsschleife ist beendet.

In Zeile 10120 wird geprüft, ob eine Nummer gelesen wurde. wenn nicht, wird sofort der nächste Menuepunkt in Zeile 10110 gelesen. Wird festgestellt, daß eine Menuenummer gelesen wurde, so wird zuerst n mit dieser neuen Nummer gelesen. Der nächste Schritt erscheint etwas unlogisch.

Als Nummer des nächsten auf das neue Menue folgenden Menues, wird der aktuelle Zähler der gelesenen Punkte gespeichert. Überprüfen Sie, ob dieser Wert wirklich die Nummer des nächsten Menues darstellt. Schließlich wird noch die Größe des zuletzt gelesenen ermittelt.

Nach dem Durchlaufen der gesamten Initialisierungsschleife ergibt sich schematisch folgendes Bild:

Nachdem das Programm initialisiert wurde, kann es in jedem Programm benutzt werden. Der Aufruf der Menueausgaberoutine hat z.B. folgendes Format:

```
1010 GOSUB 500:IF ein$=del$ THEN RETURN
1020 ON wahl GOSUB    ...Zeilennummer
```

An den so angesprungenen Zeilennummern stehen nun wieder gleichartige Zeilen, wie in 1010 und 1020, die das nächste Untermenue angeben usw.

```
10 REM menue
20 MODE 2
30 GOSUB 1000 : REM Init
40 GOSUB 300 : REM Hauptmenue
50 CLS:END
60 REM Rahmen
70 CLS:PRINT"Menuegenerator
 ";m$(mepo)
80 PRINT STRING$(80,"-");
90 LOCATE 1,23:PRINT STRING$(80,"-");
100 PRINT"Druecken Sie >ENTER< fuer Auswahl des invers Dargest
ellten"
110 PRINT"          oder >DEL< fuer ";exit$(-(mepo>0)-(mepo>1))
;
120 RETURN
130 REM Menue Aufbauen
140 GOSUB 60 :REM Rahmen
150 wahl=md(mepo,dflt)
160 LOCATE 1,5
170 FOR i=1 TO md(mepo,gros):IF i=wahl THEN PAPER 1:PEN 0
180 PRINT i;:PAPER 0:PEN 1:PRINT" - ";:IF i=wahl THEN PAPER 1:
PEN 0
190 PRINT m$(i+md(mepo,nxt)-1):PAPER 0:PEN 1
200 NEXT
210 PRINT:PRINT:PRINT:PRINT"Ihre Wahl :";:PAPER 1:PEN 0:PRINT
wahl;:PAPER 0: PEN 1:LOCATE POS(#0)-3,VPOS(#0)
220 ein$=INKEY$:IF ein$="" THEN 220
230 IF ein$=del$ THEN RETURN
240 IF ein$=CHR$(13) THEN md(mepo,dflt)=wahl:mepo=md(mepo,nxt)
+wahl-1:RETURN
250 IF ein$=CHR$(241) OR ein$=CHR$(242) OR ein$=CHR$(32) THEN
wahl=wahl+1+(wahl=md(mepo,gros))*md(mepo,gros):GOTO 290
260 IF ein$=CHR$(240) OR ein$=CHR$(243) THEN wahl=-(wahl=1)*md
(mepo,gros)+wahl-1:GOTO 290
270 n=VAL(ein$):IF (n<1) OR (n>md(mepo,gros)) THEN 220
280 wahl=n
290 PAPER 1:PEN 0:PRINT wahl;:PAPER 0:PEN 1:GOTO 160
300 REM Hauptmenue
310 GOSUB 130:IF ein$=del$ THEN RETURN
```

```
320 ON wahl GOSUB 340,3000,4000,5000,6000
330 GOTO 310
340 REM Neben1
350 GOSUB 130:IF ein$=del$ THEN mepo=md(mepo,ruck): RETURN
360 ON wahl GOSUB 370,2200,2300,2400,2500,2600:GOTO 350
370 REM unter1
380 GOSUB 130:IF ein$=del$ THEN mepo=md(mepo,ruck): RETURN
390 ON wahl GOSUB 2150,2160,2170:RETURN
1000 REM Init
1010 del$=CHR$(127)
1020 FOR i=0 TO 2:READ exit$(i):NEXT
1030 DATA "Ende Programm"
1040 DATA "Hauptmenue"
1050 DATA "letztes Menue"
1060 DIM m$(100),md(100,3):ruck=0:nxt=1:gros=2:dflt=3:n=-1:i=-
1
1070 i=i+1:READ m$(i):md(i,ruck)=n:md(i,dflt)=1:IF m$(i)="!" T
HEN i=i-1:md(md(i,ruck),gros)=i-md(md(i,ruck),nxt)+1:RETURN
1080 IF LEN(m$(i))<3 THEN n=VAL(m$(i)):md(n,nxt)=i:i=i-1:IF md
(i,ruck)>=0 THEN md(md(i,ruck),gros)=i-md(md(i,ruck),nxt)+1
1090 GOTO 1070
1100 RETURN
1110 DATA "Hauptmenue"
1120 DATA 0
1130 DATA "Drucken"
1140 DATA "Zeigen"
1150 DATA "editieren"
1160 DATA "Suchen"
1170 DATA "Dienst"
1180 DATA 1
1190 DATA "Normal","Proportional","Blocksatz","Blocksatz+Propo
rtional"
1200 DATA 6
1210 DATA "letztes Menue","Nur fuer Testzwecke","usw. etc. usf
."
1220 DATA "!"
```

3.2 Eingabemaskengenerator

Im Zusammenhang mit dem Erstellen von anwenderfreundlichen Programmen ist es auch notwendig, alle Benutzerschnittstellen gegen Fehlbedienungen abzusichern.

Benutzerschnittstellen sind grundsätzlich alle Stellen des Programms, in denen der Anwender per Tastatur, Joystick, Lightpen oder einem anderen Eingabegerät die Möglichkeit hat, den Programmablauf in irgendeiner Weise durch seine Eingabe zu verändern. Der im vorhergehenden Kapitel beschriebene Menuegenerator ist so geschrieben, daß alle nicht zulässigen Eingaben abgefangen und ignoriert werden. Beim Eingabemaskengenerator soll das für beliebige Eingaben realisiert werden. Wir wollen das Programm an einem Beispiel entwickeln:

Eingegeben werden sollen, wie z.B. bei einem Adreßprogramm, Name, Adresse, Ort, Telefon und Datum. Dabei soll für jede Eingabe nur ein bestimmter Raum vorgesehen sein. Bei der Eingabe des Namens sollen nur Buchstaben erlaubt sein, bei der Eingabe der Telefonnummer jedoch nur Zahlen und der Schrägstrich usw.

Die Daten für eine Eingabemarke sollen dabei durch DATA-Zeilen gegeben sein.
Folgende Angaben sind dazu wichtig:

- Kommentar zur Eingabe (z.B. Name)
- Bildschirmposition des Kommentares
- Länge des Eingabefeldes
- Kennziffer für die Menge der erlaubten Eingaben

Die Eingabe einer Maske mit all ihren Feldern wird insgesamt abgeschlossen, d.h., daß ein Korrigieren im ersten Eingabefeld auch nicht möglich sein soll, wenn man sich schon in einem anderen Feld befindet. Eine Eingabe über die vorgegebene Länge hinaus soll nicht möglich sein. Die Cursortasten für "nach rechts" und "nach links" bewegen den Cursor innerhalb des

Feldes. RETURN und "Cursor nach unten" bewegen den Cursor ins nächste Feld, während er durch "Cursor nach oben" in das vorhergehende zurückgebracht wird. Durch Drücken der COPY-Taste wird die gesamte Eingabe übernommen.

```
10 MODE 2
20 GOSUB 820
30 GOSUB 80: REM Eingabe holen
40 REM Eingabe Stet zur verfuegung
50 LOCATE 1,20
60 FOR i=1 TO anzfeld:PRINT ein$(i):NEXT
70 END
80 CLS:feldnum=1
90 FOR i=1 TO anzfeld
100 LOCATE x(i),y(i)
110 PRINT koment$(i);":";STRING$(l(i),"."):ein$=STRING$(l(i),"
 ")
120 ein$(i)=STRING$(l(i)," ")
130 NEXT
140 x=x(feldnum):y=y(feldnum)
150 lm=l(feldnum):kenzif=ken(feldnum)
160 LOCATE x,y:PRINT koment$(feldnum);":";
170 x=x+LEN(koment$(feldnum))+1
180 IF x>80 THEN y=y+x\80:x=x MOD 80
190 e$=ein$(feldnum)
200 GOSUB 280
210 ein$(feldnum)=e$
220 IF endflg THEN RETURN: REM Eingabe Fertig
230 IF hochflg THEN feldnum=feldnum-1:IF feldnum=0 THEN feldnu
m=1
240 IF runtflg THEN feldnum=feldnum+1:IF feldnum>anzfeld THEN
feldnum=anzfeld
250 runtflg=0:hochflg=0
260 GOTO 140
270 PRINT CHR$(7);:GOTO 360
280 ' Eingabe holen
290 ' x,y : Startkoordinaten
300 ' lm : Eingabefeldlaenge maximal
310 ' aus : Window #
320 ' rueckgabe : e$
330 LOCATE x,y
340 l=1
350 CALL &BB81: REM Cursor an
360 a$=INKEY$:IF a$="" THEN 360
```

```
370 a%=ASC(a$)
380 FOR i=1 TO anstz:IF a%<>stz%(i) THEN NEXT
390 ON i GOTO 470,470,510,540,570,590
400 REM Test auf erlaubte eingabe
410 okflg=-1
420 ON kenzif GOSUB 610,660,720,770
430 IF okflg=0  THEN 270
440 IF POS(#aus)>=x+lm THEN 270
450 e$=LEFT$(e$,POS(#aus)-x)+a$+RIGHT$(e$,lm-(POS(#aus)-x)-1)
460 PRINT a$;:GOTO 360
470 REM Return und Cursor nach Unten
480 runtflg=-1
490 CALL &BB84: REM Cursor aus
500 RETURN
510 REM chr$(242) cursor rechts
520 IF POS(#aus)=x THEN 270
530 PRINT CHR$(8);:GOTO 360
540 REM chr$(243) cursor links
550 IF POS(#aus)>=x+lm THEN 270
560 PRINT CHR$(9);:GOTO 360
570 REM CURSOR Hoch
580 hochflg=-1:GOTO 490
590 REM Copy
600 endflg=-1:GOTO 490
610 REM Test erlaubt Buchstaben
620 IF a%>64 AND a%<91 THEN RETURN
630 IF a%>96 AND a%<123 THEN RETURN
640 IF a%=32 THEN RETURN
650 okflg=0:RETURN
660 REM Test erlaubt Buchstaben und Zahlen
670 GOSUB 610
680 IF okflg THEN RETURN ELSE okflg=-1
690 REM Test auf Zahlen
700 IF a%>47 AND a%<59 THEN RETURN
710 okflg=0:RETURN
720 REM Test erlaubt Telefon, also Zahlen und "/"
730 GOSUB 690
740 IF okflg THEN RETURN ELSE okflg=-1
750 IF a%=47 THEN RETURN
```

```
760 okflg=0:RETURN
770 REM Test erlaubt Datum, also Zahlen und Punkt
780 GOSUB 690
790 IF okflg THEN RETURN ELSE okflg=-1
800 IF a%=46 THEN RETURN
810 okflg=0:RETURN
820 REM init
830 ' anstz : Anzahl Steuerzeichen
840 ' stz%(anstz) : ASCII der erlaubten Steuerzeichen
850 anstz=6:FOR i=1 TO anstz:READ stz%(i):NEXT
860 DATA 13,241,242,243,240,224
870 aus=0:REM Windownummer
880 REM Eingabemaske
890 READ anzfeld : REM Anzahl der Eingabefelder
900 FOR i=1 TO anzfeld
910 READ koment$(i),x(i),y(i): REM Kommetar,x- und y-Position
920 READ l(i): REM laenge des Eingabefeldes
930 READ ken(i):REM Kennziffer fuer erlaubte Eingaben
940 ' 1:Buchstabe, 2:Buchstaben und Zahlen, 3:Telefon, 4:Datum
950 NEXT i
960 RETURN
970 REM Menue Data
980 DATA 5
990 DATA "Name    ",1,4,20,1
1000 DATA "Datum   ",50,4,8,4
1010 DATA "Strasse",1,6,20,2
1020 DATA "Ort     ",1,9,30,2
1030 DATA Telefon,50,9,12,3
```

Mit dem XREF-Befehl erzeugte Crossreferenceliste:

```
AUS ! 440 450 450 520 550 870
ANSTZ ! 380 850 850
A % 370 380 620 620 630 630 640 700 700 750 800
A $ 360 360 370 450 460
ANZFELD ! 60 90 240 240 890 900
ENDFLG ! 220 600
E $ 190 210 450 450 450
EIN $ 60 110 120 190 210
FELDNUM ! 80 140 140 150 150 160 170 190 210 230 230 230 230 2
40 240 240 240
HOCHFLG ! 230 250 580
I ! 60 60 90 100 100 110 110 110 120 120 380 380 390 850 850 9
00 910 910 910 920 930 950
KEN ! 150 930
KENZIF ! 150 420
KOMENT $ 110 160 170 910
LM ! 150 440 450 550
L ! 110 110 120 150 340 920
LIWST !
OKFLG ! 410 430 650 680 680 710 740 740 760 790 790 810
RUNTFLG ! 240 250 480
STZ % 380 850
X ! 100 140 140 160 170 170 180 180 180 180 330 440 450 450 52
0 550 910
Y ! 100 140 140 160 180 180 330 910
```

4. Circle

Das hier gezeigte Listing stellt eine Möglichkeit dar, den auch bei den neuen CPC-Versionen nicht vorhandenen Circle-Befehl vom BASIC aus mit hervorragender Geschwindigkeit zu simulieren.

Da die Routine nur je einen 1/4 Kreis berechnet und die anderen Viertel durch Spiegelung der Werte erzeugt, ist sie schneller als "normale"-Circle Routinen.

Bei Aufruf der Routine mit GOSUB muß R den Radius, X und Y die Position und e die Exzentrizität ("Beulungsfaktor") enthalten.

```
10 RAD
20 MODE 2
30 INPUT"radius,exzentriz. ",r,e
40 INPUT"x Koor.,y Koor.   ",xk,yk
42 GOSUB 60
43 END
60 x(0)=0:y(0)=r*e
70 x(1)=0:y(1)=-r*e
80 x(2)=0:y(2)=-r*e
90 x(3)=0:y(3)=r*e
100 FOR al=0 TO PI/2 STEP PI/18
110 x=r*SIN(al):y=SQR(r*r-x*x)*e
120 i=0:GOSUB 180
130 i=1:y=-y:GOSUB 180
140 i=2:x=-x:GOSUB 180
150 i=3:y=-y:GOSUB 180
160 NEXT
170 RETURN
180 MOVE xk+x(i),yk+y(i):x(i)=x:y(i)=y
190 DRAW xk+x(i),yk+y(i):RETURN
```

5. Schützen eigener Programme

Eine einfache Möglichkeit, eigene Programme vor neugierigen Naturen zu schützen, ist das Abspeichern mit dem Zusatz 'p'. Das 'p' wird, durch Komma getrennt, an den Programmnamen angehängt.

```
SAVE "name.xxx",p
```

Ein solches Programm kann nur durch den Befehl RUN "name.xxx" gestartet werden. Der Programmlauf kann aber noch mit der ESC-Taste abgebrochen werden. Dann ist direkt zwar noch kein Listing möglich, aber Freaks kommen sehr schnell auch dahinter.

Wenn man den Speicherplatz &BDEE mit dem hexadezimalen Wert &C9 belegt

```
POKE &BDEE,&C9
```

wird eine Programmunterbrechung durch Betätigen der ESC-Taste unterbunden. Der gleiche Effekt ist auch mit dem BASIC-Befehl KEY DEF möglich:

```
KEY DEF 66,1,0      ESC-Taste aus
KEY DEF 66,1,252    ESC-Taste an
```

Es versteht sich von selbst, daß das Programm dabei natürlich so ausgefuchst erstellt sein muß, daß es nicht abstürzen kann oder durch beabsichtigt falsche Eingaben in eine Error-Meldung gezwungen wird, was ja ebenfalls einen Programmstop mit sich bringen würde.
Wenn ein Programm zu umfangreich ist, um alle Eventualitäten zu prüfen, welche zu einer Fehlermeldung führen können, kann man einen Programmabbruch durch den Befehl

```
ON ERROR GOTO zeilennummer
```

vermeiden. Ob in der angegebenen Zeile nun eine selbst definierte Fehlermeldung durchgeführt wird oder z.B. ein CALL 0 bleibt dem Programmierer dann selbst überlassen.

Beispiel:

```
10 ZAEHLER=0
20 INPUT"Teiler",T
30 PRINT 10/T
40 ON ERROR GOTO 100
50 GOTO 20
100 ZAEHLER=ZAEHLER+1
110 IF ZAEHLER<4 THEN PRINT"falsche Eingabe" ELSE CALL 0
120 GOTO 50
```

Dem Programmbediener ist es in diesem Beispiel drei mal gestattet eine falsche Eingabe zu tätigen. Bei einer weiteren fehlerhaften Eingabe wird durch CALL 0 ein RESET ausgelöst.

Zu einem guten Programm gehört aber auch ein ordentlicher Abschluß. Damit das Programm nach diesem Abschluß nicht geknackt werden kann, muß es im Rechner zerstört werden; dies ist immer noch der sicherste Schutz vor dem Auslisten.
Sollte also die Abfrage nach dem Programmende positiv beantwortet werden, braucht man lediglich den Befehl CALL 0 zu setzen. Dieser löst dann einen RESET aus, wie es auch mit der Tastenkombination CTRL/SHIFT/ESC möglich ist.

6. Erleichterung bei der Programmeingabe

Bei der Eingabe längerer Programme kommt es häufig vor, daß unnötige Blanks gesetzt werden. Diese beanspruchen Speicherplatz, der dem Programmierer an einer anderen Stelle möglicherweise hinterher fehlt.
Es ist nun aber kaum machbar, sich beim Eintippen der Programme einerseits auf die Syntax zu konzentrieren und andererseits auch noch im Auge zu behalten, welche Blanks erforderlich, und welche überflüssig sind.

Wenn man vor dem Beginn der Eingabe den Speicher &AC00 mit einem Wert ungleich null POKEt, so ist einem diese Sorge abgenommen. In diesem Fall kontrolliert der Rechner, welche Blanks zur Syntax gehören. Alle anderen werden unterdrückt. Ja, man kann sogar, wenn man nicht sicher ist, 'für alle Fälle' noch ein paar Blanks mehr eingeben. Letztendlich werden nur die maßgeblichen übernommen.

7. Beschleunigung von BASIC-Programm

Obwohl das Schneider-BASIC sich in Bezug auf Geschwindigkeit nicht zu verstecken braucht, gibt es Probleme, bei denen der Zeitfaktor eine große Rolle spielt (z.B. bei Sortierverfahren). Wenn eine vom Algorithmus her nicht mehr zu beschleunigende Version vorliegt, sind oft durch geschickte Programmiertechniken noch Geschwindigkeitsvergrößerungen zu erreichen. Einige dieser Möglichkeiten wollen wir im folgenden aufzeigen.

Mit der BASIC-Variablen TIME haben wir ein einfaches Werkzeug, um BASIC-Ausführungzeiten zu messen. Betrachten wir noch einmal das Bubblesort-Verfahren. Anhand dieses Beispiels wollen wir allgemein zeigen, wie Programmbeschleunigungen zu erreichen sind. In dem folgenden Programm haben wir einen ständig gleichbleibenden Teil und einen variablen Teil geschrieben, um die Beispiele möglichst deutlich zu machen. Die Zeilen 10 bis 90 werden wir also, da sie immer gleich bleiben, nicht ständig neu drucken.

```
10 REM BASIC Geschwindigkeit
20 anz=10:anz=anz-1
30 DIM Feldelement(anz)
40 FOR i=0 TO anz:READ Feldelement(i):NEXT
50 DATA 10,4,7,3,2,24,8,17,5,19
60 t!=TIME
70 GOSUB 100
80 PRINT (TIME-t!)/300
90 END
100 REM Bubblesort
110 feldgros=anz : REM  zu sortierendes Feld ist zuerst Gesamt
feld
120   index=1 : REM  Beginne Vergleich mit ersten Elementen
130     IF Feldelement(index-1)>Feldelement(index) THEN GOSUB
190 : REM  Austauschen
140     index=index+1 : REM  naechstes Element verleichen
150     IF index<=feldgros THEN 130 : REM  zum Vergleich
160   feldgros=feldgros-1 : REM  zu sortierendes Feld verklein
ern
170   IF feldgros>=1 THEN GOTO 120 : REM  verkleinertes Feld s
ortieren
180 RETURN
190 REM Unterprogramm : Austausch zweier Elemente
200 zwischenspeicher=Feldelement(index)
210 Feldelement(index)=Feldelement(index-1)
220 Feldelement(index-1)=zwischenspeicher
230 RETURN
```

Zeit: 0.62 Sekunden mit REM-Zeilen

REM-Zeilen

Die erste Verbesserung erreichen wir durch das Entfernen sämtlicher REM- bzw. "'" (Shift 7) -Zeilen.

Zeit: 0.56 Sekunden ohne REM-Zeilen

Variablenwahl

Einer der wichtigsten Punkte ist die Variablenwahl. Das BASIC kennt INTeger- und REAL-Variablen. REAL-Variablen sind 5 Bytes lang und benötigen bei der Behandlung sehr viel mehr Zeit als die 2 Byte langen INT-Variablen. Daher sollten immer, wenn es möglich ist, INT-Variablen eingesetzt werden. In den meisten Fällen kann auf REAL-Variablen verzichtet werden.

```
5 DEFINT a-z
100 feldgros=anz
120 index=1
130 IF Feldelement(index-1)>Feldelement(index) THEN GOSUB 190
140 index=index+1
150 IF index<=feldgros THEN 130
160 feldgros=feldgros-1
170 IF feldgros>=1 THEN GOTO 120
180 RETURN
190 zwischenspeicher=Feldelement(index)
210 Feldelement(index)=Feldelement(index-1)
220 Feldelement(index-1)=zwischenspeicher
230 RETURN
```

Zeit: 0.42 Sekunden ohne REAL-Variablen

INT-Variablen können nur Werte von -32768 bis 32767 annehmen. Bei der Behandlung von Adressen, die zwischen 0 und

65535 (&0000 bis &FFFF) liegen, treten daher Probleme auf. Anstatt aber nun doch REAL-Variablen zu benutzen, kann man alle Adressen, die größer als &7FFF sind, auch als negative Zahlen behandeln.

FOR-NEXT-Schleifen

Sollen Schleifen programmiert werden, so ist die FOR-NEXT Schleife schneller als z.B. eine Schleife mit IF-THEN-GOTO.

Zeit: 0.34 Sekunden mit FOR-NEXT

Variblennamen und Definitionen

Je kürzer der Variablenname, desto schneller seine Verarbeitung. Da auch das Typenkennzeichen zu seiner Länge beiträgt, sollten alle Variablen vorher mit DEF INT/ REAL/ STR definiert werden.

```
5 DEFINT a-z
10 REM BASIC Geschwindigkeit
20 a=10:a=a-1
30 DIM e(a)
40 FOR i=0 TO a:READ e(i):NEXT
100 FOR f=a TO 1 STEP -1
110 FOR i=1 TO f
120 IF e(i-1)>e(i) THEN GOSUB 150
130 NEXT i,f
140 RETURN
150 z=e(i)
160 e(i)=e(i-1)
170 e(i-1)=z
180 RETURN
```

Zeit: 0.35 Sekunden mit kurzen vordefinierten Variablennamen

BASIC-Zeilen

Bei der internen Verarbeitung geht es auf jeden Fall schneller, eine lange BASIC-Zeile zu verarbeiten, als mehrere kurze. Programmzeilen, die nicht als Ansprungzeile benötigt werden, sollten deshalb mit anderen Zeilen zusammengefaßt werden.

```
100 FOR f=a TO 1 STEP -1:FOR i=1 TO f:IF e(i-1)>e(i) THEN GOSU
B 120
110 NEXT i,f:RETURN
120 z=e(i):e(i)=e(i-1):e(i-1)=z:RETURN
```

Zeit: 0.34 Sekunden mit wenigeren Programmzeilen

Wie Sie sehen, haben wir gegenüber der ersten Version eine beträchtliche Zeitersparnis erreicht. Wem dies noch immer nicht schnell genug ist, der wird wohl oder übel zur Maschinensprache greifen müssen.

Noch einige Tips, die nicht am vorigen Programm demonstriert werden können.

- Bevor Variablen mit demselben Anfangsbuchstaben verwendet werden, sollten die Anfangsbuchstaben genommen werden, die noch frei sind.

- Kommen Variablen mit demselben Anfangsbuchstaben vor, sollten die Variablen, die voraussichtlich am häufigsten benutzt werden, als erste im Programm initialisiert werden, auch wenn das eigentlich nicht notwendig ist. Beispiel:

  I ist häufiger benutzt als IN$. IN$ taucht in Zeile 10, I erst in Zeile 100 auf. Daher sollte vor Zeile 10 z.B. einfach Zeile 9 I=0 eingefügt werden.

Die zuerst gefundenen Variablen werden als erste in die Variablentabelle eingetragen und dadurch später schneller wiedergefunden.

- Bei Berechnungen sind die Grundrechenarten immer schneller. Also z.B. X*X anstelle von X^2 verwenden. Auch sollten Klammern nur gesetzt werden, wenn sie unbedingt notwendig sind.

- Bringen Sie keine unnötigen Anweisungen in die Schleifen hinein. Soll z.B. für den Schleifendurchlauf eine bestimmte Farbe gewählt werden, so kann das schon vor der Schleife passieren.
  Aber auch solche Befehle, die an sich keine Ausführungszeit beanspruchen, wie REM oder DATA, gehören nicht in eine Programmschleife. Sie werden zwar nicht 'ausgeführt', müssen aber immer wieder interpretiert werden, was ebenfalls zeitintensiv ist.

- Vermeiden Sie alle unnötigen Blanks im Programmtext. Auch diese müssen interpretiert werden und verbrauchen dementsprechend kostbare Zeit.

- Wenn Sie sehr viel mit String-Zuweisungen arbeiten, kann es des öfteren zur Garbage Collection kommen. Bei dieser 'Müllbeseitigungsaktion' werden alle nicht mehr benötigten Werte im Speicher gelöscht und damit wieder Platz geschaffen. Diese Aktion dauert meist viele Sekunden, in denen der Rechner nicht zu beeinflussen ist. Die Garbage Collection ist auch nicht zu vermeiden. Man kann sie aber stetig in kleinen Stücken durchführen, so daß immer nur wenige Millisekunden verbraucht werden.

Der Befehl PRINT FRE("") löst die Garbage Collection aus. Wenn man ihn geschickt plaziert, beispielsweise hinter einer INPUT-Anweisung, wo meist eine gewisse Verzögerung bis zur Eingabe auftritt, oder mit dem Befehl EVERY-GOSUB in bestimmten Zeitabschnitten immer wieder aufruft, so kann ein Zeitverlust durch die Garbage Collection vollkommen vermieden werden.

8. Spezielle Befehle des Schneider-BASIC

8.1 EVERY a,b GOSUB c

Mit dem Befehl EVERY-GOSUB ist die phantastische Möglichkeit gegeben, hardwaregesteuerte Unterbrechungen mit BASIC-Befehlen gezielt nutzen zu können. Das ermöglicht dem Programmierer Multi-Tasking-Programme in BASIC zu schreiben, ohne mühselige Abstecher in die Maschinensprache machen zu müssen.
Durch geschickte Anwendung dieses Befehles kann der Rechner mehrere Programme, scheinbar zur selben Zeit, bearbeiten. Somit kann man besonders zeitkritische Unterprogramme, wie z.B. einen Zeitzähler, optimal und mit geringstem Aufwand verwalten.
Wird der Befehl EVERY mit den entsprechenden Parametern, die noch erklärt werden, aufgerufen, läßt der BASIC-Interpreter alles 'stehen und liegen', und verzweigt zu dem, mit GOSUB angegebenen Programmteil.

Bevor nun eine eingehende Klärung erfolgt, soll ein kurzes Programm den Nutzen dieses starken Befehls andeuten.

```
5 MODE 2 : ZEIT=0
10 EVERY 50,0 GOSUB 100
20 FOR I=0 TO 2*PI STEP PI/360
30 PLOT 320+300*SIN(I),200+195*COS(I)
40 NEXT I
50 END
100 ZEIT=ZEIT+1
110 LOCATE 50,10
120 PRINT ZEIT
130 RETURN
```

Dieses Beispiel der Lissajou-Figur hat zwar schon einige Male herhalten müssen, ist aber zur Demonstration des EVERY-Befehles sehr gut geeignet.
Hier ruft EVERY ein UP auf, welches einen Zeitzähler inkrementiert und darstellt. Dazu muß dieses UP natürlich in gleich-

mäßigen Abständen angewählt werden. Da aber die Berechnungsdauer der Funktionen SIN und COS in gewissen Grenzen abhängig von den Funktionswerten ist - so dauert z.B. die Berechnung des Sinus von PI/3 deutlich länger als die von PI/4 - kann man in diese PLOT-Schleife keinen Zeitgeber einbauen.
Diesen legt man in das UP, welches dann, durch den Befehl EVERY gesteuert, in gleichmäßigen Zeitabständen aufgerufen wird.

Die Parameter a, b und c:

a - gibt die Anzahl der Zeiteinheiten (0.02 sec) an, nach denen das UP aufgerufen werden soll.

b - ist die Nummer des gewählten Zeitgebers. Vier Zeitgeber mit den Nummern 0 bis 3 stehen zur Verfügung, wobei 3 die höchste Priorität hat. Wenn mehrere UP-Aufrufe im Programm eingebaut sind, so werden sie nach ihrer Rangstufe behandelt. Erfolgt z.B. ein Aufruf mit Priorität 2, während noch ein Ablauf mit niederer Rangstufe behandelt wird, so wird dieser unterbrochen, der mit der höheren Priorität durchgeführt und danach erst der andere zu Ende gebracht.
Aus diesem Grund muß die Vergabe der Prioritäten genauestens überdacht sein.

c - steht für die Zeilennummer des gewünschten UPs. Zu dieser Zeilennummer verzweigt das Betriebssystem in den durch a angegebenen Intervallen.

Uhren höherer Priorität haben also den unbedingten Vorrang vor denen niederer Prioritäten. Alle Uhren haben Vorrang vor dem Hauptprogramm. Das kann darin gipfeln, daß bei ungünstiger Programmierung keine Zeit mehr fürs Hauptprogramm übrigbleibt, wenn die durch die Uhr bedingte Verzweigung zu viel Zeit beansprucht. Folgendes Beispiel, schnell mal eingetippt, zeigt dies deutlich.

```
5 MODE 2
10 EVERY 30 GOSUB 100
20 FOR I=1 TO 639 : PLOT I,100 : NEXT I
50 END
100 REM Unterprogramm
110 FOR J=1 TO 30 : LOCATE 20,5 : PRINT J : NEXT J
120 RETURN
```

Das 'Unterprogramm' ab Zeile 100 verschlingt soviel Zeit, daß zum eigentlichen Programm, dem PLOTten einer Linie, kaum noch Rechnerzeit übrig bleibt.
So sollte man diesen Befehl nicht gebrauchen.

Um den Befehl EVERY voll ausschöpfen zu können, sollten Sie auch direkt die Erklärungen zu den Befehlen DI und EI lesen, die bei der Erläuterung des Befehles AFTER ebenfalls beschrieben werden.

8.2 AFTER a,b GOSUB c

Der Befehl AFTER bewirkt einen bedingten Sprung zu einem Unterprogramm. Der Sprung wird erst nach einer bestimmten Zeit ausgeführt.

Die Parameter:

a - Anzahl der Zeiteinheiten (0.02 sec), nach denen der Sprungbefehl ausgeführt werden soll.

b - Angabe des gewählten Zeitgebers, es stehen die vier Zeitgeber 0 bis 3 zur Verfügung, wobei 3 die höchste Vorrangstufe besitzt.

c - gibt die erste Zeilennummer des UPs an.

Im Gegensatz zu dem Befehl EVERY, der einen immer wiederkehrenden UP-Aufruf zur Folge hat, bewirkt der Befehl AFTER nur jeweils einen Sprung zu dem gewählten UP.
AFTER-GOSUB-Befehle mit höheren Prioritäten unterbrechen im Falle einer zeitlichen Überschneidung die UPs, die von AFTER-GOSUB-Befehlen niederer Priorität aufgerufen wurden. Diese werden später zu Ende geführt.

Die Befehle EVERY und AFTER simulieren softwaremäßig eine Interruptsteuerung, die beim Z80 hardwaremäßig vorhanden ist. So sind auch die Z80-Befehle DI und EI im Schneider-BASIC wiederzufinden.

DI steht für Disable Interrupt (Interrupts sperren)
EI steht für Enable Interrupt (Interrupts zulassen)

Soll ein Unterprogramm, das durch EVERY oder AFTER aufgerufen wurde, unbedingt vor Interrupts höherer Priorität geschützt werden, so kann dies durch setzen des DI-Befehles geschehen. Dieser unterbindet dann eine Unterbrechung von einem anderen Zeitgeber, egal ob nun EVERY oder AFTER einen Interrupt anfordert.
Ebenso schützt DI ein Unterprogramm, das z.B. mit EVERY 15,2 GOSUB aufgerufen wurde, vor einer Unterbrechung mittels AFTER 10,2 GOSUB. Zu beachten ist an diesem Beispiel, daß derselbe Zeitgeber benutzt wurde.

Mit dem Befehl EI werden wieder alle Unterbrechungen zugelassen. Die bis dahin gesperrten Interrupts sind aber nicht völlig unterdrückt worden. Die zwischenzeitlich aufgetretenen Unterbrechungen wurden gespeichert und können nun nach EI im 'Eildurchgang' abgearbeitet werden.

Starten Sie dazu folgendes Programm:

```
10 EVERY 20,0 GOSUB 100
20 EVERY 10,2 GOSUB 200
30 GOTO 30
90 REM------------------------------
100 DI
110 FOR I=1 TO 10 : PRINT I; : NEXT
120 PRINT : EI : RETURN
130 REM------------------------------
200 PRINT"UNTERBRECHUNG PRIORITAET 2"
210 RETURN
```

Ändern Sie jetzt den Endwert der FOR-NEXT-Schleife von 10 auf 50.
Man erkennt, daß die Schleife, geschützt durch DI, immer erst ganz abgearbeitet werden kann, bevor die Verzweigung in die Zeile 200 durch die höhere Priorität erfolgt. Dann aber wird der Text aus Zeile 200 nicht nur einmal, sondern so oft hintereinander gedruckt, wie Interruptanforderungen während der FOR-NEXT-Schleife auftraten.

8.3 Der Befehl MOD

Die Stärke des Schneider-Computers begründet sich u.a. in solchen Befehlen, die längere Befehlssequenzen ersetzen. Dadurch wird der Quellcode nicht nur kürzer, sondern auch deutlich übersichtlicher gehalten.

Wir wollen die ASCII-Codes von 33 bis 255 mit den zugehörigen Zeichen in vier Spalten auf dem Bildschirm ausgeben.

```
10 FOR I=33 TO 255
20 PRINT I;" ";CHR$(I),
30 IF I/4=INT(I/4) THEN PRINT
40 NEXT
```

Zeile 10 und 40 bilden die Schleife. In Zeile 20 wird die Laufvariable und das zugehörige Zeichen ausgegeben. Das Komma hinter der PRINT-Zeile bewirkt eine hintereinander durchgeführte Ausgabe. Damit nur die gewünschten vier Spalten bei der Ausgabe entstehen, muß nach vier PRINTs mit Komma eines ohne erfolgen, damit der Wagenrücklauf ausgelöst wird.
In der Zeile 30 wird mittels INTeger jeder vierte Durchgang ermittelt. Einfacher, verständlicher und übersichtlicher geht dies aber mit der Funktion MOD (Modulo), deren Ergebnis der sogenannte Rest der Division ist.

```
 4 MOD  4 = 0
 6 MOD  4 = 2
10 MOD 11 = 10
```

Wenn nun der Rest der Division durch 4 gleich 0, also das Ergebnis der Funktion MOD gleich 0 ist, dann ist die Bedingung für den Wagenrücklauf erfüllt.
Man kann Zeile 30 wie folgt verbessern:

```
30 IF I MOD 4=0 THEN PRINT
```

TEIL 2

Befehlserweiterungen und andere nützliche Maschinenprogramme

9. Programmierhilfen

9.1 Der Aufbau des Variablenspeichers

Im folgenden werden wir uns mit der internen Verarbeitung der BASIC Variablen beschäftigen. Am Ende dieses Abschnittes finden Sie die Befehlserweiterungen DUMP und XREF. DUMP gibt alle Variablen, die mit bestimmten Buchstaben anfangen, mit ihren Werten aus. XREF (X=Kreuz: engl. Cross, also Cross-reference) gibt zu beliebigen Variablennamen alle BASIC-Programmzeilen aus, die diese Variable enthalten. DUMP und XREF sind sinnvolle Befehlserweiterungen, die besonders das Programmieren und das meistens darauf folgende Fehlersuchen sehr vereinfachen.

Das Locomotive-BASIC hat gerade in Bezug auf Variablen hervorragende Eigenschaften zu bieten:

Wie üblich, gibt es drei Standardtypen von Variablen: Integer (Ganzzahl), Real (Fließkomma) und String (Kette, also alphanumerische Daten). Angenehm ist zunächst die Möglichkeit, Variablen mit bestimmten Anfangsbuchstaben von vornherein einem festen Typ zuzuordnen. Von dieser Möglichkeit sollte grundsätzlich Gebrauch gemacht werden (siehe Kapitel: Beschleunigung von BASIC-Programmen).

Vollkommen neu in dieser Klasse von Rechnern ist jedoch die Möglichkeit, Variablennamen mit bis zu 40 signifikanten Zeichen zu vergeben. D.h., daß alle Buchstaben bzw. Zahlen des Variablennamens bei der Unterscheidung von anderen beachtet werden. Bisher war es üblich nur die zwei ersten Buchstaben zu berücksichtigen. Zum einen ergibt sich dadurch die Möglichkeit, eine fast unbegrenzte Zahl von Variablennamen zu benutzen, viel wichtiger ist jedoch, daß zum anderen die Variablen "beim Namen genannt" werden können. Man kann z.B. jetzt anstelle von AK Ausgabekanal schreiben.

Natürlich darf man es mit der Länge der Namen nicht übertreiben, da die Programme dadurch verlangsamt werden. Um die Variablennamen unterschiedlicher Länge zu speichern, haben

sich die "Locomotivler" einige besondere Tricks ausgedacht. Dadurch erhöht sich evtl. sogar die Verarbeitungsgeschwindigkeit gegenüber der sonst üblichen Methode, obwohl dies aufgrund der festen Namenslänge weniger aufwendig ist.

Doch zuvor kurz einiges zu den Grundlagen der Variablenbehandlung im Rechner.

Zur Verwaltung der Variablenwerte befindet sich im Anschluß an jedes BASIC-Programm die Variablentabelle. In dieser Tabelle sind sämtliche Variablen mit Namen, Typ und Wert eingetragen. Wird bei der Ausführung eines BASIC-Befehls, z.B. in einer Formel, ein Variablenname gefunden, wird dieser Name in der Tabelle gesucht und der jeweilige Wert gelesen.
Wird eine Variable zum ersten Mal benutzt, wird sie sofort in die Tabelle eingetragen, da sie noch nicht in dieser vorhanden ist.

In einigen BASIC-Dialekten gibt es die Funktion VARPTR (VARiablen PoinTeR), die die Adresse einer Variablen in der Variablentabelle ermittelt. Auch im Schneider BASIC gibt es diese Funktion, allerdings ist sie nicht im Handbuch erwähnt. Der Variablenpointer einer Variablen läßt sich beim Schneider mit @Variablenname ermitteln.

Der so erhaltene Wert (der VARPTR) zeigt auf die Adresse in der Variablentabelle, an der der Wert der Variablen steht. Der Wert ist für jeden der drei möglichen Typen unterschiedlich abgespeichert.

Integer-Variablen

Bei INT-Variablen besteht der Wert einfach aus High und Low Byte der Zahl. An der niedrigeren Adresse, also der, auf die der VARPTR zeigt, steht immer das Low Byte (LB).

```
A%=100
PRINT PEEK(@A%)+256*PEEK(@A%+1)
```

ergibt wieder 100, den zugewiesenen Wert. Enthält die Variable negative Zahlen, so muß der mit PEEK erhaltene Wert mit UNT umgerechnet werden. Das ist notwendig, da es INT Zahlen ohne und mit Vorzeichen gibt. -256 ist eine vorzeichenbehaftete INT Zahl, &8100 eine vorzeichenlose Zahl. Dezimal hat letztere eigentlich den Wert 20736. PRINT &8100 ergibt jedoch -256.

Bei der vorzeichenbehafteten Darstellung, die außer bei Hexadezimalzahlen immer benutzt wird, wird Bit 15 der Zahl als Vorzeichen interpretiert.

Real-Variablen

Bei diesem Typ werden Zahlen in der Exponentialdarstellung gespeichert. Die Zahl wird so dargestellt, daß sie nur eine Vorkommastelle besitzt. Die wirkliche Position des Kommas wird durch den Exponenten angegeben:

$$54321=5.4321*10^4.$$

Allerdings wird dies intern im Dual- und nicht im Dezimalsystem erledigt.

Zum Speichern einer Realvariablen werden fünf Bytes benutzt. Die ersten vier Bytes bilden die Mantisse, also den Zahlenteil des Wertes, wobei, wie vereinbart, das Komma immer an 2ter Stelle steht. Das MSB (Most Significant Bit - höchstwertiges Bit) von Byte 4 stellt das Vorzeichen der Zahl dar. Im Byte 5 ist schließlich der Exponent enthalten. Um den wirklichen Wert des Exponenten zu erhalten, muß von Byte fünf 129 abgezogen werden, oder anders ausgedrückt:

Der Exponent ist mit Offset 129 dargestellt.

Wie bereits erwähnt, ist der Wert als Dualzahl gespeichert. Also besteht auch die Mantisse nur aus Nullen und Einsen. Da die Mantisse genau eine Vorkommastelle hat, und Null keine Vorkommastelle ist, bleibt nur die Eins. D.h., die Vorkommastelle der Dualzahl ist immer eins. Deshalb braucht die Eins nicht mit-

gespeichert zu werden, an ihrer Stelle (Bit 7 von Byte 4) wird das Vorzeichen gespeichert. Wie bei den INT Zahlen das Low Byte den niederwertigen Wert besitzt, haben die Bytes mit niedrigen Adressen auch hier niedrige Werte. Die vier Mantissenbytes nennen wir m1 bis m4. Damit erhält man die Mantisse durch:

```
PRINT (m1+256*m2+256^2*m3+256^3*(m4 OR 128))/256^4
```

"OR 128" fügt die nicht mitgespeicherte 1 wieder an. Die Division durch 256^4 ist notwendig, damit auch eine Kommazahl mit einer Vorkommastelle entsteht.

Den Wert einer Realzahl können wir also mit Hilfe des VARPTRs mit folgendem kleinen Programm berechnen.

```
100 a=13:'untersuchte Fliesskommavariable
110 ad=@a:'Adresse von a
120 m1=PEEK(ad):m2=PEEK(ad+1):m3=PEEK(ad+2)
130 m4=PEEK(ad+3):ex=PEEK(ad+4)
140 PRINT (1-2*SGN(m4 AND 128))*2^(ex-129)*(1+(m4 AND 127)+(m3+(m2+m1
/256)/256)/128)
```

String-Variablen

Die letzte Gruppe von Variablen ist die, in denen alphanumerische Daten gespeichert werden können. Ein String ist intern nur eine Reihe aufeinanderfolgender Bytes, die die ASCII Codes der jeweiligen Zeichen enthalten. Da Strings zwischen Länge 0 und Länge 255 variieren können, werden die eigentlichen Inhalte (die Codes) nicht direkt in der Variablentabelle gespeichert. Zum Speichern der Strings gibt es einen eigenen Speicherbereich im RAM (am oberen Ende des BASIC RAMs), in dem nur Zeichenketten gespeichert werden. Die Variablentabelle enthält dann nur noch die Startadresse des Strings in der Stringtabelle und die Länge des Strings. Beide Werte zusammen bilden den sog. Stringdescripter. Das folgende Programm macht dieses Verfahren deutlich.

```
100 INPUT a$
110 ad=@a$
120 i=PEEK(ad)
130 stad=PEEK(ad+1)+256*PEEK(ad+2)
140 FOR i=stad TO stad+i-1:PRINT CHR$(PEEK(i));:NEXT i
```

Damit haben wir alle Variablentypen besprochen. Schauen wir uns jetzt den vollständigen Aufbau der Variablentabelle an.

Die Variablentabelle

Die Startadresse der Variablentabelle kann über PEEK aus dem systembenutzten RAM gelesen werden.
An der Adresse &AE85/6 (664,6128:&AE68/9) steht die Startadresse der Variablentabelle.
Die Variablentabelle ist nach folgender Datenstruktur aufgebaut:

Anzahl Bytes	Bedeutung
2	Verkettungsadresse
bis max.39	Variablenname (ohne letzten Buchstaben), letzter Buchstabe des Variablennamens +128
1	Typ der Variablen
2, 3 oder 5	Wert der Variablen
	Anfang der nächsten Eintragung nach gleicher Struktur.

Auf die Bedeutung der Verkettungsadresse gehen wir gleich ein. Der Namen der Variablen wird prinzipiell als String, d.h. in Form seiner ASCII-Codes gespeichert. Leider ist dieses Prinzip nicht durchgängig. Im Variablennamen enthaltene Ziffern werden nicht durch ihre ASCII-Codes dargestellt.

Um das Ende des Variablennamens anzuzeigen, ist beim Code für den letzten Buchstaben (bzw. Ziffer) Bit 7 gesetzt oder an-

ders ausgedrückt 128 zum Code addiert. Der Typ der Variablen wird durch eine Ziffer gekennzeichnet. Es gilt:

1 Integer
2 String
4 Real

Wie Sie wissen, wird bei Variablennamen Groß- und Kleinschreibung nicht unterschieden. Die Variablennamen werden immer in Großbuchstaben abgespeichert. Die Umwandlung von evtl. vorhandenen Kleinbuchstaben wird durch das Löschen von Bit 5 des Codes erreicht. Probieren Sie:

```
PRINT CHR$(ASC(b) AND NOT 2^5)
```

Durch diese Operationen werden alle Kleinbuchstaben in Großbuchstaben umgewandelt. Ist nun eine Ziffer im Variablennamen, so funktioniert diese Umwandlung natürlich nicht. Das Ergebnis ist ein Steuercode, der kleiner als 32 ist. Vor der Ausgabe des Zeichencodes muß also durch OR &20 Bit 5 gesetzt werden. Der Name wird dann in Kleinbuchstaben mit den korrekten Ziffern ausgegeben. OR &20 wandelt Groß- in Kleinbuchstaben um.

Die in der Variablentabelle verwendete Typenkennziffer ist gleich der Länge des Wertes der Variablen minus 1 (2, 3 bzw. 5 Bytes).

Der Wert, dessen Länge wie beschrieben 2, 3 oder 5 Bytes ist, enthält auf die oben behandelte Weise verschlüsselt den Wert (bei Zahlen) oder die Adresse des "Wertes" (bei Strings).

Anhand dieser Struktur wäre es nun möglich, eine beliebige Variable in der Tabelle zu suchen, indem einfach alle Variablennamen auf Übereinstimmung mit dem gesuchten geprüft werden. Das ist aber, besonders bei umfangreichen Programmen zu zeitaufwendig.

Das Auffinden eines Variablennamens innerhalb der Tabelle wird durch die Benutzung verketteter Listen extrem beschleu-

nigt. Um dies zu verwirklichen, wird die am Anfang jedes Eintrags stehende Verkettungsadresse gebraucht.

Das Prinzip der Datenverwaltung durch verkettete Listen ist, daß jeder Datensatz, in unserem Fall die Eintragung der Variablen, einen Zeiger zugeordnet bekommt, der auf den nächsten relevanten Datensatz zeigt. Das ist die Verkettungsadresse.

Beim Schneider sind alle Variablen, die mit demselben Anfangsbuchstaben beginnen(!), auf diese Weise miteinander verkettet. Dadurch reduziert sich die Suchzeit für eine Variable auf ca. das 1/26-fache, da es 26 unabhängig voneinander verkettete Listen gibt (zu jedem Buchstaben eine). Die verketteten Listen benötigen einen etwas größeren Platz. Doch macht sich das auf jeden Fall bezahlt.
Um eine verkettete Liste zu verwalten, sind zwei zusätzliche Informationen erforderlich:

> Zuerst wird die Adresse des ersten und des letzten Listenelementes benötigt. Diese Elemente sind an kein anderes "gekettet", deshalb müssen ihre Adressen separat gespeichert werden. Die Adresse des letzten Elementes ist bei der hier verwendeten Methode nicht notwendig, da vereinbart ist, daß die Verkettungsadresse mit dem Wert 0 bedeutet, daß das letzte Element erreicht ist. Die Adresse des ersten Listenelementes der Liste jedes Buchstaben wird jeweils auf aktuellem Stand im RAM gespeichert.

Das Suchen einer Variablen geschieht dann nach folgendem Schema:

1) Ersten Buchstaben des Namens holen
2) Zum Buchstaben gehörende Anfangsadresse des ersten Elementes holen
3) Nächste Verkettungsadresse lesen und speichern
4) Variablennamen auf Gleichheit prüfen
5) Wenn Variablennamen ungleich, dann mit gespeicherter Verkettungsadresse (wenn ungleich 0) bei Punkt drei weitermachen, wenn die Verkettungsadresse gleich 0 ist, dann nicht gefunden

6) Wenn die Variablennamen gleich sind, dann gefunden

Da die Variablentabelle verschiebbar sein muß, weil sie z.B. bei jeder Veränderung des BASIC-Programms verschoben wird, sind die Verkettungsadressen immer als Differenz zur Startadresse der Tabelle abgespeichert.

Auf dieser Struktur der verketteten Listen der Variablentabelle ist das folgende Programm "DUMP" aufgebaut.

9.2 DUMP - Ausgabe aller Variablenwerte

DUMP gibt alle Variablen, die mit den eingegebenen Anfangsbuchstaben beginnen, mit ihren Werten aus.

Da DUMP mit RSX eingebunden ist (siehe Kapitel RSX für genauere Informationen), muß vor der Eingabe des Befehlswortes DUMP der "Strich" (Shift+@) eingegeben werden.
Sollen alle Variablen ausgegeben werden, kann dann einfach RETURN gedrückt werden. Um bestimmte Buchstabenbereiche einzugeben, wird zuvor der "'" (Shift+7) eingegeben. Die Eingabe der Buchstabenbereiche erfolgt dann wie bei den Befehlen DEFINT, DEFREAL und DEFSTR. Ein kompletter Befehl wäre also zum Beispiel:

```
|DUMP'A-C,F-L,X,Y
```

Das Assemblerlisting des Programms gilt gleichzeitig für den XREF Befehl. Lassen Sie sich dadurch nicht stören, und überspringen Sie die XREF betreffenden Zeilen. Wir werden sie später noch besprechen.

Um intern arithmetische Operationen durchzuführen, gibt es noch die sog. FACs (Fließkomma Akkumulatoren). Da das Rechnen mit Realzahlen die Behandlung von 5 Bytes erfordert, ist die Speicherung in Registern nicht mehr sinnvoll. D.h., daß grundsätzlich alle Zahlen betreffenden Operationen mit den FACs durchgeführt werden.

Der FAC ist ein Bereich im RAM, der 6 Bytes lang ist. Das erste Byte des FAC enthält den Typ der aktuell dort gespeicherten Variablen (Adresse &B0C1 für 464). Die folgenden Bytes enthalten den Wert in der jeweiligen Darstellung. Das Typenkennzeichen im FAC entspricht genau der Anzahl der zur Speicherung des Wertes benötigten Bytes, also 2 für INT, 3 für String und 5 für Real.
Der FAC wird benutzt, um mit der "PRINT FAC" Routine den Wert auszudrucken.

Das Assemblerlisting ist absichtlich sehr ausführlich dokumentiert, um auch dem Maschinenspracheeinsteiger eine gute Lesbarkeit zu ermöglichen.

```
A000             10                    ; XREF
A000             20                    ; [Cross]-REFference
A000             30
; gibt BASIC Zeilennummern aus
A000             40
; in denen bestimmte Variablen
A000             50                    ; enthalten sind
A000             60                    ; Format :
A000             70
; "|xref'<Buchstabenbereich(e)>
A000             80                    ; z.B : "|xref'c,f,i,w-z
A000             90                    ;
A000            100                    ; DUMP
A000            110
; gibt alle definierten Vari-
A000            120                    ; ablen und ihren Wert aus
A000            130                    ; Format :
A000            140
; "|dump'<Buchstabenbreich(e)>
A000            150                    ; Beispiel siehe oben
A000            160                    ;
A000            170
; Druckausgabe : POKE &ac06,8
A000            180                    ; (464: poke &ac21,8) direkt
A000            190                    ; vor Befehl
A000            200
A000            210                    ; Einbindung mit RSX
A000 010000     220  DEFRSX LD    bc,tabrsx
A003 210000     230         LD    hl,kernal
A006 CDD1BC     240         CALL  &bcd1 ; Log in Extention
A009 3EC9       250         LD    a,&c9 ; RET
A00B 3200A0     260         LD    (defrsx),a
; Redefinition verhindern
A00E C9         270         RET
**** Zeile  220  : TABRSX=&A00F
A00F 0000       280  TABRSX DW    table
A011 C30000     290         JP    xref
A014 C30000     300         JP    dump
```

```
**** Zeile  280  : TABLE=&A017
A017 585245   310  TABLE  DM   "XRE"
A01A C6       320         DB   &c6 ; "F"+&80
A01B 44554D   330         DM   "DUM"
A01E D0       340         DB   &d0 ; "P"+&80
A01F 00       350         DB   0
**** Zeile  230  : KERNAL=&A020
A020          360  KERNAL DS   4
A024          370
**** Zeile  290  : XREF=&A024
A024 3E01     380  XREF   LD   a,1
A026 320000   390  AUFRUF LD   (prgken),a
A029 DF       400         RST  &18 ; Aufruf mit Far Call,
A02A 0000     410         DW   vektor ; da upper ROM enabled
A02C C9       420         RET   ; sein soll
A02D          430
**** Zeile  410  : VEKTOR=&A02D
A02D 0000     440  VEKTOR DW   start
A02F FD       450         DB   253 ; LROM off / UROM on
A030          460
**** Zeile  300  : DUMP=&A030
A030 3E00     470  DUMP   LD   a,0
A032 18F2     480         JR   aufruf
A034          490
A034          500
;  CPC       6128    464    664
A034          510  ALBAPC EQU  &ae58
; &ae75 , &ae58  zwischenspeicher fuer BASIC PC
A034          520  BASPC  EQU  &ae1d
; &ae36 , &ae1d  Original BASIC PC
A034          530  TESBUC EQU  &ff92
; &ff71 , &ff92  Test auf Buchstabe und UPPER
A034          540  SYNTER EQU  &d0d7
; &d07b , &d0da  Syntax error ausgeben
A034          550  GTVPTR EQU  &d619
; &d5db , &d61c  get variablentabellepointer
A034          560  BADOBC EQU  &e9b9
; &e8ff , &e9be  durchlaufe Basicprogramm und springe jeweil
s nach Adresse BC
```

```
A034            570  CHKBRK EQU  &c472
; &c43c , &c475  check for Break
A034            580  LNFEED EQU  &c398
; &c34e , &c39b  Line Feed ausgeben
A034            590  SKPCMD EQU  &e9fd
; &e943 , &ea02  skip command
A034            600  VAINIT EQU  &d6ec
; &d6b3 , &d6ef  traegt Variable in die Var.Tab. ein
A034            610  CHRGET EQU  &de2c
; &dd3f , &de31  holt naechstes Byte
A034            620  CHRGOT EQU  &de37
; &dd4a , &de3c  liest letztes Byte
A034            630  CHRNEX EQU  &de25
; &dd37 , &de2a  prueft folgendes Byte
A034            640  CHKKOM EQU  &de41
; &dd55 , &de46  prueft auf Komma
A034            650  PRINT  EQU  &c3a0 ; &c356 , &c3a3
A034            660  PTLNNU EQU  &ef44
; &ee79 , &ef49  print line number
A034            670  VARFAC EQU  &ff6c
; &ff4b , &ff6c  kopiere Variable in FAC
A034            680  PRTFAC EQU  &f2d5
; &f236 , &f2da  print FAC
**** Zeile  390  : PRGKEN=&A034
A034            690  PRGKEN DS   2
A036            700  WRTADR DS   2
A038            710
**** Zeile  440  : START=&A038
A038 3A34A0     720  START  LD   a,(prgken)
A03B FE00       730         CP   0 ; Dump ?
A03D 28FE       740         JR   z,weiter
A03F 010000     750         LD   bc,initva
; alle Variablen eintragen beim
A042 CDB9E9     760         CALL badobc
; BASIC Programm durchsuchen
**** Zeile  740  : WEITER=&A045
A045 2A58AE     770  WEITER LD   hl,(albapc)
A048 0641       780         LD   b,&41 ; "A" Default Start
A04A 0E5A       790         LD   c,&5a ; "Z" Default End
```

```
A04C CD37DE    800          CALL chrgot ; Ende ??
A04F B7        810          OR   a ; wenn ja,
A050 28FE      820          JR   z,anfang
; dann mit Defaultwerten beginnen
A052 23        830          INC  hl ; PC auf "'"
A053 CD25DE    840          CALL chrnex
; Test auf folgendes Zeichen
A056 C0        850          DB   &c0 ; Zeichen muss "'" sein
A057 7E        860  VONVOR LD   a,(hl) ; Buchstaben lesen
A058 CD92FF    870          CALL tesbuc
A05B 38FE      880          JR   c,ok ; Buchstabe ist ok
A05D C3D7D0    890  ERROR  JP   synter
**** Zeile  880  : OK=&A060
A060 47        900  OK      LD   b,a ; Buchstabe ist Startwert
A061 4F        910          LD   c,a ; und auch Defaultendwert
A062 CD2CDE    920          CALL chrget
A065 FE2D      930          CP   &2d ; "-" ??
A067 20FE      940          JR   nz,fangan
A069 CD2CDE    950          CALL chrget
A06C CD92FF    960          CALL tesbuc
A06F 30EC      970          JR   nc,error
; wenn kein Buchstabe
A071 4F        980          LD   c,a ; Buchstabe ist letzter
A072 CD2CDE    990          CALL chrget
**** Zeile  940  : FANGAN=&A075
A075 E5       1000  FANGAN PUSH hl ; PC retten
A076 CD0000   1010          CALL anfang
A079 E1       1020          POP  hl ; PC
A07A CD41DE   1030          CALL chkkom ; pruefen auf Komma
A07D 38D8     1040          JR   c,vonvor
A07F 2258AE   1050          LD   (albapc),hl
A082 C9       1060          RET
A083          1070
**** Zeile  820  : ANFANG=&A083
**** Zeile  1010  : ANFANG=&A083
A083 79       1080  ANFANG LD   a,c ; Ende
A084 90       1090          SUB  b ; minus Anfang
A085 38D6     1100          JR   c,error
; end<anf, dann Syntax Error
```

```
A087 78      1110  NEXBUC LD   a,b
; Variablen mit naechstem Buchstaben XREFen
A088 04      1120         INC  b ; fuer naechstes Mal
A089 C5      1130         PUSH bc ; erhoehen und retten
A08A CD19D6  1140         CALL gtvptr
; holt Zeigeradresse des Buchstabens
A08D 7E      1150  GLEBUC LD   a,(hl) ; HL wird mit der
A08E 23      1160         INC  hl ; Differenz vom Anfang der
A08F 66      1170         LD   h,(hl) ; Variablentabelle zur
A090 6F      1180         LD   l,a ; Variablen geladen
A091 B4      1190         OR   h
; wenn Differenz 0 ist, ist keine
A092 20FE    1200         JR   nz,namaus
; Variable mit dem aktuellen
A094 C1      1210         POP  bc
; Anfangsuchstaben vorhanden
A095 79      1220         LD   a,c ; aktuellen Stand (b)
A096 B8      1230         CP   b ; mit Ende (c) vergleichen
A097 30EE    1240         JR   nc,nexbuc
; noch nicht Ende, dann naechsten Buchstabe versuchen
A099 C9      1250         RET   ; sonst Fertig
A09A         1260
**** Zeile  1200  : NAMAUS=&A09A
A09A 09      1270  NAMAUS ADD  hl,bc
; Var.tab.start+Differenz
A09B E5      1280         PUSH hl
; ist Adresse der Verkettungsadresse
A09C C5      1290         PUSH bc ; Var.tab.start retten
A09D 23      1300         INC  hl ; Verkettung
A09E 23      1310         INC  hl ; ueberspringen
A09F 7E      1320  AUSGA  LD   a,(hl) ; Buchstaben lesen
A0A0 23      1330         INC  hl
A0A1 F5      1340         PUSH af
A0A2 E67F    1350         AND  &7f
; Bit 7 fuer Ausgabe loeschen
A0A4 FE20    1360         CP   32
; Zahlen im Variablennamen ?
A0A6 30FE    1370         JR   nc,allkla
; nein, dann alles klar
```

```
A0A8 F620    1380          OR   &20 ; Bit 5 bei Zahlen setzen
**** Zeile  1370  : ALLKLA=&A0AA
A0AA CDA0C3  1390  ALLKLA CALL print
A0AD F1      1400          POP  af ; gelesener Wert
A0AE 17      1410          RLA   ; Test ob Bit 7 gesetzt
A0AF 30EE    1420          JR   nc,ausga
; nein, dann weiter ausgeben
A0B1 3E20    1430          LD   a,&20 ; Leerzeichen
A0B3 CDA0C3  1440          CALL print
A0B6 7E      1450          LD   a,(hl) ; Typenkennziffer
A0B7 23      1460          INC  hl
A0B8 C601    1470          ADD  a,1
; Anzahl der Bytes = Typ fuer FAC
A0BA CD0000  1480          CALL typtes
A0BD F5      1490          PUSH af
A0BE 3A34A0  1500          LD   a,(prgken)
A0C1 B7      1510          OR   a
A0C2 CA0000  1520          JP   z,fordum
; Fortsetzung wenn DUMP
A0C5 F1      1530          POP  af
A0C6 C1      1540          POP  bc
A0C7 B7      1550          OR   a
A0C8 ED42    1560          SBC  hl,bc ; Adresse des Var.
A0CA 2236A0  1570          LD   (wrtadr),hl ; wertese fuer
A0CD C5      1580          PUSH bc
; spaeteren Vergleich speichern
A0CE 010000  1590          LD   bc,suchva
; Variablensuchroutine
A0D1 CDB9E9  1600          CALL badobc
; durchlaeuft BASIC Programm
A0D4 CD72C4  1610  SOFERT  CALL chkbrk ; Break Test
A0D7 CD98C3  1620          CALL lnfeed ; Line Feed
A0DA C1      1630          POP  bc ; anf/end Buchstaben
A0DB E1      1640          POP  hl ; Verkettungsadresse
A0DC 18AF    1650          JR   glebuc
; naechste Variable mit gleichem Buchstaben
A0DE         1660
A0DE         1670                ; Varaiableninitroutine
**** Zeile  750  : INITVA=&A0DE
```

```
A0DE E5      1680  INITVA PUSH hl ; PC
A0DF CDFDE9  1690         CALL skpcmd
; ueberliest einen Befehl
A0E2 D1      1700         POP  de
A0E3 FE02    1710         CP   2 ; Ende ?
A0E5 D8      1720         RET  c ; ja, naechste Zeile
A0E6 FE0E    1730         CP   &e ; Variable ??
A0E8 30F4    1740         JR   nc,initva
; nein, dann Weitersuchen
A0EA FE07    1750         CP   7
A0EC 28F0    1760         JR   z,initva
A0EE FE08    1770         CP   8
A0F0 28EC    1780         JR   z,initva
A0F2 EB      1790         EX   de,hl
A0F3 D5      1800         PUSH de ; PC nach skip
A0F4 CD2CDE  1810         CALL chrget
A0F7 CDECD6  1820         CALL vainit
; Namens Suche, Variable eintragen
A0FA E1      1830         POP  hl
A0FB 18E1    1840         JR   initva
A0FD         1850
A0FD         1860              ; Variablensuchroutine
**** Zeile  1590  : SUCHVA=&A0FD
A0FD E5      1870  SUCHVA PUSH hl ; PC
A0FE CDFDE9  1880         CALL skpcmd
; ueberliest einen Befehl
A101 D1      1890         POP  de
A102 FE02    1900         CP   2 ; Ende ?
A104 D8      1910         RET  c ; ja, naechste Zeile
A105 FE0E    1920         CP   &e ; Variable ??
A107 30F4    1930         JR   nc,suchva
; nein, dann Weitersuchen
A109 FE07    1940         CP   7
A10B 28F0    1950         JR   z,suchva ; nein
A10D FE08    1960         CP   8
A10F 28EC    1970         JR   z,suchva
A111 EB      1980         EX   de,hl
A112 D5      1990         PUSH de
A113 CD2CDE  2000         CALL chrget
```

```
A116 23       2010          INC  hl ; HL auf Startadresse Var
A117 5E       2020          LD   e,(hl)
A118 23       2030          INC  hl
A119 56       2040          LD   d,(hl)
A11A 2A36A0   2050          LD   hl,(wrtadr)
; mit der aktuellen
A11D B7       2060          OR   a
A11E ED52     2070          SBC  hl,de ; Adresse vergleichen
A120 E1       2080          POP  hl ; PC nach Skip
A121 20DA     2090          JR   nz,suchva
; ungleich, dann weitersuchen
A123 E5       2100          PUSH hl ; Adr.Typ Prog.
A124 2A1DAE   2110          LD   hl,(baspc) ; BASIC PC lesen
A127 7E       2120          LD   a,(hl)
A128 23       2130          INC  hl
A129 66       2140          LD   h,(hl)
A12A 6F       2150          LD   l,a
A12B CD44EF   2160          CALL ptlnnu ; Print Line Number HL
A12E 3E20     2170          LD   a,&20
A130 CDA0C3   2180          CALL print
A133 E1       2190          POP  hl
A134 18C7     2200          JR   suchva
A136          2210
**** Zeile  1520  : FORDUM=&A136
A136 F1       2220  FORDUM  POP  af ; FORtstzung DUMp
A137 FE03     2230          CP   3
A139 20FE     2240          JR   nz,nostri
A13B 46       2250          LD   b,(hl) ; Laenge String
A13C 97       2260          SUB  a ; Akku loeschen
A13D B8       2270          CP   b ; Laenge ist 0 ???
A13E 28FE     2280          JR   z,skip ; dann nichts ausgeben
A140 E5       2290          PUSH hl ; Descriptoradresse
A141 23       2300          INC  hl
A142 7E       2310          LD   a,(hl) ; low Byte
A143 23       2320          INC  hl
A144 66       2330          LD   h,(hl) ; high Byte
A145 6F       2340          LD   l,a
A146 7E       2350  NESTCH  LD   a,(hl)
A147 23       2360          INC  hl
```

```
A148 CDA0C3  2370          CALL print
A14B 10F9    2380          DJNZ nestch
A14D E1      2390          POP  hl
**** Zeile  2280  : SKIP=&A14E
A14E 3E03    2400  SKIP    LD   a,3 ; Laenge Descriptor
A150 18FE    2410          JR   skval
**** Zeile  2240  : NOSTRI=&A152
A152 E5      2420  NOSTRI  PUSH hl
A153 CD6CFF  2430          CALL VARFAC
; Typ setzen und VAR (HL) -> FAC
A156 E1      2440          POP  hl
A157 CDD5F2  2450          CALL prtfac ; print FAC
**** Zeile  2410  : SKVAL=&A15A
A15A C3D4A0  2460  SKVAL   JP   sofert
A15D         2470
**** Zeile  1480  : TYPTES=&A15D
A15D F5      2480  TYPTES  PUSH af ; gibt der Type
A15E E5      2490          PUSH hl
; entsprechendes Kenzeichen aus
A15F E607    2500          AND  7 ; nur Bit 0-2 betrachten
A161 EE27    2510          XOR  &27
A163 FE22    2520          CP   &22
A165 20FE    2530          JR   nz,ok1
A167 D601    2540          SUB  1
**** Zeile  2530  : OK1=&A169
A169 CDA0C3  2550  OK1     CALL print
A16C 3E20    2560          LD   a,&20 ; " "
A16E CDA0C3  2570          CALL print
A171 E1      2580          POP  hl
A172 F1      2590          POP  af
A173 C9      2600          RET

Programm :xdum
Start : &A000    Ende : &A173
Laenge : 0174
 0  Fehler
Variablentabelle :
DEFRSX A000  TABRSX A00F  TABLE  A017  KERNAL A020
XREF   A024  AUFRUF A026  VEKTOR A02D  DUMP   A030
```

ALBAPC	AE58	BASPC	AE1D	TESBUC	FF92	SYNTER	D0D7
GTVPTR	D619	BADOBC	E9B9	CHKBRK	C472	LNFEED	C398
SKPCMD	E9FD	VAINIT	D6EC	CHRGET	DE2C	CHRGOT	DE37
CHRNEX	DE25	CHKKOM	DE41	PRINT	C3A0	PTLNNU	EF44
VARFAC	FF6C	PRTFAC	F2D5	PRGKEN	A034	WRTADR	A036
START	A038	WEITER	A045	VONVOR	A057	ERROR	A05D
OK	A060	FANGAN	A075	ANFANG	A083	NEXBUC	A087
GLEBUC	A08D	NAMAUS	A09A	AUSGA	A09F	ALLKLA	A0AA
SOFERT	A0D4	INITVA	A0DE	SUCHVA	A0FD	FORDUM	A136
NESTCH	A146	SKIP	A14E	NOSTRI	A152	SKVAL.	A15A
TYPTES	A15D	OK1	A169				

Eine Eigenschaft der Variablentabelle haben wir noch nicht besprochen.

Auch Funktionen, die mit DEF FN definiert sind, sind in der Variablentabelle abgespeichert. Das Typenkennzeichen einer Funktionsdefinition ist &41, &42 oder &44. D.h., Bit 6 steht für "Funktion", das Low Byte (1, 2 oder 4) gibt in der bekannten Weise an, von welchem Typ das Ergebnis der Funktion ist.

Alle Funktionsnamen sind untereinander über Verkettungsadressen verbunden. An Adresse &AE04/5 (664/6128: &ADEB/C) steht die Adresse des ersten Listenelementes. Der "Wert" eines Funktionsnamens besteht aus 2 Bytes, die auf die Stelle im BASIC-Programm zeigen, an der die Funktionsdefinition steht.

Die RAM-Adressen des ersten Elementes für jeden Anfangsbuchstaben stehen an Adressen &ADD0 bis &AE03 (für 664/6128: &ADB7 bis &ADEA), wobei für jeden Buchstaben 2 Bytes benutzt werden. Beachten Sie bei der Benutzung dieser Pointer immer, das Sie nicht die Adresse selbst, sondern die Differenz der Adresse zur Startadresse der Variablentabelle &AE85/6 (664/6128: &AE68/9) angeben.

9.3 XREF (Cross REFerence)

Die Syntax des XREF-Befehls ist gleich der des DUMP Befehls. Da große Programmteile sowohl von DUMP als auch vom XREF benutzt werden, wurden sie gemeinsam in einem Programm verwirklicht. Wie bei DUMP werden durch die Schleifen NEXBUC und GLEBUC die erforderlichen Variablen aus der Tabelle herausgesucht.

Ganz am Anfang des Programms benutzt XREF eine sehr interessante Systemroutine, um alle Variablen in die Variablentabelle einzutragen. An Adresse &E8FF (6128: &E9B9 / 664: &E9BE) steht die Routine BADOBC, die ein BASIC-Programm zeilenweise durchläuft, und als Besonderheit, nachdem der Anfang einer Zeile gefunden ist, eine beliebige andere Routine aufruft, die dann die Zeile untersuchen kann o.ä. Die Adresse der

Routine, die die Untersuchung durchführen soll, wird bei Aufruf von BADOBC im BC-Register übergeben. Im XREF-Programm ist das einmal die Adresse der INITVA Routine, später noch einmal die der SUCHVA Routine.

Um die SUCHVA-Routine zu verstehen, müssen wir uns mit dem Aufbau eines BASIC-Programms, speziell mit der Ablage von Variablen im Programm, beschäftigen.

Vor jeder Variablen steht eine Kennziffer. Die Kennziffer gibt an, welchen Typs die Variable ist, und ob das Typenkennzeichen (%, $ oder !) mit angegeben wurde. Dabei gilt:

02 integer mit %
03 string mit $
04 real mit !
0B Integer
0C string
0D real

Auf die Kennziffer folgt die Differenz der Adresse der Variablen in der Tabelle zur Startadresse der Tabelle.
Auf diesen Adressenzeiger folgt direkt der Variablenname, wobei wie üblich, das Bit 7 des letzten Buchstaben gesetzt ist.

SUCHVA prüft zuerst, ob es sich um eine der Kennziffern handelt, wenn ja, wird der Name verglichen, und schließlich die Kennziffer umgewandelt und auch verglichen. Bei Übereinstimmung wird die BASIC Zeilennummer der aktuellen Zeile ausgegeben.

Für Programmierer, die nicht den Assembler aus dem Buch "Maschinensprache mit den CPCs" oder einen anderen besitzen, befindet sich im folgenden noch ein BASIC-Lader für das Programm.

Die Einbindung der neuen Befehle mit RSX erfolgt durch CALL &A000. Danach stehen DUMP und XREF im besprochenen Format zur Verfügung.

```
10 ' XREF und DUMP fuer 464
20 ' RSX Einbindung mit call &a000
30 FOR i=&A000 TO &A173
40 READ a$:w=VAL("&H"+a$)
50 s=s+w:POKE i,w:NEXT
60 IF s<> 49982 THEN PRINT"Fehler in Datas":END
70 PRINT"ok!":END
80 DATA 01,0F,A0,21,20,A0,CD,D1
90 DATA BC,3E,C9,32,00,A0,C9,17
100 DATA A0,C3,24,A0,C3,30,A0,58
110 DATA 52,45,C6,44,55,4D,D0,00
120 DATA FC,A6,0F,A0,3E,01,32,34
130 DATA A0,DF,2D,A0,C9,38,A0,FD
140 DATA 3E,00,18,F2,01,D0,5F,03
150 DATA 3A,34,A0,FE,00,28,06,01
160 DATA DE,A0,CD,FF,E8,2A,75,AE
170 DATA 06,41,0E,5A,CD,4A,DD,B7
180 DATA 28,31,23,CD,37,DD,C0,7E
190 DATA CD,71,FF,38,03,C3,7B,D0
200 DATA 47,4F,CD,3F,DD,FE,2D,20
210 DATA 0C,CD,3F,DD,CD,71,FF,30
220 DATA EC,4F,CD,3F,DD,E5,CD,83
230 DATA A0,E1,CD,55,DD,38,D8,22
240 DATA 75,AE,C9,79,90,38,D6,78
250 DATA 04,C5,CD,DB,D5,7E,23,66
```

```
260 DATA 6F,B4,20,06,C1,79,B8,30
270 DATA EE,C9,09,E5,C5,23,23,7E
280 DATA 23,F5,E6,7F,FE,20,30,02
290 DATA F6,20,CD,56,C3,F1,17,30
300 DATA EE,3E,20,CD,56,C3,7E,23
310 DATA C6,01,CD,5D,A1,F5,3A,34
320 DATA A0,B7,CA,36,A1,F1,C1,B7
330 DATA ED,42,22,36,A0,C5,01,FD
340 DATA A0,CD,FF,E8,CD,3C,C4,CD
350 DATA 4E,C3,C1,E1,18,AF,E5,CD
360 DATA 43,E9,D1,FE,02,D8,FE,0E
370 DATA 30,F4,FE,07,28,F0,FE,08
380 DATA 28,EC,EB,D5,CD,3F,DD,CD
390 DATA B3,D6,E1,18,E1,E5,CD,43
400 DATA E9,D1,FE,02,D8,FE,0E,30
410 DATA F4,FE,07,28,F0,FE,08,28
420 DATA EC,EB,D5,CD,3F,DD,23,5E
430 DATA 23,56,2A,36,A0,B7,ED,52
440 DATA E1,20,DA,E5,2A,36,AE,7E
450 DATA 23,66,6F,CD,79,EE,3E,20
460 DATA CD,56,C3,E1,18,C7,F1,FE
470 DATA 03,20,17,46,97,B8,28,0E
480 DATA E5,23,7E,23,66,6F,7E,23
490 DATA CD,56,C3,10,F9,E1,3E,03
500 DATA 18,08,E5,CD,4B,FF,E1,CD
510 DATA 36,F2,C3,D4,A0,F5,E5,E6
520 DATA 07,EE,27,FE,22,20,02,D6
530 DATA 01,CD,56,C3,3E,20,CD,56
540 DATA C3,E1,F1,C9


10 ' XREF und DUMP fuer 664
20 ' RSX Einbindung mit call &a000
30 FOR i=&A000 TO &A173
40 READ a$:w=VAL("&H"+a$)
50 s=s+w:POKE i,w:NEXT
60 IF s<> 50380 THEN PRINT"Fehler in Datas":END
70 PRINT"ok!":END
80 DATA 01,0F,A0,21,20,A0,CD,D1
```

```
90 DATA BC,3E,C9,32,00,A0,C9,17
100 DATA A0,C3,24,A0,C3,30,A0,58
110 DATA 52,45,C6,44,55,4D,D0,00
120 DATA 38,A0,3E,0D,3E,01,32,34
130 DATA A0,DF,2D,A0,C9,38,A0,FD
140 DATA 3E,00,18,F2,E1,D0,18,D8
150 DATA 3A,34,A0,FE,00,28,06,01
160 DATA DE,A0,CD,BE,E9,2A,58,AE
170 DATA 06,41,0E,5A,CD,3C,DE,B7
180 DATA 28,31,23,CD,2A,DE,C0,7E
190 DATA CD,92,FF,38,03,C3,DA,D0
200 DATA 47,4F,CD,31,DE,FE,2D,20
210 DATA 0C,CD,31,DE,CD,92,FF,30
220 DATA EC,4F,CD,31,DE,E5,CD,83
230 DATA A0,E1,CD,46,DE,38,D8,22
240 DATA 58,AE,C9,79,90,38,D6,78
250 DATA 04,C5,CD,1C,D6,7E,23,66
260 DATA 6F,B4,20,06,C1,79,B8,30
270 DATA EE,C9,09,E5,C5,23,23,7E
280 DATA 23,F5,E6,7F,FE,20,30,02
290 DATA F6,20,CD,A3,C3,F1,17,30
300 DATA EE,3E,20,CD,A3,C3,7E,23
310 DATA C6,01,CD,5D,A1,F5,3A,34
320 DATA A0,B7,CA,36,A1,F1,C1,B7
330 DATA ED;42,22,36,A0,C5,01,FD
340 DATA A0,CD,BE,E9,CD,75,C4,CD
350 DATA 9B,C3,C1,E1,18,AF,E5,CD
360 DATA 02,EA,D1,FE,02,D8,FE,0E
370 DATA 30,F4,FE,07,28,F0,FE,08
380 DATA 28,EC,EB,D5,CD,31,DE,CD
390 DATA EF,D6,E1,18,E1,E5,CD,02
400 DATA EA,D1,FE,02,D8,FE,0E,30
410 DATA F4,FE,07,28,F0,FE,08,28
420 DATA EC,EB,D5,CD,31,DE,23,5E
430 DATA 23,56,2A,36,A0,B7,ED,52
440 DATA E1,20,DA,E5,2A,1D,AE,7E
450 DATA 23,66,6F,CD,49,EF,3E,20
460 DATA CD,A3,C3,E1,18,C7,F1,FE
470 DATA 03,20,17,46,97,B8,28,0E
```

```
480 DATA E5,23,7E,23,66,6F,7E,23
490 DATA CD,A3,C3,10,F9,E1,3E,03
500 DATA 18,08,E5,CD,6C,FF,E1,CD
510 DATA DA,F2,C3,D4,A0,F5,E5,E6
520 DATA 07,EE,27,FE,22,20,02,D6
530 DATA 01,CD,A3,C3,3E,20,CD,A3
540 DATA C3,E1,F1,C9
```

```
10 ' XREF und DUMP fuer 6128
20 ' RSX Einbindung mit call &a000
30 FOR i=&A000 TO &A173
40 READ a$:w=VAL("&H"+a$)
50 s=s+w:POKE i,w:NEXT
60 IF s<> 50608 THEN PRINT"Fehler in Datas":END
70 PRINT"ok!":END
80 DATA 01,0F,A0,21,20,A0,CD,D1
90 DATA BC,3E,C9,32,00,A0,C9,17
100 DATA A0,C3,24,A0,C3,30,A0,58
110 DATA 52,45,C6,44,55,4D,D0,00
120 DATA 20,A0,0F,A0,3E,01,32,34
130 DATA A0,DF,2D,A0,C9,38,A0,FD
140 DATA 3E,00,18,F2,01,D0,D1,00
150 DATA 3A,34,A0,FE,00,28,06,01
160 DATA DE,A0,CD,B9,E9,2A,58,AE
170 DATA 06,41,0E,5A,CD,37,DE,B7
180 DATA 28,31,23,CD,25,DE,C0,7E
190 DATA CD,92,FF,38,03,C3,D7,D0
200 DATA 47,4F,CD,2C,DE,FE,2D,20
210 DATA 0C,CD,2C,DE,CD,92,FF,30
220 DATA EC,4F,CD,2C,DE,E5,CD,83
230 DATA A0,E1,CD,41,DE,38,D8,22
240 DATA 58,AE,C9,79,90,38,D6,78
250 DATA 04,C5,CD,19,D6,7E,23,66
260 DATA 6F,B4,20,06,C1,79,B8,30
270 DATA EE,C9,09,E5,C5,23,23,7E
280 DATA 23,F5,E6,7F,FE,20,30,02
290 DATA F6,20,CD,A0,C3,F1,17,30
```

```
300 DATA EE,3E,20,CD,A0,C3,7E,23
310 DATA C6,01,CD,5D,A1,F5,3A,34
320 DATA A0,B7,CA,36,A1,F1,C1,B7
330 DATA ED,42,22,36,A0,C5,01,FD
340 DATA A0,CD,B9,E9,CD,72,C4,CD
350 DATA 98,C3,C1,E1,18,AF,E5,CD
360 DATA FD,E9,D1,FE,02,D8,FE,0E
370 DATA 30,F4,FE,07,28,F0,FE,08
380 DATA 28,EC,EB,D5,CD,2C,DE,CD
390 DATA EC,D6,E1,18,E1,E5,CD,FD
400 DATA E9,D1,FE,02,D8,FE,0E,30
410 DATA F4,FE,07,28,F0,FE,08,28
420 DATA EC,EB,D5,CD,2C,DE,23,5E
430 DATA 23,56,2A,36,A0,B7,ED,52
440 DATA E1,20,DA,E5,2A,1D,AE,7E
450 DATA 23,66,6F,CD,44,EF,3E,20
460 DATA CD,A0,C3,E1,18,C7,F1,FE
470 DATA 03,20,17,46,97,B8,28,0E
480 DATA E5,23,7E,23,66,6F,7E,23
490 DATA CD,A0,C3,10,F9,E1,3E,03
500 DATA 18,08,E5,CD,6C,FF,E1,CD
510 DATA D5,F2,C3,D4,A0,F5,E5,E6
520 DATA 07,EE,27,FE,22,20,02,D6
530 DATA 01,CD,A0,C3,3E,20,CD,A0
540 DATA C3,E1,F1,C9
```

10. BASIC-Zeile vom BASIC aus erzeugen

Ein Befehl, der bisher in allen gängigen BASIC-Dialekten fehlt, ist der zum Erzeugen einer BASIC-Zeile vom Programm aus.

Dabei ist es gerade dieser Befehl, der einige vollkommen neue Aspekte in die BASIC-Programmierung bringt.

Gibt es einen Befehl, bei dem es vom Programm aus möglich ist, Programmzeilen zu erzeugen?

Wenn das möglich ist, liegt der Gedanke an Programme sehr nahe, die ihrerseits Programme erstellen, wobei die so erzeugten Programme wiederum Programme erzeugen usw. Wozu lernt man denn überhaupt noch das Programmieren, wenn es sowieso bald von Computern erledigt wird?

Soweit ist es glücklicherweise nocht nicht gekommen. Dennoch bietet der "Programmerzeugungsbefehl" ansatzweise in diesem Sinne hochinteressante Möglichkeiten. Doch wenden wir uns zunächst einmal dem Befehl selbst zu.

Bei der Eingabe von BASIC-Zeilen im Direktmodus werden diese quasi als String in einem Puffer gespeichert bis RETURN eingegeben wird. Danach wird versucht, den String als BASIC-Zeile (erkennbar an der vorn stehenden Zeilennummer!) zu interpretieren. Wenn das funktioniert, wird der String in das interne BASIC-Zeilenformat umgewandelt. Dabei werden z.B. erkannte Befehlswörter in Tokens umgewandelt und dann in dieser Kurzform gespeichert. Schließlich wird die so übersetzte BASIC-Zeile anhand ihrer Zeilennummer in das vorhandene Programm eingefügt. Dazu werden vorher alle Zeilen mit größerer Nummer im Speicher um das entsprechende Stück nach oben verschoben.

Um unsere Idee zu verwirklichen, benutzen wir dieselben Routinen, die beim oben beschriebenen Vorgang vom Interpreter benutzt werden. Wir brauchen lediglich vom Anwender die Adresse des Strings, der die zu erzeugende Zeile enthält, mit @Stringvariable an das Programm zu übergeben.

Bei der Übergabe von nur einem Wert steht dieser nach Aufruf mit CALL oder auch bei RSX-Erweiterungen im DE-Register zur Verfügung. DE enthält also die Stringdescriptoradresse des umzuwandelnden Strings. Nach dem Überlesen der Stringlänge im Stringdescriptor wird die Stringadresse gelesen und ins HL-Register geladen. Mit der CHRSKP-Routine werden evtl. vorhandene Leerzeichen und Cursorbewegungszeichen (HT und LF) überlesen. Wenn kein Nullbyte gefunden wurde, wird mit TESTER geprüft, ob am Anfang der Zeile eine Zeilennummer steht. Wenn ja, wird mit ASSEMB die Zeile übersetzt und eingefügt. Dabei wird das Ende der Zeile anhand von Nullbytes erkannt. Im folgenden steht das Assemblerlisting und darauffolgend der BASIC-Lader des eben beschriebenen Programms.

```
A000              10                ; liner
A000              20                ; erzeugt BASIC Zeilen
A000              30                ; 1. im Direktmodus
A000              40                ; 2. im Programm, wenn
A000              50                ; Zeilenno. > als aktuelle
A000              60                ; a$ = Zeile im ASCII-
A000              70                ; Code, Ende = Nullbyte
A000              80                ; Format : call &a000,@a$
A000              90
A000             100          ORG  &a000
A000             110
;    CPC      6128 ;  464   ,  664
A000             120  CHRSKP EQU  &de4d ; &dd61 , &de52
A000             130  TESTER EQU  &eecf ; &ee04 , &eed4
A000             140  ASSEMB EQU  &e7a5 ; &e6c6 , &e7aa
A000 DF          150          RST  &18 ; Far Call
A001 0000        160          DW   vektor ; da das BASIC ROM
A003 C9          170          RET   ; eingeschaltet sein muss
**** Zeile  160  : VEKTOR=&A004
A004 0000        180  VEKTOR DW   start
A006 FD          190          DB   253 ; low ROM off, upp ROM on
**** Zeile  180  : START=&A007
A007 EB          200  START  EX   de,hl
; uebergebene Adresse nach HL
A008 23          210          INC  hl ; Stringlaenge ueberlesen
A009 5E          220          LD   e,(hl) ; Lo Byte Stringadresse
A00A 23          230          INC  hl
A00B 56          240          LD   d,(hl) ; Hi Byte Stringadresse
A00C EB          250          EX   de,hl ; Stringadresse nach HL
A00D CD4DDE      260          CALL chrskp
A010 B7          270          OR   a
A011 C8          280          RET  z
A012 CDCFEE      290          CALL tester
A015 D0          300          RET  nc
A016 CDA5E7      310          CALL assemb
; Zeile uebersetzen und einfuegen; HL muss auf erstes Byte der
 ASCII Zeile zeigen
A019 C9          320          RET   ; Fertig !!!
```

```
Programm :liner
Start : &A000     Ende : &A019
Laenge : 001A
 0  Fehler
Variablentabelle :
CHRSKP DE4D  TESTER EECF  ASSEMB E7A5  VEKTOR A004
START  A007
```

```
10 REM BASIC Lader fuer Liner
20 FOR i=&A000 TO &A019
30 READ a$:w=VAL("&H"+a$)
40 s=s+w:POKE i,w:NEXT
50 IF s<> 4275 THEN PRINT"Fehler in Datas":END
60 '464: IF s<> 4123 THEN .....
70 '664: IF s<> 4290 THEN .....
80 PRINT"ok!":END
90 DATA DF,04,A0,C9,07,A0,FD,EB
100 DATA 23,5E,23,56,EB,CD,4D,DE
110 '664:..,..,..,..,..,..,52,DE
120 DATA B7,C8,CD,CF,EE,D0,CD,A5
130 '464:..,..,..,04,EE,..,..,c6
140 '664:..,..,..,D4,EE,..,..,AA
150 DATA E7,C9
160 '464:E6,C9
170 '664:E7,C9
```

Zur Demonstration der Funktionsweise des Befehls folgt hier ein kleines Programm.

```
10 MEMORY &9FFF: REM Ist Liner geladen?
20 ZEINU=100
30 Z$=STR$(ZEINU)+"REM Dies ist die Zeile"+STR$(ZEINU)
40 Z$=Z$+CHR$(0): REM Endekennzeichen
50 CALL &A000,@Z$
60 ZEINU=ZEINU+10
70 IF ZEINU<210 THEN 30
80 Z$=STR$(ZEINU)+"LIST"+CHR$(0)
90 CALL &A000,@Z$
```

Folgende Einschränkungen bestehen für die Benutzung des Programms:

- Die Zeilennummer der zu erzeugenden Zeile muß größer als die Zeilennummer sein, in der der jeweilige CALL-Befehl erfolgt.

- Der Befehl darf nicht im Inneren einer FOR-NEXT-Schleife oder WHILE-WEND-Schleife stehen.

Diese Einschränkungen sind unbedingt zu berücksichtigen, da sonst die Programmausführung nach CALL nicht an die richtige Stelle zurückfindet und/oder Variablen nicht wiedergefunden werden. Die Ursache dafür ist die Verschiebung des BASIC-Programmes für das Einfügen der neuen Zeile.

Die einfachste Anwendung für den Befehl ist sicherlich das Erzeugen von DATA-Zeilen für z.B. eine Dateiverwaltung. Mit Hilfe des Zeilenerzeugers können die Daten direkt im Programm gespeichert werden.

Im Buch "Maschinensprache für 464, 664, 6128" von DATA BECKER finden Sie ein Programm, das den Befehl zum automatischen Erzeugen eines BASIC-Laders aus den im Speicher befindlichen Codes verwendet. Eine professionelle Anwendung des Befehls bzw. des dahinterstehenden Prinzips könnte so aussehen:

Eine Softwarefirma vertreibt Buchhaltungsprogramme. Um nun nicht für jeden Kunden das Programm auf die jeweils spezifischen Bedürfnisse zugeschnitten umschreiben zu müssen, wird ein Softwareentwicklungsprogramm geschrieben, das alle Möglichkeiten der Anpassung vorsieht. Nach der Eingabe besonderer Wünsche erzeugt das Entwicklungssystem ein weitgehend individuell für den jeweiligen Fall zugeschnittenes Programm.

11. Grafik-Hardcopy

Damit Sie die mit dem 3-D-Funktionsplotter erstellten Zeichnungen auch auf das Papier bringen können, stellen wir Ihnen nun noch ein Hardcopy Programm vor. Das Programm läuft ohne Änderung auf dem Drucker EPSON FX-80 und Kompatiblen. Zur Anpassung an andere Druckertypen muß nur die Steuersequenz, die den Drucker auf Grafikmodus schaltet, geändert werden, vorausgesetzt, daß ein 8-Punkt Bitmustermodus existiert.

Für die Hardcopy wird direkt der Video-RAM des Schneiders von Adresse &C000 - &FFFF im Modus 2 ausgelesen. Der Bildschirmspeicher ist beim CPC auf die bekannte, etwas merkwürdige Art und Weise aufgebaut. Grundsätzlich gilt aber, wie bei fast allen anderen Rechnern, daß ein Byte im Bildschirmspeicher auf dem Bildschirm einer gewissen Anzahl (im MODE 2 sind es 8) von Punkten entspricht. Das Problem besteht nun darin, daß der Drucker jeweils 8 untereinander und nicht, wie beim Bildschirmspeicher, nebeneinanderliegende Punkte druckt.

Betrachten wir als Beispiel das folgende Sonderzeichen. Dieses Zeichen soll auf dem Drucker ausgegeben werden.

								Bildschirm-speichercodes:
0	0	0	0	0	0	0	0	= 00
0	0	1	0	1	0	0	0	= 40
0	1	0	0	0	1	0	0	= 68
1	0	1	1	1	0	1	0	= 186
0	1	0	0	0	1	0	0	= 68
0	0	1	0	1	0	0	0	= 40
0	0	0	1	0	0	0	0	= 16
1	1	1	1	1	1	1	0	= 254
Druckercodes:	17	41	85	19	85	41	17	0

Das Zeichen wird durch

```
MODE 2
FOR I=&C000 TO &F8000 STEP &800
POKE I,A:NEXT
DATA 16,40,68,16,68,40,16,254
```

auf dem Bildschirm ausgegeben.

Für die Druckerausgabe müssen die spaltenweise gebildeten Werte gesendet werden.

```
PRINT#8, CHR$(27);"*"CHR$(1);CHR$(8);CHR$(0);
FOR I=0 TO 7:READ A
PRINT#8, CHR$(A);:NEXT
DATA 17,41,85,19,85,41,17,0
```

Der erste PRINT#8-Befehl ist die Steuercodefolge für den EPSON FX-80, die die Ausgabe von 8 Grafikcodes im Bitmustermodus ankündigt.

Zum Jahresende 1985 wird das "Große Epson Druckerbuch" von DATA BECKER erscheinen, in dem alle Steuerbefehle der meisten EPSON-Drucker und vieles mehr beschrieben werden.

Das Hardcopy-Programm muß nun den gesamten Bildschirmspeicher Schritt für Schritt umwandeln, um die "Spaltenwerte" senden zu können. Da das im BASIC einige Stunden dauern kann, ist hier nur eine Lösung in Maschinensprache sinnvoll.

Das Harddcopy-Programm benutzt einige sehr nützliche Systemroutinen. Wir werden es im folgenden kurz beschreiben.

Beim Schneider ist die Hardcopy ohne hardwaremäßige Änderungen etwas schwieriger als üblich. Der Grund dafür ist, daß die Schneider-Entwickler zwar eine Centronics-Schnittstelle für den Drucker vorgesehen haben, unverständlicherweise aber nur 7 Daten-Bit-Leitungen belegt haben. Normalerweise sind dies natürlich 8 Leitungen, da ein Byte ja 8 Bits hat und der Drucker byteweise angesteuert wird.

Solange man sich nur mit dem Druck von Texten und Listings beschäftigt, ist diese Einschränkung auch nicht weiter störend. Bei der Hardcopy bewirkt das fehlende achte Bit jedoch einen weißen Strich im erzeugten Bild, der alle acht Punktzeilen auftritt.

Das Programm muß dieses Manko also ausgleichen indem, immer nur sieben Punktzeilen gedruckt werden und auch nur ein Zeilenvorschub von sieben Punktbreiten erzeugt wird. Da aber der Bildschirmspeicher des Schneiders in vertikaler Richtung in Einheiten zu je acht Byte organisiert ist und da auch jedes Zeichen acht Bit hoch ist, sind etwas umfangreichere Berechnungen erforderlich, um jeweils nur sieben Bits in einer annehmbaren Geschwindigkeit zu drucken.

Der Bildschirm besitzt 25 Zeilen zu je acht Punktreihen, also 200 Punktreihen. Werden je sieben Punktreihen abgearbeitet, so müssen 28*7 Punktreihen gesendet werden, wobei das Problem auftritt, daß am Ende noch 4 (200-28*7) Punktreihen übrig sind. Diese müssen gesondert behandelt werden. Weiterhin hat der Bildschirm in horizontaler Richtung 640=&280 Punkte.

Dem FX-80 (und kompatible) wird beim Einschalten des Bildschirmmodus mitgeteilt, wieviele Grafikbytes empfangen werden sollen. Das Low-Byte von &280 ist &80. Bei diesem Byte ist das achte Bit gesetzt, also kann dieser Wert nicht gesendet werden. Um dieses Problem ohne großen Mehraufwand lösen zu können, werden dem Grafikdrucker nur &27F Grafikbytes angekündigt und gesendet. Das bedeutet, daß die letzte Punktreihe des Bildschirmes nicht auf der Hardcopy erscheint. Das ist eigentlich ein Behelf, jedoch wird man in fast allen Fällen auf die letzte Punktreihe verzichten können.

Doch nun zum Programm selbst.

Zum Auslesen der jeweils sieben untereinanderliegenden Bytes wird eine Tabelle aufgebaut, in der die Adressen der sieben aktuellen Bytes gespeichert sind. Die achte Adresse in der Tabelle enthält immer die Adresse des ersten von den sieben Bytes, die

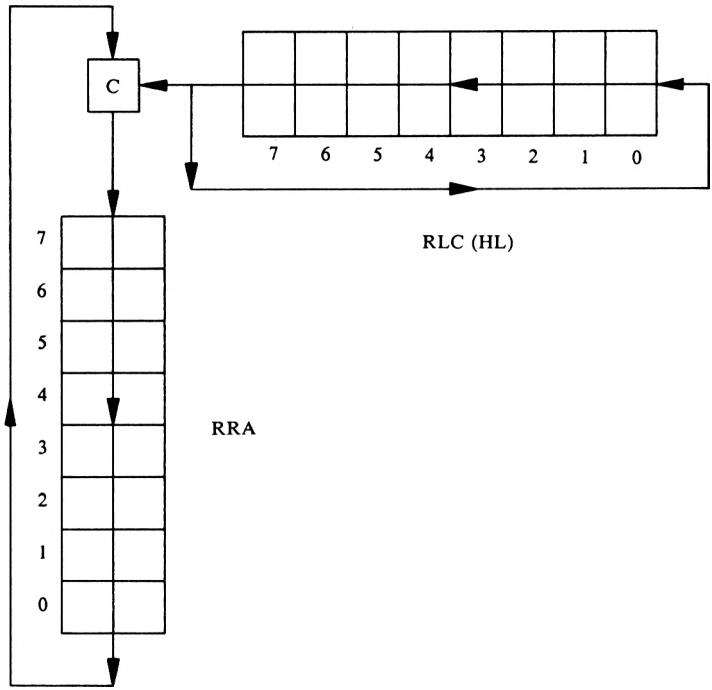
als nächstes behandelt werden sollen. Diese Tabelle wird ab Label NELI1 für die jeweils nächste siebener-Zeile neu erzeugt. Dabei wird die Systemroutine SCR NEXT LINE benutzt. Sie erhöht die in HL übergebene Bildschirmadresse so, daß sie auf die nächste Punktzeile des Bildschirmes zeigt. In der Schleife Label NEBIT findet nun die Umwandlung von waagerechten 8-Bit- ins senkrechte 7-Bit-Format statt.

Dazu wird, angefangen mit dem letzten Tabellenelement (TABLET), die jeweilige Byteadresse ins HL-Register geladen. Durch RLC (HL) wird nun das jeweilige Byte nach links rotiert, wobei das MSB (höchstwertige Bit) zusätzlich ins Carry geladen wird.

Der Bildpunkt ist nun also im Carry gespeichert und wird durch RRA in den Akku hineinrotiert. Dann beginnt die Schleife von vorn. Nach sieben Durchläufen enthält der Akku der Reihe noch die sieben höchsten Bits (=Punkte) der Bildschirmzeilen.

Diese Schleife ist ein hervorragendes Beispiel für die Leistungsfähigkeit der Maschinenbefehle bei einem geschickten Einsatz.

Überlegen Sie sich die Funktionsweise der Schleife noch einmal anhand der folgenden Skizze:



Nach dem Ende der NEBIT-Schleife wird der Akkuinhalt nochmals rotiert. Damit unsere ermittelte Bitmatrix nicht mehr das Bit Nr. 7, das nicht gesendet werden kann, belegt. Schließlich wird das erzeugte Byte an den Drucker ausgegeben.

Dieser Vorgang läuft insgesamt achtmal bei jedem Byte ab. Damit ist jedes Bit schrittweise in den Akku rotiert worden. Ein interessanter Nebeneffekt dieser Programmmethode ist, daß Sie bei genauem Hinsehen die Bytes auf dem Bildschirm wirklich rotieren sehen. Meist ist das jedoch nur am Anfang einer Zeile zu sehen, da dann das Programm "auf den Drucker warten muß".

Sind alle acht Bit eines Bytes gesendet, so wird durch die Schleife ab NEBY1 mit Hilfe der Systemroutine SCR NEXT BYTE jedes Tabellenelement auf das nächste Byte gesetzt. Bei der Übergabe der aktuellen Bildschirmadresse im HL-Register erhöht SCR NEXT LINE den Wert von HL, so daß er das nächste Byte in derselben Zeile adressiert.

Anhand des Bytezählers C wird, solange nicht das Ende der Zeile erreicht ist, wieder zur Umsetzung und Ausgabe der neuen adressierten Bytes gesprungen.

Wurde das Ende einer Zeile erreicht, so wird mittels des Zählers E ermittelt, ob weitere Zeilen zu drucken sind. Wenn dem so ist, wird zum Label NELINE verzweigt. Dort beginnt der gesamte Vorgang von neuem. Handelt es sich um die letzte zu druckende Zeile (LD A,E; CP 1), dann wird die Anzahl der untereinander auszugebenden Bits auf 4 gesetzt und die Tabelle ab der Mitte nur mit vier Startadressen beschrieben.

Hier haben wir außerdem noch einen kleinen Programmtrick angewendet. Nachdem mit LD DE,TABMIT und LD B,4 die Tabellendefinitionswerte für die letzte Zeile gesetzt wurden, muß natürlich der Befehl LD DE,TABANF übersprungen werden. Normalerweise geschieht dies durch einen JR-Befehl. Kürzer ist jedoch, ein Byte mit Wert &21 anstelle des JR-Befehles abzulegen. Der Rechner interpretiert &21 als Opcode des Befehls LD HL,nn. Damit sind die beiden folgenden Bytes diesem Befehl zugehörig. Das dritte Byte ist das High Byte von TABANF, also &A0. &A0 steht für den Befehl AND B. Beide so erzeugten Befehle schaden unserem Programmablauf nicht und verändern auch keine wichtigen Registerinhalte. Nach diesen beiden Befehlen wird nun die reguläre Programmausführung mit dem Befehl LD HL,(TABEND) fortgesetzt. Der gewünschte Effekt, nämlich das Überspringen des Befehles LD DE,TABEND, wird auf diese Weise erreicht.

Zwei Unterprogramme sind noch erwähnenswert: TABOUT und PRINT.

TABOUT steht für Tabellenausgabe. Dabei sind Tabellen von Steuerbefehlen an den Drucker gemeint. Die erste Steuersequenz ist PRINT. Sie setzt den Zeilenvorschub auf 7/72 Zoll und sendet einen Zeilenvorschub. Um diese Steuersequenz an den Drucker zu senden, wird die Adresse des ersten Bytes der Liste nach HL geladen und dann TABOUT aufgerufen. TABOUT er-

kennt das Ende einer Sequenz an dem am Ende stehenden Nullbyte.

Außerdem gibt es die Sequenzen PRLIIN und PRREIN. PRLIIN (Printer Line Init) sendet die Steuersequenz, die den Drucker für 639 Bytes (also für eine Zeile) in den 8-Bit Nadel Bituntermodus versetzt. PRREIN (Printer Re-init) schließlich normiert den Drucker wieder für den Standardbetrieb.

Damit Sie das Hardcopyprogramm auch an andere Drucker anpassen können, haben wir jeweils noch vier Bytes bei jeder Sequenz freigelassen. Sollte Ihr Drucker im übrigen keinen Zeilenvorschub ausführen, so ersetzen Sie den Code 24 der PRLIIN-Sequenz durch den Code 10 (LF).

Die Print-Routine erledigt die Druckausgabe. Zuerst wird der Zähler PRTCOU erniedrigt und zurückgesprungen, wenn der Zähler Null ist. Dadurch werden nur 639 anstelle von 640 Bytes pro Zeile gesendet. Dann wird mit MCPRBU getestet, ob der Drucker "busy" (engl.: beschäftigt) ist. Wenn ja, wird weiter getestet, ansonsten wird mit MCPRCH das Byte, das im Akku enthalten ist, an den Drucker gesendet.

```
A000             10                  ; Hardcopy von H.D. 18/10/85
A000             20
; fuer Epson FX80 und Kompatible
A000             30
; lauffaehig auf CPC 464,664 und 6128
A000             40                   ; ohne Aenderungen
A000             50          ORG   &a000
A000             60  SCGELO  EQU   &bc0b ; Screen Get Location
A000             70  SCNELI  EQU   &bc26 ; Screeen next Line
A000             80  SCNEBY  EQU   &bc20 ; Screen next Byte
A000             90  MCPRCH  EQU   &bd2b ; MC print Character
A000            100  MCPRBU  EQU   &bd2e ; MC Pinter Busy ?
A000            110
A000 210000     120          LD    hl,prinit ; printer Initia-
A003 CD0000     130          CALL  tabout ; lisierung
A006 CD0BBC     140          CALL  scgelo
A009 57         150          LD    d,a
A00A 1E00       160          LD    e,0
A00C 19         170          ADD   hl,de
; Adresse des Punktes oben links
A00D 220000     180          LD    (tabend),hl
A010 1E1D       190          LD    e,29 ; Zeilen zaehler
A012 3E07       200          LD    a,7
A014 320000     210          LD    (bitanz),a
; Bits pro Druckzeile
A017 D5         220  NELINE  PUSH  de ; Zeilenzaehler retten
A018 0608       230          LD    b,8
; 8 Zeilenstartadresse aendern
A01A 7B         240          LD    a,e ; wenn nicht
A01B FE01       250          CP    1 ; letzte Zeile
A01D 20FE       260          JR    nz,ok ; nicht, alles ok
A01F 3E04       270          LD    a,4 ; sonst nur 4 Bit pro
A021 320000     280          LD    (bitanz),a ; Druckzeile
A024 110000     290          LD    de,tabmit ; und Tabelle mit
A027 0604       300          LD    b,4 ; nur 4 Elementen fuellen
A029 21         310          DB    &21
; naechsten Befehl "zerstoeren"
**** Zeile  260  : OK=&A02A
```

```
A02A 110000    320  OK      LD   de,tabanf
A02D 2A0000    330          LD   hl,(tabend) ; HL ist neue erste
A030 EB        340  NELI1   EX   de,hl
; Eintagung in die Tabelle
A031 73        350          LD   (hl),e
A032 23        360          INC  hl
A033 72        370          LD   (hl),d
A034 23        380          INC  hl
A035 EB        390          EX   de,hl
A036 CD26BC    400          CALL scneli
; naechste Zeilenadresse
A039 10F5      410          DJNZ neli1 ; holen und speichern
A03B 210000    420          LD   hl,prliin ; Bit Mustermodus
A03E CD0000    430          CALL tabout ; einschalten
A041 217F02    440          LD   hl,&27f ; aber nur
A044 220000    450          LD   (prtcou),hl ; 639 Bytes senden
A047 015008    460          LD   bc,&850 ; c=Bytezaehler=80
A04A C5        470  NEBY    PUSH bc
A04B 3A0000    480          LD   a,(bitanz) ; Anzahl der
A04E 47        490          LD   b,a ; Druckbits
A04F 97        500          SUB  a ; Akku loeschen
A050 210000    510          LD   hl,tablet
; mit letztem Tabellen-
A053 56        520  NEBIT   LD   d,(hl) ; Element beginnen
A054 2B        530          DEC  hl
A055 5E        540          LD   e,(hl) ; Byte adresse
A056 2B        550          DEC  hl ; lesen
A057 EB        560          EX   de,hl
A058 CB06      570          RLC  (hl) ; Byte rotieren
A05A 1F        580          RRA   ; Caay in den Akku rotieren
A05B EB        590          EX   de,hl
A05C 10F5      600          DJNZ nebit ; naechstes Bit
A05E 1F        610          RRA   ; Bit 7 nicht benutzen
A05F CD0000    620          CALL print ; ausgeben
A062 C1        630          POP  bc
A063 10E5      640          DJNZ neby ; 8 mal pro Byte
A065 210000    650          LD   hl,tabanf ; 7 Tabellenelemente
A068 0607      660          LD   b,7 ; um je ein
A06A 5E        670  NEBY1   LD   e,(hl) ; Byte nach
```

```
A06B 23       680          INC  hl ; rechts erhoehen
A06C 56       690          LD   d,(hl)
A06D 2B       700          DEC  hl
A06E EB       710          EX   de,hl
A06F CD20BC   720          CALL scneby
A072 EB       730          EX   de,hl
A073 73       740          LD   (hl),e
A074 23       750          INC  hl
A075 72       760          LD   (hl),d
A076 23       770          INC  hl
A077 10F1     780          DJNZ neby1
A079 0608     790          LD   b,8
A07B 0D       800          DEC  c ; noch nicth Zeilenende ?
A07C 20CC     810          JR   nz,neby ; dann naechstes Byte
A07E D1       820          POP  de
A07F 1D       830          DEC  e ; Letzte Zeile ?
A080 2095     840          JR   nz,neline ; nein, dann naechste
A082 210000   850          LD   hl,prrein
; Printer re-initialisieren
A085 18FE     860          JR   tabout ; dann ende
A087          870               ;
**** Zeile  130  : TABOUT=&A087
**** Zeile  430  : TABOUT=&A087
**** Zeile  860  : TABOUT=&A087
A087 7E       880  TABOUT LD   a,(hl) ; Tabelle an
A088 FE00     890          CP   0 ; Adresse HL
A08A C8       900          RET  z ; bis zum
A08B 23       910          INC  hl ; Nullbyte
A08C E5       920          PUSH hl ; ausgeben
A08D CD0000   930          CALL print
A090 E1       940          POP  hl
A091 18F4     950          JR   tabout ; naechstes Byte
A093          960               ;
**** Zeile  620  : PRINT=&A093
**** Zeile  930  : PRINT=&A093
A093 2A0000   970  PRINT  LD   hl,(prtcou) ; Byte Zaehler
A096 2B       980          DEC  hl ; erniedrigen
A097 220000   990          LD   (prtcou),hl
; und wiederspeichern
```

```
A09A 4F       1000          LD   c,a
; Charakter zwischenspeichern
A09B 7C       1010          LD   a,h ; Test ob
A09C B5       1020          OR   l ; Bytezaehler
A09D C8       1030          RET  z ; gleich null ist
A09E 79       1040          LD   a,c ; nein, dann Akku ausgeben
A09F CD2EBD   1050  WAIT    CALL mcprbu ; Printer Busy ??
A0A2 38FB     1060          JR   c,wait ; Ja, weiterwarten
A0A4 CD2BBD   1070          CALL mcprch ; Zeichen ausgeben
A0A7 C9       1080          RET
A0A8          1090
**** Zeile  120  : PRINIT=&A0A8
A0A8 1B       1100  PRINIT  DB   27 ; ESC
A0A9 31       1110          DM   "1" ; 7/72 Zoll Zeilenvorschub
A0AA 0D00     1120          DW   &000d ; CR und 0-Byte
A0AC 0000     1130          DW   0 ; Platz fuer
A0AE 0000     1140          DW   0 ; Aenderungen
A0B0          1150
**** Zeile  420  : PRLIIN=&A0B0
A0B0 0D       1160  PRLIIN  DB   13
A0B1 18       1170          DB   24
; CANcel; evtl. durch 10=LF ersetzen
A0B2 1B       1180          DB   27
A0B3 2A       1190          DM   "*" ; Bitmustermodus setzen
A0B4 01       1200          DB   1 ; Modus 1
A0B5 7F02     1210          DW   &027f
A0B7 00       1220          DB   0 ; 0-Byte
A0B8 0000     1230          DW   0 ; Platz fuer
A0BA 0000     1240          DW   0 ; Aenderungen
A0BC          1250
**** Zeile  850  : PRREIN=&A0BC
A0BC 1B       1260  PRREIN  DB   27 ; ESC
A0BD 40       1270          DM   "@" ; Normieren
A0BE 0D00     1280          DW   &000d ; CR und 0-Byte
A0C0 0000     1290          DW   0 ; Platz fuer
A0C2 0000     1300          DW   0 ; Aenderungen
A0C4          1310
**** Zeile  210  : BITANZ=&A0C4
**** Zeile  280  : BITANZ=&A0C4
```

```
**** Zeile  480  : BITANZ=&A0C4
A0C4 07      1320  BITANZ DB   7
**** Zeile  450  : PRTCOU=&A0C5
**** Zeile  970  : PRTCOU=&A0C5
**** Zeile  990  : PRTCOU=&A0C5
A0C5 7F02    1330  PRTCOU DW   &27f
**** Zeile  320  : TABANF=&A0C7
**** Zeile  650  : TABANF=&A0C7
A0C7         1340  TABANF DS   6
**** Zeile  290  : TABMIT=&A0CD
A0CD         1350  TABMIT DS   7
**** Zeile  510  : TABLET=&A0D4
A0D4         1360  TABLET DS   1
**** Zeile  180  : TABEND=&A0D5
**** Zeile  330  : TABEND=&A0D5
A0D5         1370  TABEND DS   2

Programm :hardcopy[
Start : &A000    Ende : &A0D6
Laenge : 00D7
 0  Fehler
Variablentabelle :
SCGELO BC0B  SCNELI BC26  SCNEBY BC20  MCPRCH BD2B
MCPRBU BD2E  NELINE A017  OK     A02A  NELI1  A030
NEBY   A04A  NEBIT  A053  NEBY1  A06A  TABOUT A087
PRINT  A093  WAIT   A09F  PRINIT A0A8  PRLIIN A0B0
PRREIN A0BC  BITANZ A0C4  PRTCOU A0C5  TABANF A0C7
TABMIT A0CD  TABLET A0D4  TABEND A0D5
```

```
10 REM Hardcopy fuer alle Schneider
20 REM Aufruf mit call &a000 in MODE 2
30 FOR i=&A000 TO &A0C6
40 READ a$:w=VAL("&H"+a$)
50 s=s+w:POKE i,w:NEXT
60 IF s<> 19650 THEN PRINT"Fehler in Datas":END
70 PRINT"ok!":END
80 DATA 21,A8,A0,CD,87,A0,CD,0B
90 DATA BC,57,1E,00,19,22,D5,A0
100 DATA 1E,1D,3E,07,32,C4,A0,D5
110 DATA 06,08,7B,FE,01,20,0B,3E
120 DATA 04,32,C4,A0,11,CD,A0,06
130 DATA 04,21,11,C7,A0,2A,D5,A0
140 DATA EB,73,23,72,23,EB,CD,26
150 DATA BC,10,F5,21,B0,A0,CD,87
160 DATA A0,21,7F,02,22,C5,A0,01
170 DATA 50,08,C5,3A,C4,A0,47,97
180 DATA 21,D4,A0,56,2B,5E,2B,EB
190 DATA CB,06,1F,EB,10,F5,1F,CD
200 DATA 93,A0,C1,10,E5,21,C7,A0
210 DATA 06,07,5E,23,56,2B,EB,CD
220 DATA 20,BC,EB,73,23,72,23,10
230 DATA F1,06,08,0D,20,CC,D1,1D
240 DATA 20,95,21,BC,A0,18,00,7E
250 DATA FE,00,C8,23,E5,CD,93,A0
260 DATA E1,18,F4,2A,C5,A0,2B,22
270 DATA C5,A0,4F,7C,B5,C8,79,CD
280 DATA 2E,BD,38,FB,CD,2B,BD,C9
290 DATA 1B,31,0D,00,00,00,00,00
300 DATA 0D,18,1B,2A,01,7F,02,00
310 DATA 00,00,00,00,1B,40,0D,00
320 DATA 00,00,00,00,07,7F,02
```

12. Das richtige Timing für CPCs

Damit Sie beim Programmieren die Zeit nicht vergessen, haben wir eine Softwareuhr entwickelt. Sie kann Ihnen Ärger mit Familie und Freunden ersparen und ist auch noch bei einigen Anwenderprogrammen eine sinnvolle Erweiterung.

Programme wie eine Softwareuhr können nur mit Hilfe der Interruptsteuerung geschrieben werden. Die Interruptprogrammierung ist noch schwieriger als die "normale" Programmierung in Maschinensprache. Es ist bereits bei Maschinenprogrammen schwer genug, Programme zu testen. Bei Programmen, die den Interrupt benutzen, ist es fast gänzlich unmöglich, unter realen Bedingungen zu testen, da es bei dieser Sorte von Programmen eben auf das "Timing" ankommt.

Jedoch bietet die Interruptprogrammierung ungeahnte Möglichkeiten. Überhaupt würde der Rechner ohne Interrupt gar nicht funktionieren. Eine grundsätzliche Aufgabe des Interrupts ist z.B. die Tastaturabfrage. Natürlich werden auch die BASIC-Interruptbefehle über den internen Interrupt abgewickelt.

Doch fangen wir vorn an. Durch einen in jedem Rechner vorhandenen Schwingquarz wird mit einer bestimmten Frequenz der gesamte Ablauf gesteuert und synchronisiert. Dieser Quarz ist damit die grundlegende interne Uhr des Rechners. In vielen Rechnern gibt es mehrere solcher Taktgeber. Durch einen dieser Taktgeber, beim Schneider durch das Gate Array, wird in regelmäßigen Abständen der Interrupt Request (IRQ) Pin des Z80-Prozessors auf Low geschaltet. Dadurch wird aus dem jeweils laufenden Maschinenprogramm zu einer vom Interruptmodus abhängige Adresse verzweigt.

Der Z80 wird beim Schneider im Interruptmodus 1 betrieben. Damit bewirkt ein IRQ einen RST &38 oder CALL &0038 Befehl. Der IRQ tritt beim Schneider-Computer 300 mal in einer Sekunde auf. Ist der Interrupt nicht mit DI ausgeschaltet worden, wird die aktuelle Programmausführungsadresse auf den Stapel gelegt und nach Adresse &38 verzweigt. Dort steht

wiederum ein Sprung zur Adresse &B989 (6128: &B941/ 664: &B941), also ins ständig eingeschaltete RAM, wo die eigentliche Interruptroutine beginnt.

Von hier aus wird das untere ROM selektiert und u.U. in die Interrupt-Service-Routine an Adresse &00B1 gesprungen. Dort werden nun z.B. die Tastaturabfrage oder das Erhöhen der BASIC-Variablen TIME erledigt.

Eine zweite ROM-Routine sorgt für die Bedienung der BASIC-Interrupts. Auch wird die gesamte Steuerung des Soundchips durch Interruptroutinen ausgeführt.

Um nun eine eigene Interruptroutine in die vorhandene einzubinden, legen wir einen Patch über den Sprung zur Interrupt-Service-Routine, der unsere Routine aufruft. Am Ende dieser Routine muß dann natürlich eine Verzweigung zur eigentlichen Interrupt-Service-Routine, also zur Adresse &00B1, erfolgen.

Mit diesen Voraussetzungen können wir unsere eigenen Routinen ohne Einschränkungen zunächst unabhängig entwickeln und dann einbinden. Ab dem Zeitpunkt der Einbindung wird sie dann jede 1/300stel Sekunde automatisch aufgerufen, ohne daß der sonstige Ablauf im Rechner nennenswert beeinflußt wird.

Das bedeutet für unser Beispiel einer Softwareuhr, daß diese ständig, ohne unser Dazutun, auch im Direktmodus läuft. So lange der Computer eingeschaltet ist, wird die aktuelle Zeit auf dem Bildschirm an der gewünschten Position angezeigt. Nun wollen wir die eigentliche Routine besprechen.

Am Anfang des Programms steht die kleine Initialisierungsroutine, die den CALL-Befehl zur Interrupt-Service-Routine auf unsere eigene Routine verbiegt. Die folgenden Befehle verändern die Init-Routine so, daß sie bei erneutem Aufruf wieder das Abhängen der Routine bewirkt.

Die Routine beginnt ab dem Label START. Dort wird als erstes die Rücksprungadresse für die Interrupt-Service-Routine auf den Stapel gelegt. D.h., daß die eigene Routine danach einfach

durch RET abgeschlossen werden kann und dann automatisch zur Adresse &00B1 gesprungen wird.

Der erste Zähler COUNT wird dann bei jedem Aufruf um eins erniedrigt. Dadurch wird erreicht, daß nur jedes 300ste Mal, also jede Sekunde, die neue Zeit ausgegeben wird. Es ist sinnvoll auf diese Weise möglichst wenig Zeit zu verschwenden, da sonst der Rechner unnötig verlangsamt wird. Ist der Zähler COUNT also gleich 0, wird er für die nächsten Durchgänge wieder mit 300 geladen. Nun beginnt das eigentliche Uhrprogramm.

Zum Speichern der Uhrzeit werden drei Bytes verwendet. Ein Byte für die Stunden, eins für die Minuten und ein Byte für die Sekunden. Dazu wird die aktuelle Zeit also z.B. eine 16 für 16 Uhr im Stundenbyte gespeichert. Der Wert jedes Bytes wird im BCD-Format abgespeichert. Das bedeutet, daß jede Ziffer des Dezimalwertes einzeln in je 4 Bits gespeichert wird. BCD steht für Binär Codiert Dezimal. Aufgrund der Eigenschaften des Hexadezimalsystems kann man auch folgendes schreiben. Dezimal 16 ist im BCD Format &16. Die BCD-Form ist in unserem Fall aus Zeitgründen effektiver als die einfache Speicherung des Wertes. Das Rechnen mit 1-Byte-BCD-Zahlen ist kaum langsamer als mit normalen Zahlen, da der Z80 den speziell für die BCD Arithmetik vorgesehenen Befehl DAA besitzt. Die Ausgabe einer BCD Zahl auf den Bildschirm ist jedoch im Vergleich zu normal gespeicherten Zahlen sehr viel einfacher und damit schneller. Auf diese Weise geht am wenigsten von der kostbaren Rechenzeit verloren.

Gerade bei Interruptroutinen sollte man auf den Faktor Rechenzeit besonderen Wert legen.

Zum Weiterstellen der Uhr um eine Sekunde wird die Adresse des Sekundenbytes im HL Register und die maximale Anzahl der Sekunden (also 60) an das Unterprogramm ZEIT übergeben.

Das Unterprogramm ZEIT erhöht die an der angegebenen Adresse stehende BCD-Zahl um 1 und prüft, ob der Maximalwert erreicht ist. Ist das der Fall, wird das Carry-Flag gelöscht, ansonsten gesetzt. Wurde der Maximalwert noch nicht erreicht,

wird nach dem Rücksprung zur Hauptroutine sofort zur Ausgabe der neuen Uhrzeit gesprungen. Das ist möglich, da sich die Minuten bzw. Stunden nur dann ändern, wenn 60 Sekunden abgelaufen sind.

Wurde der Maximalwert erreicht, so setzt ZEIT das jeweilige Byte wieder auf 0, denn nach 59 Sekunden bzw. Minuten kommt wieder die 0, ebenso nach 24 Stunden. Beim Erreichen des Maximalwertes der Sekunden wird HL mit der Adresse der Minuten geladen und, um diese zu erhöhen, wieder ZEIT aufgerufen. Wurde auch der Maximalwert der Minuten (60) erreicht, so werden auch noch die Stunden erhöht, wobei zuvor B mit 24 geladen wird. Spätestens jetzt sind die 3 Bytes auf die aktuelle Uhrzeit gesetzt und können ausgegeben werden.

Dazu wird die Speicherstelle POS mit der Bildschirmposition für die Ausgabe geladen. Dann werden mit der Routine BCBYOU (BCD BYte OUt) die Stunden, Minuten und Sekunden jeweils durch Doppelpunkte getrennt ausgegeben. Bei BCBYOU kommt der Vorteil der BCD Zahlen zum Tragen. Der Akku wird mit &30, dem ASCII-Code von "0" geladen. Dann wird mit RLD, wobei HL auf das jeweilige Byte zeigen muß, immer eine BCD Ziffer in die unteren 4 Bits des Akkus rotiert. Dadurch steht dann sofort der ASCII-Code der jeweiligen Ziffer im Akku und kann ausgegeben werden. Die Ausgabe erfolgt durch den Aufruf vom Unterprogramm PRINT, das bei jeder Ausgabe die Position für die nächste Ausgabe berechnet und abspeichert. Hier folgt nun das komplette Assemblerlisting:

```
A000              10
; Softwareuhr fuer "alle Schneider"
A000              20                ; H.D. 22/9/85
A000              30
A000              40                ; Initialisierungsroutine
A000 210000       50          LD    hl,start ; Interruptroutine
A003 2251B9       60          LD    (&b951),hl ; patchen
A006              70                ; fuer 464: ld (&b949),hl
A006 21B100       80          LD    hl,&b1
A009 2201A0       90          LD    (&a001),hl
A00C C9          100          RET
A00D             110
A00D             120  WRITE   EQU   &bdd3 ; (464, 664 und 6128 !!)
A00D             130  INTCOU  EQU   300
A00D             140  POSITI  EQU   &4700 ; mode 2
A00D             150  DOPPUN  EQU   &3a
A00D             160
**** Zeile  50  : START=&A00D
A00D 21B100      170  START   LD    hl,&b1 ; Ruecksprungadresse
A010 E5          180          PUSH  hl ; Interruptserviceroutine
A011 2A0000      190          LD    hl,(count) ; Zaehler holen
A014 2B          200          DEC   hl
; erniedrigen, damit nur jede
A015 220000      210          LD    (count),hl
A018 7C          220          LD    a,h
; Sekunde "der Rest" ausgefuehrt wird
A019 B5          230          OR    l ; also: Zahler ist nicht
A01A C0          240          RET   nz ; Null, dann Fertig
A01B 212C01      250          LD    hl,intcou ; Zaehler wieder
A01E 220000      260          LD    (count),hl ; Initialisieren
A021             270
A021 210000      280          LD    hl,sec ; Sekunden-Byteadresse
A024 0660        290          LD    b,&60 ; max. 60 (BCD) Sekunden
A026 CD0000      300          CALL  zeit ; Sekunden erhoehen
A029 38FE        310          JR    c,anzei
; noch nicht 60, dann Anzeigen
A02B CD0000      320          CALL  zeit ; Minuten erhoehen
A02E 38FE        330          JR    c,anzei
```

```
A030 0624      340          LD   b,&24 ; max. 24 Stunden
A032 CD0000    350          CALL zeit ;
**** Zeile  310  : ANZEI=&A035
**** Zeile  330  : ANZEI=&A035
A035 210047    360  ANZEI   LD   hl,positi
; Position fuer Ausgabe
A038 220000    370          LD   (pos),hl
A03B 210000    380          LD   hl,stund
A03E CD0000    390          CALL bcbyou
; BCD-Format Byte Ausgabe
A041 3E3A      400          LD   a,doppun
A043 CD0000    410          CALL print
A046 CD0000    420          CALL bcbyou ; Minuten Ausgeben
A049 3E3A      430          LD   a,doppun
A04B CD0000    440          CALL print
A04E CD0000    450          CALL bcbyou ; Sekunden Ausgeben
A051 C9        460          RET
A052           470
A052           480
; Unterprogramm zur erhoehung der Zeitzaehler
**** Zeile  300  : ZEIT=&A052
**** Zeile  320  : ZEIT=&A052
**** Zeile  350  : ZEIT=&A052
A052 7E        490  ZEIT    LD   a,(hl) ; alte Zeit
A053 C601      500          ADD  a,1 ; erhoehen
A055 27        510          DAA   ; Bytes sind im BCD-Format
A056 77        520          LD   (hl),a ; Speichern
A057 B8        530          CP   b ; Maximum erreicht ?
A058 D8        540          RET  c ; Nein, dann Fertig
A059 97        550          SUB  a ; Ja, dann mit
A05A 77        560          LD   (hl),a ; "0" fortsetzen
A05B 2B        570          DEC  hl
; naechstes Mal Minuten (Stunden)
A05C C9        580          RET
A05D           590
A05D           600
; Routine zur Ausgabe eines BCD-Bytes
**** Zeile  390  : BCBYOU=&A05D
**** Zeile  420  : BCBYOU=&A05D
```

```
**** Zeile  450  : BCBYOU=&A05D
A05D 3E30      610  BCBYOU LD   a,&30
; Ausgabe von Zahlen (asc("0")=&30)
A05F ED6F      620          RLD
; hoeherwertige Ziffer in Akku rotieren
A061 CD0000    630          CALL print
A064 ED6F      640          RLD
A066 CD0000    650          CALL print ; niderwertige Ziffer
A069 ED6F      660          RLD   ; Wert wiederherstellen
A06B 23        670          INC  hl
; naechstes Mal Minuten (Sekunden)
A06C C9        680          RET
A06D           690
**** Zeile  410  : PRINT=&A06D
**** Zeile  440  : PRINT=&A06D
**** Zeile  630  : PRINT=&A06D
**** Zeile  650  : PRINT=&A06D
A06D E5        700  PRINT  PUSH hl
A06E F5        710          PUSH af
A06F 2A0000    720          LD   hl,(pos)
A072 24        730          INC  h
A073 220000    740          LD   (pos),hl
A076 CDD3BD    750          CALL write
A079 F1        760          POP  af
A07A E1        770          POP  hl
A07B C9        780          RET
A07C           790
**** Zeile  370  : POS=&A07C
**** Zeile  720  : POS=&A07C
**** Zeile  740  : POS=&A07C
A07C           800  POS    DS   2
**** Zeile  190  : COUNT=&A07E
**** Zeile  210  : COUNT=&A07E
**** Zeile  260  : COUNT=&A07E
A07E 2C01      810  COUNT  DW   300
**** Zeile  380  : STUND=&A080
A080 00        820  STUND  DB   0
A081 00        830          DB   0
**** Zeile  280  : SEC=&A082
```

```
A082 00        840  SEC     DB   0

Programm :uhr
Start : &A000     Ende : &A082
Laenge : 0083
 0  Fehler
Variablentabelle :
WRITE  BDD3  INTCOU 012C  POSITI 4700  DOPPUN 003A
START  A00D  ANZEI  A035  ZEIT   A052  BCBYOU A05D
PRINT  A06D  POS    A07C  COUNT  A07E  STUND  A080
SEC    A082
```

```
10 REM BASIC Lader fuer Uhr
20 REM Initialisierung mit Call &a000
30 FOR i=&A000 TO &A07F
40 READ a$:w=VAL("&H"+a$)
50 s=s+w:POKE i,w:NEXT
60 IF s<> 14784 THEN PRINT"Fehler in Datas":END
70 ' fuer 464:IF s<> 14776 THEN .........
80 PRINT"ok!":END
90 DATA 21,0D,A0,22,51,B9,21,B1
100 '464:..,..,..,..,49,..,..,..
110 DATA 00,22,01,A0,C9,21,B1,00
120 DATA E5,2A,7E,A0,2B,22,7E,A0
130 DATA 7C,B5,C0,21,2C,01,22,7E
140 DATA A0,21,82,A0,06,60,CD,52
150 DATA A0,38,0A,CD,52,A0,38,05
160 DATA 06,24,CD,52,A0,21,00,47
170 DATA 22,7C,A0,21,80,A0,CD,5D
180 DATA A0,3E,3A,CD,6D,A0,CD,5D
190 DATA A0,3E,3A,CD,6D,A0,CD,5D
200 DATA A0,C9,7E,C6,01,27,77,B8
210 DATA D8,97,77,2B,C9,3E,30,ED
220 DATA 6F,CD,6D,A0,ED,6F,CD,6D
230 DATA A0,ED,6F,23,C9,E5,F5,2A
240 DATA 7C,A0,24,22,7C,A0,CD,D3
250 DATA BD,F1,E1,C9,00,00,2C,01
```

Für den Fall, daß Sie keinen Assembler besitzen, haben wir auch einen BASIC-Lader erstellt.

Außerdem folgt noch ein kleines BASIC-Programm, mit dem Sie die Uhr auf bequeme Weise stellen können. Wollen Sie die Uhr wieder ganz ausstellen, also die Routine aus dem Interrupt ausklinken, so brauchen Sie nur zum Zweitenmal mit CALL &A000 die Routine aufzurufen. Ein erneutes Einstellen ist dann erst wieder nach Eingabe von

```
POKE &A001,&D:POKE &A002,&A0
```

mit CALL möglich.

```
10 REM Uhrsteller
20 MEMORY &9FFF
30 MODE 2
40 ' LOAD"uhr1.obj
50 LOCATE 15,7:PRINT"U h r s t e l l e r"
60 LOCATE 1,11
70 INPUT"12 oder 24 Stunden Anzeige (12/24) ?";e
80 PRINT
90 IF e<>12 AND e<>24 THEN 70
100 POKE &A031,VAL("&"+STR$(e))
110 zeibas=&A080
120 INPUT"Uhrzeit (hh,mm,ss)";h,m,s
130 PRINT
140 POKE zeibas,VAL("&"+STR$(h)):POKE zeibas+1,VAL("&"+STR$(m)
):POKE zeibas+2,VAL("&"+STR$(s))
150 INPUT "Position (Spalte,Zeile)";sp,ze
160 IF sp<1 OR ze<1 THEN 150
170 IF sp=1 THEN sp=257
180 POKE &A036,ze-1:POKE &A037,sp-2
190 ' call &a000
```

TEIL 3

Tips & Tricks zur Maschinensprache

13. Programmieren in Maschinensprache

Immer wenn die Geschwindigkeit zu einem wichtigen Faktor der Programmierung wird, ist der Programmierer gezwungen, direkt oder indirekt auf die Maschinensprache zurückgreifen. Direkt bedeutet, daß der Maschinencode unmittelbar in den Computer eingegeben wird. Dafür gibt es mittlerweile eine große Anzahl von Assemblerprogrammen, die diese Arbeit erleichtern.
Mit indirektem Weg ist an dieser Stelle die Möglichkeit gemeint, einen Compiler zu verwenden. Verfügt man z.B. über einen BASIC-Compiler, so wird das selbst erstellte Programm wie gewöhnlich in der Sprache BASIC geschrieben. Der Compiler übersetzt dann dieses BASIC in den Maschinencode, der getrennt von dem Quellcode (BASIC) gespeichert und verarbeitet wird.

Um einen Einblick in die Maschinenebene zu bekommen, werden wir anhand der direkten Assemblierung einige kleinere Routinen erstellen und deren zeitliche Vorteile gegenüber einer Interpretersprache, wie es BASIC im CPC 464/664 ist, darlegen. Dazu benötigen wir jedoch kein Assemblerprogramm. Kleine BASIC-Ladeprogramme werden uns helfen, den Maschinencode in den Arbeitsspeicher zu bringen.

Starten Sie einmal das folgende, kleine BASIC-Programm.

```
50 MODE 2
100 FOR ADRESSE=49152 TO 65535
110 POKE ADRESSE,255
120 NEXT
```

Programmbeschreibung:

Die oberen 16K RAM des CPC-Speichers (&C000-&FFFF) sind für den Bildschirm reserviert. Die numerischen Inhalte der Speicherplätze repräsentieren die Punktemuster, die, auf dem Bildschirm zusammengefügt, dann sinnvolle Zeichen ergeben.

Mit unserem Beispielprogramm setzen wir alle Punkte auf dem Bildschirm.

50: Der Befehl MODE 2 löscht den Bildschirm und setzt den Offset auf die linke obere Ecke.

100: Die FOR-NEXT-Schleife beginnt mit der ersten Bildschirmadresse und endet mit der letzten.

110: Der Wert 255 wird in alle Speicherplätze gePOKEt.

Die Ausführung dieses Programmes benötigt ca. 40 Sekunden. Bei 16384 zu belegenden Speicherplätzen bedeutet dies, daß jede Speicherplatzbelegung etwa 2,4 Millisekunden dauert.
Um nun einen Zeitvergleich mit einem Maschinenprogramm zu bekommen, das etwa gleich aufgebaut ist wie das BASIC-Programm, lassen Sie einmal folgendes Programm laufen.

```
10 MODE 2:SUMME=0
20 FOR I=40960 TO 40979
30    READ a$:WERT=VAL("&"+a$)
40    POKE ADRESSE,WERT:SUMME=SUMME+WERT:NEXT
50 IF SUMME<>1531 THEN PRINT "Fehler in DATAs":END
60 PRINT "TASTENDRUCK STARTET MASCHINENROUTINE"
70 CALL &BB06:CALL 40960
80 END
90 DATA 21,00,40,01,01,00,11,00,C0,3E
100 DATA FF,12,13,ED,42,C2,0B,A0,C9,00
```

Mit dieser Assemblerroutine kann man die Ausführungszeit auf etwa 0,2 Sekunden verkürzen, das bedeutet pro Speicherplatz gut 12 Mikrosekunden. Obwohl diese Routine noch nicht einmal die schnellste Möglichkeit darstellt, dürfte dieser Vergleich doch die Möglichkeiten der Assemblierung verdeutlichen.

Programmerläuterung:

Das Maschinenprogramm besteht aus den hexadezimalen Werten der DATA-Zeilen 90 und 100. In der FOR-NEXT-Schleife (Zeile 20-40) werden diese Werte gelesen, in einen numerischen Wert umgewandelt (Z. 30) und dann in die Speicherplätze 40960-40979 gePOKEt. Der Befehl CALL &BB06 in Zeile 70 ruft eine Routine des Betriebssystems auf, die die Tastatur solange abtastet, bis eine beliebige Taste gedrückt wird. Dann erst wird der nächste Befehl ausgeführt. Hier folgt CALL 40960, der jene Maschinenroutine aufruft, die in der FOR-NEXT-Schleife in den Speicher geschrieben wurde.
Nach der Beendigung kehrt der Rechner in die Obhut des BASIC-Interpreters zurück und meldet sich mit READY, nachdem er in die Zeile 80 (END) gelangte.

Was bedeuten denn nun die Hex-Werte der DATA-Zeilen?

Um eine befriedigende Antwort darauf geben zu können, müssen wir erst einmal ein paar Blicke in die Z80-CPU, das Herz des Schneider Computers, werfen.

Nach der Einführung in die Sprache der CPU (Tips & Tricks Band 1) soll hier nun ein tieferer Blick in die Trickkiste des Rechners geworfen werden. Es werden viele wichtige, leistungsfähige und schnelle Befehle des Z80 vorgestellt und anhand von praktischen Beispielroutinen erläutert.

Zum Verständnis dessen brauchen wir erst einmal eine gute Kenntnis der Register, wie sie aufgebaut sind, wie sie funktionieren und wie man sie effizient anwendet.

13.1 Die Register des Z80

Die Register sind Speicherplätze innerhalb der CPU. Obwohl der Z80 ein 8-Bit-Prozessor ist, verfügt er auch über 16-Bit-Register.

Die Bezeichnungen der Register lauten wie folgt:

8-Bit-Register:	A, B, C, D, E, F, H, L
16-Bit-Register:	BC, DE, HL, SP, PC

Damit sind aber nur die wichtigsten Register aufgeführt. Die anderen wollen wir an dieser Stelle erst einmal übergehen, weil sie für das Grundverständnis nicht erforderlich sind. Auch möchte ich jene Leser auf noch folgende Abschnitte dieses Kapitels verweisen, die bereits bemerkt haben daß einige der 16-Bit-Register Bezeichnungen haben, die sich aus denen der 8-Bit Register kombinieren lassen. Dies hat seinen logischen Grund, ... aber davon später mehr.

13.1.1 8-Bit Register

Die aufgeführten 8-Bit-Register sind bei weitem nicht gleichwertig. Das A- und das F-Register nehmen gegenüber den anderen eine ganz besondere Position ein, so daß die Aufzählung besser folgendermaßen erfolgen sollte:

A,F, B,C,D,E,H,L

Das A-Register (Akkumulator)

Alle logischen und arithmetischen 8-Bit-Operationen laufen stets über das A-Register. Die Ergebnisse dieser Operationen stehen dann im Akku zur Verfügung, während das andere Register nicht verändert wird.

1. Beispiel einer Addition zweier Zahlen:

```
LD    A,6       Lade Akku mit 6
LD    D,3       Lade Reg. B mit 3
ADD   A,D       Addiere Inhalt von D zum Akku
```

Nach der Operation ADD A,D steht in A der Wert 9, während in D weiterhin der Wert 6 vorhanden ist.

2. Beispiel einer Subtraktion zweier Zahlen:

```
LD    A,&54      Lade Akku mit &54
LD    L,&54      Lade Reg.L mit &54
SUB   A,L        Subtrahiere Inhalt von D vom Akku
```

Nach dieser Subtraktion steht im Akkumulator nun der Wert 0.

Diese beschriebenen arithmetischen Operationen beeinflussen aber nicht nur den Inhalt des Akkus. Ein weiterer Einfluß erfolgt auf das Flag-Register. Dieses gibt Aufschluß über verschiedene Auswirkungen, die bei den Operationen mit dem Akku auftreten können. Um die Auswirkungen darzustellen, werden die einzelnen, speziellen Bits des F-Registers gesetzt oder zurückgesetzt.
Erst aufgrund dieser Informationen, die das Flag-Register bietet, kann man bedingte Operationen ausführen. Doch zunächst einmal die Beschreibung dieses Registers.

Das F-Register (Flag-Register)

Wie Sie bereits erkannt haben, kann man über den Inhalt des F-Registers Aufschluß über die Ergebnisse von arithmetischen und logischen Operationen erhalten.
Das Zero-Flag ist nur eines der sechs zur Verfügung stehenden Flags dieses Registers, aber es ist neben dem Carry-Bit (C) wohl das wichtigste.

Der Aufbau des Flag-Registers:

Nachfolgend sehen Sie die Anordnung der einzelnen Flags im Flagregister.

Bez.	S	Z		H		P/V	N	C
Bit Nr.	7	6	5	4	3	2	1	0

Beschreibung der Flags:

S:	Sign	0	Ergebnis positiv
		1	Ergebnis negativ
Z:	Zero	0	Ergebnis ungleich null
		1	Ergebnis gleich null
H:	Halfcarry	0	kein Übertrag von Bit 3 nach Bit 4 im Akku
		1	Übertrag von Bit 3 nach Bit 4
P/V:	Parity/Overfow	0	ungerade Parität / kein Überlauf von Bit 6 nach 7
		1	gerade Parität / Überlauf von Bit 6 nach 7 (andere Funktionen später)
N:	Subtraktions-Flag	0	nach Ausführung einer Addition
		1	nach Ausführung einer Subtraktion
C:	Carry	0	kein Übertrag von Bit 7 nach Bit 8
		1	Übertrag von Bit 7 nach Bit 8

Das Sign-Bit (S)

Mit den 8 Bit eines Bytes lassen sich 2^8=256 Werte darstellen, nämlich 0 bis 255. Bei der vorzeichenbehafteten Darstellung sind es die Werte -128 bis +127. Hierbei wird der Zahlenwert durch

die Bits 0 bis 6 und das Vorzeichen durch das Bit 7 dargestellt. Ist das Vorzeichen positiv, so ist Bit 7 zurückgesetzt (0). Folglich ist Bit 7 bei einer negativen Zahl gesetzt, d.h. gleich 1.
Die Binärzahl 11111111 steht dann für die Dezimalzahl -1.

Das Zero-Bit (Z)

Wenn ein Byte durch eine logische oder arithmetische Operation oder durch einen Rotations- oder Schiebebefehl beeinflußt wird, so zeigt das Z-Flag mit einer 1 an, daß jenes Byte zu null wurde. Dementsprechend ist das Z-Flag null, wenn der Wert des Bytes ungleich Null ist.
Bei Vergleichen zwischen zwei Bytes - z.B. CP A,E - wird das Z-Bit null, wenn die Register gleiche Werte beeinhalten, eins, wenn die Inhalte differieren. Dies ist leicht zu verstehen, wenn man weiß, daß der Befehl CP A,s den Operanden s vom Akku subtrahiert.
Von den weiteren drei Funktionen des Z-Flag möchte ich lediglich noch die Verwendung beim Testbefehl BIT b,r erläutern. Der Befehl BIT 5,D testet das fünfte Bit des Registers D auf seinen Inhalt. Ist der Inhalt eins, so ist das Z-Flag null, ist der Inhalt null wird das Z-Flag eins.

Das Half-Carry-Bit (H)

Das H-Flag zeigt einen Übertrag vom unteren Nibble (Bit 0-3) zum oberen Nibble (Bit 4-7) eines Bytes an. Haupsächliches Anwendungsgebiet ist hier die BCD-Arithmetik und der Dezimalabgleich (DAA). Wenn hierbei das untere Nibble einen Wert größer als 9 erhält, muß ein Übertrag ins obere Nibble erfolgen, da ein Nibble zur Darstellung einer Dezimalzahl nicht größer als 9 werden darf. Dieser Übertrag wird durch das H-Flag angezeigt.

Das Paritäts/Overflow-Bit (P/V)

Diesem Flag kommen verschiedene Funktionen zu.

a) Trotz einem sehr hohem Qualitätsstandard der heutigen Hardware sind Fehler bei der Übertragung von Daten in ihrer Anzahl lediglich zu verringern, nicht aber gänzlich auszuschließen. Die Parität wird meist bei der Übertragung von Zeichen im ASCII-Format (7-Bit) verwendet, indem man das Paritätsbit als achtes Bit anhängt. Ist die Anzahl der Datenbits, die 1 sind, gerade, so wird das Paritätsbit auf 1 gesetzt. Ist jene Anzahl jedoch ungerade, so wird das Paritätsbit auf 0 gesetzt. Sollte bei einem Datentransfer nun ein Bit falsch übertragen werden, so stimmt die Parität nicht mehr mit dem Paritätsbit überein. Dies wird dann erkannt, und es wird eine erneute Übertragung dieses Bytes verlangt werden.

b) Wenn bei einer Addition oder Subtraktion das Ergebnis größer als +127 wird, so wird das 7. Bit, das für Vorzeichen reserviert ist, fälschlicherweise beeinflußt. Dieses wird durch das P/V-Flag angezeigt.

c) Bei Blocktransfer- und Blocksuchbefehlen wird dieses Flag in Abhängigkeit des Registers BC beeinflußt. Dieses Register hat bei den erwähnten Befehlen die Funktion eines Zählers. Ist das Zählregister BC gleich 0, wird das P/V-Flag auf 0 gesetzt. Ist BC ungleich 0, wird P/V gleich 1.

d) Bei den Befehlen LD A,I und LD A,R erhält das P/V-Flag den Wert des Interrupt-Enable-Flip-Flops IFF2. Damit ist es möglich den Inhalt des IFF2 abzufragen und/oder zu speichern.

Das Subtraktion-Bit (N)

Der Programmierer wird für gewöhnlich keinen Gebrauch von diesem Flag machen. Der eigentliche Nutzen dieses Flags ergibt sich beim Dezimalabgleich (DAA) nach einer Addition oder Subtraktion. Dieser Abgleich wird nach einer Addition anders als nach einer Subtraktion ausgeführt. Wie die CPU nun auf den DAA zu reagieren hat, erkennt sie aus dem Inhalt des N-Flags.

Das Carry-Bit (C)

Das Übertragsbit zeigt bei einer Addition oder Subtraktion an, ob bei diesen Operationen ein Übertrag auftritt.
Bei Rotations- oder Schiebebefehlen wird das Carry-Bit als neuntes Bit gebraucht.
Das Carry wird von den logischen Operationen AND, OR und XOR zurückgesetzt. Diese Befehle des Z80 werden häufig verwendet, wenn es gilt, das Carry zu löschen.

Warum löschen diese Logik-Befehle das C-Flag?

Betrachten wir dazu das folgende Beispiel mit dem Wert &D3 im Akku:

```
        11010011
AND     11010011
        11010011
```

Da bei diesem Befehl, wie auch bei OR oder XOR, nie ein Übertrag auftreten kann, auch nicht wenn man den Akku mit einem anderen Register vergleicht, wird das Carry-Flag stets auf 0 gesetzt. Diese Befehle entpuppen sich also als regelrechte Carry-Lösch-Befehle.

Etwas über die Verwirrungen, wann und wie die Flags gesetzt, zurückgesetzt und benutzt werden

Es mag auf den ersten Blick Unverständnis aufkommen lassen, daß z.B. das Zero-Flag genau dann 1 wird, wenn das getestete Register durch eine Operation 0 wird, und umgekehrt.
Dieser Umstand wird aber sofort plausibel, wenn man die Inhalte der Flags als Wahrheitswerte betrachtet.
Ein Flag kann demnach zwei Wahrheitswerte annehmen, wahr und falsch. Die 1 steht für wahr und die 0 für falsch.
Der Z80 verfügt über verschiedene bedingte Sprungbefehle. Ob die Bedingung, die einen Sprung auslösen würde, erfüllt ist, erkennt die CPU aus den Zuständen der Flags.
Am Beispiel des häufig verwendeten Sprungbefehls JP C,adresse soll das Zusammenspiel mit dem Flag-Register dargelegt werden.
Bei der Addition des Registers A mit einem anderen Wert wurde ein Ergebnis ermittelt, dessen Wert größer als &FF ist. Wie wir wissen, ist das Ergebnis durch den Überlauf des Akku nicht zerstört worden, weil in dem Fall das Carry-Bit als neuntes Bit mit der Wertigkeit 2^8 (=256) anzusehen ist.
Wenn dieser Fall auftritt, so soll das Programm zu einem bestimmten Programmteil verzweigen. Dies erledigt der oben erwähnte bedingte Sprungbefehl JP C,adresse, der genau dann aktiv wird, wenn das C-Bit gesetzt, d.h. 1 ist.

Wenn wir genau dann einen Sprung auslösen wollen, wenn das C-Bit 0 ist, benutzen wir den Befehl JP NC,adresse, der dann aktiv wird, wenn das Carry null ist.

Die Register B, C, D, E, H, L

Diese Register sind vollkommen gleichwertig. Man kann ihnen direkt Werte zuweisen, man kann sie mit Werten aus anderen Registern laden, und man kann sie mit sich selbst laden, was in meinen Augen allerdings von zweifelhaftem Wert ist, wenn man nicht gerade eine Zeitverzögerung bewirken will.
Die Register können mit den Inhalten des RAMs geladen und wieder zurückgespeichert werden.

Man kann die Register mit dem Akkuinhalt logisch und arithmetisch verknüpfen und/oder sie mit Rotations- und Schiebebefehlen beeinflussen.
Damit sind die wichtigsten Anwendungsgebiete dieser Register als 8-Bit-Register aufgeführt.

13.1.2 Die 16-Bit-Register BC, DE, HL, PC und SP

Die 8-Bit-Register B und C, D und E bzw. H und L können zusammengesetzt werden, womit man in der Lage ist, mit speziellen Befehlen über 16-Bit-Register verfügen zu können, und dieses in einer 8-Bit-CPU.

Dies hat natürlich seine ganz besondere Bewandtnis. Wie soll man beispielweise den 40000. Speicherplatz der 65536 zur Verfügung stehenden Plätze adressieren? Mit einem 8-Bit-Wort lassen sich ja gerade 2^8=256 verschiedene Werte oder Adressen darstellen. Fügt man aber zwei Byte zu einem 16-Bit-Register zusammen, so hat man die Möglichkeit, 2^{16}=65536 Speicherplätze anzusprechen.

Den drei 16-Bit-ern kommen aber nun unterschiedliche Funktionen zu. Obwohl sie von der Struktur keine Unterschiede aufweisen, fordert der Prozessor für bestimmte Operationen die benötigten Parameter in dem einen oder anderen Register an.

So kann man das HL durchaus als abgespeckten 16-Bit-Akkumulator bezeichnen. Drei verschiedene arithmetische Verknüpfungen lassen sich über HL mit DE oder BC bewirken.

Das Register BC wird häufiger als Zähler benutzt. Dies geschieht in den sogenannten Blocktransfer- und Suchbefehlen die wir später noch genau kennen lernen werden.

Das Register PC (Program Counter)

Um kein Mißverständnis aufkommen zu lassen: Das Register PC ist ein reiner 16-Bit-er. Man hat nur einen ganz beschränkten Einfluß auf dieses Register.

Die Befehlsfolge eines Programmes steht immer in einem Speicherbereich außerhalb des Prozessors. Im Prozessor selbst befindet sich immer nur der aktuelle, auszuführende Befehl. Diesen Befehl holt sich die CPU aus demjenigen Speicherplatz, dessen Adresse gerade im PC steht. Man kann den Program Counter also als einen Zeiger bezeichnen. Dieser zeigt stets auf den nächsten Speicherplatz, in dem der folgende Befehl oder Datenwert steht.
Nachdem sich der Prozessor einen Befehl aus dem Speicher geholt hat, wird der PC auf den neuen aktuellen Stand gebracht. Entsprechend der Länge der Einträge (es gibt ja 1-, 2-, 3- und 4-Byte-Befehle) wird der PC um den entsprechenden Wert erhöht.

Das Register SP (Stack Pointer)

Der Stack Pointer hat ebenfalls eine Zeigerfunktion. Der Inhalt des SP ist die Adresse des oberen Stack-(Stapel-)elementes.

Doch was ist nun ein Stack?

Während man im allgemeinen jederzeit auf jeden Speicherplatz Zugriff hat, kann man bei der Stapelverarbeitung immer nur an das oberste Element des Stapels herankommen. Dies läßt sich leicht an dem Beispiel eines Bücherstapels veranschaulichen. Hat man mehrere Bücher in einen Karton gelegt, so kann man immer nur das oberste Buch entnehmen. Will man an das unterste, muß man erst alle darüberliegenden Bücher entfernen.
Diese Zugriffsart nennt man LIFO-Prinzip. LIFO steht für Last-In-First-Out, das zuletzt hineingegebene Element ist auch als erstes wieder zu entnehmen.

Das Beispiel mit dem Buch-Stack hat nur einen Fehler. Der Stack im Rechner wird von oben nach unten aufgebaut. Der erste Eintrag liegt an der höchsten Stelle im Stack, die weiteren Einträge erhalten demnach also Speicherplätze mit kleineren Adressnummern. Dies ist aber nicht von allzu großer Bedeutung, da die Verwaltung von der CPU übernommen wird. Man muß es trotzdem wissen, wenn man den Stack in einen anderen Bereich legen will.

Dies kommt allerdings nicht häufig vor, da im RAM des Rechners immer ein Bereich des Speichers für den Stack reserviert ist. Dieser darf aber auf keinen Fall durch andere Daten überschrieben werden. Dies würde zwangsläufig zum Absturz des Systems führen.

Der Vollständigkeit halber erwähne ich jetzt noch, daß der SP immer auf das oberste, also zuletzt eingegebene Stapelelement weist.

Nun sind die wichtigsten Register der Z80-CPU benannt und beschrieben worden.
Es gibt noch weitere Register, die ich hier aber noch nicht erwähnen will, weil sie zum allgemeinen Verständnis vorerst zu entbehren sind.

Nachdem uns der Z80 nun in groben Zügen bekannt ist, wenden wir uns jetzt den Befehlen dieses Mikroprozessors zu.
Da es unsinnig ist, erst alle Befehle kennenzulernen - derer gibt es über 500 - wollen wir uns immer jene heraussuchen, die uns bei der Lösung verschiedener Problemstellungen dienlich sein können.

13.3 Ein detailiertes Beispiel der Maschinenprogrammierung

Als erstes Beispielprogramm werde ich das zu Anfang des Kapitels aufgeführte Bildschirmlöschen erläutern.

Das eigentliche Assemblerprogramm besteht, wie wir wissen, aus der Datenfolge der DATA-Zeilen 90 und 100.

```
90 DATA 21,00,40,01,01,00,11,00,C0,3E
100 DATA FF,12,13,ED,42,C2,0B,A0,C9,00
```

Das BASIC-Programm diente ja lediglich dem Zweck, diese Werte in den Speicher ab dem Speichplatz 40960 (&A000) zu POKEn.

Diese DATA-Zeilen haben nur für Assembler-Profis eine direkte Aussagekraft. Nun wollen wir uns dieses Programm einmal in einer verständlichen Schreibweise anschauen. Diese Schreibweise nennt man die mnemotechnische Form des Assemblerlistings.
Mnemonics (hier liegt kein Druckfehler vor) heißen die aussagekräftigen Kürzel, die hier Anwendung finden.
Wenn man den Hex-Code 3E FF liest, muß man schon über ein gutes Gedächtnis und viel Erfahrung verfügen, um erkennen zu können, was sich dahinter verbirgt. Die mnemotechnische Darstellung desselben Befehles lautet LD A,&FF. Hieraus ist nun leicht zu erkennen was gemeint ist.

Lade	Akku	mit der	Hex-Zahl	FF
LD	A	,	&	FF

Das folgende Programmlisting ist äußerst aufschlußreich. In der ersten Spalte steht die Speicherplatznummer, in der zweiten Spalte ist der Hex-Code und in der dritten Spalte der mnemonische Code des Assemblerbefehles eingetragen. Abgerundet wird dieses Listing durch die Spalte 'Bemerkungen',

in der die Bedeutungen der Befehle kurz dargelegt werden können.

```
Adresse  Hex-Code      Mnemonic          Bemerkung
-------------------------------------------------------------
A000     21 00 40      LD    HL,&4000    Lade Zähler
A003     01 01 00      LD    BC,&0001    Lade Subtrahent
A006     11 00 C0      LD    DE,&C000    erster Bildschirmplatz
A009     3E FF         LD    A,&FF       Lade Akku mit FF
A00B     12            LD    (DE),A      belege Sp. DE mit A
A00C     13            INC   DE          erhöhe DE um 1
A00D     ED 42         SBC   HL,BC       subtr. BC von HL
A00F     C2 0B A0      JP    NZ,&A00B    Jump nach &A00B wenn Zero-
                                         Flag=0
A012     C9            RET               zurück ins Hauptprogr.
A013     00            NOP               NoOPeration
```

Die Adressen in der linken Spalte beziehen sich dabei übrigens immer nur auf das erste Byte in jeder Zeile. Deshalb werden nach den gezeigten 3-Byte-Befehlen immer zwei Adressen übersprungen. Daran gewöhnt man sich aber sehr schnell.

Die ersten vier Befehle des Listings sind bekannt. Es werden die verschiedenen Register lediglich mit Werten geladen, deren Bedeutungen in Kürze zu verstehen sind.

Neu in diesem Listing ist zum ersten der Befehl LD (DE),A. Wenn, wie in diesem Fall, ein Registername in Klammern steht, ist der Inhalt des Registers als Adresse zu verstehen. DE beinhaltet an dieser Stelle den Wert &C000 (siehe dritte Zeile). Damit wird mit diesem Befehl LD (DE),A der Inhalt von A in den Speicherplatz &C000 gebracht.

Der zweite neue Befehl ist INC DE. INC steht für INCrementiere. Dieser Befehl erhöht den Inhalt des angegebenen Registers um 1. Nach diesem Befehl steht nun der Wert &C001 im Register DE.

Übrigens lassen sich, bis auf den Program Counter, alle bislang erwähnten Register INCrementieren.

Weiterhin neu ist der Befehl SBC HL,BC. SBC steht für Subtrahiere mit Carry. Das Register HL wird um den Inhalt von BC verringert.
Um Schwierigkeiten mit diesem Befehl zu vermeiden, muß man immer beachten, daß das eventuell gesetzte Carry-Bit bei dieser Operation für einige Überraschungen gut ist. Im Zweifelsfall über den Zustand des C-Bits sollte man es zuvor immer löschen. Dies kann mit den Befehlen AND A, XOR A und OR A geschehen.
In unserem Beispielprogramm habe ich darauf der Einfachheit halber verzichtet.

Der letzte neue Befehl in diesem Programm (RETurn und NOP kennen Sie ja schon aus TIPS & TRICKS Band I) ist ein bedingter Sprung, JP NZ,&A00B.
Dieser wird immer dann ausgeführt, wenn die an den Sprung geknüpfte Bedingung erfüllt ist. Bei dem Sprungbefehl JP NZ,adresse lautet die Bedingung NZ, Non Zero oder auf deutsch ungleich null.
Solange das Zero-Flag 0 ist und damit anzeigt, daß das Ergebnis der vorhergehenden Operation ungleich null ist, wird der Sprungbefehl ausgeführt.
In diesem Beispiel geschieht das so lange, bis das Register HL null wird, also &4000 (16384) mal. Dies entspricht der Anzahl der Speicherplätze, die für den Bildschirmbereich reserviert sind. Wenn nun der Befehl SBC HL,BC das Register HL auf null setzt, wird zugleich das Zero-Flag gesetzt. Die Bedingung für den folgenden Sprungbefehl ist nicht mehr gegeben, womit dieser nicht mehr durchgeführt wird. Der nächste Befehl RET wird ausgeführt und der Rechner kehrt in den BASIC-Interpreter zurück. Dieser meldet sich mit READY.

Eine 'zeitkritische' Betrachtung der Routine

Wenn man zur Lösung eines Programmierproblemes die Maschinensprache wählt, um höhere Verarbeitungsgeschwindigkeiten zu erreichen, so muß man der Wahl der Befehle und dem Programmaufbau ebenfalls besondere Beachtung zuteil werden lassen.
In diesem Sinne ist die bereits vorgestellte Routine alles andere als günstig ausgefallen. Dies aber mit dem Hintergedanken dem aufmerksamen Leser zu demonstrieren, daß der direkte Lösungsweg manchmal elegant erscheint, in vielen Fällen aber Lösungen unterlegen ist, welche umständlich oder zu aufwendig erscheinen.
Bei der Assemblerprogrammierung gelten nicht unbedingt die Konventionen, die das Programmieren in einer höheren Sprache effizient machen. So ist unter anderem das Löschen eines zusammenhängenden Speicherbereiches wie der des Bildschirmes mit einem sehr langen Programm um ein vielfaches schneller zu erledigen als mit einem scheinbar effizienten, nur wenige Befehle langen Programm. Wie überall wird auch hier ein Kompromiß zwischen Geschwindigkeit und Programmieraufwand den zu begehenden Weg weisen.

Um nun die Geschwindigkeit einer Routine zu erfahren, brauchen wir uns nicht mit der Stoppuhr neben den Rechner zu setzen. Viel genauer arbeitet da ein Taschenrechner.
Dies funktioniert natürlich nur, wenn man weiß, wie lange die CPU zur Ausführung einzelner Befehle benötigt. Am Beispiel unserer Routine werde ich die Vorgehensweise bekannt machen.
Die Zahlen hinter den Mnemonics sind die Angaben über die Dauer des betreffenden Befehles in Mikrosekunden.

```
        LD      HL,&4000    5
        LD      BC,&0001    5
        LD      DE,&C000    5
        LD      A,&FF       3.5
-----------------------------------
weiter  LD      (DE),A      6.5
        INC     DE          3
        SBC     HL,BC       7.5
        JP      NZ,weiter   5
-----------------------------------
        RET                 5
```

Die ersten vier Zeilen des Programmes und die letzte werden bei dem Programmlauf nur einmal durchlaufen, d.h. die entsprechenden Zeiten werden nur einmal gerechnet. Die Summe ist $23.5*10^{-6}$ sec.

Der Programmbereich, der durch die Sprungmarke (Label) "weiter" gekennzeichnet ist, und den ich zur Übersicht zwischen die gestrichelte Linie gesetzt habe, ist eine endliche Schleife, die &4000 mal durchlaufen wird. Demnach wird die Summe dieser vier Zeilen mit &4000 multipliziert. Da jene Summe 22 ist, ergibt sich für die gesamte Ausführung der Schleife $\&4000*22^{-6}=0.360448$ Sekunden.
Plus dem einmaligen Durchlauf der übrigen fünf Zeilen ergibt sich eine Ausführungszeit des gesamten Programmes von 0.3604715 Sekunden.

Daß der Sprungbefehl etwas schneller arbeitet, wenn die Bedingung negativ ist, soll hier nur der Vollständigkeit halber erwähnt werden, fällt aber in der gesamten Rechnung mit wenigen Bruchteilen von Mikrosekunden nicht ins Gewicht.

Doch wir wollen die Routine weiter beschleunigen. Dazu müssen wir uns noch einmal klarmachen, was geschieht, wenn ein Registerinhalt fortlaufend erhöht wird. Irgendwann ist dann die größte Zahl erreicht, die dieses Register darzustellen in der Lage ist. Darüber hinaus wird das Register wieder zu null und kann von neuem aufgezählt werden.

Und damit wären wir an dem Punkt angelangt, wo unsere weiteren Überlegungen zum Beschleunigen der Löschroutine ansetzen sollen. Wenn nämlich das betreffende Register gerade null wurde, hat es auch gerade den für uns interessanten Bildschirmspeicherbereich verlassen. Dies soll für unsere Routine die Endbedingung sein.

Bisher haben wir ein separates Zählregister (HL=&4000) neben dem Adressregister DE benutzt. Die zusätzliche Zeit, die wir beim Subtrahieren von HL aufwendeten, wollen wir im folgenden Beispiel sparen. Das Adressregister DE wird ja nach der letzten Adresse &FFFF beim INCrementieren auch wieder zu null.
Nachdem nun das 'Zählwerk' wegrationalisiert wurde, sieht unser Programm wie folgt aus.

```
        LD    DE,&C000    Startadresse
        LD    A,&FF       zu speichernder Wert
weiter  LD    (DE),A      Speichern
        INC   DE          Adr. erneuern
        JP    NZ,weiter   Sprung bei Bedingung
        RET               Zurück ins Hauptprogr.
```

Dies scheint eine praktische Lösung zu sein. Drei Zeilen wurden gespart, zwei Doppelregister (HL und BC) stehen für andere Zwecke zur Verfügung, und, was sehr wichtig ist, die Zeile SBC HL,BC wird nicht mehr durchlaufen. Dies hätte eine Zeitersparnis von 0.12288 Sekunden zur Folge, das entspricht etwa 35% des ersten Programmentwurfes.

Leider gibt es dabei einen Haken, auf den jeder Assembler-Neuling irgendwann einmal (oder auch öfter) hereinfällt.

Der Befehl INC DE hat keinen Einfluß auf das Flagregister, aus dem der bedingte Sprungbefehl seine Informationen bezieht. Auch wenn DE auf null inkrementiert wird, bleibt das Zero-Flag unbeeinflußt, alle anderen Flags übrigens auch.

Die Folge daraus, daß die Abbruchbedingung nie zustande kommt:
Zuerst wird der gesamte Bildschirmbereich mit &FF belegt. Dann wird das Adressregister DE zu null erhöht. Dadurch gelangt man wieder in den untersten Speicherbereich des CPC. Dort liegen einige, für das System 'lebenswichtige' Routinen, die nun überschrieben werden. Ein Absturz des Rechners ist nun zur unvermeidlichen Tatsache geworden.

Aber unsere Vorarbeiten waren nicht nutzlos. Man muß nur auf irgendeinem Weg dafür sorgen, daß nach dem INC DE das Zero-Flag entsprechend dem Ergebnis des INC-Befehles gesetzt wird. Der Befehl BIT 7,D bietet sich da an. Wie er funktioniert, beschreibe ich nach dem folgenden Listing des verbesserten und lauffähigen Programmes.

```
        LD    DE,&C000
        LD    A,&FF
weiter  LD    (DE),A
        INC   DE
        BIT   7,D
        JP    NZ,weiter
        RET
```

Das Adressregister DE wird zu Beginn der Routine mit &C000 geladen. Aus der binären Schreibweise heraus erkennt man die Zustände der 16 Bits dieser Zahl:

Register	D		E	
hexad.	C	0	0	0
binär	1100	0000	0000	0000
Bit Nr.	7654	3210	7654	3210

Wie man hier sieht, sind die Bits 6 und 7 des Registers D von Anfang an gesetzt, während die anderen 14 Bits 0 sind. Beim Inkrementieren von DE werden also lediglich die niederwertigen 14 Bits verändert, die anfangs alle gleich 0 sind. Erst wenn DE den Wert &FFFF (11111111 11111111 binär) hat und ein weiteres Mal inkrementiert wird, kommt es zum Überlauf. Die

16 Stellen des Registers DE werden dabei null, also auch Bit 6 und 7 von D werden Null.
Auf diesen Punkt 'wartet' der Befehl BIT 7,D, wobei aber dieser Befehl, im Gegensatz zu den 16-Bit-Inkrementierungsbefehlen, das Zero-Flag beeinflußt.
Der Befehl setzt das Z-Flag dann auf 1, wenn das getestete Bit des Registers zu null wird.
In diesem speziellen Fall könnte man auch den Befehl BIT 6,D benutzen, da ja auch das Bit 6 des Registers D von Anfang an gesetzt ist, und gleichzeitig mit Bit 7 zurückgesetzt wird.

Dies aber nur der Vollständigkeit halber, denn beide Befehle sind absolut gleichwertig.

Der Zeitvergleich dieser Routine mit der ersten von uns untersuchten fällt etwas günstiger aus.

```
        LD      DE,&C000    5
        LD      A,&FF       3.5
weiter  LD      (DE),A      6.5
        INC     DE          3
        BIT     7,D         4
        JP      NZ,weiter   5
        RET                 5
```

Bei dem Zeitvergleich einer solchen Programmschleife können wir die wenigen Befehle, die nicht zur Schleife gehören, außer Betracht lassen, weil diese keine für uns wesentlichen Zeiträume beanspruchen, da sie pro Programmdurchlauf ja nur einmal benutzt werden.
Die Schleife des letzten Programmes benötigt nur noch 18.5 Mikrosekunden, während die erste noch 22 Mikrosekunden 'verschlang'.
Dies ist zwar eine klare, aber noch nicht befriedigende Verbesserung.

In den bisher beschriebenen Routinen wurden die Schleifen immer &4000 (16384) mal durchlaufen. Bei jedem Durchlauf wurde der Inhalt des Akku, also immer nur 1 Byte pro

Schleifendurchlauf, im Bildschirmspeicherbereich abgelegt. Dies geschieht mit dem Befehl LD (DE),A. Leider gibt es beim Z80 keinen Befehl der wie folgt o.ä. lautet: LD (DE),HL, denn damit könnte man mit einem Befehl direkt zwei Speicherplätze auf einmal belegen. Dies würde natürlich gewaltige Zeitvorteile mit sich bringen.
Zwar gibt es Befehle, die den Inhalt eines 16-Bit-Registers abspeichern können. Bei ihnen ist es aber nicht möglich, eine Adressierung so elegant zu gestalten, wie bei dem Befehl LD (DE),A, wo nur das Register DE INCrementiert werden muß.

Um nun aber doch zwei Byte mit einem Befehl ablegen zu können, muß man sich mit einem Trick behelfen.
Dieser Trick liegt darin, die Funktion des Stacks zu mißbrauchen.
Normalerweise wird der Stack benutzt, um beispielsweise die Inhalte von Registern kurzzeitig zwischenzuspeichern, wenn eine andere Routine angesprungen werden soll, in der diese Register ebenfalls, aber mit anderen Werten benutzt werden. Dazu liegt der Stack in einem Bereich des RAMs, der vor einem Zugriff des BASIC-Interpreters geschützt ist.
Der Trick liegt nun darin, den Stack in den Bildschirmbereich zu legen, um dann den schnellen Stapelbefehl PUSH rr benutzen zu können. Dieser Befehl ist mit 6.5 Mikrosekunden nur geringfügig schneller als ein doppelt ausgeführter LD (DE),A mit 7 Mikrosekunden, beinhaltet aber schon das Aktualisieren des Adresszählers, das bei dem dem Ladebefehl noch dazu kommt.
So steht der PUSH-Befehl mit 6.5 Mikrosekunden dem Befehl LD (DE),A plus zweimaligem INCrementieren mit 13 Mikrosekunden gegenüber. Dieser Zeitvorteil bedarf wohl keines weiteren Kommentares.

Der Befehl PUSH rr speichert den Inhalt jener Registers ab, die hier symbolisch durch rr dargestellt wurde. Diese Register sind BC, DE, HL und AF (und IX, IY).
Der benötigte Schleifenzähler muß nun nur noch auf &2000 gesetzt werden, da die Anzahl der Durchläufe mit PUSH rr halbiert wurde. Es werden anstatt &4000*1 Byte nun &2000*2 Byte gespeichert.

Bevor wir zum Programm kommen, muß noch eine fatale Fehlermöglichkeit ausgeschaltet werden.
Der Stackpointer (SP) wird im Programmlauf in den Bildschirmbereich gelegt. Damit er nach der Routine wieder mit seinem alten Wert belegt werden kann, muß man sich diesen 'merken'. Dies geschieht, indem man ihn in zwei Speicherplätze legt, die nicht verändert werden dürfen. Wir benutzen die Speicherplätze &A030/31, die oberhalb des Bereiches liegen, zu dem BASIC Zugriff hat. Nach unserer Programmschleife und vor dem RET-Befehl muß der SP seinen alten Wert wieder erhalten, damit eine ordentliche Rückkehr ins BASIC gewährleistet ist. Wird dies vergessen, folgt der unvermeidliche Absturz des Systems, unser bester Indikator für Programmfehler.
Dies liegt daran, daß der Befehl CALL nn, den wir zum Aufruf unserer Routinen verwenden, die Rückkehradresse und einige Parameter auf dem Stack ablegt, von dem diese Werte beim RET-Befehl wieder heruntergenommen werden. Dazu muß der SP logischerweise seinen alten Wert zurückbekommen, nachdem er zuvor in den Bildschirmspeicherbereich gelegt wurde.

Nun das Programm:

```
        LD    (&A030),SP  SP retten
        LD    SP,&FFFF    Stack in Bildbereich
        LD    HL,&3FFE    Zähler setzen
        LD    BC,&FFFF    Wert zum abspeichern
weiter  PUSH  BC          Speicher belegen
        DEC   HL          Zähler DECrementieren
        BIT   5,H         Flag setzen
        JP    NZ,weiter   bedingter Sprung
        LD    SP,(&A030)  SP aktualisieren
        RET               zurück zum Hauptprogramm
```

Die Schleife dieses Programmes benötigt 18.5 Mikrosekunden, genau wie die der letzten Routine. Hierbei werden aber in derselben Zeit zwei Byte abgespeichert, wogegen es bei der vohergehenden nun eines war.
Daran zeigt sich besonders deutlich, wie leistungsfähig der PUSH-Befehl ist.

Manchem wird aufgefallen sein, daß der Befehl LD (&A030),SP nur einen Speicherplatz adressiert, wogegen der abzulegende Wert des SP zwei Byte lang ist. Doch dafür ist dieser Befehl geeignet. In den Speicher &A030 wird das niederwertige Byte, in den nächsten Speicher, also &A031, das höherwertige Byte abgelegt.
Diese Form der Abspeicherung von 2-Byte-Wertenn, erst das Low- und dann das High-Byte, wird ausnahmlos angewendet. Es wird immer zuerst das niederwertige (L), dann das höherwertige (H) Byte abgelegt. Man sollte sich dies genauestens merken, weil ohne das Wissen um diese Tatsache sehr viele Unklarheiten auftreten können.

Beim Befehl LD SP,(&A030), bei dem der Datentransfer in umgekehrter Richtung erfolgt, wird abermals zuerst das L-Byte und dann erst das H-Byte übertragen.

Ein wichtiges Beispiel noch dazu: Soll ein Sprung zur Adresse &6533 erfolgen, so lautet der entsprechende Befehl z.B. CD 33 65.

Zum Schluß dieses Kapitels möchte ich noch eine sehr schnelle Löschroutine erklären. Mit der logischen OR-Verknüpfung wird hier das entsprechende Flag gesetzt, auf das der folgende Sprungbefehl abfragt.

```
        LD    (&A020),SP  retten des SP
        LD    SP,&FFFF    SP in Bildschirm
        LD    BC,&FFFF    Wert laden
        LD    HL,&C001    letzte Adresse
        XOR   A           Akku löschen
        LD    (&C001),A   letzte Adr. löschen

WEITER  PUSH  BC          Wert speichern
        OR    (HL)        Test auf Ende
        JP    Z,WEITER    bed. Sprung

        LD    SP,(&A020)  alten SP laden
        RET               zurück ins Hauptprg.
```

Zuerst wird der Wert des Stack-Pointers, der ja enorm wichtig ist, im Register &A020 gesichert. Diese Speicherstelle liegt hinter der Löschroutine.
Nun wird der SP in den Bildschirmbereich gelegt (LD SP,&FFFF), das Register BC mit dem abzuspeichernden Wert geladen (LD BC,&FFFF) und die Adresse des letzten zu belegeneden Speicherplatzes ins Register HL gebracht (LD HL,&C001).
Dann wird der Akku mit 0 geladen, indem man ihn mit sich selbst verXORt, eine schnelle und elegante Methode, die man sich merken sollte, weil viele Assemblerprogramme solche Tricks verwenden, durch die sie dann zwar schneller, aber auch schwerer zu durchschauen werden.
Da der Akku schon einmal null ist, kann man ihn im weiteren auch benutzen, um die Speicherzelle &C001 zu löschen, also mit 0 zu belegen. Dies ist hier wichtig, weil die Abbruchbedingung nur dann einwandfrei funktioniert, wenn dieser Speicherplatz und der Akku von Anfang an Null sind.
Wer hier Verständnisschwierigkeiten hat, sollte einmal den Schleifendurchlauf auf dem Papier simulieren, um den Sinn der Anweisung OR (HL) zu verstehen. Damit wird zudem direkt klar, warum das vorhergehende Löschen unbedingt erforderlich ist.

In der Schleife wird die Sprungbedingung erzeugt, indem ein Vergleich zwischen dem Inhalt des Akkus und dem einer Speicherzelle durchgeführt wird. Die Speicherzelle wird indirekt über das Register HL adressiert.

Das Ergebnis der OR-Verknüpfung ist solange null, wie sowohl der Akku als auch die Speicherzelle &C001 gleich null sind. Sobald der PUSH-Befehl den Wert &FF in den Speicher &C001 gebracht hat, ist das Ergebnis von OR (HL) ungleich Null und die Bedingung nicht mehr erfüllt. Damit endet die Schleife.

Nun will ich noch kurz auf den Zeitbedarf der Schleife eingehen. Was noch an Befehlen außerhalb der Schleife anfällt, können wir ja, wie bereits bekannt ist, vernachlässigen.

```
weiter  PUSH  BC          6.5
        XOR   A           3.5
        JP    NZ,weiter   5
```

Ein Schleifendurchlauf benötigt 15 Mikrosekunden. Das bedeutet 0.246 sec für den kompletten Bildschirm und lediglich 7.5 Mikrosekunden pro Byte.

Mit dem letzten Programm dürften wir eine maximale Effizienz beim Belegen des Bildschirmes erreicht haben. Mit einem Speicherbefehl (PUSH BC) pro Schleifendurchlauf dürfte keine andere Konstruktion schneller arbeiten. Wenn wir allerdings die Anzahl der PUSH-Befehle in der Schleife erhöhen, wird der zeitliche Anteil der störenden Befehle JP NZ und OR A immer kleiner. Bei 10 PUSH-Befehlen pro Schleife entfallen pro Byte 3.7 Mikrosekunden, während es bei einem PUSH pro Schleife noch 7.5 Mikrosekunden waren. Mit jedem PUSH pro Schleife mehr nähern wir uns der Grenze von 3.25 Mikrosekunden, was ja der Hälfte eines ganzen PUSH entspricht. Leider nähern wir uns mit einer solchen Programmiertechnik auch schnell den Speichergrenzen des Rechners, so daß hier ein vernünftiges Mittel gefunden werden sollte.

Die Schleife dürfte nun hinreichend erläutert worden sein, so daß schließlich nur noch zu erwähnen wäre, daß vor dem RET-Befehl der SP unbedingt aktualisiert werden muß. Dabei wird der Stapelzeiger wieder auf seinen ursprünglichen Wert zurückgesetzt, d.h. er weist jetzt wieder auf jene Speicherzellen, in der zu Anfang der Routine die Rückkehrparameter abgelegt wurden. Ein bekanntes Beispiel in BASIC ist der GOSUB-RETURN-Befehl. Erreicht BASIC einen GOSUB-Befehl, so merkt es sich, in welcher Zeile dieser stand, damit es bei Erreichen des nächsten RETURN-Befehles 'weiß', wohin es zurückspringen muß. Diese Analogie kommt natürlich nicht von ungefähr. Für solche Aufgaben steht dem Schneider-Rechner ein eigener BASIC-Stack zur Verfügung, der ebenfalls nach dem LIFO-Prinzip arbeitet.

Beim Abspeichern des Inhaltes des Doppelregisters BC sind wir natürlich nicht auf den Wert &FFFF angewiesen, den ich bisher immer zur Verdeutlichung herangezogen habe. Wir können jeden Wert zwischen &01 und &FF nehmen, nur nicht die null. In diesem Fall würde es nie zur Abbruchbedingung kommen und wir hätten einen wunderschönen Absturz programmiert.

Wer sich jetzt fit genug fühlt, dies trotzdem zu ermöglichen sollte sich sofort daran setzen. Die Befehlsfolge bleibt die gleiche, es brauchen nur ein paar Werte geändert zu werden.
Und nicht verzagen - bis die ersten größeren Erfolge eintreten wird der Rechner noch öfter abstürzen, und zum wiederholten Male das scheinbar so perfekte Programm als Fehlkonstruktion entlarven.

Wer in die Routine mehrere PUSH-Befehle einbauen will, dem sei gesagt, daß auch nicht ein Byte in den Speicherbereich direkt unterhalb &C000 gePUSHt werden darf. Die Folge wäre ein nun hinreichend bekannter Absturz des Systems.
Dies liegt daran, daß der Speicherplatz direkt unterhalb von &C000, also &BFFF, die Basis des Maschinenstacks ist.
Der beim Anschalten des Rechners reservierte Stackbereich ist &BF00 - &BFFF.

Um beim PUSHen nicht unter &C000 zu geraten, kann man z.B. HL mit einem höheren Wert belegen. Dies dürfte zunächst unklar sein. Eine 'Schreibtischprobe', d.h. den Programmlauf auf Papier simulieren, hilft bei solchen Problemen immer. Natürlich sollen Sie dabei nicht den gesamten Bildschirmbereich durchrechnen, sondern nur einen kleinen Teil dessen, um die genaue Wirkungsweise zu durchschauen.
Man sollte sich dabei nicht von dem Irrtum leiten lassen, ein solcher Test wäre Kinderkram. Bei Erstellung und Fehlersuche ist diese Vorgehensweise absolut üblich und effektiv.

Hier noch einmal das schnelle Löschprogramm in einen BASIC-Lader eingebunden. Ich habe für den absoluten Sprungbefehl einen relativen eingesetzt, der die Schleife zwar um 1Mikrosekunde verlangsamt, dafür die gesamte Routine aber relokativ macht. Relokativ bedeutet, daß sie überall im Speicher plaziert werden kann.

```
5 MODE 2:SUMME=0
10 FOR I=startwert TO startwert+28
20    READ WERT$:WERT=VAL("&"+WERT$)
30    POKE I,WERT:SUMME=SUMME+WERT:NEXT
50 IF SUMME<>3682 THEN PRINT "Fehler in DATAs"
60 DATA ED,73,20,A0,31,FF,FF,01,FF,FF,21,01,C0,AF
70 DATA 32,01,C0,C5,B6,28,FC,ED,7B,20,A0,C9,00,00
```

13.4 Die leistungsstarken Befehle des Z80

Der Z80-Mikroprozessor verfügt über mehrere hundert Befehle. Dies erscheint auf den ersten Blick ein kaum überwindbares Hindernis zu sein, um sich schnell in diese Technik einzuarbeiten. Aber keine Sorge. Bei näherer Betrachtung stellt sich schnell heraus, daß man große Mengen an Befehlen zu Gruppen zusammenfassen kann, die praktisch die gleiche Funktion haben, nur für die einzelnen Register spezielle Codes besitzen.

Bevor nun einige wichtige Befehle angesprochen werden, vereinbare ich an dieser Stelle, wie ich im weiteren den Einfluß der

Befehle auf das Flag-Register darstellen werde. Unter den Kürzeln der Flagnamen stehen einzelne Zeichen die den Einfluß auf das jeweilige Flag eindeutig beschreiben.

S	Z	H	P/V	N	C	
?	*	1	?	0	-	m/t/z

*: Flag wird dem Ergebnis entsprechend beeinflußt
-: Flag bleibt unbeeinflußt
1: Flag wird gesetzt (unbedingt)
0: Flag wird zurückgesetzt (unbedingt)
?: Flag ist nach Operation unbestimmt

m: Anzahl der Maschinenzyklen
t: Anzahl der Taktzyklen
z: Angabe des Zeitbedarfs in Mikrosekunden

b: Nummer eines Bit
r: steht für die Register A, B, C, D, E, H, L

13.4.1 Einzelbitbefehle

Nun wollen wir, gut gerüstet, mal eben so ca. 1/3 aller Z80-Befehle erklären.

Da sind z.B. die 168 Befehle, die sich nur mit den einzelnen Bits der Register A, B, C, D, E, H und L befassen. Es sind die Befehle:

BIT b,r , SET b,r und RES b,r .

Den Befehl BIT b,r kennen wir ja bereits, wennauch nicht mit dieser Form der Bezeichnung.

BIT b,r

Dieser Befehl testet das durch die Parameter angegebene Bit eines Registers.

Die Paramater b und r stehen in der allgemeinen Darstellungsform der Befehle für Bitnummer (b) und Registername (r), wobei das höchstwertige Bit eines Registers immer die Nummer 7 ist und ganz links steht, während das niederwertigste Bit immer die Nummer 0 hat und im Byte ganz rechts zu finden ist. Das Kürzel r steht für die 8-Bit-Registernamen, die wohl nicht mehr aufgezählt werden müssen.

S	Z	H	P/V	N	C	
?	*	1	?	0	-	2/8/4

SET b,r

Dieser Befehl setzt das entsprechende Bit des jeweiligen Registers auf den Wert 1. Die Bedeutung der Parameter sind ja bereits bekannt.

S	Z	H	P/V	N	C	
-	-	-	-	-	-	2/8/4

RES b,r

Der Befehl setzt das beschriebene Bit zurück, also auf den Wert 0.

S	Z	H	P/V	N	C	
-	-	-	-	-	-	2/8/4

So, damit wären tatsächlich etwa ein Drittel aller Befehle erläutert. Auf ähnliche Art und Weise ist es möglich, ein weiteres Drittel kurz und bündig zu erklären. Was dann noch übrig bleibt, erweist sich allerdings deutlich zeit- und papieraufwendi-

ger. Es ist aber nicht notwendig, von Anfang an alle Befehle zu kennen. Man eignet sich diese automatisch an, wenn man mal vor einem Problem steht, welches mit den bekannten Mitteln nicht oder nur umständlich zu lösen ist. Dann blättert man im Befehlsverzeichnis und pickt sich den entsprechenden Befehl heraus, der eine leichtere Problemlösung ermöglicht.
Ein entsprechendes Verzeichnis würde den Rahmen dieses Buches allerdings sprengen. Indes verweise ich Assemblerfreunde auf entsprechende Literatur, die für wenig Geld zu erstehen ist. Was aber dieses Kapitel betrifft, ist eine solche Anschaffung nicht notwendig, weil alle verwendeten Befehle ausreichend erklärt werden.

Doch bevor wir weitere Befehle kennenlernen, sollen erst alle Einzelbitbefehle erwähnt worden sein. Es gibt nämlich noch:

BIT b,(HL) BIT b,(IX+d) BIT b,(IY+d)

Kurz beschrieben werden soll der Befehl:

BIT b,(HL)

Das Register HL beinhaltet hier die Adresse des Speichers (hier also kein Register der CPU, sondern ein Speicherplatz), dessen Bit Nummer b getestet werden soll. Diese indirekte Adressierung kann einem viel zeitraubendes Hin- und Herladen ersparen.

S	Z	H	P/V	N	C	
?	*	1	?	0	-	3/12/6

SET b,(HL) / RES b,(HL)

Diese Befehle setzen das Bit Nummer b des durch HL adressierten Registers auf 1 bzw. auf 0.

S	Z	H	P/V	N	C	
-	-	-	-	-	-	4/15/7.5

13.4.2 Rotations- und Schiebebefehle

Alle Rotations- und Schiebebefehle (bis auf die eine, berühmte Ausnahme) haben in ihrer Funktion folgende Punkte gemeinsam:

a) Der gesamte Inhalt des angesprochenen Registers wird um eine Stelle nach links/rechts verschoben. Nach einem Befehl, der den Inhalt nach links versetzt, steht der alte Inhalt des Bit 0 nun in Bit 1, der alte Inhalt von Bit 1 in Bit 2 usw.

b) Das letzte zu versetzende Bit wird ins Carry-Bit übertragen, was sich im weiteren nicht als Notlösung aus Platznot sondern als äußerst zweckmäßig erweisen wird. Bei den Bewegungen nach links ist dies das Bit 7, bei den Bewegungen nach rechts das Bit 0.

c) Logischerweise ist bei diesen Befehlen das Carry-Flag immer entsprechend beeinflußt.

d) Alle Rotations- und Schiebebefehle sind für folgende Operanden, die durch 's' dargestellt werden, anwendbar:

A B C D E H L (HL) (IX+D) (IY+D)

RL s

Der Inhalt der durch den Operanden bezeichneten Stelle wird nach links verschoben. Dabei gelangt das Bit 7 ins Carry, während dessen Inhalt nach Bit 0 verschoben wird.

S	Z	H	P/V	N	C
*	*	0	*	0	*

RR s

Der Inhalt der durch den Operanden bezeichneten Stelle wird nach rechts verschoben. Dabei gelangt das Bit 0 ins Carry, während dessen Inhalt nach Bit 7 verschoben wird.

S	Z	H	P/V	N	C
*	*	0	*	0	*

RLC s

Der Inhalt der durch den Operanden bezeichneten Stelle wird nach links verschoben. Dabei gelangt das Bit 7 zurück nach Bit 0 und zusätzlich ins Carry.

S	Z	H	P/V	N	C
*	*	0	*	0	*

RRC s

Der Inhalt der durch den Operanden bezeichneten Stelle wird nach rechts verschoben. Dabei gelangt das Bit 0 in das Bit 7 und zusätzlich ins Carry.

S	Z	H	P/V	N	C
*	*	0	*	0	*

SLA s

Dieser Befehle schiebt den Inhalt der durch den Operanden bezeichneten Stelle s nach links. Das frei werdende Bit 0 wird unbedingt zurückgesetzt, d.h. mit 0 belegt. Das Bit 7 wird ins Carry geschoben.

S	Z	H	P/V	N	C
*	*	0	*	0	*

SRA s

Dieser Befehle schiebt den Inhalt der durch den Operanden bezeichneten Stelle s nach rechts. Das Bit 0 wird ins Carry geschoben. Das Bit 7 bleibt unverändert.

S	Z	H	P/V	N	C
*	*	0	*	0	*

SRL s

Dieser Befehle schiebt den Inhalt der durch den Operanden bezeichneten Stelle s nach rechts. Das Bit 0 wird ins Carry geschoben. Das Bit 7 wird unbedingt zurückgesetzt, d.h. mit 0 belegt.

S	Z	H	P/V	N	C
*	*	0	*	0	*

Für die beschriebenen Rotations- und Schiebebefehle gilt für die verschiedenen Operanden 's' folgende Zeittabelle:

s:	m	t	z
r	2	8	4
(HL)	4	15	7,5
(IX+d)	6	23	11,5
(IY+d)	6	23	11,5

Anwendungen der Rotations- und Schiebebefehle:

Nachdem wir nun einige, wohlgemerkt noch nicht alle, R-S-Befehle kennengelernt haben, wenden wir uns jetzt praktischen Beispielen zu, mit denen man den Nutzen dieser Befehle darstellen kann.

Aus verschiedensten Gründen stehen dem Datentransfer manchmal nur wenige Leitungen zur Verfügung, von denen dann lediglich eine Leitung für die eigentlichen Daten gedacht ist. Über eine Leitung kann zur selben Zeit natürlich immer nur die Dateninformation eines einzigen Bits gelangen. Diese Form der Datenübertragung, also Bit für Bit, heißt seriell.
Wir wollen uns nicht mit Einzelheiten der seriellen Datenübertragung befassen, sondern lediglich verstehen, wie man ein Byte in einzelne Bits zerlegt, so daß diese dann zur seriellen Datenübertragung verfügbar sind. An einem kurzen Beispiel ist dies leicht erklärt.

Vorausgesetzt wird, daß sich das zu übertragende Byte im Register D befindet.

```
        LD    B,8          Zähler 8-Bit pro Byte
weiter  XOR   A            Akku löschen
        SLA   D            Bit 7,D ins Carry
        JR    NC,M1        überspringe folg.Befehl
        LD    A,1          Akku ungleich 0
M1      CALL  ausgabe      springe zur Senderoutine
        DEC   B            Zähler aktualisieren
        JP    NZ,weiter    nächstes Bit
        RET
```

Zuerst wird der Zähler B mit dem Wert 8 belegt, damit wir die Anzahl der zu übertragenden Bits kontrollieren können. Dann wird der Akku mittels XOR A gelöscht. Mit SLA D wird der Inhalt des Registers D um eine Stelle nach links verschoben, wobei Bit 7 ins Carry rutscht. Beinhaltete das ehemalige Bit 7 eine 0, so ist auch das Carry-Bit nun 0. Damit der Befehl JR NC,M1 positiv aus, der Sprung nach M1 wird ausgeführt. Somit bleibt der Akku unverändert, also null. Wäre das ehemalige Bit 7 mit dem Wert 1 belegt gewesen, stünde nun im Carry ebenfalls eine 1 und der Befehl JR NC,M1 fiele negativ aus, würde also nicht ausgeführt. Damit würde im folgenden Befehl LD A,1 der Akkumulator mit 1 belegt.

Bei der Marke M1 wird nun mittels CALL ausgabe zu einer Ausgaberoutine verzweigt, die so geartet sein sollte, daß sie eine logische 0 sendet, wenn der Akku gleich null ist, oder die eine logische 1 sendet wenn der Akku ungleich null ist. Wie dies im einzelnen geschieht und wie logisch 1 und 0 aussehen, ist zum Verständnis an dieser Stelle unwesentlich.

Wichtig ist, zu erkennen, wie man mittels des Befehles SLA D das Carry-Bit so beeinflußt, daß man aufgrund seines Inhaltes einen Sprungbefehl beeinflussen kann, und damit Parameter für die Senderoutine entsprechend gestalten kann.

Ein anderes eindrucksvolles Beispiel für Verwendbarkeit des SLA r ist die multiplikative Verdoppelung eines Register- oder Speicherinhaltes.

Zuerst aber ein Beispiel, wie man eine solch einfache Multiplikation nicht programmieren sollte: 13*2

```
        LD    B,13        Multiplikant
        LD    C,2         Multiplikator
        XOR   A           Akku loeschen
weiter  ADD   A,B         additive Mult.
        DEC   C           Zähler aktual.
        JP    NZ,weiter   Sprung bei c<>0
        RET
```

Zuerst werden Multiplikand und Multiplikator in die Register B bzw. C gebracht. Dann wird der Akku gelöscht, in dem das Ergebnis aufaddiert wird. In der Schleife 'weiter' wird der Akku dann C-mal mit dem Wert des Registers B addiert.

Das folgende Listing führt zum gleichen Ergebnis:

```
        LD    A,13        Akku=13
        SLA   A           rotiere A nach links
        RET
```

Wir wollen uns die Funktion dieser Multiplikationsweise kurz anhand der binären Schreibweise verdeutlichen.

Register A	00001101 =	8+4+1=13	(vorher)
Register A	00011010 =	16+8+2=26	(nachher)

Rechts neben den Binärzahlen sind die Summanden der gesetzten Binärstellen in dezimaker Form aufgeführt. Verschiebt man die Binärstellen nach links, so bekommt jede Stelle die nächst höhere Wertigkeit, nämlich immer die doppelte. Wenn sich also die Summanden verdoppeln, verdoppelt sich folgerichtig auch die resultierende Summe.

Frage: Was passiert, wenn die zu verdoppelnde Zahl größer als 127 ist?

Es tritt ein Carry auf. Um aber dieses Ergebnis ordentlich verwalten zu können, muß ein zweites Register her, um anfallende Carrys aufzufangen.
Anhand des folgenden Beispieles 160*4, in dem das Register L mit 160 belegt und zwei mal verdoppelt wird, soll das Zusammenspiel von zwei Registern verdeutlicht werden.

```
        LD    H,0         Reg. H löschen
        LD    L,160        "   L =160
        LD    B,2         Zähler
weiter  SLA   L           schiebe L links
        SLA   H           schiebe H links
        OR    A           Carry löschen
        DEC   B           Zähler aktual.
        JP    NZ,weiter
        RET
```

Zuerst wird H gelöscht, L mit 160 geladen und B als Zähler gesetzt. In der folgenden Schleife wird immer zuerst das L-Register geschoben, weil ja dessen Inhalt beim Schieben das

Carry setzt, welches erst dann in H hineingeschoben werden soll. Da man die Register H und L als ein 16-Bit-Register behandeln kann, hat man nach dem Durchlauf dieser Routine in HL das korrekte Ergebnis.
Der Befehl OR A steht nur der Vollständigkeit halber in der Schleife. Er soll ein mögliches Carry von H löschen, weil dieses ansonsten beim erneuten Schleifendurchlauf in Bit 0 des Registers L geschoben würde.

Zur Veranschaulichung dieser ungewohnten Rechenweise ist dieses Beispiel im folgenden noch einmal binär dargestellt.

Register	H	L	
	00000000	10100000	= 128+ 32=160 (vorher)
	00000001	01000000	
	00000010	10000000	= 512+128=640 (nachher)

Wer schon einmal eine Assembler-Befehlsliste durchgestöbert hat, wird dabei erkannt haben, daß der Z80 keine 16-Bit-Rotations- oder Schiebebefehle zur Verfügung stellt.

Und er hat doch einen.

In den vorhergehenden Beispielen wurde u.a. ein nicht vorhandener Multiplikationsbefehl durch Linksschieben eines Register substituiert. Nun drehen wir den Spieß um, und 'erzeugen' uns einen Schiebebefehl mit einem Additionsbefehl.

Wie wir wissen, kann man ja das Doppelregister HL als Quasi-16-Bit-Akkumulator betrachten. Zu HL kann man die Register BC, DE, SP und HL addieren. Da man HL mit sich selbst additiv verknüpfen kann, was ja der Multiplikation mit dem Faktor 2 entspricht, muß das Ergebnis dem eines Linksschiebebefehls entsprechen.

Der 16-Bit-Befehl ADD HL,HL entspricht prinzipiell also dem 8-Bit-Befehl SLA r. Die Inhalte der betreffenden Register werden nach links geschoben, das höchstwertige Bit kommt ins Carry und in das niederwertigste wird eine 0 geschoben.

13.4.3 16-Bit-Arithmetikbefehle

ADD HL,rr

Der Inhalt des durch rr bezeichneten Doppelregisters wird mit dem Inhalt von HL addiert und im Register HL abgelegt. Für rr kommen die Register BC, DE, HL und SP in Frage. Das Half-Carry-Bit wird durch einen Übertrag von Bit 11 nach Bit 12 gesetzt, d.h. vom niederwertigen Nibble des Registers H zum höherwertigen. Es wird nicht angezeigt, ob von L nach H ein Übertrag entsteht.

S	Z	H	P/V	N	C	
-	-	*	/	0	*	3/11/5.5

Mit diesem Befehl kann auf einfache Weise eine nicht im Z80 implimentierte 16-Bit-Rotation verwirklichen.

Wie bei dem echten Akkumulator kann man mit dem Doppelregister HL neben der einfachen Addition auch eine mit Carry ausführen. Der Befehl dazu lautet:

ADC HL,rr

Die Inhalte der Register HL und des durch rr dargestellten Doppelregisters werden addiert. Zu dieser Summe wird noch der Inhalt des Carry-Flags addiert. Für rr kommen die Register BC, DE, HL und SP in Frage. Bis auf das N-Flag werden alle Flags entsprechend dem Ergebnis beeinflußt.

S	Z	H	P/V	N	C	
*	*	*	*	0	*	4/15/7.5

Mit dem folgenden 16-Bit-Befehl sind dann alle arithmetischen Fähigkeiten des HL-Akkus beschrieben.

SBC HL,rr

Der Inhalt des mit rr gekennzeichneten Doppelregisters und der Übertrag werden vom Inhalt des Registers HL subtrahiert. Das Ergebnis dieser Subtraktion wird wieder in HL abgelegt. Für 'rr' können die Register BC, DE, HL und SP eingesetzt werden. Bis auf das N-Flag werden alle Flags entsprechend dem Ergebnis beeinflußt.

S	Z	H	P/V	N	C	
*	*	*	*	1	*	4/15/7.5

13.4.4 Block-Befehle

Wenn man zusammenhängende Speicherbereiche des RAM an einen anderen Platz transferieren will, stehen einem dazu die wohl leistungsfähigsten Befehle des Z80 zur Verfügung. Es sind die Befehle:

LDI	Laden mit INCrementieren
LDIR	Laden mit INCrementieren mit Abbruch
LDD	Laden mit DECremenetieren
LDDR	Laden mit DECremenetieren mit Abbruch

Jeder dieser Befehle besteht im Prinzip aus mehreren Assemblerbefehlen, die nacheinander ablaufen.

Blocktransfer-Befehle

LDI

LDI LD (DE),(HL); INC HL; INC DE; DEC BC

Der Inhalt des Speicherplatzes, der durch HL angegeben ist, wird in den Speicherplatz geladen, welcher durch DE adressiert ist. Dann werden die Doppelregister HL und DE INCrementiert. Zum Schluß wird dann BC DECrementiert.

S	Z	H	P/V	N	C	
-	-	0	*	0	-	4/16/8

LDIR

LDI LD (DE),(HL); INC HL; INC DE; DEC BC bis BC=0

Der Inhalt des Speicherplatzes, der durch HL angegeben ist, wird in den Speicherplatz geladen, der durch DE adressiert ist. Dann werden die Doppelregister HL und DE INCrementiert. Zum Schluß wird dann BC DECrementiert.
Diese Befehlssequenz wird solange wiederholt, bis das Register BC zu null DECrementiert wurde.

S	Z	H	P/V	N	C	
-	-	0	0	0	-	4/16/8 (BC=0) 5/21/10.5 (BC<>0)

LDD

LDI LD (DE),(HL); DEC HL; DEC DE; DEC BC

Der Inhalt des Speicherplatzes, der durch HL angegeben ist, wird in den Speicherplatz geladen, der durch DE adressiert ist. Dann werden die Doppelregister HL und DE DECrementiert. Zum Schluß wird dann BC DECrementiert.

S	Z	H	P/V	N	C	
-	-	0	*	0	-	4/16/8

LDDR

LDI LD (DE),(HL); DEC HL; DEC DE; DEC BC bis BC=0

Der Inhalt des Speicherplatzes, der durch HL angegeben ist, wird in den Speicherplatz geladen, der durch DE adressiert ist. Dann werden die Doppelregister HL und DE DECrementiert. Zum Schluß wird dann BC DECrementiert.
Diese Befehlssequenz wird solange wiederholt, bis das Register BC zu null DECrementiert wurde.

S	Z	H	P/V	N	C	
-	-	0	0	0	-	4/16/8 (BC=0) 5/21/10.5 (BC<>0)

Hier noch einmal eine knappe Zusammenfassung:

LDI	LD (DE),(HL); INC HL; INC DE; DEC BC
LDIR	LD (DE),(HL); INC HL; INC DE; DEC BC bis BC=0
LDD	LD (DE),(HL); DEC HL; DEC DE; DEC BC
LDDR	LD (DE),(HL); DEC HL; DEC DE; DEC BC bis BC=0

Für alle Besitzer eines CPC 464 oder 664, die etwas neidvoll auf die zusätzlichen Bankbefehle des 6128 schauen, ist die folgende

Routine gedacht, die das Kopieren von Speicherblöcken ermöglicht.

```
LD    BC,&4000    Zähler setzen
LD    DE,&C000    Zieladresse
LD    HL,&4000    Quelladresse
LDIR              Blockladebefehl
RET
```

Sie können dieses Hochleistungsprogrämmchen mit dem folgenden BASIC-Lader eingeben:

```
5 MODE 2:SUMME=0
10 FOR I=startwert TO startwert+11
20    READ WERT$:WERT=VAL("&"+WERT$)
30    POKE I,WERT:SUMME=SUMME+WERT:NEXT
40 IF SUMME<>985 THEN PRINT "Fehler in DATAs"
50 DATA 01,00,40,11,00,40,21,00,C0,ED,B0,C9
```

Nachdem Sie die Routine mit diesem Programm geladen haben, starten Sie es mittels CALL startwert.
Außer einer READY-Meldung ist noch nichts zu erkennen. Diese Routine hat nämlich den Inhalt des Speicherblock 3, also den Bildschirm-Speicherbereich in den Block 1 des RAM kopiert.
Tauschen Sie jetzt bitte die unterstrichenen Werte der DATA-Liste gegeneinander aus. Es sind die High-Bytes der Block-Anfangsadressen. Nach dem erneuten Start des Ladeprogramms wird durch CALL startwert der Block 1 in den Bildschirmbereich zurück kopiert. Bevor Sie allerdings mit CALL starten, sollten Sie den Bildschirm kurz mit MODE 2 löschen, damit der Effekt besser zu erkennen ist.

Wohin man den Bildschirmbereich kopiert, bleibt Ihnen überlassen. Sie sollten nur darauf achten, daß nach unten keine Kollision mit BASIC-Programmen und nach oben keine Überschneidungen mit dem vom Betriebssystem reservierten Bereich erfolgen. Die Länge ist ebenfalls frei zu wählen. Praktischerweise habe ich zur Demonstration die Länge des Bildschirmspeichers gewählt.

14. Einige Maschinenroutinen zur Bildschirmbehandlung

14.1 Weiches Bildschirmscrollen

Wenn der Rechner bei einer Ausgabe auf dem Bildschirm an den unteren Rand gelangt, schafft er sich neuen Platz für die Ausgabe, indem er den Bildschirminhalt um jeweils eine Schriftzeichenhöhe nach oben rückt. Die Schriftzeichenhöhe beträgt acht Pixelreihen. Der Rechner verschiebt dabei aber nicht die Speicherinhalte, sondern ändert den Offset.
Der Speicherplatz, dessen Inhalt die linke obere Ecke darstellt, ist dann nicht mehr &C000. Ein anderer Speicher wird nun als Bildschirmbasis benutzt.
Mit folgendem Minilader können Sie diese Art von Scrollen ausprobieren:

```
10 FOR I%=&A000 TO &A005
20     READ WERT$:WERT=VAL("&"+WERT$)
30     POKE I%,WERT
30     NEXT
40 DATA 06,01,CD,4D,BC,C9
```

Soll der Bildschirm herunter gescrollt werden, tauschen Sie den gekennzeichneten Wert gegen 0 aus. Sie sollten immer darauf achten, daß der Bildschirm nicht bereits gescrollt ist, da es sonst durch den verschobenen Offset zu unbrauchbaren Ergebnissen kommen kann.
Im folgenden ist ein BASIC-Lader aufgeführt, der ein Scrollen realisiert, das jede Pixelreihe einzeln verschiebt. Der Inhalt wird hierbei oben aus dem Bidschirm herausgescrollt.

```
10 MODE 2:SUMME=0
20 FOR I%=&A000 TO &A036
30    READ WERT$:WERT=VAL("&"+WERT$)
40    POKE I%,WERT:SUMME=SUMME+WERT
50    NEXT
60    IF SUMME<>5695 THEN PRINT"Fehler in DATAs"
100 DATA 01,C7,C7,21,00,C0,22,00,B0,C5,2A,00,B0,54,5D,CD
110 DATA 26,BC,22,00,B0,01,50,00,ED,B0,C1,CD,09,BB,38,15
120 DATA 0D,20,E6,C5,11,80,FF,21,36,A0,01,50,00,ED,B0,C1
130 DATA 0E,C7,05,20,CE,C9,00,00,00
```

Soll der Bildinhalt unten herausscrollen, so ersetzen Sie die Zeilen 100 und 110 wie folgt:

```
100 DATA 01,C8,C7,21,80,FF,22,00,B0,C5,2A,00,B0,54,5D,CD
110 DATA 29,BC,22,00,B0,01,50,00,ED,B0,C1,CD,09,BB,38,15
120 DATA 0D,20,E6,C5,11,00,C0,21,36,A0,01,50,00,ED,B0,C1
```

Die Prüfsumme ist in diesem Fall 5699.

Wenn nun nicht der gesamte Bildbereich gescrollt werden soll, sondern nur ein Teil am linken Rand, so geben Sie z.B. folgende Befehle nach dem obigen Programmlauf ein:

```
POKE &A016,10 : POKE &A02B,10
```

Die 10 steht dabei für die Breite des zu scrollenden Bereiches am linken Bildrand.

Nun noch kurz das obige Programm in abgespeckter und beschleunigter Form.
Es ist deutlich schneller, kann aber nicht abgebrochen werden. Es stoppt automatisch nach dem Rausscrollen des gesamten Screens. Das oben benutzte Ladeprogramm kann verwendet

werden. Die FOR-NEXT-Schleife läuft dabei von &A000 - &A02F und die SUMME ist 4368. Die DATA-Zeilen werden gegen die folgenden getauscht:

```
100 DATA 01,C8,C8,21,80,FF,22,00,B0,C5,2A,00,B0,54,5D,01
110 DATA 00,08,ED,42,CB,74,20,04,01,B0,3F,09,22,00,B0,01
120 DATA 50,00,ED,B0,C1,0D,20,E1,0E,C8,05,20,D6,C9,00,00
```

14.2 Seitliches Scrollen der untersten Bildschirmzeile

Zur hervorhebenden Darstellung von Textelementen gibt es verschiedene Mittel. Zum einen die inverse oder andersfarbige Darstellung von Zeichen, das Einrahmen und Unterstreichen mit markanten Sonderzeichen, die Anzeige in blinkender Form usw. Einen interessanten, neuen Effekt bietet eine Laufanzeige, die Text oder Textabschnitte unübersehbar verschiebt.
Mit dem folgenden Programm kann die unterste Bildschirmzeile nach links gescrollt werden, wobei der links aus dem Bildbereich austretende Text von rechts wieder hereingescrollt wird.

```
A000  117702    ANF   LD    DE,&277      ein Durchgang
A003  D5              PUSH  DE
A004  114F08          LD    DE,&84F      Summand-Anf.Adresse
A007  ED5341A0        LD    (&A041),DE   Summand. retten
A00B  01084F    M2    LD    BC,&4F08     Zähler-8-Z./&4F-Sp.
A00E  21CEC7          LD    HL,&C7CE     Anf.Adresse
A011  B7              OR    A            lösche Carry
A012  CB16      M1    RL    (HL)
A014  2B              DEC   HL           neue Adresse
A015  05              DEC   B            Zeile zu Ende?
A016  20FA            JR    NZ,M1
A018  114F00          LD    DE,&004F
A01B  3005            JR    NC,M3        Carry Zeilenende überneh?
A01D  19              ADD   HL,DE        alte erste Adresse
A01E  CBC6            SET   0,(HL)       rechtes Bit setzen
A020  1803            JR    M4           überspringen RES
A022  19        M3    ADD   HL,DE        alte erste Adresse
A023  CB86            RES   0,(HL)       rechtes Bit löschen
A025  ED52      M4    SBC   HL,DE
```

```
A027  064F      LD    B,&4F       Zeilenzähler neu
A029  ED5B41A0  LD    DE,(&A041)  Summand holen
A02D  19        ADD   HL,DE
A02E  0D        DEC   C
A02F  20E1      JR    NZ,M1
A031  D1        POP   DE          Zähler Anzahl-Sp.
A032  1B        DEC   DE
A033  CD09BB    CALL  &BB09       Taste gedrückt?
A036  3808      JR    C,ENDE      Abbruch
A038  D5        PUSH  DE
A039  CB7A      BIT   7,D         Abfrage Zählerende
A03B  28CE      JR    Z,M2
A03D  D1        POP   DE          Stackkorrektur
A03E  18C0      JR    ANF         neue Runde
A040  C9        ENDE  RET
A041                              Speicher für Summanden
A042                              Speicher für Summanden
```

Der Bildschirm darf vor dem Aufruf dieser Routine nicht gescrollt werden, da sonst der Offset verändert würde. Das Programm ist für den Betrieb in MODE 2 ausgelegt. In den beiden übrigen MODEs entstehen unbrauchbare, aber interessante Effekte.
Mit den relativen Sprungbefehlen ist dieses Programm prinzipiell relokativ. Es kann überall im freien Speicherbereich des RAM plaziert werden. Vorsicht ist aber bei den Speicherplätzen &A041 und &A042 geboten. In diese wird das Register DE zwischenzeitlich gespeichert. Sollte das Programm einmal an eine andere Stelle geladen werden, so sollten auch diese beisen Speicherplätze entsprechend verändert werden.

Der BASIC-Lader:

```
10 MODE 2:SUMME=0
20 FOR I=&A000 TO &A042
30 READ WERT$:WERT=VAL("&"+WERT$)
40 POKE I,WERT:SUMME=SUMME+WERT
50 NEXT I
60 IF SUMME<>6591 THEN PRINT"Fehler in den DATA-Zeilen"
70 END
100 DATA 11,77,02,D5,11,4F,08,ED,53,41,A0,01,08,4F,21
110 DATA CE,C7,B7,CB,16,2B,05,20,FA,11,4F,00,30,05,19
120 DATA CB,C6,18,03,19,CB,86,ED,52,06,4F,ED,5B,41,A0
130 DATA 19,0D,20,E1,D1,1B,CD,09,BB,38,08,D5,CB,7A,28
140 DATA CE,D1,18,C0,C9,00,00,00
```

Nachdem Sie das Ladeprogramm gestartet haben, ist die Routine im Speicher abgelegt und kann mit CALL &A000 gestartet bzw. mit einem beliebigen Tastendruck abgebrochen werden. Das Ladeprogramm wird nun nicht mehr gebraucht und kann mit DELETE 1- gelöscht werden. Wenn Sie den Speicherbereich der Maschinenroutine mittels MEMORY &A000-1 geschützt haben, kann das Ladeprogramm auch mit NEW gelöscht werden.

Beispiel für die Nutzung der Routine:

```
10 LOCATE 5,25 : PRINT"Wiederholen Sie die Eingabe"
20 CALL &A000
30 LOCATE 1,25 : PRINT STRING$(80," ");
40 MODE 2
50 INPUT"Neue Eingabe";A
60 REM ...etc...
```

14.3 Screencopy für CPC 464/664

Dieser Abschnitt ist besonders für diejenigen unter den CPC-Besitzern gedacht, die einen 464 oder einen 664 ihr eigen nennen und damit nicht die Möglichkeiten bezüglich der Bild-

schirmverwaltung haben, die den Besitzern eines 6128 zur Verfügung stehen.

Mit SCREENCOPY kann man den Inhalt eines Blocks des RAM in einen anderen Block kopieren. Ein Block ist ein zusammenhängender Speicherbereich mit einer Länge von 16 KByte. Da der RAM-Bereich des 464/664 insgesamt 64 KByte groß ist, besteht er aus vier Blocks.

	Anfangsadresse	Endadresse
Block 0	&0000	&3FFF
Block 1	&4000	&4FFF
Block 2	&8000	&BFFF
Block 3	&C000	&FFFF

Obwohl nach dem Einschalten des Rechners dem Bediener mehr als 42000 Byte frei zur Verfügung stehen, ist nur der Block 1 völlig frei zu nutzen.

Anmerkungen zu den Speicherblocks:

Im Block 0 liegen in den unteren (ca. 60) Speicherplätzen 'lebensnotwendige' Routinen des Betriebssystems. Ansonsten ist auch dieser Block frei verfügbar.
Warum 'lebensnotwendig'??
Bringen Sie einmal den Wert 0 in den Speicher 8 mit POKE 8,0, aber nur wenn keine wichtigen Daten oder Programme im Rechner sind!!!

Wie schon erwähnt, ist Block 1 komplett verfügbar.

Im Block 2 sind die Speicherplätze &8000 - &A67B nach dem Einschalten zu benutzen. Der Bereich von &A67C bis &BFFF (dez. 42620 - 49151) ist ebenfalls unverzichtbar für das Betriebssystem. Veränderungen in diesem Bereich sollten wie auch im unteren Bereich mit großer Vorsicht geschehen, d.h. es sollten keine wesentlichen Daten im Rechner sein, und die

Diskette sollte entfernt werden. Diese könnte bei einem Systemabsturz 'übergebügelt' werden.

Der Block 3 enthält die Bildschirminformationen. Verändert man seinen Inhalt, so verändert sich auch die Anzeige.

Wie der Befehl SCREENCOPY arbeitet, kann man dem Beispielprogramm entnehmen, das im Anschluß an die Beschreibung des Maschinenbefehls LDIR gezeigt wurde. In diesem Beispielprogramm wurde der Bildschirmspeicherbereich in den Block 9 kopiert. Wenn in diesem Block 1 erst einmal ein Bildinhalt gespeichert ist, so kann man mit einem Kniff innerhalb weniger tausendstel Sekunden die Bildschirmanzeige umschalten.

Erklärung:

Nach dem Einschalten oder nach einem RESET des Rechners holt sich die Hardware die Informationen für den Monitor aus Block 3, also aus dem Speicherbereich, der mit der Adresse &C000 beginnt. Wenn man die Anfangsadresse jedoch verändert, z.B. auf den Wert &4000 (erste Adresse von Block 1) wird das Bild aus den Daten der 16383 auf &4000 folgenden Speicherplätzen zusammengesetzt.
Liegt dort bereits ein anderer Bildinhalt, so wird dieser nun angezeigt.
Mit folgendem Programm können Sie die Bildschirmbasis verändern:

```
10 FOR I=&A000 TO &A005
20    READ WERT$
30    WERT=VAL("&"+WERT$):POKE I,WERT
40    NEXT
50    DATA 3E,00,CD,08,BC,C9
60 INPUT"Welcher Block   1 / 3 ",B
70 IF B=1 THEN POKE &A001,&40 ELSE POKE &A001,&C0
80 CALL &A000
```

Das disassemblierte Listing der Maschinenroutine sieht folgendermaßen aus:

```
A000    3E xx          LD      A,xx
A002    CD 08 BC       CALL    BC08
A005    C9             RET
```

In der ersten Zeile wird der Akku mit dem signifikanten Byte der Bildschirmadresse geladen. Im Akku wird also der Parameter für die Routine SCR SET BASE übergeben, die mit CALL &BC08 aufgerufen wird. Das RETurn in der letzten Zeile bedarf keines Kommentares mehr.
Die POKE-Befehle in Zeile 70 des BASIC-Programmes dienen also dazu, das High-Byte der gewählten Bildschirm-Anfangsadresse so in den Speicher zu POKEn, daß der Akku-Ladebefehl einen sinnvollen Wert bekommt.

Eine interessante Variante für den Bildinhalt entsteht, wenn man nach dem Starten des BASIC-Programmes den Ablauf mit zweimaligem ESC abbricht, MODE 2 und dann den Befehl CALL &BC08 eingibt.
Auf dem Bildschirm erscheinen mehrere Reihen von Punkten. Die oberste zeigt die 'lebensnotwendigen' Systemroutinen und die anderen vier Reihen stellen das BASIC-Programm dar, mit dem Sie die Bildschirm-Anfangsadresse verändert haben.
Sie haben nämlich den Bildbereich in den ersten Block (Block 0) gelegt und sehen nun die dort abgelegten Programmteile.
Geben Sie nun einmal ein paar BASIC-Zeilen mit Zeilennummern ein. Beachten Sie die Veränderungen wenn Sie die Eingaben mit der ENTER-Taste bestätigen.

14.4 SCREENSWAP für den 464/664

Der Unterschied von SCREENCOPY und SCREENSWAP besteht darin, daß beim SWAP die Inhalte der angegebenen Bereiche ausgetauscht werden. Beim COPY wird ein Bereich in den anderen kopiert, worauf beide Inhalte identisch sind.

Wir wissen, daß zum Austauschen der Inhalte zweier Variablen eine dritte als Zwischenspeicher gebraucht wird.
Beispiel: Vertausche Inhalte von A und B (mit x als Zwischenspeicher)

```
x=A : A=B : B=x
```

Wollen wir nun zwei Speicherblöcke austauschen, in denen Bildinhalte liegen, so haben wir keinen vollständigen dritten Block als Zwischenspeicher zu Verfügung. Deshalb vertauschen wir die Inhalte in kleineren Stücken. Diese Stücke sind in folgendem Beispiel &100 Byte lang. Sie könnten auch eine andere Länge haben, doch zu kurz sollten sie nicht sein, da die Routine dann zu langsam werden würde. Werden die Stücke dagegen zu lang, benötigen sie zuviel des freien Speicherplatzes, der ja durch einen zweiten Bildschirminhalt schon um 1/4 reduziert wurde.

Der BASIC-Lader zum SCREENSWAP 464/664:

```
5 START=&A000:SUMME=0
10 FOR I%=START TO START+&40
20    READ WERT$:WERT=VAL("&"+WERT$)
30    POKE I%,WERT:SUMME=SUMME+WERT:NEXT
40 IF SUMME<>5098 THEN PRINT "Fehler in DATAs"
50 DATA 21,00,C0,22,00,81,21,00,40,22,02,81,01
60 DATA 40,00,C5,01,00,01,11,00,80,2A,00,81,ED
70 DATA B0,01,00,01,ED,5B,00,81,2A,02,81,ED,B0
80 DATA 01,00,01,ED,5B,02,81,21,00,80,ED,B0,21
90 DATA 01,81,34,23,23,34,C1,0D,C5,20,D1,C1,C9
```

Das Assembler-Listing zum SCREENSWAP 464/664:

```
A000   21 00 C0          LD    HL,&C000
A003   22 00 81          LD    (&8100),HL
A006   21 00 40          LD    HL,&4000
A009   22 02 81          LD    (&8102),HL
A00C   01 40 00          LD    BC,&0040
```

```
A00F   C5                  PUSH   BC
A010   01 00 01   weiter    LD     BC,&0100
A013   11 00 80            LD     DE,&8000
A016   2A 00 81            LD     HL,(&8100)
A019   ED B0                LDIR
A01B   01 00 01            LD     BC,&0100
A01E   ED 5B 00 81         LD     DE,(&8100)
A022   2A 02 81            LD     HL,(&8102)
A025   ED B0                LDIR
A027   01 00 01            LD     BC,&0100
A02A   ED 5B 02 81         LD     DE,(&8102)
A02E   21 00 80            LD     HL,&8000
A031   ED B0                LDIR
A033   21 01 81            LD     HL,&8101
A036   34                  INC    (HL)
A037   23                  INC    HL
A038   23                  INC    HL
A039   34                  INC    (HL)
A03A   C1                  POP    BC
A03B   0D                  DEC    C
A03C   C5                  PUSH   BC
A03D   20 D1               JR     NZ,weiter
A03F   C1                  POP    BC
A040   C9                  RET
```

Das Programm selbst ist relokativ ausgelegt. Trotzdem kann es nicht an jeden Platz gelegt werden. Zum einen darf es nicht im Block 1 des Speichers liegen, da dieser ja als Bildschirm-Nebenspeicher durch das Programm beeinflußt wird. Zum anderen liegt der Zwischenspeicher für das SWAPen und vier Reserve-Speicherplätze unmittelbar hinter dem Block 1 in dem Bereich von &8000 - &8103. Damit fallen die Speicherplätze &4000 - &8103 als Bereich für das Programm aus.

Wenn Sie in Zeile 5 des BASIC-Laders die STARTadresse auf &8104 legen, haben Sie einen übersichtlichen Bereich (&4000 - &8144) geschaffen, in dem dann alle für das SWAPen benötigten Speicherplätze liegen.

15. Eine einfache Schnittstelle aus BASIC zu den Z80-Maschinenregistern

Es ist bekannt, daß man sich beim Programmieren manchmal mit den Maschinenroutinen des CPC besser behelfen kann als mit den entsprechenden BASIC-Befehlen.
Ein gutes Beispiel ist die Routine, die mit CALL &BB06 aufgerufen wird. Diese Routine wartet auf den nächsten Tastendruck, wobei jede beliebige Taste betätigt werden kann. Dann kehrt das Programm wieder zu dem BASIC-Programm zurück.
Die Routine &BB06 braucht keine Einsprung-Parameter.

Mit dem Befehl SPEED WRITE x kann man über die Parameter 0 und 1 die Schreibgeschwindigkeit für die Cassette wählen, wobei die höchste Geschwindigkeit mit 1 eingestellt wird. Damit ist die Ausgabe auf 2000 Baud gesetzt. Dieses ist natürlich nicht das Maximum, was der CPC zu leisten vermag.
Höhere Übertragungsraten lassen sich mittels der Routine, die über &BC68 angewählt wird, erreichen. Dazu müssen allerdings 3 Register des Z80 gezielt gesetzt werden. Mit dem folgenden Programm lassen sich alle Register leicht laden.

```
10 MODE 2:SUMME=0:GOSUB 100
20 FOR I=1 TO 10
30    READ REG$,PLATZ$:PRINT REG$;:INPUT" ";WERT$
40    IF WERT$="" THEN WERT$="00"
50    WERT=VAL("&"+WERT$):PLATZ=VAL("&"+PLATZ$)
60    POKE &A000+PLATZ,WERT
70    NEXT
80 CALL &A000:END
100 FOR I=&A000 TO &A012:READ W$:W=VAL("&"+W$)
110 POKE I,W:SUMME=SUMME+W:NEXT
120 IF SUMME<>960 THEN PRINT "Fehler in DATAs":END
130 RETURN
140 DATA 21,0,0,E5,F1,1,0,0,11,0,0,21,0,0,CD,0,0,C9,0
150 DATA A,2,F,1,B,7,C,6,D,A,E,9,H,D,L,C,HB,10,LB,F
```

Die Schleife in den Zeilen 100 und 110 plus den Daten der Zeile 140 ergeben die eigentliche Schnittstelle in Form einer kleinen Assembler-Routine ab der Speicherstelle &A000. Das Programm

fragt nach den Werten, die in die Register A, F, B, C, D, E, H, und L geladen werden sollen. Die folgenden Eingaben HB und LB beziehen sich auf das High-Byte und Low-Byte der gewählten Routine. Die Eingaben, die nicht interessieren, können mit ENTER übergangen werden.

Mit diesem einfachen Werkzeug haben wir aus BASIC heraus eine gute Möglichkeit, die Firmware des CPC kennenzulernen und zu nutzen.
Noch einmal zurück zur Schreibgeschwindigkeit des Cassettenteils. Die Übergabeparameter sind die Länge für ein halbes Null-Bit in HL und die Vorprüflänge im Register A.

1000 Baud:	HL=333	A=25
2000 Baud:	HL=167	A=50

Diese Werte ergeben eine hohe Sicherheit beim Datentransfer. Wie stark man die Ladegeschwindigkeit erhöhen kann, hängt nicht zuletzt vom Bandmaterial ab.

16. Die Unterbringung von Maschinenprogrammen im Speicher

Die übliche Methode zum Speichern von Maschinenprogrammen kennen Sie sicherlich. Die aus Datenzeilen gelesenen Werte werden oberhalb des BASIC RAMs mit POKE in den Speicher geschrieben. Zuvor wurde der benutzte Speicherbereich mit MEMORY reserviert. Diese Methode hat einen Nachteil:

Da die eigenen Symboldefinitionen auch oberhalb des BASIC-Endes gespeichert werden, kommt es hier zu Überlappungen. In dem Wissen um diese Probleme haben die Systemprogrammierer eine Sperre eingebaut. Wurde mit MEMORY die Speicherobergrenze unterhalb des Endes der Symboldefinition gelegt, so gibt jeder darauffolgende "SYMBOL AFTER"-Befehl die Fehlermeldung "inproper argument". Damit wird verhindert, daß bei der Vergrößerung des Symbolbereiches eventuell vorhandene Maschinenprogramme überschrieben werden. Einerseits ist dies für den Schutz der Maschinenprogramme sinnvoll, andererseits erzeugt jedes "SYMBOL AFTER" sofort eine Fehlermeldung. Das bedeutet, daß z.B. bei jedem Programmablauf bis auf den ersten die Programmausführung bei "SYMBOL AFTER" abbricht. Eine Möglichkeit wäre von vornherein "SYMBOL AFTER 0" im Direktmodus einzugeben.

Für "anständige" Programme ist das aber keine befriedigende Lösung.

Eine zweite Möglichkeit bestünde darin, den Befehl in die erste Zeile zu setzen und nach dem ersten Durchlauf zu löschen(!).

Für den CPC 464 gilt folgendes:

```
10 SYMBOL AFTER 100
20 a=PEEK(&AE81)+256*PEEK(&AE82)+1
30 POKE a+4,&C5
40 FOR i=a+5 TO a+PEEK(a)+256*PEEK(a+1)-2:POKE i,32:NEXT
50 MEMORY &A000
        .
        .
        .
```

Für den CPC 664/6128 gilt:

```
10 SYMBOL AFTER 100
20 a=PEEK(&AE64)+256*PEEK(&AE65)+1
30 POKE a+4,&C5
40 FOR i=a+5 TO a+PEEK(a)+256*PEEK(a+1)-2:POKE i,32:NEXT
50 MEMORY &A000
        .
        .
```

Eine andere Methode, die variable Symboldefinitionen auch mitten in Programmen zuläßt, verändert vorübergehend den internen Himem-Zeiger, so daß "SYMBOL AFTER" wieder erlaubt ist. Wenn Sie diese Methode benutzen, müssen Sie jedoch auf jeden Fall darauf achten, daß Ihre Maschinenprogramme nicht doch durch Symboldefinitionen überschrieben werden.

Für den CPC 664 und CPC 6128 gilt:

```
10 MEMORY &9FFF
20 REM ....
30 oldhimem=PEEK(&AE5E)+256*PEEK(&AE5F)
40 POKE &AE5E,PEEK(&AE60):POKE &AE5F,PEEK(&AE61)
50 POKE &B735,0
60 SYMBOL AFTER 200
70 MEMORY oldhimem
80 REM ....
```

Für den 464 gilt:

```
10 MEMORY &9FFF
20 REM ....
30 oldhimem=PEEK(&AE7B)+256*PEEK(&AE7C)
40 POKE &AE7B,PEEK(&AE7D):POKE &AE7C,PEEK(&AE7E)
50 POKE &B295,0
60 SYMBOL AFTER 200
70 MEMORY oldhimem
80 REM ....
```

Da alle diese Methoden jedoch nur Behelfe sind, lernen Sie im folgenden noch einige andere kennen, die von Fall zu Fall unterschiedlich vorteilhaft eingesetzt werden können.
Zunächst zeigen wir Ihnen zwei Möglichkeiten, die prinzipiell dasselbe Verfahren, aber einen anderen Speicherbereich benutzen.

Anstatt Maschinenprogramme oberhalb des BASIC-Bereiches zu speichern, ist ein Abspeichern natürlich auch unterhalb des BASIC-Programmspeichers möglich.

Der "Start des BASIC-Programm minus eins Zeiger" steht an Adresse &AE81/2 (bei 664/6128: &AE64/5). Normalerweise zeigt er auf die Adresse &16F, d.h., BASIC-Programme beginnen an Adresse &170.

Stellen wir diesen Zeiger auf einen größeren Wert, den wir "Lowmem" nennen, so steht der Bereich von &170 bis Lowmem für Maschinenprogramme zur Verfügung. Also:

```
POKE &AE64,INT(Lowmem/256)
POKE &AE65,Lowmem-INT(Lowmem/256)*256
NEW
```

(464: für &AE69/5 steht 6AE81/2)

NEW muß direkt anschließend eingegeben werden, da sonst diverse Zeiger, z.B. für die Variablentabelle, nicht richtig gesetzt werden.

Die Verlegung des Bereiches ist bei der Programmierung im BASIC nicht bemerkbar. Auch "SYMBOL AFTER" funktioniert einwandfrei. Damit eignet sich der Bereich unterhalb des BASIC-Programms hervorragend zum Speichern von Maschinenprogrammen.

Mit gewissen Einschränkungen ist es möglich, Maschinenprogramme im System RAM-zu speichern. Ein so zu benutzender RAM-Bereich ist der Keybelegungsspeicher. Seine Startadresse steht an Adresse &B4E1/2 (664/6128: &B62B/C). Normalerweise beginnt die Tabelle an Adresse &B446 (664/6128: &B590); sie ist 152 Bytes lang. Der Nachteil liegt darin, daß die KEY-Belegung der Tasten nicht mehr verwendbar ist. Damit das nicht trotzdem geschieht, sollten die erweiterten Tasten mit KEY DEF auf ihre normale Funktion zurückgesetzt werden.

Auch eine Speicherung der Tabelle der vom Anwender selbstdefinierten Zeichen ist möglich. Allerdings können die überschriebenen Zeichendefinitionen nicht mehr benutzt werden. Der Code des ersten in der Symboltabelle stehenden Zeichens steht an Adresse &B294 (664/6128: &B734). Die Größe ergibt sich aus:

```
PRINT (256-PEEK(&B294))*8   für 464
PRINT (256-PEEK(&B734))*8   für 664/6128
```

Natürlich müssen, wenn definiert, die Zeichencodes 32 - 127 unverändert bleiben.

Die Startadresse der Tabelle steht an Adresse &B296/7 (664/6128: &B736/7), die Endadresse an Adresse &B096/7 (664/6128: &B075/6). Schließlich ist es auch möglich, ein Maschinenprogramm im Video-RAM (Adresse &C000 bis &FFFF) abzuspeichern. Dazu folgende Überlegung:

Der Video-RAM ist 16K (16K=16*1024=16384) groß. In MODE 2 sind darin aber "nur" 25*80*8 Punkte gespeichert. Die Differenz, nämlich 384=8*48 Byte werden nicht genutzt.

Beim Einschalten bzw. nach dem MODE-Befehl liegen je 48 Bytes ungenutzt im Bereich von &C7D0-&C7FF, &CFD0-&CFFF, &D7D0-&D7FF usw. vor. Das Problem bei dieser Methode liegt darin, daß der Bildschirm nie (!) scrollen darf.

Die ausgegebenen freien Bereiche würden sich dadurch verschieben und die Programme überschrieben werden. Trotzdem mag dies für das Abspeichern einiger Programme ein guter Platz sein, da er ansonsten keinerlei Einschränkungen auferlegt.

Im folgenden sind noch zwei weitere Methoden aufgeführt, die jedoch nur für bestimmte Maschinenprogramme funktionieren. Alle Maschinenprogramme, die abgespeichert werden sollen, müssen verschiebbar (relocativ) sein, d.h. sie dürfen keine absolut angegebenen Adressen, die den Adreßbereich des Programms selbst betreffen, enthalten. Oft kann man Maschinenprogramme glücklicherweise auch ohne absolute Adressen, beispielsweise nur mit relativen Sprüngen innerhalb des Programms schreiben. Die Programme müssen diese Eigenschaft haben, da wir sie sozusagen "in die Hände" des Systems geben, das sie verwaltet. Dabei ist nur garantiert, daß sich kein Byte des Programms ändert, die Startadresse ist aber variabel und wird je nach Bedarf verändert

Wie ist das möglich?

Ganz einfach: Das BASIC erledigt ständig diese Aufgabe der Verwaltung von Ketten von Bytes, nämlich von Strings. Ein String besteht aus aufeinanderfolgenden Codes. Seine Länge kann bis zu 255 Zeichen betragen. Folglich können wir auch ein Maschinenprogramm (= eine Folge von Codes) als String speichern.

Mit einer Schleife werden wie üblich die Codes aus DATA-Zeilen gelesen und dann mit CHR$ der Reihe nach zu einem String addiert, z.B.:

```
MAPRO$=MAPRO$+CHR$(Byte)
```

Damit ist das Maschinenprogramm gespeichert. Der Aufruf erfolgt mit Hilfe der @-Funktion, der auf den Stringdescriptor

zeigt (siehe Programmierhilfen). Für unser Programm, das in MAPRO$ enthalten ist, kann man schreiben:

```
CALL PEEK(@MAPRO$+1)+256*PEEK(@MAPRO$+2)
```

Die zweite Möglichkeit, die auf demselben Prinzip aufbaut, erlaubt sogar die Speicherung beliebig langer Maschinenprogramme.

Diesmal benutzen wir zum Speichern einfach eine Feldvariable des Typs Integer. Eine Integerzahl besteht aus High und Low Byte. In einem Feld werden die zu den Indices gehörenden Werte einfach hintereinander abgespeichert. Aus diesem Grund können wir Felder von INT-Zahlen zur Speicherung von Maschinenprogrammen benutzen. Genaueres über Feldvariablen und ihre interne Verarbeitung finden Sie im Kapitel 22. Das folgende Programm macht das Prinzip deutlich. Wichtig ist:

1) Das Feld muß vom Typ Integer sein (%)
2) Hat das Programm eine ungerade Länge, so muß zusätzlich &00 an die DATA-Zeile gehängt werden.

```
10 lang=26: ' Anzahl Bytes im Programm
20 DIM meldung%(INT((lang-1)/2))
30 FOR i=0 TO INT((lang-1)/2)
40 READ l$,h$
50 meldung%(i)=VAL("&"+h$+l$)
60 NEXT i
70 END
80 DATA 3e,01,cd,0e,bc,cd,15,b9
90 DATA 7c,21,57,86,fe,01,28,06
100 DATA 2e,5c,38,02,2e,77,cd,0b
110 DATA 00,c9
120 REM Aufruf mit CALL @meldung%(0)
```

Nach Eingabe von RUN kann das Programm jederzeit mit CALL @MELDUNG%(0) aufgerufen werden. Der beim Aufruf angegebene Index muß 0 sein.

Haben Sie das Programm ausprobiert? Vielleicht haben auch Sie erst einmal einen Schrecken bekommen und gedacht, der Rechner wäre abgestürzt bzw. ein Reset ausgeführt worden.
Dem ist nicht so. Es wurde lediglich noch einmal die Einschaltmeldung angegeben. Das kleine Programm "Meldung" ist insofern interessant, als daß es automatisch die richtige Rechnerversion erkennt. Das wird durch die Systemroutine "KL Version Number" bewerkstelligt.

Hier das Assemblerlisting des Programms:

```
A000             10                ; Meldung
A000 3E01        20          LD    a,1
A002 CD0EBC      30          CALL  &bc0e ; SCR Mode a
A005 CD15B9      40          CALL  &b915 ; KL Versions Nummer
A008 7C          50          LD    a,h
; BASIC Version 0,1 oder 2
A009 215786      60          LD    hl,&8657 ; fuer 664
A00C FE01        70          CP    1
A00E 28FE        80          JR    z,ok
A010 2E5C        90          LD    l,&5c ; fuer 464
A012 38FE       100          JR    c,ok
A014 2E77       110          LD    l,&77 ; fuer 6128
**** Zeile  80  : OK=&A016
**** Zeile  100  : OK=&A016
A016 CD0B00     120  OK      CALL  &b
A019 C9         130          RET

Programm :meldung
Start : &A000     Ende : &A019
Laenge : 001A
 0  Fehler
Variablentabelle :
OK      A016
```

17. Abspeichern von Assemblerroutinen und Speicherbereichen

Mit dem hervorragenden BASIC-Interpreter des CPC sind Sie in der Lage, einzelne Bereiche des RAM gezielt abzuspeichern. Die Syntax dieser Form des SAVE-Befehles sieht folgendermaßen aus:

SAVE "programmname",B,startadresse,länge

Auf diese Art und Weise können u.a. Maschinenroutinen auf Diskette oder Cassette gespeichert werden. Beispiel:

Das abzuspeichernde Programm liegt in dem Bereich von &A000 bis &A013.

SAVE "DEMONAME",B,&A000,&14

Beachten Sie bitte genau, daß die Längenangabe dabei nicht zu kurz gerät. Im Beispiel ist die Länge der Routine &14 Byte, obwohl das letzte Byte die Nummer &xx13 hat. Fehlt nämlich das letzte Byte, es ist häufig das RET, stürzt das Programm nach dem Aufruf ab.

Außer der Möglichkeit, Assemblerroutinen abzuspeichern, können auch ganze Bildinhalte geSAVEt werden.

SAVE "SCREEN",B,&C000,&4000

In diesem Beispiel wird der reguläre Bildschirmspeicherbereich ab der Adresse &C000 mit der Länge &4000 als Binärfile unter dem Namen SCREEN abgespeichert.
Jetzt bieten sich gute Möglichkeiten, die bereits erwähnten Routinen für das Screencopy, Screenswap oder das Umschalten der Bildschirmbasis anzuwenden. Da der Ladevorgang eines

kompletten Bildes einige Sekunden dauert, bietet es sich an, das Bild zuerst in den Block 1 des RAM zu laden, und dann mit einem der Befehle schnell in den Bildschirmbereich zu bringen.

```
LOAD "SCREEN",&4000 : CALL screencopy
```

18. Nützliche Betriebssystem-Routinen

Adresse &0000: **RST 0 - RESET**

Der Aufruf dieser Routine mit CALL 0 wirkt wie ein Aus-/Anschalten des Rechners.

Adresse &0008: **RST &08 - Low Jump**

Diese Routine springt an eine Adresse im Betriebssystem-ROM oder im darüberliegenden RAM. Bit 14 und 15 bestimmen die ROM/RAM-Selektion. Ein gesetztes Bit bedeutet RAM und ein nichtgesetztes Bit ROM. Bit 14 bestimmt über den unteren Adressbereich (&0-&3FFF) und Bit 15 über den oberen (&C000 bis &FFFF).

Adresse &000C: **JP (HL) mit ROM/RAM-Selektion**

Indirekt adressierter Sprung zur Adresse im HL-Register. Bit 14 und 15 von HL üben dieselbe Funktion wie bei RST &08 aus.

Adresse &0010: **RST &10 : Side Call**

Dient zum Aufruf einer Routine in einem Expansions-ROM.

Adresse &0018: **RST &18 - Far Call**

Dient zum Aufruf einer Routine irgendwo im ROM oder RAM. Hinter dem Befehl RST &18 steht die Adresse eines Vektoren, der Zieladresse und ROM/RAM Status enthalten muß.

Adresse &0020: RST &20 - RAM Lam LD A,(HL)

Der Akku wird mit dem Wert der Adresse, auf die HL, zeigt geladen. Durch diese Routine wird dabei immer RAM selektiert.

Adresse &0028: RST &28 - Firm Jump

Dient zum Aufruf einer Routine im Betriebssystem (Firmware). Dabei wird die Adresse direkt hinter dem Befehl angegeben.

Adresse &0030: RST &30 - User Restart

Diese Routine steht für eigene Programme zur Verfügung.
Ein Beispiel für die Anwendung des RST &30 finden Sie in Kapitel 23.

Adresse &0038: RST &38 - Interrupt Einsprung

Über diese Adresse werden die Interruptroutinen aufgerufen.

Adresse &BB06: KM (Key Manager) - Wait Char

ASCII-Code der gedrückten Taste wird im Akku zurückgegeben.

Adresse &BB15: KM EXP Buffer

Diese Routine legt den RAM Bereich fest, in dem die Funktionstastenbelegungen gespeichert werden sollen. Da dieser Bereich eigentlich ein bißchen kurz geraten ist, kann er mit dieser Routine vergrößert

werden. Dazu wird im DE Register die Startadresse und im HL Register die Länge des Puffers übergeben. Wurde eine ausreichende Länge definiert, so kann jeder Funktionstaste ein String von bis zu 32 Zeichen zugeordnet werden. Alte Eintragungen werden durch den Ablauf dieser Routine gelöscht.

Adresse &BB24: KM Get Joystick

H-Register enthält Zustand des Joystick 1, L-Register entsprechend vom 2ten.

Adresse &BB39: KM Set Repeat.

Die Repeat-Funktion für die Taste mit der im Akku enthaltenen Nummer wird eingeschaltet, wenn B<>0 ist, bei B=0 wird sie ausgeschaltet.

Adresse &BB3C: KM Get Repeat

Das Z-Flag wird gesetzt, wenn die Taste mit der im Akku enthaltenen Nummer nicht auf Repeat gestellt ist, ansonsten zurückgesetzt.

Adresse &BB3F: KM Set Delay

Übergabe Tastennummer: A, Verzögerungszeit im H-Register, Wiederholungsgeschwindigkeit im L-Register.

Adresse &BB42: KM Set Delay

Übergabe: Tastennummer in A
Rückgabe: H:Verzögerungszeit, L: Wiederholungsgeschwindigkeit.

Adresse &BB5A: TXT Output

Gibt dem Wert des Akkus entsprechend das Zeichen auf den Bildschirm aus.

Adresse &BB60: TXT Read Char

Liest ein Zeichen vom Bildschirm ein. In HL wird die Position des Zeichens übergeben (H-Zeile / L-Spalte). Wird ein gültiges Zeichen erkannt, wird Carry gesetzt und der ASCII-Code des Zeichens in den Akku geladen.

Adresse &BB6C: TXT Clear Window

Löscht das aktuelle Bildschirmfenster.

Adresse &BB75: TXT Set Cursor

H/L entspricht Zeile/Spalte.

Adresse &BB78: TXT Get Cursor

Adresse &BB81: TXT Cursor On

Adresse &BB84: TXT Cursor Off

Adresse &BB90: TXT SET PEN

Die Pen-Farbe wird der im Akku enthaltenen INK-Nummer zugeordnet.

Adresse &BB93: TXT GET PEN

Liest die aktuelle PEN-INK-Nummer in den Akku.

Adresse &BB96: TXT SET PAPER (s. PEN)

Adresse &BB99: TXT GET PAPER (s. PEN)

Adresse &BB9C: TXT INVERSE

Adresse &BBA5: TXT GET MATRIX

Die Adresse der Zeichendefinition des Codes A wird im HL-Register zurückgegeben. Carry ist gesetzt, wenn es sich um ein vom Benutzer definiertes Zeichen handelt.

Adresse &BBA8: TXT SET MATRIX

Die Matrix des vom Benutzer definierbaren Zeichens mit dem Code A wird mit der Matrix ab Adresse HL geladen.

Adresse &BBC0: GRA Move Absolute

Das DE-Register enthält die X-Koordinate, das HL-Register enthält die Y-Koordinate.

Adresse &BBC3: GRA MOVE Relative

Das DE-Register enthält relative X-Koordinate, das HL-Register die relative Y-Koordinate.

Adresse &BBC6: GRA Ask Cursor

Registerbelegung wie bei Move Absolute.

Adresse &BBC9: GRA SET PEN A=ink

Adresse &BBE1: GRA GET PEN A=ink

Adresse &BBE4: GRA SET PAPER A=ink

Adresse &BBE7: GRA GET PAPER A=ink

Adresse &BBEA: GRA Plot Absolut

Adresse &BBED: GRA Plot relative

DE: relative X-Koordinate
HL: relative Y-Koordinate

Adresse &BBF0: GRA TEXT Absolut

DE: X-Koordinate
Hl: Y-Koordinate
Rückgabe: A enthält INK des Punktes

Adresse &BBF3: GRA Text relativ

Adresse &BBF6: GRA DRAW LINE

Die Linie wird von dem aktuellen Grafikcursor zu den Zielkoordinaten gezogen.

Adresse &BBF9: GRA DRAW LINE relative

Adresse &BBFC: GRA WRITE Char

Auf der Grafikcursorposition wird das Zeichen A ausgegeben.

Adresse &BC0B: SET Location

Die aktuelle Startadresse des linken oberen Bildschirmzeichens wird ermittelt. A enthält das High-Byte der Basisadresse (normalerweise &C0) und HL den Offset von der Basisadresse zur Startadresse.

Adresse &BC0E: SCR Mode A

Der dem Akkuinhalt entsprechende Bildschirmmodus wird eingeschaltet.

Adresse &BC11: SCR GET MODE

Die aktuelle MODE-Nummer wird in den Akku geladen. Außerdem sind die Flags beeinflußt:

MODE 0: C
MODE 1: Z
MODE 2: NC

Adresse &BC14: **SCR Clear screen mit ink 0**

Adresse &BC1A: **SCR CHAR POS**

Übergabe: H-Zeile / L-Spalte der Zeilenposition
Rückgabe: HL: Adresse der ersten Bytes
B: Breite des Zeichens in Byte (1, 2 oder 3)

Adresse &BC1D: **SCR DOT POS**

Übergabe: DE: X-Koordinate / HL: Y-Koordinate
Rückgabe: HL: Bildschirmadresse
C: Bit-Maske, die nur den betroffenen Punkt berücksichtigt.
B: Anzahl der Punkte -1 pro Byte (1, 2 oder 3)

Adresse &BC20: **SCR NEXT BYTE**

Bildschirmadresse HL um ein Byte nach rechts erhöhen.

Adresse &BC23: **SCR PREV BYTE**

Die Bildschirmadresse HL um ein Byte nach links verrücken.

Adresse &BC26: **SCR NEXT LINE**

Bildschirmadresse HL auf die nächste Zeile setzen.

Adresse &BC29: **SCR PREV LINE**

Bildschirmadresse HL auf vorhergehende Zeile setzen.

Adresse &BC32: **SET INK Color**

A: INK-Nummer; B und C Farben.

Adresse &BC35: **SCR GET INKcolor (wie oben)**

Adresse &BC38: **SCR SET Bordercolor (B,C)**

Adresse &BC3B: **SCR GET Bordercolor (B,C)**

Adresse &BC3E: **SCR SET Flash Time**

H: Zeit für erste Farbe
L: Zeit für zweite Farbe

Adresse &BC41: **SCR GET Flash Time**

Adresse &BC5F: **SCR waagerechte Linie**

A: INK, DE: X-Anfang, BC: X-Ende, HL: Y-Koordinate.

Adresse &BC62: **CR senkrechte Linie**

A: INK, DE: X-Koordinate, BC: Y-Ende, HL: Y-Anfang.

Adresse &BC9B: **CAS (bzw. Disk) Catalog**

Adresse &BCD1: KL (Kernel) Log Ext

Bindet RSX Erweiterungen ein.

Adresse &BD0D: KL Time please

Gibt den Wert des Timers als 4-Byte-Wert in DE und HL zurück

Adresse &BD10: KL Time Set

Adresse &BD2B: MC Print Char

Gibt den Wert des Akkus zum Drucker aus.

Adresse &BD2E: MC Printer Busy

Prüft, ob der Drucker empfangsbereit ist. Wenn nicht, wird das Carry-Flag gesetzt.

Adresse &BD37: Jump Restore : Notbremse

Setzt alle eventuell verbogenen Sprungvektoren auf ihre Ausgangswerte zurück.

Adresse &BDD3: Write

Der Akkuinhalt wird als ASCII-Zeichen auf dem Bildschirm ausgegeben.

19. Nützliche Routinen des BASIC-Interpreters

Die Reihenfolge der Adressen bedeutet:

Adresse 1 für 464, Adresse 2 für 664 und Adresse 3 für 6128

Adresse &C356: &C3A0: &C3A3: Print

Gibt das dem Akkuinhalt entsprechende Zeichen auf den aktuellen Kanal aus. Die Kanalnummer ist dabei in Speicherstelle &AC21, &AC06, &AC06 enthalten. Die Ausgabe unser Programme XREF/DUMP kann z.B. mit POKE &AC06,8:|XREF (464: POKE &AC21,8:|XREF) gedruckt werden.

Adresse &C34E: &C398: &C39B: Line Feed

Gibt Line-Feed auf aktuellen Kanal aus.

Adresse &C43C: &C472: &C475: BREAK Test

Prüft, ob ESC gedrückt wurde. Wenn ja, springt die Routine in den Wartemodus (wie vom BASIC bekannt).

Adresse &CA94: &CB55: &CB58: Error-Ausgabe

Bei Übergabe der Fehlernummer im Akku (bei 664 und 6128 im E-Register) wird die entsprechende Fehlermeldung ausgegeben und das Programm unterbrochen.

Adresse &D5DB: &D619: &D61C: Buchstaben-Variablentabellenpointer holen

Im Akku wird der Anfangsbuchstabe der zu suchenden Variablen übergeben. Als Rückgabe enthält BC die Startadresse der Variablentabelle und HL die Adresse, an der der Zeiger auf die erste Variable mit dem jeweiligen Buchstaben gespeichert ist.

Adresse &D6B3: &D6EC: &D6EF: Variablentabellenpointer holen

Zur Übergabe muß HL auf den Anfang einer Variablen im Programm zeigen, genau gesagt auf das Typenkennzeichen, das jeder Variablen vorausgeht. Existiert die Variable bereits in der Tabelle, wird ihre Startadresse zurückgegeben. Ansonsten wird die Variable in die Liste eingetragen und die neue Startadresse zurückgegeben.

Adresse &DD37: &DE25: &DE2A: CHR NEXT

Prüft, ob das nächste Byte (Adresse HL+1) des BASIC-Programms gleich dem Byte ist, das auf den Aufruf der Routine folgt. Bei Ungleichheit wird ein Syntax Error ausgegeben.

Adresse &DD3F: &DE2C: &DE31: CHRGET

Diese Routine dient zum Holen des nächsten Bytes. Dazu wird zunächst HL erhöht. Das an Adresse HL stehende Byte wird gelesen. Ist es ein Leerzeichen (Code 32), wird das nächste Byte gelesen usw.
Das erste "nicht Leerzeichen" wird im Akku zurückgegeben. Handelt es sich um ein Null-Byte, so wird

das Z-Flag gesetzt. Die Routine wird zum schrittweisen Lesen von BASIC-Programmen benutzt. Dabei bedeutet das Null-Byte das Ende einer Zeile.

Adresse &DD51: &DE3D: &DE42: CHRGOT

Liest noch einmal das zuletzt gelesene Byte und prüft, ob das Ende des Befehls erreicht ist. Wenn ja, ist C gesetzt.

Adresse &DD55: &DE41: &DE46: CHKKOMMA

Prüft, ob das aktuelle Byte ein Komma darstellt. Wenn ja, wird das folgende Byte gelesen und Carry gesetzt. Sonst ist das Carry-Flag zurückgesetzt.

Adresse &E8FF: &E9B9: &E9BE: BASIC DO Routine BC

Diese Routine durchläuft das aktuelle BASIC-Programm und springt nach dem Lesen des Anfangs jeder Zeile zu einer beliebigen Routine an Adresse BC. Dort kann die Zeile untersucht werden. Nach Behandlung der Zeile wird mit RET zurückgesprungen und automatisch die nächste Zeile "geliefert".

Adresse &E943: &E9FD: &EA02: SKIP COMMAND

U.U. im Zusammenhang mit der zuletztgenannten Routine überspringt diese Routine eine Befehlseinheit in einer Zeile.

Adresse &EE79: &EF44: &EF49: PRINT HL dezimal

Zum Ausgeben der Zeilennummern eines BASIC-Programms kann diese Zeile benutzt werden. Übergeben werden muß nur die Zeilennummer im HL-Register als 2-Byte-Wert.

Adresse &F236: &F2D5: &F2DA: PRINT FAC

Diese Routine gibt den zur Zeit im FAC enthaltenen Wert in dezimaler Form aus.

Adresse &FF4B: &FF6C: &FF6C: VAR FAC

Kopiert den Wert einer Variablen in den FAC. Dazu muß HL die Adresse des Wertes enthalten.

Adresse &FF71: &FF92: &FF92: TEST Buchstabe

Der Akkuinhalt wird als ASCII interpretiert und wenn möglich, in Großbuchstaben verwandelt. Handelte es sich dabei nicht um den ASCII-Code eines Buchstabens, ist das Carry-Flag zurückgesetzt.

Adresse &CFEE: &D058: &D055: Operand Missing

Adresse &C205: &CB30: &C210: Improper Argument

20. Kompatibilität zwischen den drei CPCs

In diesem Kapitel wollen wir kurz festhalten, wo die wichtigen Unterschiede bzw. "Gleichheiten" zwischen den drei Schneider-Rechnern CPC 464, CPC 664 und CPC 6128 bzgl. der Maschinenspracheprogrammierung liegen.

Grundsätzlich kann man sagen, daß die ROMs der drei Computer sehr ähnlich sind. Leider bedeutet sehr ähnlich nur, daß fast alle Routinen gleichermaßen bei allen Rechnern existieren. Der Haken an der Sache ist, daß auch bei fast allen Routinen die direkten Einsprungadressen unterschiedlich sind. Davon ausgenommen sind lediglich und glücklicherweise die meisten Sprungvektoren im zentralen RAM. Im Speziellen sind das die Sprungvektoren zum:

- HIGH KERNEL
  Adressen &B900 bis &B920

- zum KEY-Manager
  Adressen &BB00 bis &BB4D

- zum TEXT-Pack
  Adressen &BB4E bis &BBB9

- zum GRAfik-Pack
  Adressen &BBBA bis &BBFE

- zum SCReen-Pack
  Adressen &BBFF bis &BC64

- zum CAS- bzw. DISK-Manager
  Adressen &BC65 bis &BCA6

- zum SOUND-Pack
  Adressen &BCA7 bis &BCC5

- zum KERNEL (Low)
  Adressen &BCC8 bis &BD12

- zum MaChine-Pack
  Adressen &BD13 bis &BD35

- sowie der JUMP RESTORE-Vektor &BD37

Leider konnten wir noch nicht jede einzelne der o.g. Routinen prüfen, jedoch haben wir bei unserer Arbeit bisher keine Routine aus diesen Bereichen gefunden, die nicht bei allen drei Schneider-Computern die gleiche Funktion hat.

Nach dem JUMP RESTORE-Vektor an Adresse &BD37 ist es jedoch vorbei mit der Kompatibilität.

Beim 464 folgen nun der Aufruf des Line-Editors und dann die Sprünge auf die Arithmetik-Routinen. Beim 664 und 6128 folgen hier jedoch einige neue Sprungvektoren wie z.B. zum Aufruf der Fill-Routine.
Der LINE EDITOR-Vektor steht an Adresse &BD3A, dann erst folgen die Arithmetikroutinen.
Die Jump-Vektoren (Vektoren, die nur bei eingeschaltetem Betriebssystem aufgerufen werden dürfen) von Adresse &BDCD bis Adresse &BDF3 sind dagegen wieder bei allen Rechnerversionen gleich.

Wollen Sie irgendwelche Routinen direkt anspringen, wie z.B. alle BASIC-Routinen, so sind in aller Regel die Einsprungadressen aller drei Rechner unterschiedlich, wobei zwischen dem CPC 664 und dem CPC 6128 nur geringe Differenzen auftreten.

Ein weiterer wichtiger Unterschied ist, daß beim CPC 664 und 6128 die bei CPC 464 vorhandenen Indirections des BASIC-Interpreters aus Platzgründen vollkommen weggefallen sind. Dadurch haben sich natürlich auch alle Systemdatenadressen ("PEEKs und POKEs") gegenüber dem 464 verschoben. Diese Adressen sind jedoch beim CPC 664 und 6128 weitgehend gleich.

Fazit:

Bei der Programmierung sollten nach Möglichkeit ausschließlich die erwähnten Vektoren benutzt werden. Ist das nicht möglich, so wird man meist nicht umhinkommen, jeweils drei leicht unterschiedliche Versionen schreiben zu müssen. Einziger Lichtblick in diesem Adreßdschungel ist die Routine KL VERSION, die glücklicherweise, ansonsten wäre alles vergeblich, an einer genormten Adresse (&B915) steht. Diese Routine gibt die Nummer des aktuellen High-ROMs zurück. Dabei enthält H die ROM-Nummer (1 für BASIC-ROM) und L die Versionsnummer 0=CPC 464, 1=CPC 664 und 2=CPC 6128.

Mit Hilfe dieser Routine ist das Programm "Meldung" aus dem Kapitel über Maschinenprogrammeingabe für alle drei Rechner lauffähig gemacht worden.

21. Befehlserweiterung mit RSX

Ein Erweitern des Schneider-BASIC Befehlsvorrates ist auf verschiedene Weisen möglich. Beim 464 kann man über die "Patchbereiche" im RAM die vorhandenen ROM Routinen erweitern. Doch diese Möglichkeit ist relativ eingeschränkt, da nur 9 solcher Patches möglich sind. Beim CPC 6128 sind sie außerdem aus Platzgründen ganz weggelassen worden. Es gibt jedoch bei allen CPCs die standardmäßig vorgesehene Methode, die die Erweiterung mit eigenen Befehlen ermöglicht. Sie alle kennen diese Methode. Sie wird für alle AMSDOS-Diskettenbefehle verwendet.

Sicherlich haben Sie schon einmal mit |CPM den CP/M-Modus aufgerufen, auf jeden Fall wenn Sie ein Diskettenlaufwer besitzen. Der CPM-Befehl beginnt mit einem Strich ("|"). Bei Eingabe von "CPM" (ohne Strich) erhalten Sie nur einen "Syntax Error". Betreiben Sie den Rechner ohne Floppy (nur bei 464 möglich), erhalten Sie nach |CPM ein "Unknown Command".

Es handelt sich also um einen Befehl, der erst vorhanden ist, wenn eine Floppy angeschlossen ist. Damit ist er eine Erweiterung.

Der "|" teilt dem System mit, daß es sich im folgenden um einen Erweiterungsbefehl handelt. Die Methode, von vorneherein evtl. Erweiterungen einheitlich durch ein Sonderzeichen zu kennzeichnen, um sie dann zu erkennen und auszuführen, ist bei den Schneider Rechnern fest eingebaut. Diese Art der Einbindung von Befehlen bezeichnet man als RSX. RSX steht für "Resident System Extention", was soviel bedeutet wie "feste Systemerweiterung". D.h., daß RSX ursprünglich für die Einbindung von neuen Befehlen, die in zusätzlichen ROMs (Expansions ROMs) vorhanden sind, gedacht war. Für den Floppybetrieb gibt es einen solchen Expansions ROM, in dem auch die Befehle |CPM, |ERA usw. enthalten sind.

Wir können die RSX-Methode aber auch benutzen, um unsere selbstgeschriebenen Routinen wie wirkliche Befehle einzubinden und somit komfortabel zu benutzen.

21.1 DOKE - Zwei-Byte-Wert in den Speicher schreiben

Die Einbindung mit der RSX-Methode wollen wir am Beispiel des DOKE-Befehls verdeutlichen.

DOKE ist quasi ein "Doppelter POKE-Befehl". Mit POKE kann eine Speicherstelle mit einem Wert zwischen 0 und 255 beschrieben werden. DOKE beschreibt zwei aufeinanderfolgende Speicherstellen mit einem Wert zwischen 0 und 65500. Dazu wird der Wert in Low-Byte und High-Byte zerlegt. Das Low-Byte wird an der unteren Speicherstelle abgelegt, das High-Byte an der Speicherstelle mit der höheren Adresse. Mit DOKE wird das Ändern vieler Systemparameter die ins RAM gespeichert sind (üblicherweise als PEEKs und POKEs) vereinfacht.

Die Parameterübergabe bei den RSX ist analog der des CALL-Befehls.

Der DOKE Befehl benötigt zwei Parameter: Die Adresse, ab der der Wert gespeichert werden soll und den Wert, der ab dieser Adresse abgelegt werden soll. Damit hat der Befehl folgendes Format:

```
|DOKE, Adresse, Wert
```

Die Parameter werden folgendermaßen übergeben:

A: Enthält Anzahl der übergebenen Parameter

Flags: Zero-Flag=1 wenn keine Parameter übergeben wurden, sonst 0

B: Enthält 32-Anzahl der Parameter

DE: Enthält letzten übergebenen Parameter

IX: Adresse des letzten Elementes. Der vorletzte Parameter steht dann an Adresse IX+2, der nächste an Adresse IX+4 usw.

Die DOKE Routine sieht dann so aus:

```
100 ' LD L,(IX+2) ; übergebene Adresse
110 ' LD H,(IX+3) ; ins HL Register laden
120 ' LD (HL),E ; Low Byte speichern
130 ' INC HL ; Adresse auf High Byte setzen
140 ' LD (HL),D ; High Byte speichern
150 ' RET ; fertig!
```

Mit CALL &A000,Adresse,Wert könnten wir jetzt den DOKE-Befehl aufrufen, vorausgesetzt, er wurde ab &A000 abgespeichert. Probieren Sie z.B. einmal:

CALL &A000,..,..

Damit haben wir eine funktionierende DOKE-Routine. Allerdings ist der Aufruf mit CALL etwas lästig. DOKE soll jetzt auch mit DOKE aufgerufen werden. Dazu muß eine kleine Einbindungsroutine geschrieben werden. Die Hauptarbeit nimmt uns dabei die Routine "LOGEXT" des Betriebssystems ab, die die eigentliche Einbindung vornimmt. Wir müssen lediglich einige Werte an Logext übergeben.

Um DOKE einzubinden, muß das System folgendes "wissen":

1) Name der Erweiterung
2) Adresse der Routine
3) Adresse von 4 unbenutzten Bytes, die intern zur Verwaltung der Routine benutzt werden

Die Adresse der vier Systembytes wird vor dem Aufruf von Logext ins HL Register geladen. Außerdem wird BC mit der Startadresse einer Tabelle geladen, in der die weiteren Informationen enthalten sind.

Diese Tabelle enthält als erstes die Adresse einer zweiten Tabelle, nämlich der, in welcher der Name oder auch mehrere Namen verschiedener einzubindender Routinen enthalten ist. Auf die Adresse der 2ten Tabelle in der 1ten Tabelle folgt ein Sprungbefehl auf die einzubindende Routine bzw. Routinen. Werden mehrere Routinen eingebunden, müssen Sprungbefehle und Routinennamen einander entsprechen.

Die 2te Tabelle enthält, wie gesagt, die Namen der Routinen. Dabei muß, um die einzelnen Namen voneinander zu trennen, das Bit 7 des letzten Buchstaben eines Namens immer gesetzt sein.

Doch betrachten wir unser Beispiel:

```
10 ' LD BC,RSXTAB ; Adresse der 1ten Tabelle
20 ' LD HL,SYSBYT ; Adresse des Systembytes
30 ' CALL &BCD1 ; Routine Logext aufrufen
40 ' RET ; Einbindung fertig
50 ' RSXTAB Dw NAMTAB ; Tabelle 1 enthält als erste Adresse die
Anfangsadressen der 2ten Tabelle (Namenstabelle)
60 ' JP START ; und dann Sprungbefehle auf die Routine
70 ' NAMTAB DM "DOK" ; Namenstabelle
80 ' DB &C5 : =ASC("E")+128 damit Bit 7 gesetzt ist
90 ' DB 0 ; Nullbyte kennzeichnet das Ende der Namenstabelle
95 ' SYSBYT DS 4 ; 4 Bytes für Kernal reservieren
100 ' START  LD.... hier beginnt die Routine
                :                :
```

Nach dem Assemblieren des gesamten Programms (bzw. BASIC-Lader) wird einmal mit CALL &A000 der DOKE Befehl eingebunden. Von diesem Zeitpunkt an ist DOKE eine Befehlserweiterung (RSX), d.h. der Befehl kann wie jeder andere im Direktmodus oder im Programm benutzt werden.

Achten Sie darauf, die Einbindungsroutine nur einmal aufzurufen. Die 4 Systembytes werden u.a. dazu benutzt, auf evtl. weitere RSX-Tabellen zu zeigen. Wird zum zweiten Mal dieselbe Routine initialisiert, so wird der Zeiger, der auf die nächste

RSX Tabelle zeigen soll, logischerweise wieder auf dieselbe Tabelle gerichtet.

Dadurch kann es passieren, daß die Befehlserkennung in einer Endlosschleife hängen bleibt, was natürlich recht ungünstig ist. Folgendes kleines Programm verhindert zusätzlich, daß ein Wiederaufruf stattfinden kann, indem es an den Anfang der Init-Routine einen RET-Befehl schreibt.

```
A000 010000     10  INIT    LD    bc,rsxtab
A003 210000     20          LD    hl,sysbyt
A006 CDD1BC     30          CALL  &bcd1
A009 3EC9       40          LD    a,&c9
A00B 3200A0     50          LD    (init),a
A00E C9         60          RET
**** Zeile  10  : RSXTAB=&A00F
A00F 0000       70  RSXTAB  DW    namtab
A011 C30000     80          JP    start
**** Zeile  70  : NAMTAB=&A014
A014 444F4B     90  NAMTAB  DM    "DOK"
A017 C5        100          DB    &c5
A018 00        110          DB    0
**** Zeile  20  : SYSBYT=&A019
A019           120  SYSBYT  DS    4
**** Zeile  80  : START=&A01D
A01D DD6E02    130  START   LD    l,(ix+2)
A020 DD6603    140          LD    h,(ix+3)
A023 73        150          LD    (hl),e
A024 23        160          INC   hl
A025 72        170          LD    (hl),d
A026 C9        180          RET

Programm :doke
Start : &A000    Ende : &A026
Laenge : 0027
 0  Fehler
Variablentabelle :
INIT   A000  RSXTAB A00F  NAMTAB A014  SYSBYT A019
START  A01D
```

21.2 RPEEK - Beliebiges Lesen aus RAM oder ROM

Das folgende Listing zeigt, wie man die @-Funktion in Verbindung mit RSX oder CALL benutzt.
In unserem Fall wird die Startadresse des Wertes einer Integervariablen mit @ ermittelt und dann an das Maschinenprogramm übergeben. Der ermittelte Wert wird dann vom Programm an die übergebene Adresse geschrieben und steht danach im BASIC unter der jeweiligen Variablen zur Verfügung.

Das nachfolgende Programm implementiert einen erweiterten PEEK-Befehl mit Hilfe von RSX. Mit diesem neuen Befehl ist es möglich, Adressen sowohl im RAM als auch im ROM und im Expansions-ROM (z.B. Disketten-ROM) zu lesen.

Der Befehl hat das Format:

```
|RPEEK,@Integervariable,Adresse,Status
```

Also würde z.B. durch

```
|RPEEK,@w%,&C000,-7
```

der Wert der an der ersten Adresse des Disketten-ROMs steht, in die Variable w% geladen. Wichtig ist, daß w% zuvor mindestens einmal benutzt wurde, da ansonsten @w% den Fehler "improper argument" hervorrufen würde. Für den Status gilt folgende Zuordnung:

1: RAM
2: ROM
0,-1,-2 bis -251 selektieren die jeweiligen Expansions-ROMs

Alle anderen Werte für den Status verursachen einen "improper argument".

```
A000             10                ; Erweiterung mit RSX
A000             20
; "|RPEEK,@Intvar.,Adr.,Status"
A000             30                ; Satus   1 :RAM
A000             40                ;         2 :ROM
A000             50                ; 0,-1,...,-251 :Exp. ROM
A000             60
A000             70  AOPMIS EQU  &d055
; 464 : &cfee / 664 : &d058
A000             80  AIMPAR EQU  &c21d
; 464 : &c205 / 664 : &cb50
A000 010000      90  INIT   LD   bc,rsxtab
A003 210000     100          LD   hl,sysbyt
A006 CDD1BC     110          CALL &bcd1
A009 3EC9       120          LD   a,&c9
A00B 3200A0     130          LD   (init),a
A00E C9         140          RET
**** Zeile  90  : RSXTAB=&A00F
A00F 0000       150  RSXTAB DW   namtab
A011 C30000     160          JP   start
**** Zeile  150 : NAMTAB=&A014
A014 52504545 170  NAMTAB DM   "RPEE"
A018 CB         180          DB   &cb
A019 00         190          DB   0
**** Zeile  100 : SYSBYT=&A01A
A01A            200  SYSBYT DS   4
A01E DF         210  OPMIS  RST  &18
A01F 0000       220          DW   vek1
A021 C9         230          RET
**** Zeile  220 : VEK1=&A022
A022 55D0       240  VEK1   DW   aopmis
A024 FD         250          DB   253
A025 DF         260  IMPARG RST  &18
A026 0000       270          DW   vek2
A028 C9         280          RET
**** Zeile  270 : VEK2=&A029
A029 1DC2       290  VEK2   DW   aimpar
A02B FD         300          DB   253
```

```
**** Zeile  160  : START=&A02C
A02C FE03     310  START  CP   3 ; zwei Parameter ?
A02E 20EE     320         JR   nz,opmis
; nein, dann Operand missing
A030 7A       330         LD   a,d
A031 FE00     340         CP   0
A033 28FE     350         JR   z,noexro
; Status>0, dann kein Expansionsrom
A035 FEFF     360         CP   &ff ; Betrag<255 ?
A037 20EC     370         JR   nz,imparg
; ja, dann improper Argument
A039 7B       380         LD   a,e
A03A ED44     390         NEG
A03C FEFC     400         CP   252
A03E 30E5     410         JR   nc,imparg
A040 18FE     420         JR   setsta
**** Zeile  350  : NOEXRO=&A042
A042 7B       430  NOEXRO LD   a,e
A043 FE03     440         CP   3
A045 30DE     450         JR   nc,imparg
A047 C6FE     460         ADD  a,254
A049 30FE     470         JR   nc,setsta
A04B D604     480         SUB  4
**** Zeile  420  : SETSTA=&A04D
**** Zeile  470  : SETSTA=&A04D
A04D 320000   490  SETSTA LD   (status),a
A050 DD6E02   500         LD   l,(ix+2)
A053 DD6603   510         LD   h,(ix+3)
A056 DF       520         RST  &18
A057 0000     530         DW   vektor
A059 DD6E04   540         LD   l,(ix+4)
A05C DD6605   550         LD   h,(ix+5)
A05F 77       560         LD   (hl),a
A060 97       570         SUB  a
A061 23       580         INC  hl
A062 77       590         LD   (hl),a
A063 C9       600         RET
**** Zeile  530  : VEKTOR=&A064
A064 0000     610  VEKTOR DW   anfang
```

```
**** Zeile  490  : STATUS=&A066
A066 FD       620  STATUS DB   253
**** Zeile  610  : ANFANG=&A067
A067 7E       630  ANFANG LD   a,(hl)
A068 C9       640          RET

Programm :rpeek
Start : &A000    Ende : &A068
Laenge : 0069
 0  Fehler
Variablentabelle :
AOPMIS D055  AIMPAR C21D  INIT   A000  RSXTAB A00F
NAMTAB A014  SYSBYT A01A  OPMIS  A01E  VEK1   A022
IMPARG A025  VEK2   A029  START  A02C  NOEXRO A042
SETSTA A04D  VEKTOR A064  STATUS A066  ANFANG A067
```

```
10 FOR i=&A000 TO &A068
20 READ a$:w=VAL("&H"+a$)
30 s=s+w:POKE i,w:NEXT
40 IF s<> 17592 THEN PRINT"Fehler in Datas":END
50 '464: IF s<> 17720 THEN ...
60 '664: IF s<> 17655 THEN ...
70 PRINT"ok!":END
80 DATA 01,0F,A0,21,1A,A0,CD,D1
90 DATA BC,3E,C9,32,00,A0,C9,14
100 DATA A0,C3,2C,A0,52,50,45,45
110 DATA CB,00,20,A0,0F,A0,DF,22
120 DATA A0,C9,55,D0,FD,DF,29,A0
130 '464:..,..,ee,cf,..,..,..,..
140 '664:..,..,58,d0,..,..,..,..
150 DATA C9,1D,C2,FD,FE,03,20,EE
160 '464:..,05,c2,..,..,..,..,..
170 '664:..,50,cb,..,..,..,..,..
180 DATA 7A,FE,00,28,0D,FE,FF,20
190 DATA EC,7B,ED,44,FE,FC,30,E5
200 DATA 18,0B,7B,FE,03,30,DE,C6
210 DATA FE,30,02,D6,04,32,66,A0
220 DATA DD,6E,02,DD,66,03,DF,64
230 DATA A0,DD,6E,04,DD,66,05,77
240 DATA 97,23,77,C9,67,A0,FD,7E
250 DATA C9
```

22. Der Aufbau von Feldvariablen

In diesem Kapitel wollen wir uns kurz mit der internen Speicherung von Feldvariablen beschäftigen. Als praktische Anwendung werden wir Ihnen ein Programm vorstellen, mit dem Sie die Summe aller Elemente in Maschinensprache berechnen können.

Feldvariablen werden wie normale Variablen in einer Tabelle hinter dem BASIC-Programm gespeichert. Die Startadresse der Tabelle steht an Adresse &AE87 (664/6128: &AE68). Da Feldvariablen sehr viel Speicherplatz in Anspruch nehmen können, muß dieser mit dem DIM-Befehl vorzeitig reserviert werden. Die Feldvariablen sind, wie schon von den normalen Variablen bekannt, über Verkettungsadressen nach Anfangsbuchstaben in verkettete Listen eingeordnet.

Der Aufbau der Feldvariablentabelle ist folgendermaßen:

Zuerst stehen zwei Bytes, die die Verkettungsadresse enthalten. Darauf folgt der Name der Variablen, wobei, wie schon bekannt, zum ASCII Code des letzten Buchstabens 128 = &80 addiert ist, wodurch Bit 7 dieses Codes gesetzt wird. Auf den Namen folgt die Typenkennziffer. Soweit ist die Feldvariablentabelle der der normalen Variablen ähnlich.

Nun folgt die Länge der gesamten folgenden Eintragung, die als Zwei-Byte-Zahl abgespeichert wird. Als nächstes steht die Anzahl der Indices des Feldes. Darauf folgt für jeden Index des Feldes seine maximale Größe.

Nach diesen Informationen folgen der Reihe nach alle Werte des Feldes. Dabei sind in einem INTeger-Feld für jede Eintragung zwei Bytes vorgesehen, die den Wert selbst enthalten. Bei REAL-Variablen beträgt die Länge einer Eintragung fünf Bytes. Auch dabei ist der Wert selbst dargestellt. Bei Stringfeldern wird jeweils der drei Byte lange Stringdescriptor abgespeichert.

Der Aufbau noch einmal in der Übersicht:

2 Bytes	Verkettungsadresse
n Bytes	für die Darstellung des Namens
1 Byte	Typenkennziffer
2 Bytes	Länge der gesamten folgenden Eintragung
1 Byte	Anzahl der Indices
je 2 Bytes	für Maximalwert jedes Index'
m Bytes	Wert jedes Feldelementes
	Länge 2 für INTeger
	Länge 3 für String
	Länge 5 für REAL

Das folgende Programm benutzt diese Struktur, um die Summe aller Elemente eines Zahlenfeldes zu bilden. Für das Format des Befehles schauen Sie bitte im folgenden Assemblerlisting nach.

```
A000             10                ; Summe eines Feldes
A000             20                ; gibt die Summe aller Werte
A000             30                ; einer Feldvariablen aus
A000             40
A000             50
; Format: "call &a000,@<variablenname(indices 0,..)>,Anzahl In
dicices,@<Variable fuer Ergebnis>
A000             60
A000             70          ORG   &a000
A000 DF          80          RST   &18
A001 0000        90          DW    vektor
A003 C9         100          RET
**** Zeile  90  : VEKTOR=&A004
A004 0000       110  VEKTOR DW    sumar
A006 FD         120          DB    253 ; UROM on
A007            130
A007            140
; Adr. fuer 6128 ;  464   ,   664
A007            150  IMPARG EQU   &c21d
; &FA9C , &cb50  improper arg.
A007            160  TYPMIS EQU   &ff62
; &FF40 , &ff62  type mismatch
A007            170  VARHND EQU   &ff8c
; &FF66 , &ff87  Variable (hl) nach (de)
A007            180  VARFAC EQU   &ff6c
; &ff4b , &ff6c  Variable (hl) nach FAC
A007            190  CPHLDE EQU   &ffd8 ; &ffb8 , &ffd8  CP hl,de
A007            200  READHD EQU   &bd7c
; &bd58 , &bd79  Real arithm. (hl)+(de)
A007            210  FAC    EQU   &b0a0 ; &b0c2 , &b0a0
A007            220  TYP    EQU   &b09f ; &b0c1 , &b09f
A007            230
**** Zeile  110  : SUMAR=&A007
A007 D5         240  SUMAR  PUSH de
A008 210000     250         LD    hl,FAC1 ; Die fuenf Bytes
A00B 0605       260         LD    b,5 ; des FAC1 auf
A00D 3600       270  NEXT   LD    (hl),0 ; null setzen
A00F 23         280         INC   hl
```

```
A010 10FB      290          DJNZ next
A012 DD6E04    300          LD   l,(ix+4)
A015 DD6605    310          LD   h,(ix+5)
; Adresse erstes Element nach HL
A018 DD7E02    320          LD   a,(ix+2) ; Anzahl Indices
A01B 47        330          LD   b,a
A01C E5        340          PUSH hl ; Adresse erstes Element
A01D 87        350          ADD  a,a ; Anzahl Inices * 2
A01E C601      360          ADD  a,1 ; um je 2 Bytes
A020 1600      370          LD   d,0 ; fuer jeden Index
A022 5F        380          LD   e,a ; von hl abziehen
A023 B7        390          OR   a ; Carry loeschen
A024 ED52      400          SBC  hl,de ; HL zeigt auf Anzahl
A026 7E        410          LD   a,(hl) ; der Indices
A027 B8        420          CP   b ; vergleichen
A028 C21DC2    430          JP   nz,imparg
; wenn Anzahl Indices falsch: "improper Argument"
A02B 2B        440          DEC  hl ; Feldlaenge
A02C 2B        450          DEC  hl ; ueberspringen
A02D 2B        460          DEC  hl ; HL zeigt dann
A02E 7E        470          LD   a,(hl) ; auf Typenkennziffer
A02F C601      480          ADD  a,1
A031 FE03      490          CP   3 ; wenn String, dann
A033 CA62FF    500          JP   z,typmis ; Type mismatch
A036 4F        510          LD   c,a
A037 329FB0    520          LD   (TYP),a
A03A 23        530          INC  hl
A03B 5E        540          LD   e,(hl) ; Laenge des
A03C 23        550          INC  hl ; Feldes in
A03D 56        560          LD   d,(hl) ; Bytes zu HL
A03E 19        570          ADD  hl,de ; addieren = Ende Feld
A03F E3        580          EX   (sp),hl
; Anfang mit Ende tauschen
A040 0600      590          LD   b,0
A042 F3        600          DI    ; weil's schneller ist
A043 E5        610  WEITER  PUSH hl ; Adresse naechstes Element
A044 C5        620          PUSH bc ; Typ in c
A045 CD6CFF    630          CALL varfac ; var (hl) nach FAC
A048 21A0B0    640          LD   hl,fac
```

```
A04B 110000    650          LD   de,fac1
A04E FE05      660          CP   5 ; Real ?
A050 28FE      670          JR   z,real
A052 7E        680          LD   a,(hl)
A053 23        690          INC  hl ; Wert des aktuellen
A054 66        700          LD   h,(hl) ; Elementes holen
A055 6F        710          LD   l,a
A056 EB        720          EX   de,hl
A057 7E        730          LD   a,(hl) ; Wert der Summe
A058 23        740          INC  hl ; holen
A059 66        750          LD   h,(hl)
A05A 6F        760          LD   l,a
A05B 19        770          ADD  hl,de
A05C 220000    780          LD   (FAC1),hl ; FAC1=hl
A05F 18FE      790          JR   break
**** Zeile  670  : REAL=&A061
A061 EB        800  REAL     EX   de,hl
A062 CD7CBD    810          CALL readhd ; real (hl)=(hl)+(de)
**** Zeile  790  : BREAK=&A065
A065 C1        820  BREAK    POP  bc ; Typ in c
A066 E1        830          POP  hl ; HL auf naechste Adresse
A067 09        840          ADD  hl,bc ; setzen
A068 D1        850          POP  de ; Endadresse
A069 D5        860          PUSH de
A06A CDD8FF    870          CALL cphlde ; Vergleich hl mit de
A06D 38D4      880          JR   c,weiter
A06F FB        890          EI    ; Ende der Summenbildung
A070 D1        900          POP  de
A071 210000    910          LD   hl,FAC1
A074 D1        920          POP  de ; Zieladresse Ergebnis
A075 CD8CFF    930          CALL varhnd ; kopieren
A078 C9        940          RET
**** Zeile  250  : FAC1=&A079
**** Zeile  650  : FAC1=&A079
**** Zeile  780  : FAC1=&A079
**** Zeile  910  : FAC1=&A079
A079           950  FAC1     DS   5

Programm :sumfeld
```

```
Start : &A000     Ende : &A07D
Laenge : 007E
 0  Fehler
Variablentabelle :
VEKTOR A004   IMPARG C21D   TYPMIS FF62   VARHND FF8C
VARFAC FF6C   CPHLDE FFD8   READHD BD7C   FAC     B0A0
TYP    B09F   SUMAR  A007   NEXT   A00D   WEITER A043
REAL   A061   BREAK  A065   FAC1   A079
```

```
10 REM BASIC Lader fuer 6128
20 REM Programm sumfeld
30 FOR i=&A000 TO &A07D
40 READ a$:w=VAL("&H"+a$)
50 s=s+w:POKE i,w:NEXT
60 IF s<> 15294 THEN PRINT"Fehler in Datas":END
70 PRINT"ok!":END
80 DATA DF,04,A0,C9,07,A0,FD,D5
90 DATA 21,79,A0,06,05,36,00,23
100 DATA 10,FB,DD,6E,04,DD,66,05
110 DATA DD,7E,02,47,E5,87,C6,01
120 DATA 16,00,5F,B7,ED,52,7E,B8
130 DATA C2,1D,C2,2B,2B,2B,7E,C6
140 DATA 01,FE,03,CA,62,FF,4F,32
150 DATA 9F,B0,23,5E,23,56,19,E3
160 DATA 06,00,F3,E5,C5,CD,6C,FF
170 DATA 21,A0,B0,11,79,A0,FE,05
180 DATA 28,0F,7E,23,66,6F,EB,7E
190 DATA 23,66,6F,19,22,79,A0,18
200 DATA 04,EB,CD,7C,BD,C1,E1,09
210 DATA D1,D5,CD,D8,FF,38,D4,FB
220 DATA D1,21,79,A0,D1,CD,8C,FF
230 DATA C9,06,08,0D,20,CC
```

```
10 REM Sumfeld fuer 464
20 FOR i=&A000 TO &A07D
30 READ a$:w=VAL("&H"+a$)
40 s=s+w:POKE i,w:NEXT
50 IF s<> 15372 THEN PRINT"Fehler in Datas":END
60 PRINT"ok!":END
70 DATA DF,04,A0,C9,07,A0,FD,D5
80 DATA 21,79,A0,06,05,36,00,23
90 DATA 10,FB,DD,6E,04,DD,66,05
100 DATA DD,7E,02,47,E5,87,C6,01
110 DATA 16,00,5F,B7,ED,52,7E,B8
120 DATA C2,9C,FA,2B,2B,2B,7E,C6
130 DATA 01,FE,03,CA,40,FF,4F,32
140 DATA C1,B0,23,5E,23,56,19,E3
150 DATA 06,00,F3,E5,C5,CD,4B,FF
160 DATA 21,C2,B0,11,79,A0,FE,05
170 DATA 28,0F,7E,23,66,6F,EB,7E
180 DATA 23,66,6F,19,22,79,A0,18
190 DATA 04,EB,CD,58,BD,C1,E1,09
200 DATA D1,D5,CD,B8,FF,38,D4,FB
210 DATA D1,21,79,A0,D1,CD,66,FF
220 DATA C9,06,08,0D,20,CC
```

```
10 REM Sumfeld fuer 664
20 FOR i=&A000 TO &A07D
30 READ a$:w=VAL("&H"+a$)
40 s=s+w:POKE i,w:NEXT
50 IF s<> 15346 THEN PRINT"Fehler in Datas":END
60 PRINT"ok!":END
70 DATA DF,04,A0,C9,07,A0,FD,D5
80 DATA 21,79,A0,06,05,36,00,23
90 DATA 10,FB,DD,6E,04,DD,66,05
100 DATA DD,7E,02,47,E5,87,C6,01
110 DATA 16,00,5F,B7,ED,52,7E,B8
120 DATA C2,50,CB,2B,2B,2B,7E,C6
130 DATA 01,FE,03,CA,62,FF,4F,32
140 DATA 9F,B0,23,5E,23,56,19,E3
150 DATA 06,00,F3,E5,C5,CD,6C,FF
160 DATA 21,A0,B0,11,79,A0,FE,05
170 DATA 28,0F,7E,23,66,6F,EB,7E
180 DATA 23,66,6F,19,22,79,A0,18
190 DATA 04,EB,CD,79,BD,C1,E1,09
200 DATA D1,D5,CD,D8,FF,38,D4,FB
210 DATA D1,21,79,A0,D1,CD,87,FF
220 DATA C9,06,08,0D,20,CC
```

23. Maschinenprogramme beweglich gemacht: Relokalisation

23.1 Warum Relokalisation?

Fast jeder, der Maschinenprogramme - besonders Befehlserweiterungen - schon mehrfach benutzt hat, seien sie nun selbst geschrieben oder von jemand anderem übernommen, kennt dieses Problem:
Jedes für sich genommen, laufen diese Programme einwandfrei. Will man aber zwei oder gar mehrere dieser Programme kombinieren, also gleichzeitig im Speicher verfügbar halten, so ergeben sich schon die Probleme. Das eine dieser schwerwiegenden Probleme bei der Kombination von Befehlserweiterungen resultiert aus der Umlegung von Vektoren. Es führt dazu, daß die zuletzt geladene Erweiterung die durch die zuerst geladene gemachten Änderungen an den Vektoren zur Befehlserkennung wieder rückgängig macht. Dieses Problem existiert beim CPC aufgrund der komfortablen Möglichkeit der RSX-Erweiterungen (RSX = Resident System Expansion) zum Glück nicht. Durch die Verwendung des RSX-Mechanismus können in dieser Hinsicht Befehlserweiterungen praktisch beliebig kombiniert werden. Es brauchen nämlich für die Erkennung von Erweiterungsbefehlen keine Vektoren des Betriebssystems umgelegt zu werden; die Erkennung von externen Befehlen ist im Betriebssystem bereits vorgesehen.

Das andere Problem, das sich bei der Kombination von Maschinenprogrammen ergibt, ist jedoch nicht vom Betriebssystem, sondern von einer Eigenschaft der Maschinensprache selbst abhängig: Ein Maschinenprogramm ist prinzipiell an einen bestimmten festen Speicherbereich, für den es geschrieben wurde, gebunden. Es kann zwar an einer beliebigen Stelle im Speicher stehen, ist aber nur an dem ihm zugedachten Ort auch ausführbar. Dies rührt daher, daß viele Befehle der Maschinensprache sich auf absolute Speicheradressen beziehen. Bei den Befehlen, die sich auf Adressen innerhalb des Maschinenprogrammes selbst beziehen, ergibt sich daher als logische Konsequenz, daß diese Befehle nur dann an der angesprochenen

Adresse den Programmteil oder die Daten vorfinden, auf die sie zugreifen sollen, wenn das Programm auch an der richtigen Stelle im Speicher steht.
Da nun aber Befehlserweiterungen in der Regel möglichst weit am oberen Ende des verfügbaren Speichers stehen, um möglichst wenig des kostbaren Speicherplatzes für BASIC-Programme zu beanspruchen, ergibt sich ein schwerwiegendes Problem bei der Kombination von Befehlserweiterungen: Für gewöhnlich kollidieren diese bei der Belegung des Speicherplatzes, wodurch eine später geladene Befehlserweiterung zumindest einen Teil einer vorher geladenen Erweiterung auslöscht. Die fatale Folge dieser Kollision ist zumeist ein Absturz des Rechners oder mindestens eine Fehlfunktion der zuerst geladenen Befehlserweiterung.

23.2 Wie Relokalisation im allgemeinen funktioniert

Das Zauberwort zur Behebung der bis hierhin beschriebenen Schwierigkeiten bei der Kombination von Maschinenprogrammen heißt "Relokalisation". Hinter diesem klangvollen Fremdwort verbirgt sich eine Technik, die ein Maschinenprogramm durch Anpassung der sich auf das Programm selbst beziehenden Adressen zumindest teilweise von der absoluten Adresse des Programmes unabhängig macht. Um dies zu erreichen, gibt es verschiedene Möglichkeiten:

Die erste dieser Möglichkeiten beruht auf der Verwendung von speziellen Zwischenfiles, die einen speziellen Lader-Code enthalten, der neben dem eigentlichen Programm noch für die Relokalisation wichtige Informationen enthält. Dieser Lader-Code ist noch nicht als Maschinenprogramm ausführbar. Er muß erst noch mittels eines speziellen Ladeprogrammes in den ihm zugedachten Speicherbereich gebracht werden, wobei er automatisch in ein in diesem Speicherbereich lauffähiges Programm umgewandelt wird. Bei dem für diese Aufgabe verwendeten Ladeprogramm handelt es sich in der Regel nicht nur um ein einfaches Ladeprogramm, sondern um einen sogenannten Linker. Ein Linker ist nicht nur in der Lage, Programme zu laden und zu starten, sondern auch, mehrere relokative Programm-Module

automatisch zu einem Gesamtprogramm zusammenzufügen (daher der Name: engl. "to link" = "verbinden").
Auf diese Weise arbeitet zum Beispiel die Firma Microsoft bei ihrem MACRO80-Assembler. Der dazugehörige Linker bietet neben der Auswahl, ob das endgültige Programm gestartet, als ausführbares Maschinenprogramm auf Diskette gespeichert oder einfach im Speicher belassen werden soll, noch die Möglichkeit des automatischen Einbindens von Routinen aus einer Bibliothek an. Auch der Zugriff von einem Programm-Modul auf Teile oder Daten eines anderen gleichzeitig geladenen Programm-Moduls ist kein Problem; er wird über sogenannte "globale Labels" abgewickelt, die praktisch komplett vom Linker verwaltet werden.
Was die gebotenen Möglichkeiten angeht, ist diese Methode zweifellos als die beste zu bezeichnen. Es ergeben sich hier Möglichkeiten, deren Flexibilität und Vielseitigkeit praktisch nur noch durch Speicherplatz, Rechnergeschwindgkeit und Einfallsreichtum begrenzt sind. Größere Maschinenprogramme können mit diesem Softwarepaket nach allen Regeln der Kunst strukturiert und modular aufgebaut werden; sie sind besonders übersichtlich und änderungsfreundlich.
Es ist zum Beispiel möglich, Programm und Daten in beliebigen, voneinander getrennten Speicherbereichen unterzubringen, was sich als sehr praktisch erweist, wenn man sein Programm in ein EPROM brennen möchte.
Die erheblichen Vorteile dieser Lösung müssen aber mit einem recht hohen Aufwand bezahlt werden: Es ist ein spezieller Assembler vonnöten, der den speziellen Lader-Code erzeugt. Dieser Assembler ist nicht nur wesentlich umfangreicher als ein normaler, sondern aufgrund der erheblich erweiterten Möglichkeiten ist es auch erheblich schwieriger, ihn in seinem vollen Umfang zu nutzen. Auch der Linker ist ein nicht gerade kleines Programm.

Die zweite Möglichkeit geht von einem, mit einem beliebigen, ganz normalen Assembler geschriebenen Programm aus. Das Programm wird mit dem Assembler auf irgendeine prinzipiell beliebige Basisadresse festgelegt. (Zum Test des Programmes empfiehlt sich allerdings eine Adresse, an der das Programm ausführbar ist.) Nun beginnt die Knochenarbeit: Das relokativ zu

machende Programm muß mit einem speziell auf dieses Programm ausgelegten Ladeprogramm versehen werden, das die gewünschte Basisadresse erfragt oder sonstwie ermittelt und zwischen Laden und Starten des Programmes die Adressen, die sich auf Ziele innerhalb des Programmes selbst beziehen, automatisch ändert.
Das eigentliche Ladeprogramm sieht zwar prinzipiell immer wieder gleich aus und kann ohne weiteres auch in BASIC geschrieben sein, doch muß die Tabelle der anzupassenden Adressen für jedes Programm einzeln erstellt und bei fast jeder Programmänderung immer wieder entsprechend angepaßt werden. Diese Tabelle ist schon bei mittleren Programmen recht umfangreich und wächst drastisch mit der Länge des Programmes.
Eine automatische Erstellung dieser Tabelle, beispielsweise durch ein Hilfsprogramm, das seine Informationen aus speziellen Kommentaren im Assembler-Sourcecode bezieht, ist zwar denkbar, doch ist sowohl die Erstellung dieses Hilfsprogrammes als auch dessen Handhabung relativ aufwendig.

Es gibt da noch eine weitere Methode, die allerdings mehr experimentellen Charakter hat: Ein "automatischer Relokalisator" disassembliert quasi das zu behandelnde Programm und nimmt gegebenenfalls eine entsprechende Änderung der Adresse vor, und zwar abhängig davon, ob ein Befehl, der eine absolute Adresse enthalten könnte (16-Bit-Ladebefehle, absolute Sprungbefehle und Ladebefehle mit erweiterter oder erweiterter unmittelbarer Adressierung), sich auf eine Adresse innerhalb des Programmes selbst bezieht. Diese Methode ist sehr bequem zu handhaben, aber äußerst unzuverlässig: Sobald es in dem behandelten Programm eine sich auf das Programm selbst beziehende Adressentabelle gibt, funktioniert sie schon nicht mehr. Außerdem muß ja ein Wert hinter einem 16-Bit-Ladebefehl, selbst dann, wenn er einer Adresse innerhalb des Programmes entspricht, nicht immer eine Adresse sein; Zähler-Startwerte und andere Konstanten werden von solchen Relokalisatoren in diesem Fall rücksichtslos geändert.
Aus diesen Gründen kann es also bei Programmen von nennens-

wertem Umfang praktisch nur noch als Zufall bezeichnet werden, wenn ein solches Programm ein brauchbares Ergebnis liefert.

23.3 Es geht auch einfacher!

Die ersten beiden der im vorigen Abschnitt vorgestellten Relokalisationsmethoden sind recht weit verbreitet - was man von der dritten Methode aus gutem Grund nicht sagen kann. Diese beiden Methoden haben aber jeweils entscheidende Nachteile: Die erste Methode erfordert die Neuanschaffung nicht gerade billiger Programme und die Einarbeitung in ein neues System der Assemblerprogrammierung. Die zweite Methode erfordert ausgiebige "Handarbeit".

Die Methode, die wir Ihnen hier vorstellen möchten, hat mit den drei bisher beschriebenen Methoden recht wenig gemeinsam. Sie werden im folgenden eine Relokalisationsmethode kennenlernen, die mit minimalem Aufwand (Das hierfür nötige spezielle Maschinenprogramm ist nur 42 Bytes lang!) fast alle anfallenden Relokalisationsaufgaben übernehmen kann.

Das Prinzip, auf dem diese Methode basiert, ist ebenso einfach wie genial: Das Problem der Unbeweglichkeit von Maschinenprogrammen bestünde fast gar nicht, wenn es Äquivalente zu den relativen Sprungbefehlen (JR) auch für die JP- und CALL-Befehle (mit 16-Bit-Sprungdistanz - versteht sich -, um auf den gesamten Speicher zugreifen zu können) sowie für die Ladebefehle mit erweiterter oder erweiterter unmittelbarer Adressierung gäbe. Und da haben wir uns gedacht: Wenn diese Befehle so sehr fehlen, bauen wir sie doch einfach zusätzlich ein. Das "Erfinden" neuer Befehle ist nun aber nicht gerade eine alltägliche Sache, weswegen wir die Vorgehensweise einmal etwas genauer beschreiben werden.
Bei der Konstruktion unserer neuen Befehle bedienen wir uns einer Methode, die die Konstrukteure des Z80 bereits vorgezeichnet haben, der Prefix-Methode. Sie beruht darauf, daß einem bereits definierten Opcode durch Voranstellung eines so-

genannten Prefix eine abgewandelte Bedeutung zu geben. So wird zum Beispiel aus einem Befehl, der sich auf das HL-Register bezieht, durch das vorangestellte Prefix &DD ein Befehl für das IX-Register, durch &FD einer für das IY-Register. Durch unser neu zu schaffendes Prefix soll nun ein beliebiger Befehl mit 16-Bit-Operand dahingehend geändert werden, daß der Operand nicht absolut genommen wird, sondern sich relativ auf den jeweiligen Stand des Programmzähler-Registers (PC) bezieht. Auf diese Weise können dann innerhalb eines Programmes Programmteile oder Daten relativ zur Adresse des zugreifenden Befehls angesprochen werden, wodurch die tatsächliche absolute Adresse des Programmes für diese Zugriffe unerheblich wird.

Natürlich können wir zur Einführung unserer zusätzlichen Befehle nicht einfach die Funktion des Z80-Prozessors ändern. Statt dessen benötigen wir ein kleines (wie gesagt: 42 Bytes) Hilfsprogramm, das beim Auftreten unseres Relokalisations-Prefix aufgerufen wird und entsprechend reagiert. Als Prefix benutzen wir die im CPC-Betriebssystem für Benutzerzwecke freigehaltene Instruktion RST &30 (entspricht von der Funktion her einem CALL &0030). Es ginge zwar auch mit einem einfachen CALL-Befehl, doch hat dies, vor allem im Hinblick auf Verarbeitungsgeschwindigkeit und Speicherplatzausnutzung, recht große Nachteile. Der Vorteil der RST-Instruktion liegt in der Tatsache, daß sie, im Gegensatz zum CALL-Befehl, praktisch ohne Angabe einer Sprungadresse funktioniert und nur ein Byte belegt.
Die Instruktion RST &30 wird einfach vor den relativ zu interpretierenden Befehl gestellt. Bevor beim Lauf des Programmes der eigentliche Befehl ausgeführt wird, kommt der Prozessor zu unserem RST &30 und führt das dazugehörige Unterprogramm aus. Dieses Unterprogramm pickt sich nun aus dem Befehl, zu dem es gehört, die relative Adresse heraus (wo der Befehl steht, geht aus der auf dem Stack stehenden Rücksprungadresse des RST-Befehls hervor), berechnet aus dieser und der Basisadresse (sie ist gleich der Rücksprungadresse) die effektive Adresse, schreibt diese in den Befehl hinein, überschreibt den RST-Befehl, der es aufgerufen hat, durch eine NOP-Instruktion und kehrt zum eigentlichen Programm zurück, wo dann der eigent-

liche Befehl ausgeführt wird. Durch diese Methode ist das Programm, sobald es einmal ausgeführt wurde, doch wieder an seinen Speicherplatz gebunden, doch kommt der Fall, daß ein Programm während seiner Benutzung noch im Speicher bewegt werden muß, praktisch nicht vor. Es ginge zwar auch anders, doch wäre dafür ein erheblich umfangreicheres Hilfsprogramm notwendig, das auch wesentlich mehr Rechenzeit benötigen würde. Ein solches Programm müßte, im Gegensatz zum hier verwendeten Programm, die jeweiligen Befehle erkennen und deren Funktion eigenständig ausführen. Unser Programm versieht den Befehl jedoch nur mit der korrekten Adresse und läßt den Prozessor den Befehl selbst ausführen. Der erhebliche Geschwindigkeitsvorteil unseres Programmes beruht dabei nicht nur auf der Tatsache, daß das Hilfsprogramm nicht nur auf die Erkennung und Simulation des auf die RST-Instruktion folgenden Befehls verzichten kann, sondern vor allem daraus, daß es nur beim ersten Durchlauf des Befehls überhaupt in Aktion tritt; danach steht hinter dem behandelten Befehl die korrekte absolute Adresse und der Befehl kann ganz normal ausgeführt werden. Zusätzliche Rechenzeit wird dann nur noch für die vor dem Befehl stehende NOP-Instruktion gebraucht, die sehr schnell abgearbeitet wird.

23.4 Das Relokalisations-Programm

Nun soll vorläufig einmal genug der grauen Theorie sein. Sehen Sie sich erst einmal das Listing unseres Hilfsprogrammes an. In der nachfolgenden eingehenden Funktionsbeschreibung werden sicherlich die theoretischen Grundlagen dieser Methode wesentlich klarer.

Hier das Listing des Relokalisations-Hilfsprogrammes:

```
0000   E3        RELRST   EX     (SP),HL     ; HL sichern, Rueck-
                                               sprungadresse nach HL
0001   D5                 PUSH   DE          ; DE sichern
0002   C5                 PUSH   BC          ; BC sichern
0003   F5                 PUSH   AF          AF sichern
```

```
0004  E5           PUSH  HL          ; Ruecksprungadr. sichern
0005  7E           LD    A,(HL)      ; 1. Byte des Befehls
0006  FE ED        CP    &ED         ; Test auf Prefix
0008  28 08        JR    Z,SKPPFX
000A  FE DD        CP    &DD
000C  28 04        JR    Z,SKPPFX
000E  FE FD        CP    &FD
0010  20 01        JR    NZ,NOPFX
0012  23    SKPPFX INC   HL          ; Prefix ueberspringen
0013  23    NOPFX  INC   HL          ; HL zeigt auf Adressfeld
0014  5E           LD    E,(HL)      ; Offset Low-Byte
0015  23           INC   HL
0016  56           LD    D,(HL)      ; und High-Byte holen
0017  2B           DEC   HL          ; HL zeigt auf Anfang des
                                     Adressfeldes
0018  EB           EX    DE,HL       ; Zeiger auf Adressfeld
                                     nach DE, Offset nach HL
0019  C1           POP   BC          ; Ruecksprungadresse
                                     (=Basisadresse) nach BC
001A  C5           PUSH  BC          ; und wieder auf Stack
001B  09           ADD   HL,BC       ; Offset zu Basisadresse
                                     addieren
001C  EB           EX    DE,HL       ; Adressfeld-Zeiger nach
                                     HL, eff. Adr. nach DE
001D  73           LD    (HL),E      ; Eff. Adresse Low-Byte
001E  23           INC   HL
001F  72           LD    (HL),D      ; und High-Byte in Adress-
                                     feld schreiben
0020  E1           POP   HL          ; Ruecksprungadr. nach HL
0021  2B           DEC   HL          ; HL zeigt auf RST-
                                     Instruktion
0022  36 00        LD    (HL),&00    ; RST-Instruktion durch
                                     NOP ersetzen
0024  23           INC   HL          ; HL enthaelt wieder
                                     Ruecksprungadresse
0025  F1           POP   AF          ; AF zurueckholen
0026  C1           POP   BC          ; BC zurueckholen
```

```
0027  D1          POP   DE          ; DE zurueckholen
0028  E3          EX    (SP),HL     ; HL zurueckholen, Rueck-
                                     sprungadresse auf Stack
0029  C9          RET               ; Ruecksprung
```

Daß die Basisadresse dieses Assemblerlistings bei &0000 liegt, bedeutet auf den ersten Blick, wie Ihnen vielleicht schon aufgefallen ist, daß es an dieser Adresse nie und nimmer ausführbar ist. Der RAM-Bereich von &0000 bis &003F ist für Benutzerprogramme absolut tabu, weil er die wichtigsten und meistverwendeten Betriebssystemvektoren enthält. (Eine Ausnahme davon stellt lediglich der bereits erwähnte RST &30-Vektor dar.)
Diese Tatsache ist allerdings kein Beinbruch, denn es handelt sich dabei um relative Adressen. Sie stellen also nicht die tatsächliche absolute Speicheradresse dar, sondern lediglich die Distanz zu einer noch zu bestimmenden Basisadresse.
Die Angabe von relativen Adressen ist bei relokativen Assemblern üblich; es hat ja ohnehin keinen Wert, absolute Speicheradressen anzugeben, wenn die tatsächliche Basisadresse noch gar nicht feststeht.
Sollte Sie dies stören, so denken Sie sich einfach ein A an Stelle der ersten 0 in jeder Adresse. Dadurch ändert sich am Sinn des Listings und an der Funktionsfähigkeit des Programmes nichts, außer daß es mit der Basisadresse &A000 auch ausführbar ist.
Unser kleines Hilfsprogramm ist übrigens, da in ihm kein einziger Befehl vorkommt, der sich auf eine absolute Adresse bezieht, schon von vorne herein relokativ, das heißt, es kann ohne jegliche besondere Maßnahme in jedem beliebigen, für Programme zugänglichen Speicherbereich betrieben werden.
Über die Ermittlung der tatsächlichen Basisadresse erfahren Sie im folgenden Abschnitt mehr.

Trotz des geringen Umfangs des Programmes erfüllt es vollkommen die gestellten Anforderungen.

Eine eigentlich selbstverständliche Eigenschaft dieses Programmes, die bis hier hin noch nicht erwähnt wurde, soll nicht verschwiegen bleiben: Da es quasi als Befehlssatzerweiterung des

Z80-Prozessors fungieren soll, muß darauf geachtet werden, daß es nicht irgendwelche Register verändert. Ein wirklich universeller Einsatz ist nur dann möglich, wenn das Unterprogramm nach außen hin keinen Einfluß auf die Register hat. Aus diesem Grunde werden sämtliche benutzten Register am Beginn des Programmes gerettet (PUSH) und vor dem Rücksprung wieder vom Stack geholt (POP).
Eine weitere Eigenschaft dieses Relokalisationsprogrammes soll hier noch erwähnt werden, die zwar in den allermeisten Fällen unerheblich ist, doch wenn sie einmal zu einem Fehler führt, dann sucht man lange danach, wenn man folgendes nicht weiß: Da das Betriebssystem des CPC-Rechners das untere ROM ausschalten muß, um an den Vektor für den RST-Befehl heranzukommen (der zeigt ja später auf unser Programm), ist dieses ROM, auch wenn es vorher eingeschaltet war, nach dem Aufruf des RST &30 ausgeschaltet.

23.5 Das Ladeprogramm

Bevor wir nun zu einer eingehenden Funktionsbeschreibung anhand eines Beispiels kommen, sollten wir noch zwei wichtige Fragen im Zusammenhang mit dem Betrieb des Programmes klären: "Wie kommt das Programm an den ihm zugedachten Ausführungsort?" (Das Listing enthielt aus gutem Grund keine Angabe über die Basisadresse des Programmes!) und "Wie wird dafür gesorgt, daß beim Aufruf des RST &30 auch wirklich das Relokalisationsprogramm angesprungen wird?".

Beantworten wir zunächst einmal die zweite Frage. Abgesehen davon, daß der Befehl RST &30 nur ein Byte lang ist und schneller ausgeführt wird, entspricht er genau dem Befehl CALL &0030. An der Stelle &0030 im RAM stehen nun dem Benutzer acht Bytes zur Verfügung, von &0030 bis einschließlich &0037, die er zum Aufruf seiner RST-Routine verwenden kann. Da kaum eine Routine in den acht Bytes Platz haben dürfte, muß von dort aus ein Vektor auf die gewünschte Routine gelegt werden. Dieser besteht lediglich aus einer unbedingten Sprunganweisung (JP). Es müssen also nur der entsprechende Opcode (&C3) nach &0030 und die Ansprungadresse

unseres Programmes nach &0031 und &0032 gebracht werden. Immer, wenn nun ein RST &30 ausgeführt wird, wird über diesen Sprungbefehl die dazugehörige Routine aufgerufen.
Da wir für das Setzen des RST-Vektors auf unsere Routine deren Startadresse (sie ist hier gleich der Basisadresse) benötigen, führt uns dies zu unserer ersten Frage zurück: Wie wird die Basisadresse ermittelt und wie gelangt das Programm dorthin?

Bei der Unterbringung unseres Hilfsprogrammes im Speicher kommt uns die Tatsache, daß es bereits ohne besondere Vorkehrungen vollkommen relokativ ist, sehr entgegen. Es ist dadurch möglich, ohne besonderen Aufwand einen Lader für unser Programm zu schreiben, der es automatisch dahin bringt, wo es am günstigsten liegt: an das obere Ende des freien Speichers.
Zum Laden des Programmes benutzen wir der Einfachheit halber eine Variante des bekannten BASIC-Laders. Dieser Lader dürfte aufgrund des geringen Umfang des Hilfsprogrammes auch keine größeren Schwierigkeiten beim Abtippen bereiten.

Hier ist das Ladeprogramm:

```
10 REM BASIC-Lader fuer Relokalisations-Hilfsprogramm    Ver. 1.0
20 REM (hr) 11/85
30 MEMORY HIMEM-42
40 FOR mptr=HIMEM+1 TO HIMEM+42
50 READ byte
60 POKE mptr,byte
70 cs=cs+byte
80 NEXT
90 IF cs<>&H165C THEN PRINT CHR$(7);"Pruefsummenfhler!    ";
HEX$(cs):STOP
100 REM RST-Vektor setzen:
110 POKE &H0030,&HC3:POKE &H0031,(HIMEM+1) AND 255:POKE &H0032,
(HIMEM+1)/256
120 PRINT "Relokalisations-Hilfsprogramm korrekt geladen bei ";
HEX$(HIMEM,4)
130 END
1000 DATA &HE3,&HD5,&HC5,&HF5,&HE5,&H7E,&HFE,&HED
```

```
1005 DATA &H28,&H08,&HFE,&HDD,&H28,&H04,&HFE,&HFD
1010 DATA &H20,&H01,&H23,&H23,&H5E,&H23,&H56,&H2B
1015 DATA &HEB,&HC1,&HC5,&H09,&HEB,&H73,&H23,&H72
1020 DATA &HE1,&H2B,&H36,&H00,&H23,&HF1,&HC1,&HD1
1025 DATA &HE3,&HC9
```

Programmbeschreibung:

30: Durch Herunterstzen von HIMEM wird Patz am oberen Speicherende reserviert.

40-80: Der Maschinencode wird aus den DATA-Zeilen gelesen und in dem reservierten Bereich gespeichert. In Zeile 70 wird dabei die Prüfsumme aufsummiert.

90: Hier wird die ermittelte Prüfsumme mit der vorgegebenen vergichen und gegebenenfalls ein Fehler gemeldet (unter Angabe der falschen Prüfsumme) und das Programm abgebrochen.

110: Umlegung des RST &30-Vektors auf die Startadresse des Hilfsprogrammes.

120: Es wird unter Angabe der Basisadresse des Relokalisationsprogrammes der Erfolg des Ladeprozesses gemeldet.

130: Hier wird das Programm beendet. Statt des END-Befehls kann hier auch der Befehl NEW eingesetzt werden, der das nach erfolgtem Laden unnötige BASIC-Programm automatisch löscht. Diese Version des Programmes muß natürlich vor dem ersten Start gespeichert werden, da es sich nach erfolgreichem Durchlauf selbsttätig löscht!

Dieser BASIC-Lader erledigt alles, was zur Installation unseres Hilfsprogrammes nötig ist: Die Ermittlung der Basisadresse

(HIMEM minus Programmlänge), die Reservierung von Speicherplatz (MEMORY-Anweisung), den Transfer des Programmes in den resrvierten Bereich und die Einrichtung des Vektors. Nun brauchen wir es nur noch zu benutzen...

23.6 Die Anwendung des Relokalisationsprogrammes

Wie hat nun ein Assembler-Programm auszusehen, wenn es für den Betrieb mit dem Relokalisations-Hilfsprogramm geeignet sein soll?
Die Änderungen, die an einem normalen Assembler-Programm vorzunehmen sind, um es fit für unser Relokalisationsprogramm, sind nicht sehr gravierend. Es brauchen an dem Programm selbst nur die folgenden beiden Änderungen vorgenomen zu werden:

* Jedem Befehl, der sich auf eine in seinem Operandenfeld angegebene Adresse bezieht, muß der Befehl RST &30 direkt vorangestellt werden.

* In denselben Befehlen muß die absolute Adresse durch eine PC-relative Adresse von 16 Bit Länge ersetzt werden.
  Eine PC-relative Adresse gibt nicht den Abstand von der Basisadresse des Programmes an, sondern die Distanz vom darauf Bezug nehmenden Befehl aus (ähnlich JR-Befehl).

Das Einbauen der RST-Befehle in ein vorhandenes Programm stellt kein Problem dar; aber wie sieht das mit den "PC-relativen Adressen" aus?
Auch diese Adressierungsmethode ist ohne besonderen Aufwand in den Griff zu bekommen. Die Verwendung von PC-relativen Adressen ist für unseren Zweck sehr vorteilhaft, da das Programm die kompletten Daten, die es vom Hauptprogramm benötigt, direkt oder indirekt über den Stack bekommt. Aus diesem Grunde ist die Eintragung der zu behandelnden Programme in irgendeine Tabelle der Basisadressen unnötig. Durch die 16-Bit-Distanzangabe haben wir auch einen in jedem

Falle ausreichenden Adressbereich zur Verfügung, der den gesamten Speicher des CPC (beim 6128 nur eine Bank) erlaubt. Die Eingabe der PC-relativen Adresse in ein Assembler-Programm ist auch absolut unkompliziert. Sie brauchen der Adresse lediglich die Zeichen -$ anzufügen - fertig ist die PC-relative Adresse.

Angenommen, Sie haben folgenden Unterprogramm-Aufruf, der innerhalb eines zu behandelnden Programmes steht (der Aufruf bezieht sich auf eine Adresse innerhalb desselben Programmes):

```
CALL   NUMOUT
```

Um diesen Befehl 'umzurüsten', muß statt dessen geschrieben werden:

```
RST    &30
CALL   NUMOUT-$
```

Die Geschichte mit dem '-$' ist dabei kein bislang unbekannter Zauberbefehl des Assemblers. Der Assembler bietet die Möglichkeit, mittels des Symbols $ auf den Programm-Zähler des Assemblers und somit auf die Adresse des Befehls zuzugreifen, in dessen Zeile das $ steht. Der Befehl

```
JP     $
```

stellt somit beispielsweise eine Endlosschleife dar.

Wir subtrahieren also die Adresse des zugreifenden Befehls von der absoluten Adresse (gemäß der vom Assembler her vorgegebenen Basisadresse des Programmes), auf die der Befehl zugreifen soll, und haben die relative Adresse.

Hinweis: Der Zugriff auf den Programmzähler ist in den meisten Assemblern möglich. Das Dollarzeichen ($) stellt die dafür übliche Bezeichnung dar. Abweichende Bezeichnungen, wie z.B. ein * sind aber möglich.

Sind alle Befehle, die sich auf Ziele innerhalb des Programmes selbst beziehen, so behandelt, ist das Programm schon fertig und kann in jedem freien Speicherbereich ablaufen, wenn das Relokalisations-Hilfsprogramm installiert ist.

23.7 Ein relokatives Beispielprogramm

Nun möchten wir Ihnen einmal die Anwendung des Relokalisationsprogrammes an einem praktischen Beispielprogramm demonstrieren.
Das, was das Programm tut, ist nichts weltbewegendes: Nach einem Aufruf mit CALL zählt es rückwärts von 100 bis 0 und kehrt danach zum BASIC zurück.

Obwohl das Programm selbst, abgesehen von seinem Beispiel-Charakter, nicht praktisch einsetzbar ist, bietet es doch recht interessante Routinen. Das Programm wurde nämlich, um den relativen Aufruf von Unterprogrammen zu demonstrieren, mit Ausnahme der Routine TXT OUTPUT vollkommen unabhängig vom Betriebssystem ausgelegt. Es enthält somit eine eigene dezimale Ausgaberoutine: die Routine NUMOUT. Diese Ausgaberoutine ruft wiederum die Routine DIV168 auf, die eine binäre Division durchführt. Wir werden diese Routinen noch näher betrachten, da sie recht übersichtliche Beispiele für etwas kompliziertere arithmetische Routinen darstellen.

Zunächst einmal das Listing des Beispielprogrammes:

```
; Demo-Programm fuer RELRST

0000   21 00 65   RRDEMO   LD     HL,101      ; Startwert+1
0003   2B         LOOP00   DEC    HL          ; Zaehlen
0004   E5                  PUSH   HL          ; Zaehlerstand sichern
0005   F7                  RST    &30         ; Relokalisations-Prefix
```

```
0006  CD 0D 00          CALL  NUMOUT-$    ; Zaehlerstand ausgeben
0009  E1               POP   HL          ; Zaehlerstand zurueck-
                                          holen
000A  7C               LD    A,H         ; HL
000B  B5               OR    L           ;     =0?
000C  20 F5            JR    NZ,LOOP00   ; Nein: weiter zaehlen...
000E  3E 0D            LD    A,0DH       ; CR ausgeben
0010  C3 5A BB         JP    TXTOUT      ; TXT OUTPUT


                ; 16-Bit-Zahl in HL auf Bildschirm ausgeben
                ; Veraenderte Register AF, HL, DE, BC, IX

0013  E5        NUMOUT PUSH  HL          ; Auszugebende Zahl
                                          sichern
0014  F7               RST   &30
0015  21 4B 00         LD    HL,OUTBUF-$ ; Ausgabe-Puffer loeschen
0018  F7               RST   &30
0019  11 48 00         LD    DE,OUTBUF+1-$
001C  01 00 05         LD    BC,5
001F  36 20            LD    (HL),' '
0021  ED B0            LDIR
0023  E1               POP   HL          ; Auszugebende Zahl
                                          zurückholen
0024  F7               RST   &30
0025  DD 21 3F 00      LD    IX,OUTBUF+4-$ ; IX zeigt auf Ausgabe-
                                          puffer
0029  0E 0A            LD    C,10        ; Divisor (konstant)
002B  F7        LOOP01 RST   &30
002C  CD 1D 00         CALL  DIV168-$    ; HL durch 10 dividieren
002F  C6 30            ADD   A,'0'       ; Divisionsrest --> ASCII-
                                          Ziffer
0031  DD 77 00         LD    (IX),A      ; Ziffer in Puffer
                                          schreiben
0034  DD 2B            DEC   IX
0036  EB               EX    DE,HL       ; Quotient ist neue Zahl
0037  7C               LD    A,H         ; Zahl
0038  B5               OR    L           ;     =0?
0039  20 F0            JR    NZ,LOOP01   ; Nein: weiter umwandeln
003B  F7               RST   &30
```

```
003C   21 24 00          LD     HL,OUTBUF-$ ; Puffer ausgeben...
003F   06 05             LD     B,5         ; Zaehler fuer Zeichen
0041   7E        LOOP02  LD     A,(HL)      ; Zeichen holen
0042   CD 5A BB          CALL   TXTOUT      ; Zeichen ausgeben
0045   23                INC    HL          ; Naechstes Zeichen
0046   10 F9             DJNZ   LOOP02      ; Weiter, wenn nicht
                                            fertig
0048   C9                RET                ; Ende der Ausgabe


; Division 16 Bit durch 8 Bit
; Eingaba: Dividend in HL, Divisor in C
; Ausgabe: Quotient in DE, Rest in A
; Veraenderte Register: AF, HL, DE, BC

0049   11 00 00  DIV168  LD     DE,0        ; Quotienten-Register
                                            vorbereiten
004C   06 10             LD     B,16        ; Bit-Zaehler
004E   AF                XOR    A           ; Arbeits- und Rest-
                                            Register vorbereiten
004F   CB 23      LOOP03  SLA    E           ; Quotient
0051   CB 12              RL     D           ;     linsks schieben
0053   CB 25              SLA    L           ; Dividend
0055   CB 14              RL     H           ;     links schieben
0057   17                RLA                ; Hoechstwertigstes Bit in
                                            Akku
0058   B9                CP     C           ; Subtraktion moeglich?
0059   38 02             JR     C,SKIP00    ; Nein: ueberspringen
005B   13                INC    DE          ; Quotient erhoehen
005C   91                SUB    C           ; Divisor subtrahieren
005D   10 F0      SKIP00  DJNZ   LOOP03      ; Nicht fertig: weiter...
005F   C9                RET


0060              OUTBUF  DEFS   5           ; 5 Bytes Ausgabepuffer
```

Im Listing des Demonstrationsprogrammes finden sich wieder die bereits bekannten relativen Adressen.

In dem Demo-Programm sind allen Befehlen, die eine absolute Adresse innerhalb des Programmes selbst enthalten, RST &30-Befehle vorangestellt worden; die Adressen sind durch Anhängen von '-$' in PC-relative Adressen umgewandelt worden. Beispiele dafür sind alle Befehle, die sich auf die Adresse des Ausgabepuffers (OUTBUF) beziehen (an den Adressen &0014, &0018, &0024 und &003C). An diesen Zugriffen zeigt sich, daß auch die Verwendung eines Offsets (z.B. in OUTBUF+4) mit dem Relokalisationsprogramm problemlos möglich ist. Die Ladebefehle, die keine sich auf das Programm selbst beziehende Adresse enthalten, wie beispielsweise das Laden der Konstante bei Adresse &0000, wurden nicht verändert. Der Befehl LD IX,OUTBUF+1-$ an Adresse &0024 demonstriert, wie die RST-Erweiterung mit Befehlen verwendet wird, die ein Index-Register-Prefix enthalten. Der RST-Befehl wird dem Prefix vorangestellt. Das Hilfsprogramm erkennt dieses und berücksichtigt es beim Zugriff auf das Adressfeld des behandelten Befehls. (Siehe Adressen &0005 bis &0013 im Listing des Relokalisations-Programmes.)
Die CALL-Befehle zu den Unterprogrammen NUMOUT und DIV168 wurden ebenfalls mit einem RST &30 versehen. Nicht behandelt wurden dagegen die Aufrufe der Routine TXT OUTPUT, da sich diese auf das Betriebssystem und nicht auf das Beispielprogramm selbst beziehen.
Nicht behandelt wurden natürlich auch die relativen Sprungbefehle JR und DJNZ. Dies ist nicht erforderlich, da sie sich ja ohnehin nicht auf eine absolute Adresse beziehen.

Die Devise 'lieber einmal zuviel als einmal zu wenig' gilt nicht für das Relokalisationsprogramm. Unnötig gesetzte RST &30-Befehle zerstören den auf sie folgenden Code und führen somit in der Regel zu einem Programmabsturz. Wird aber in einem relokativen Programm auch nur ein RST &30 vergessen, so ist mit ziemlicher Sicherheit ein Absturz zu erwarten, wenn das Programm bei der Ausführung an den fälschlicherweise nicht behandelten Befehl gelangt.

Doch nun Schluß mit der Theorie. Mit dem folgenden BASIC-Lader können Sie das Demonstrationsprogramm selber einmal an verschiedenen Ausführungsadressen testen:

```
10 INPUT "Basisadresse";basis
20 oldhimem=himem
30 MEMORY basis-1
40 FOR mp=basis TO basis+100
50 READ byte
60 POKE mp,byte
70 cs=cs+byte
80 NEXT
90 IF cs<>&2751 THEN PRINT "Pruefsummenfehler!":STOP
100 CALL basis
110 ? "Fertig!"
120 MEMORY oldhimem
130 END
1000 DATA &21,&65,&00,&2B,&E5,&F7,&CD,&0D
1005 DATA &00,&E1,&7C,&B5,&20,&F5,&3E,&0D
1010 DATA &C3,&5A,&BB,&E5,&F7,&21,&4B,&00
1015 DATA &F7,&11,&48,&00,&01,&05,&00,&36
1020 DATA &20,&ED,&B0,&E1,&F7,&DD,&21,&3F
1025 DATA &00,&0E,&0A,&F7,&CD,&1D,&00,&C6
1030 DATA &30,&DD,&77,&00,&DD,&2B,&EB,&7C
1035 DATA &B5,&20,&F0,&F7,&21,&24,&00,&06
1040 DATA &05,&7E,&CD,&5A,&BB,&23,&10,&F9
1045 DATA &C9,&11,&00,&00,&06,&10,&AF,&CB
1050 DATA &23,&CB,&12,&CB,&25,&CB,&14,&17
1055 DATA &B9,&38,&02,&13,&91,&10,&F0,&C9
1060 DATA &00,&00,&00,&00,&00
```

Dieser BASIC-Lader erfragt zu Beginn die gewünschte Basisadresse, reserviert an dieser Stelle Speicherplatz und kopiert das Demonstrationsprogramm anschließend dort hin. Vor dem neuen Setzen von HIMEM mittels der MEMORY-Anweisung wird der alte Wert von HIMEM gespeichert, um nach Beendigung des Maschinenprogrammes den alten Zustand wiederherzutellen.

Damit dieses Programm allerdings lauffähig ist, muß natürlich das Relokalisations-Hilfsprogramm installiert sein. Ansonsten dürfte es mit ziemlicher Sicherheit zu einem Absturz des Rechners führen.
Mit diesem BASIC-Lader kann das Programm an jeder für den Betrieb von Programmen zulässigen Adresse betrieben werden. Das sind prinzipiell bei der geringen Länge des BASIC-Programms etwa die Adressen von &1000 bis &9FFF. Dabei muß natürlich darauf geachtet werden, daß durch das Laden des Demo-Programmes nicht das Relokalisations-Hilfsprogramm überschrieben wird. Die höchste zulässige Basisadresse für das Demo-Programm ist also die Basisadresse des Hilfsprogrammes (die wird ja von dem Lader des Hilfsprogrammes angegeben) abzüglich der Länge des Demonstrationsprogrammes (101 Bytes). Unter Einhaltung der genannten Grenzen ist das Programm nun an einer beliebigen Basisadresse ausführbar. Durch Entfernen der RST &30-Befehle aus dem Demonstrationsprogramm kann es in ein ganz normales, an eine feste Adresse gebundenes Maschinenprogramm umgewandelt werden. Bei einem Zeitvergleich werden Sie feststellen, daß das, was die Geschwindigkeit des Programmes angeht, praktisch keinen Unterschied macht.

Nach dieser intensiven Betrachtung des Demo-Programmes dürfte es Ihnen nunmehr nicht schwerfallen, eigene Programme auf die Benutzung des Relokalisations-Hilfsprogrammes umzustellen.

23.7.1 Die Unterprogramme des Demo-Programmes

Wie bereits im vorigen Abschnitt angekündigt, wollen wir jetzt noch etwas näher auf die von dem Demonstrationsprogramm verwendeten Unterprogramme NUMOUT und DIV168 zu sprechen kommen.
Wir können in diesem Kapitel natürlich keine komplette Abhandlung über Arithmetik-Routinen unterbringen, da diese den Rahmen des Kapitels bei weitem sprengen würde. Wir werden deshalb die Funktion der Routinen nur relativ kurz anreißen, um Ihnen Hinweise zu geben, wie Sie diese Routinen

für Ihre eigenen Zwecke umgestalten oder als Anregung für eigene Routinen verwenden können. Wir hoffen aber, daß diese Beschreibung zusammen mit den Kommentaren zu den Routinen im Listing Ihnen einen Einblick in die Funktionsweise der Routinen vermittelt.
In jedem Fall aber können diese Routinen, besonders die Divisionsroutine, so, wie sie im Listing stehen, übernommen werden, wenn keine abgewandelten Anforderungen an die Routinen bestehen. Die Kommentare im Listing zu Beginn der jeweiligen Routine geben Auskunft über die Übergabe der Ein- und die Ausgabeparameter sowie die veränderten Register. Beide Routinen werden mit einem einfachen CALL-Befehl aufgerufen.

Die Ausgaberoutine NUMOUT

Diese Routine gibt die im Register HL übergebene 16-Bit-Binärzahl dezimal aus. Die Ausgabe erfolgt rechtsbündig in einem Feld von fünf Zeichen ab der aktuellen Cursor-Postion; nicht benutzte Ausgabestellen werden mit Leerzeichen aufgefüllt. Die Zahl wird vorzeichenlos behandelt; es können also ganzzahlige Werte von 0 bis 65535 ausgegeben werden.

Die Routine benutzt zur Ausgabe den fünf Zeichen (Bytes) langen Ausgabepuffer OUTBUF. Dies ist nicht nur für die rechtsbündige Ausrichtung erforderlich, sondern auch, weil die Zahl von dieser Routine rückwärts aufgebaut wird. Die Ziffern werden von hinten beginnend erst in den Puffer geschrieben und nach beendeter Umwandlung als ganzes ausgegeben.

Zu Beginn der Ausgaberoutine wird der Ausgabepuffer mit Leerzeichen gefüllt (Adressen &0013 bis &0023). Dies geschieht hier durch gezielt falschen Einsatz des Befehls LDIR. Durch die Befehlssequenz

```
LD      HL,START
LD      DE,START+1
LD      BC,LAENGE-1
LD      (HL),BYTE
LDIR
```

kann ein Block ab der Adresse START der Länge LAENGE mit dem Wert BYTE gefüllt werden, indem der Block quasi auf sich selbst kopiert wird. Dies ist eine relativ schnelle (wenn auch nicht die schnellste) und handliche Methode, um einen Speicherplatz mit einer beliebigen konstanten Byte-Sequenz - also auch mit einem einzigen Byte - zu füllen.

Danach wird das Register IX als Zeiger auf das Ende des Puffers initialisiert und das Register C mit der für die Umwandlung benötigten Konstante 10 geladen.

Anschließend beginnt ab dem Label LOOP01 die eigentliche Umwandlung. Zu diesem Zweck wird die Zahl wiederholt durch 10 dividiert (DIV168) und der sich dabei ergebende Divisionsrest nach der Umwandlung in eine ASCII-Ziffer (ADD A,'0') im Puffer gespeichert (LD (IX),A); anschließend wird der Pufferzeiger um ein Zeichen nach vorne gesetzt (DEC IX). Der im Register DE von der Divisionsroutine übergebene ganzzahlige Quotient wird vor dem Rücksprung an den Anfang der Schleife als neue Zahl in HL gespeichert (EX DE,HL).
Dieser Prozeß wird wiederholt, bis der Quotient nach einem Schleifendurchlauf zu null wird. Der hier verwendete Null-Test für 16-Bit-Register mit den Befehlen LD und OR ist eine einfache und schnelle für die Register HL, DE und BC anwendbare Testmethode, die sich besonders im Zusammenhang mit 16-Bit-INC- und -DEC-Befehlen als sinnvoll erweist, da diese Befehle nicht die Flags beeinflussen.

Ist der Quotient null, so sind keine weiteren Ziffern zu ermitteln. Die Umwandlung selbst ist beendet, die Zahl steht rechtsbündig, mit vorangestellten Leerzeichen im Puffer. Der gesamte Pufferinhalt wird in einer einfachen Schleife (LOOP02) mittels der Systemroutine TXT OUTPUT auf den Bildschirm ausgegeben.

Um die Arbeitsweise der Ausgaberoutine besser verständlich zu machen, seien hier einmal anhand des Beispiels der Umwandlung der Zahl 523 die Inhalte der entscheidenden Register und des Puffers während der einzelnen Durchläufe der Schleife

LOOP01 an der Adresse &0036 aufgelistet. Die Inhalte der Register sind dabei dezimal angegeben, der Inhalt des Puffers entsprechend den darin entlatenen ASCII-Zeichen. Der Inhalt des Registers HL wird, da er durch die Divisionsroutine verändert wird, nicht wirklich an Adresse &0036, sondern vor Aufruf der Divisionsroutine an Adresse &002B angegeben.

Durchlauf	HL	DE	A	Puffer
1	523	52	3	3
2	52	5	2	23
3	5	0	5	523

Nach dem dritten Durchlauf der Schleife ist der Quotient in DE gleich null und somit die Umwandungsschleife beendet.

Am Ende der Routine erfolgt mittels RET der Rücksprung in das aufrufende Programm.

Die Divisionsroutine DIV168

Diese Routine dividiert eine binäre 16-Bit-Zahl durch eine 8-Bit-Zahl und übergibt einen 16-Bit-Quotienten und einen 8-Bit-Divisionsrest zurück. Alle Zahlen sind vorzeichenlose ganze Zahlen.

Zum Verständnis der Funktionsweise dieser Routine sind recht intime Kenntnisse der binären Arithmetik erforderlich. Da die fundierte Vermittlung dieser Kenntnisse den Rahmen dieses Kapitels bei weitem sprengen würde, müssen wir an dieser Stelle leider darauf verzichten.
Als Hinweis auf die Funktionsweise der Divisionsroutine sei hier lediglich erwähnt, daß sie nach dem ins Binärsystem übertragenen Prinzip der bekannten schriftlichen Division funtioniert.
Verzweifeln Sie aber trotzdem nicht am Verständnis dieser Routine. Eine binäre Division ist eine der kompliziertesten elementaren Routinen in Maschinensprache. Spielen Sie die Routine einfach einmal anhand einiger Beispiele auf dem Papier

durch und experimentieren Sie etwas. Ich persönlich habe die binäre Division auch erst verstanden, als ich sie zum erstem Mal selbst programmiert hatte.

23.8 Die Grenzen dieser Relokalisationsmethode

In diesem Abschnitt sollen einmal die Grenzen der in diesem Kapitel vorgestellten Relokalisationsmethode abgesteckt werden.

Mit unserem kleinen Relokalisationsprogramm können wir ohne Probleme jeglichen Zugriff auf ein Objekt behandeln, dessen Adresse im Operandenfeld eines Befehls steht. Etwas komplizierter wird die Sache, wenn wir über eine innerhalb einer Datenstruktur, beispielsweis eine Tabelle von Sprungadressen, stehende Adresse auf das Objekt zugreifen wollen. Aber auch dieses Problem läßt sich lösen.

Zu diesem Zweck muß das Maschinenprogramm allerdings herausbekommen, wo es selbst im Speicher steht. Zu diesem Zweck muß lediglich dieses winziges Unterprogramm aufgerufen werden:

```
GETPC   POP   HL
        JP    (HL)
```

Nach dem Aufruf mit CALL GETPC holt dieses Unterprogramm mittels des POP-Befehls seine Rücksprungadresse vom Stack und springt diese mittels des JP (HL) an. Dies hat denselben Effekt wie ein simpler RET-Befehl, liefert aber Zusätzlich die Rücksprungadresse, also die Adresse direkt hinter dem dieses Unterprogramm aufrufenden CALL-Befehl, in HL zurück. Damit haben wird schon fast alles, was wir brauchen. Mit der Befehlssequenz

```
        RST   &30
        CALL  GETPC-$
BASE    RST   &30
        LD    (BASADR-$),HL
```

kann die tatsächliche absolute Adresse des Labels BASE ermittelt werden. Einträge in einer Adressentabelle (z.B. einer Sprungtabelle) können nun relativ zu BASE angegeben wereden. Statt

```
DW      adresse
```

schreibt man also nun

```
DW      adresse-BASE
```

in die Tabelle. Vor dem Zugriff auf das Objekt muß dann noch die Adresse von BASE, die in dem 16-Bit-Speicher BASADR aufbewahrt wird, zu der relativen Adresse addiert werden. Angenommen, die relative Adresse befände sich im Register HL, könnte dies beispielsweise mit der Sequenz

```
RST     &30
LD      DE,(BASADR-$)
ADD     HL,DE
```

geschehen. Die dazu benötigten Unterprogramme und Speicher nehmen so wenig Platz in Anspruch, daß sie kaum stören dürften.
In unseren Beipielen haben wir das Relokalisations-Prefix (RST &30), soweit es nötig war, bereits mit angegeben.

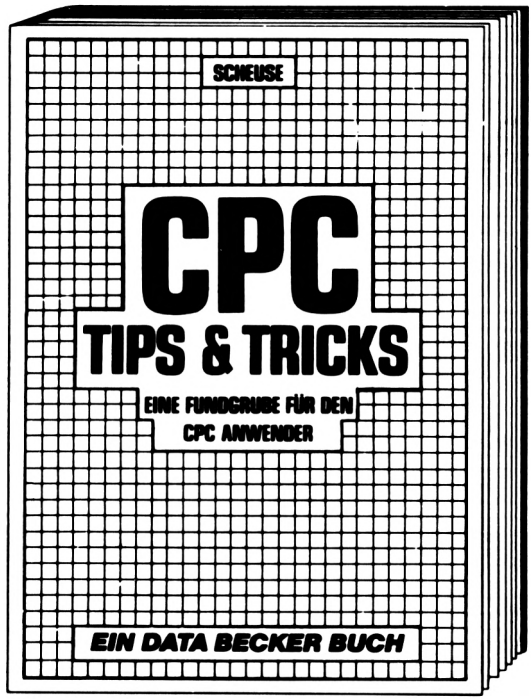
Zusammen mit der hier gegebenen Zusatzhinweisen kann man mit unserem kleinen Relokalisations-Hilfsprogramm praktisch alle Zugriffe, die sich auf Adressen innerhalb desselben Programmes beziehen, relokativ gestalten. Lediglich der Zugriff auf den Adressbereich eines anderen Programmes ist noch mit recht hohem Aufwand verbunden, weil wir nicht über die Möglichkeit eines globalen Zugriffes verfügen.

23.9 Das Laden eines relokativen Programmes

Zum Laden kleinerer Programme erweist sich der BASIC-Lader, wie zum Beispiel bei unserem Demo-Programm, als recht praktisch. Der BASIC-Lader kann ohne großen Aufwand mit Hilfe

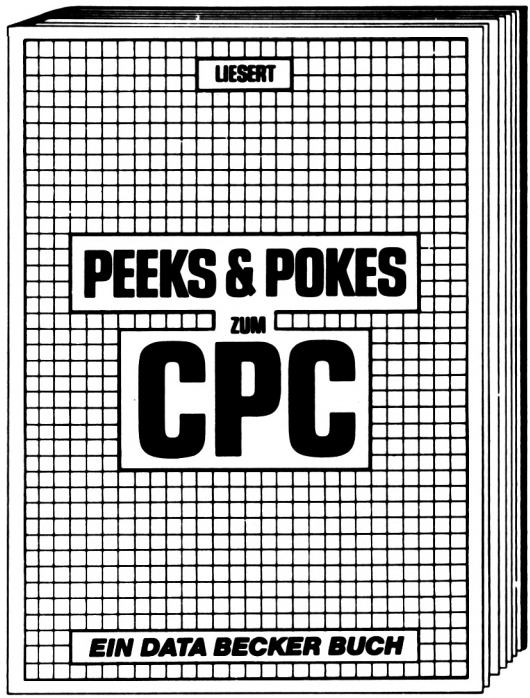
eines Programmes erstellt werden. Der Lader kann dabei direkt so gestaltet werden, daß er die Ermittlung der Basisadresse anhand von HIMEM gestattet, wie das bei dem Lader für das Relokalisations-Hilfsprogramm geschieht.

Für längere Programme ist dies nicht adäquat, da die Länge des BASIC-Laders recht schnell mit der Länge des Programmes wächst (der BASIC-Lader ist grundsätzlich mehr als dreimal so lang wie das eigentliche Programm). Hier bietet sich eher ein Ladeprogramm an, das im Stande ist, ein auf die Basisadresse &0000 gelegtes Binärfile relokativ einzulesen, wobei auch hier eine Ermittlung der Basisadresse mittels HIMEM sinnvoll ist. Ein solches Dienstprogramm ist in Vorbereitung und wird wahrscheinlich auf der Diskette zu diesem Buch erscheinen.



Rund um den CPC 464 viele Anregungen und wichtige Hilfen. Von Hardwareaufbau, Betriebssystem, BASIC-Tokens, Anwendungen der Windowtechnik und sehr vielen interessanten Programmen bis zu einer umfangreichen Dateiverwaltung, Soundeditor, komfortablem Zeichengenerator und kompletten Listings spannender Spiele bietet dieses Buch eine Fülle von Möglichkeiten. Diese Tips kommen von den DATA-BECKER-Spezialisten!

Englisch/Germer/Scheuse/Thrun
CPC 464 Tips & Tricks
271 Seiten, DM 39,–
ISBN 3-89011-039-8

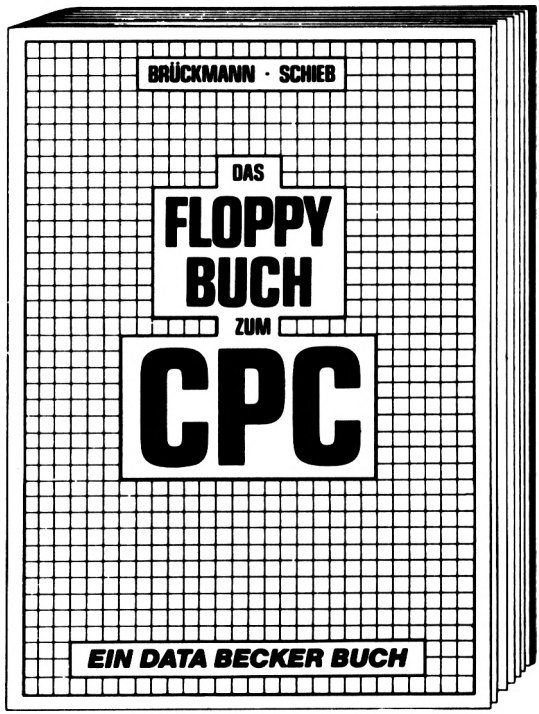


Wer PEEKS und POKES zum CPC 464 kennen und anwenden will, der findet hier umfassende Information! Sie reicht vom Adreßbereich des Prozessors über Betriebssystem und Interpreter bis hin zur Einführung in die Maschinensprache. Dazu Programmierhilfen, Routinen sowie reichlich Material zu den Themen Grafikfunktionen, Massenspeicherung und Peripherie, Tricks und Formeln in BASIC und RAM-Pages!

Liesert
Peeks & Pokes zum CPC
180 Seiten, DM 29,–
ISBN 3-89011-092-4

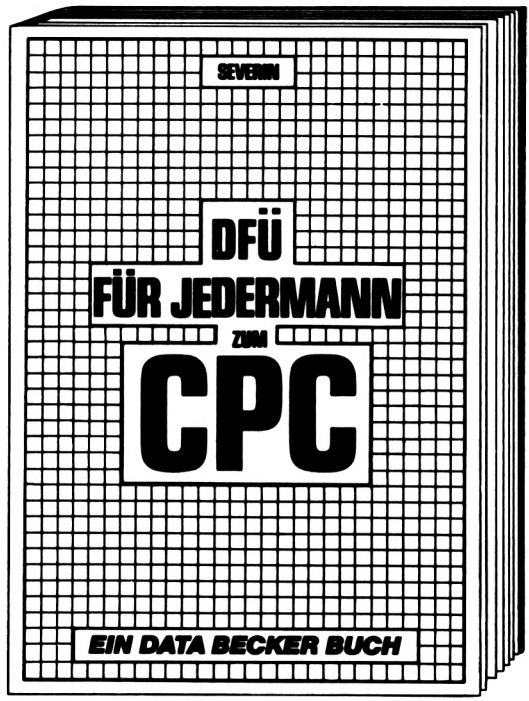
Von den Grundlagen der Maschinensprache-programmierung über die Arbeitsweise des Z80-Prozessors und einer genauen Beschreibung seiner Befehle bis zur Benutzung von Systemroutinen ist alles ausführlich und mit vielen Beispielen erklärt. Im Buch enthalten sind Assembler, Disassembler und Monitor als komplette Anwenderprogramme. So wird der Einstieg in die Maschinensprache leicht-gemacht!

Dullin/Straßenburg
Das Maschinensprachebuch zum CPC 464
330 Seiten, DM 39,–
ISBN 3-89011-070-3



Alles über Floppyprogrammierung vom Einsteiger bis zum Profi. Natürlich mit ausführlichem ROM-Listing, einer äußerst komfortablen Dateiverwaltung, einem hilfreichen Disk-Monitor und einem ausgesprochen nützlichen Disk-Manager. Dazu eine Fundgrube verschiedener Programme und Hilfsroutinen, die das Buch für jeden Floppy-Anwender zur Pflichtlektüre machen!

Brückmann/Schieb
Das Floppy-Buch zum CPC
250 Seiten, DM 49,–
ISBN 3-89011-093-2



DFÜ für jedermann mit dem CPC bietet eine ausführliche und verständliche Einführung in das Gebiet der Datenfernübertragung: was ist DFÜ, BTX, DATEX, Mailbox, alles über Modems und Koppler. Begriffserklärung: Originate, Answer, Half-Duplex usw., eine serielle Schnittstelle am CPC, RS-232/V.24 simuliert, Mailboxsoftware – selbstgestrickt, Postbestimmungen u. v. m. Steigen Sie mit diesem Buch in die Welt der Datennetze und Datenfernübertragung ein!

Severin
DFÜ für jedermann zum CPC
ca. 250 Seiten, DM 39,–
ISBN 3-89011-140-8
Erscheint ca. Dezember 85

Endlich CP/M beherrschen! Von grundsätzlichen Erklärungen zu Speicherung von Zahlen, Schreibschutz oder ASCII, Schnittstellen und Anwendung von CP/M-Hilfsprogrammen. Für Fortgeschrittene: Fremde Diskettenformate lesen, Erstellen von Submit-Dateien u. v. m. Dieses Buch berücksichtigt die Versionen CP/M 2.2 und 3.0 für Schneider 464, 664 und 6128.

Schieb/Weiler
Das CP/M-Trainingsbuch zum CPC
260 Seiten, DM 49,–
ISBN 3-89011-089-4

Empfängerabschnitt

DM 29,-- | Pf ---

für Postgirokonto Nr. 789 - 436

Absender (mit Postleitzahl)

Verwendungszweck: Diskette z. Buch 376 131

DM 29,-- | Pf --- | für Postgirokonto Nr. 789 - 436

Absender:

Zahlkarte (Mit Schreibmaschine, Tinte oder Kugelschreiber deutlich ausfüllen)

DM 29,-- | Pf --- | (DM-Betrag in Buchstaben wiederholen) Neunundzwanzig----------

für DATA BECKER GmbH
Merowingerstr. 30
in 4000 Düsseldorf

Postgirokonto Nr. 789 - 436

Postgiroamt Essen

Postvermerk

Für Vermerke des Absenders

Einlieferungsschein
– Bitte sorgfältig aufbewahren –

DM 29,-- | Pf ---

für DATA BECKER GmbH
Merowingerstr. 30
4000 Düsseldorf

Postgirokonto Nr. 789 - 436

Postvermerk

LADEN, STARTEN - KLAR!

Für alle diejenigen, die sich fleißiges Abtippen ersparen und trotzdem alle Programme nutzen wollen, gibt es einen Ausweg:

Mit der nebenstehenden Post-Zahlkarte einfach die Diskette zum Buch bestellen!

Diskette zum Buch „CPC Tips & Tricks (II)" DM 29,--

Diskette zum Buch
CPC Tips & Tricks (II)
376 131

Für Mitteilungen an den Empfänger

Bedienen Sie sich der Vorteile
eines eigenen Postgirokontos!
Auskunft hierüber erteilt jedes Postamt

Feld
für
postdienstliche
Zwecke

Einlieferungsschein

(nicht zu Mitteilungen an den Empfänger benutzen)

Gebühr für die Zahlkarte
(wird bei der Einlieferung bar erhoben)

bis 10 DM . . . . . 90 Pf
über 10 DM (unbeschränkt) 1,50 DM

DAS STEHT DRIN:

Der 2. Band CPC Tips & Tricks ist für alle CPC Besitzer interessant, die einen 464, 664 oder 6128 besitzen und Insider-Tips suchen zum BASIC, für Befehlserweiterungen und zur Maschinensprache. Viele effektive Hilfsroutinen!

Aus dem Inhalt:

- Sortierverfahren
- 3-D-Grafik
- Menügenerator
- Eingabemaskengenerator
- Schützen eigener Programme
- Variablendump
- Grafik-Hardcopy
- BASIC-Zeile von BASIC aus erzeugen
- Wichtige Facts zur Programmierung in Maschinensprache
- Soft-Scrolling
- Schnittstelle von BASIC zu den Z80-Registern
- Insiderroutinen des Interpreters und des Betriebssystems
- Kompatibilität zwischen den 3 CPC-Rechnern
- Relokative Maschinenprogramme

UND GESCHRIEBEN HABEN DIESES BUCH:

Holger Dullin und Hardy Straßenburg, erfolgreich als Spezialisten-Team für DATA-BECKER (Das Maschinensprachebuch zum CPC), sowie Jochen Schneider, begeisterter Programmierer und DV-Kaufmann im Hause DATA-BECKER, und Helmut Retzlaff, der jahrelange Erfahrungen mit verschiedenen Z80-Rechnern hat.

ISBN 3-89011-131-9

AMSTRAD CPC

MÉMOIRE ÉCRITE
MEMORY ENGRAVED
MEMORIA ESCRITA

https://acpc.me/