LIESERT

PEEKS & POKES

ZUM

CPC

EIN DATA BECKER BUCH

LIESERT

PEEKS & POKES

ZUM

CPC

EIN DATA BECKER BUCH

ISBN-Nr. 3-89011-092-4

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

VORWORT

Haben Sie sich schon einmal gefragt, wie Ihr CPC-464 eigentlich funktioniert? Wollten Sie immer schon wissen, was ein Betriebssystem ist, und was es bewirkt? Zu solchen Fragen kann und will eine Gebrauchsanleitung (auch wenn sie noch so ausführlich geschrieben ist) keine erschöpfende Auskunft geben. Viele der Tricks und Kniffe, die einem Programmierer das Leben erleichtern, wurden aber erst durch einen solchen Blick hinter die Kulissen möglich. Daher soll dieses Buch erklären, wie Ihr Computer funktioniert, was in ihm abläuft, wenn Sie die Return-Taste drücken usw.

Der Titel PEEKS & POKES läßt leicht den Eindruck entstehen, hier handele es sich um ein Buch nur über diese beiden Befehle. Dem ist zum Glück nicht so (das wäre auch für mich sehr langweilig gewesen). Der Titel wurde so gewählt, weil der Vorläufer dieses Buches "PEEKS & POKES zum Commodore 64" hieß und das alte Konzept und die Erscheinungsweise beibehalten werden sollten. Und außerdem wird auch ein Commodore 64 nicht nur mit PEEK und POKE programmiert. Schließlich haben die beiden genannten Befehle bei BASIC-Programmierern allgemein den Ruf, einen ziemlich leichten Einstieg in Betriebssystem und Maschinensprache zu ermöglichen. Genau das wollen wir hier tun.

Natürlich werden auch die zugehörigen Tricks frei Haus geliefert, und das gleich in zweifacher Form. Nach der ausführlichen Vorstellung eines Tips ist am Ende des Kapitels noch eine Zusammenfassung abgedruckt, damit man beim späteren Nachschlagen nicht unbedingt alle Erläuterungen mitlesen muß. Begriffe, die im Stichwortverzeichnis aufgeführt sind, erscheinen im Text kursiv.

Auch die spartanischen Grafikbefehle des CPC werden erweitert - und das leicht verständlich. Sie müssen kein ausgebildeter Informatiker sein, um alles zu kapieren. Und falls Ihnen die Maschinensprache immer noch als Buch mit

sieben Siegeln erscheint, so finden Sie einen Schnupper-Kurs, der Ihnen erste Einblicke in diese Programmiersprache verschaffen soll. Danach können Sie selbst entscheiden, ob Sie sich weiter mit Assembler befassen wollen, oder ob Sie vielleicht PASCAL oder etwas ähnliches lernen.

Mir bleibt nur noch, Ihnen viel Spaß bei der Lektüre dieses Buches und beim Ausprobieren zu wünschen.

Hans Joachim Liesert
Münster, im November 1984

1. WIE FUNKTIONIERT EIN COMPUTER?

In den folgenden Abschnitten lernen Sie den CPC und seine Funktionsweise kennen. Diejenigen, die schon einigermaßen sattelfest in der Computertechnik sind, können getrost weiterblättern. Die "Supercracks" unter Ihnen mögen mir etwaige Vereinfachungen verzeihen, die ich des besseren Verständnisses wegen gemacht habe.

1.1. AUCH EIN MIKROCOMPUTER HAT BUSSE

Zunächst etwas Grundsätzliches. Jeder Mikroprozessor kann einen bestimmten *Speicherbereich* adressieren, d.h. eine gewisse Anzahl von Speicherzellen oder Bytes ansprechen. Dies ist abhängig von der Zahl der Adressleitungen, die der Prozessor besitzt. Jede Adressleitung repräsentiert ein Bit (was das ist, wissen Sie hoffentlich noch aus Ihrem CPC-Handbuch) und kann demzufolge zwei Zustände einnehmen: 0 und 1.
Der Z-80-Mikroprozessor, der das "Gehirn" Ihres CPC bildet, hat 16 Adressleitungen, die alle zusammen auch *Adressbus* genannt werden. Damit kann er 2^{16} = 65536 = 64K Speicherzellen ansprechen.
Vieleicht ist Ihnen schon aufgefallen, daß der CPC über 64K RAM und 32K ROM = 96K verfügt und so eigentlich mehr Speicher als Möglichkeiten zur Nutzung hat. Wie das trotzdem funktionieren kann, werde ich Ihnen später erklären.
Außer dem Adressbus gibt es noch zwei weitere Busse in Ihrem Z-80. Zum einen ist da der *Steuerbus*. Das ist eigentlich nur der Name für die Gesamtheit aller Steuerleitungen, die z.B. den Speicher von Eingabe auf Ausgabe umschalten o.ä.
Außerdem gibt es noch den viel wichtigeren *Datenbus*. Er besitzt 8 Leitungen und hat die Aufgabe, Daten (genauer gesagt 8 Bit = 1 Byte) innerhalb des Computers zu transportieren.
Sowohl Daten- als auch Adressbus sind mit allen Bauteilen

des Computers verbunden, die vom Mikroprozessor angesprochen werden müssen - also mit RAM, ROM und anderen Chips.
Immer wenn der Mikroprozessor einen Befehl ausführt, der Daten außerhalb des Z-80 betrifft, so gibt er zunächst die Speicheradresse (sozusagen die Telefonnummer) auf dem Adressbus aus. Der Speicherbaustein erkennt, welches Byte "gemeint" ist. Dann wird der Mikroprozessor entweder die Daten auf den Datenbus geben, wo sie der Speicher übernimmt, oder der Speicher gibt den Inhalt der Speicherzelle über den Bus an den Z-80 weiter. So einfach ist das!

All dies gilt für jeden 8-Bit-Prozessor (8 nach der Datenbusbreite). Doch ab jetzt wollen wir uns auch mit den speziellen Eigenschaften Ihres CPC-464 beschäftigen.

1.2. DER HARDWAREAUFBAU DES CPC

Keine Angst, auch hier wird es nicht zu technisch. Es ist für das Verständnis der folgenden Kapitel sehr nützlich, wenn man etwas über das Innenleben des CPC-464 weiß.
Am Ende dieses Abschnitts finden Sie ein stark vereinfachtes Blockschaltbild Ihres Rechners (Abb. 1). Wie Sie aus der Abbildung entnehmen können, liegen ROM und RAM teilweise nebeneinander, d.h. sie belegen die gleichen Adressen. Nun kann aber ein Zugriff auf ein Byte nicht zwei verschiedene Werte liefern (es gibt ja nur einen Datenbus). Irgendwo muß also entschieden werden, was gemeint ist: RAM oder ROM. Je nach Wunsch des Prozessors wird dann der eine oder andere Teil vom Datenbus "abgehängt". Dies funktioniert, weil der Prozessor die verschiedenen Bereiche auch für verschiedene Zwecke benutzt. Immer, wenn bestimmte Daten benötigt werden, schaltet der Z-80 vorher rechtzeitig um.
Vom BASIC aus können Sie übrigens nur das RAM ansprechen - per PEEK und POKE. Mit einem kleinen Trick kann man trotzdem das ROM erreichen - wie, das erkäre ich später.

Einen Teil des RAMs belegt der Bildschirmspeicher. Aus

diesem erhält der sogenannte *6845*-Chip seine Informationen. Dieser IC hat die Aufgabe, die Daten aus dem Video-RAM in ein Video-Signal für den Monitor umzuwandeln.

Neben dem 6845 gibt es noch weitere Chips. Der *AY-3-8912* (keine Angst, sie brauchen diese Kombinationen nicht auswendig zu lernen) ist für den Sound zuständig, d.h. er setzt SOUND-Befehle in hörbare Töne um.
Der 8912 ist aber aus technischen Gründen nicht direkt mit dem Z-80 verbunden, sondern erhält seine Daten vom Interfacebaustein *8255* (siehe Abb. 1.).
Dieser integrierte Schaltkreis gibt Daten vom Prozessor an Peripheriegeräte wie Cassettenrecorder, Schnittstellen und auch die Tastatur (die für den Z-80 auch ein externes Gerät darstellt, sie ist nur zufällig im gleichen Gehäuse eingebaut) weiter.
Wenn der Z-80 z.B. die Tastatur abfragen möchte, so teilt er dies dem Schnittstellenbaustein mit. Dieser "sieht" seinerseits nach, ob und welche Taste gedrückt wurde. Das wird dann über den Datenbus an den Z-80 zurückgemeldet. Soll ein Byte über eine Schnittstelle ausgegeben werden, so sendet es der Prozessor an den 8255. Dort wird es dann festgehalten und - sobald das angesprochene Gerät bereit ist - weitergegeben.
Ein weiterer Chip regelt spezielle, interne Abläufe. Er ist für uns nicht von großer Bedeutung.

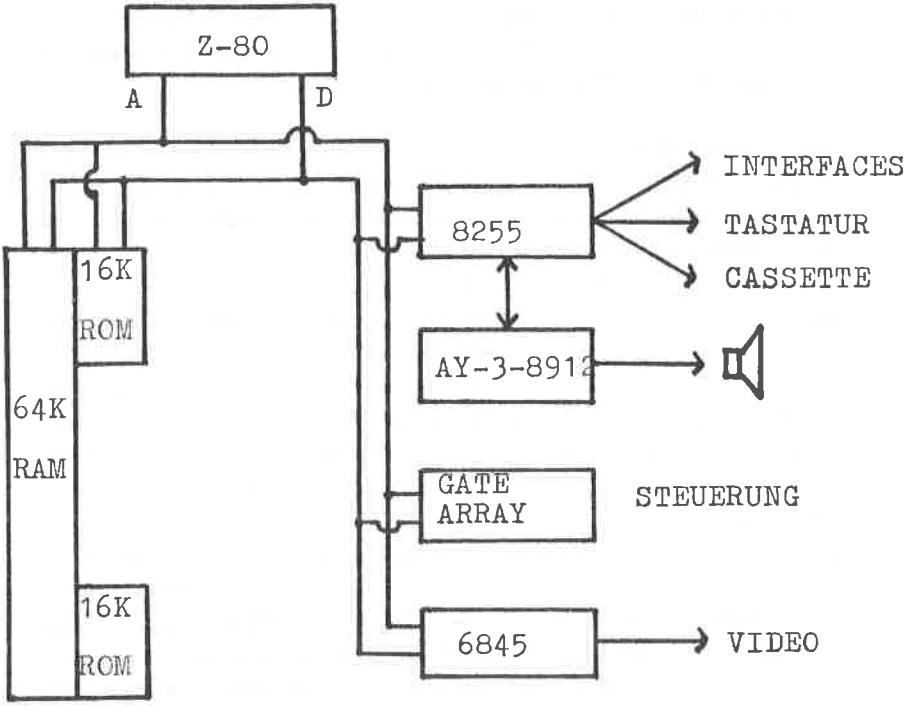


Abb. 1 Der Aufbau des CPC-464

1.3. SPEICHERAUFTEILUNG

Wie Sie schon aus dem letzten Kapitel wissen, besitzt der CPC 64 K RAM. Für die BASIC-Programmierung stehen aber nur 42,5 K zur Verfügung. Mit Recht fragen Sie sich, wo die fehlenden 21,5 K bleiben.

Auf den ersten Blick scheint es möglich, einen viel größeren Speicher für das BASIC zu reservieren. Doch leider braucht ein Computer auch Speicherplätze für interne Funktionen. Zuerst wäre da das *Video-RAM* zu nennen. Es hat die Aufgabe, das Aussehen des Bildschirms zu speichern. Immer wenn ein Punkt auf dem Schirm gesetzt oder gelöscht werden soll, so ändert der Prozessor einen dazugehörigen Wert im Video-RAM. Der Chip, der für die Erzeugung der Video-Signale zuständig ist, sieht dann in regelmäßigen Abständen nach, welche Punkte aufleuchten sollen. Auf diese Weise wird der Prozessor nicht mehr als nötig mit der Bilderzeugung belastet.
Das Video-RAM beansprucht 16 Kilobyte (die deshalb auch nicht für das BASIC benutzt werden können) und liegt im Bereich von 49152 bis 65536. Damit belegt es die gleichen Adressen wie der BASIC-Interpreter.
Ziehen wir die 16 K von den 64 K Gesamt-RAM ab, so bleiben noch 48. In unserer Rechnung fehlen also noch ca 5,5 K.

Jeder Prozessor braucht eine gewisse Anzahl Bytes im Speicher, genannt *Stack* oder *Stapel*, wo er seine "persönlichen" Daten zwischenlagern kann. So muß sich der Z-80 z.B. bei jedem Unterprogrammstart merken, von wo es aufgerufen wurde, um am Routinenende zurückspringen zu können. Dies geschieht mit Hilfe des Stapels (siehe auch Kap. 1.4.). Er liegt im Bereich von 48896 bis 49151 und belegt damit genau 256 Bytes. Sie sollten nie der Versuchung erliegen, in diesen Bereich mittels POKE einzugreifen. Das Ergebnis kann ein Absturz Ihres Rechners sein, aus dem Sie ihn höchstwahrscheinlich nur durch Ausschalten zurückholen können.

Um sowohl im RAM als auch im ROM arbeiten zu können, stellen die Speicherplätze 0 - 63 im RAM eine Kopie der parallelen ROM-Bytes dar. Arbeitet der Z-80 gerade im RAM und braucht ROM-Daten, so kann er mittels der Routinen in diesen 64 Bytes zwischen beiden Bereichen umschalten. Außerdem finden Sie noch weitere Routinen im RAM zwischen BASIC-Speicher und Video-RAM. Sie sollten nie versuchen, in diese Bereiche irgendwelche Werte durch POKE zu verändern. In den allermeisten Fällen wird sich der Rechner aufhängen, d.h. er reagiert nicht mehr auf Tastendrücke o.ä. Dann bleibt Ihnen nur noch das Ausschalten des Computers!

Aber damit noch nicht genug. Betriebssystem und Interpreter brauchen eine gewisse Anzahl Speicherplätze, um sich z.B. Zwischenergebnisse aus arithmetischen Ausdrücken merken zu können, Daten zwischen beiden ROM-Programmen auszutauschen, Tastatureingaben zwischenzuspeichern usw.

In diesem Zusammenhang ist der *Tastaturpuffer* sehr interessant. Er ermöglicht Zeicheneingaben, bevor das BASIC diese überhaupt entgegennehmen kann. Dies können Sie selbst testen. Geben Sie folgendes "Prográmmchen" ein und starten Sie es:

```
1 FOR i = 1 TO 10000: NEXT i
```

Wenn das Programm läuft, tippen Sie bitte eine beliebige Zeichenfolge ein. Solange kein BREAK veranlaßt wird, zeigt sich kein einziger Buchstabe auf dem Bildschirm. Sobald die Schleife aber nach 10000 Durchläufen ihr natürliches Ende findet, erscheinen die Zeichen auf dem Schirm. Während das Programm lief, hat das Betriebssystem bis zu 20 Zeichen gespeichert. So können Sie bei langen Berechnungen die Eingabe für den nächsten INPUT-Befehl schon vorbereiten. Ein "Breaken" eines BASIC-Programmes löscht aber den gesamten Tastaturpuffer!
Dagegen werden die Zeichen auch während langer Befehlsausführungen (z.B. bei LOAD etc., allerdings mit gewissen Ausnahmen) noch gespeichert.

Wenn Sie sich genaueren Überblick über den Speicheraufbau Ihres CPC verschaffen möchten, sollten Sie den Anhang dieses Buches aufschlagen. Dort finden Sie einen Speicherbelegungsplan mit Adressangabe der einzelnen Bereiche.

1.4. POINTER & STACKS

Zwei Fachbegriffe, auf die Sie immer wieder stoßen werden, sind Pointer und Stack.
Pointer (engl. *Zeiger*) zeigen auf bestimmte Stellen im Speicher und werden auch *Vektoren* genannt. Dort können entweder Informationen oder Unterprogramme stehen. Der Cursorpointer beispielsweise zeigt auf die Stelle im Bildschirmspeicher, wo der Cursor gerade steht und gibt daher die Speicherzellen an, wo das nächste Zeichen abgelegt wird.
Zeiger auf Unterprogramme wurden eingeführt, um die Arbeitsweise von Maschinenspracheprogrammen flexibel zu gestalten. Beispielsweise kann ein Programm mittels einer Tabelle von *Unterprogrammzeigern* je nach Erfordernis den richtigen Vektor auswählen, bei PRINT#8 z.B. den achten Zeiger für die Druckeransteuerung (auch wenn der PRINT-Befehl vielleicht nicht so arbeitet, es ist auf jeden Fall ein gutes Beispiel).

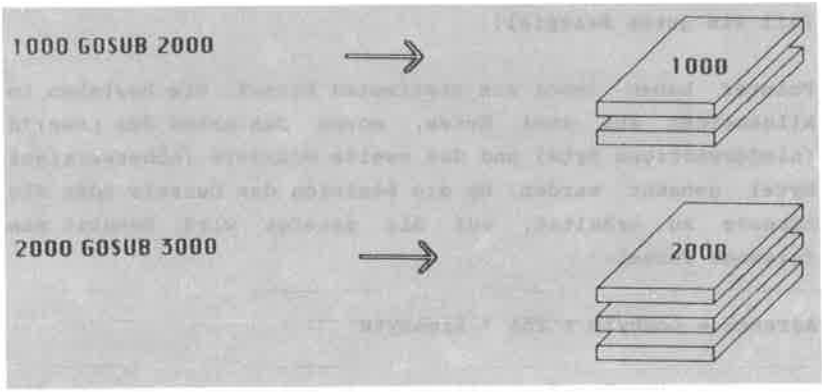
Pointer haben immer ein bestimmtes Format. Sie bestehen im Allgemeinen aus zwei Bytes, wovon das erste das *LOWBYTE* (niederwertiges Byte) und das zweite *HIGHBYTE* (höherwertiges Byte) genannt werden. Um die Position des Cursors oder die Adresse zu erhalten, auf die gezeigt wird, benutzt man folgende Formel:

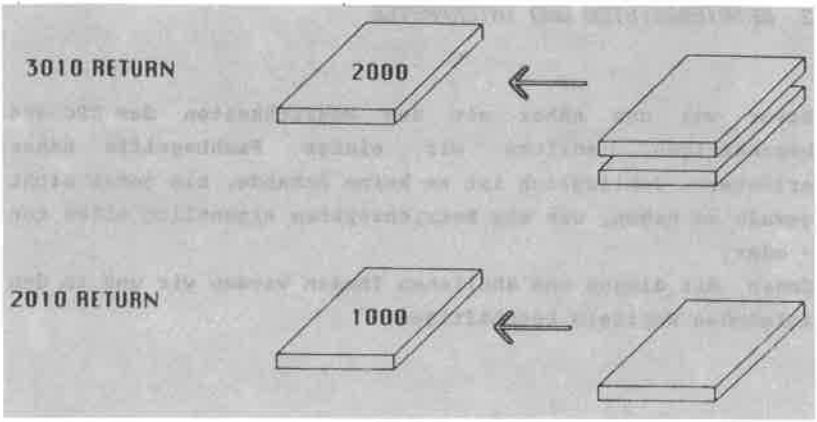
Adresse = Lowbyte + 256 * Highbyte

Wenn Sie im Hexadezimalsystem arbeiten, so setzen Sie einfach das Lowbyte auf die rechte, das Highbyte auf die

linke Seite und erhalten so die vierstellige Hexadresse.
Es gehört zu den Besonderheiten der Computer, daß das Lowbyte immer vor dem Highbyte im Speicher steht.
Ein-Byte-Pointer zeigen innerhalb eines bestimmten Bereichs auf die aktuelle Position (0-255) und werden zu einer *BASISADRESSE* addiert.

Ein *STACK* (engl. *Stapel*) hat die Aufgabe, Daten zwischenzuspeichern und in der umgekehrten Reihenfolge wieder zurückzugeben, wenn sie benötigt werden. Wie bei einem richtigen Stapel kann man immer nur das oberste Element herunternehmen und auch nur ganz oben neue Daten dazulegen. Dies wird vor allem für Unterprogramme benötigt. Beim Aufruf des Unterprogramms wird die gegenwärtige Stelle im Programm auf dem Stack zwischengespeichert, beim RETURN holt sich der Rechner diese Adresse zurück und setzt das Programm fort.
Vorausgesetzt, daß alle Daten, die einmal auf den Stapel gelangt sind, auch wieder heruntergeholt werden, bevor das nächste Element an der Reihe ist, ist gewährleistet, daß jedes Unterprogramm wieder an die richtige Adresse springt.
Ein Beispiel:





Das funktioniert bei Maschinenspracheprogrammen genauso wie bei BASIC. Übrigens werden auch für Schleifen wie FOR-NEXT u.ä. die Rücksprungziele in Stapeln gespeichert. Hieraus erkennen Sie auch den Grund dafür, daß Unterprogramme und Schleifen sich nicht überlappen dürfen. Würde der Rücksprung aus dem Unterprogramm 3000 vor dem Rücksprung aus 2000 erfolgen, so erhielte der Computer falsche Rücksprungziele vom Stapel (die 2000 liegt ja zuoberst)!

```
Zusammenfassung: Zeiger
Adresse = Lowbyte + 256 * Highbyte
Lowbyte = Adresse - INT(Adresse/256)*256
Highbyte = INT (Adresse/255)
Zeiger bestehen im Normalfall aus zwei Bytes, die immer in
der Reihenfolge LOW/HIGH angeordnet sind.
```

2. BETRIEBSSYSTEM UND INTERPRETER

Bevor wir uns näher mit den Möglichkeiten des CPC-464 beschäftigen, sollten wir einige Fachbegriffe näher erläutern. Schließlich ist es keine Schande, bis jetzt nicht gewußt zu haben, was ein Betriebssystem eigentlich alles tut - oder?
Genau mit diesem und ähnlichen Themen werden wir uns in den folgenden Kapiteln beschäftigen.

2.1. DIENSTBARE GEISTER FÜR DIE KLEINARBEIT

Jeder Computer braucht es, viele Namen dafür kennen Sie schon, doch nur wenige Anfänger wissen, was dahintersteckt - gemeint ist das *Betriebssystem*.
Wie der Name schon sagt, ist das Betriebssystem für das Funktionieren Ihres Computers unverzichtbar. Um zum Beispiel ein Byte vom Prozessor zum Drucker zu schicken, sind jedesmal kleine Maschinenprogramme nötig. Zwar läuft die reine Übertragung automatisch ab, doch muß der Schnittstellenbaustein programmiert und kontrolliert werden, damit Fehler u.ä. schnell und sicher festgestellt werden.
Allgemein steuert das Betriebssystem also die Zusammenarbeit zwischen Computer und Peripherie (also auch Tastatur, Drucker usw.). Für jedes Gerät gibt es deshalb eigene Unterprogramme, die vom BASIC-Interpreter oder von Ihren Programmen benutzt werden können. Um Zeichen zur Peripherie zu schicken, muß das BASIC also nur die Daten bereitstellen und dann die entsprechende Routine anspringen.

Sicher haben Sie auch schon von anderen Betriebssystemen wie *CP/M* oder *MS-DOS* (um die beiden berühmtesten zu nennen) gehört und in diesem Zusammenhang auch von "Standard-Betriebssystemen".
Diese Systeme sind so aufgebaut, daß nicht mehr Unterprogramme angesprungen werden, sondern in speziellen

Registern Befehlsnummern und Daten übergeben werden. Auf diese Weise kann das Betriebssystem, das ja für verschieden konstruierte Computer auch jedesmal verändert werden muß, mit den gleichen Codes auf mehreren Rechnern arbeiten. Dazu muß man wissen, daß die Maschinensprache einen unangenehmen Nebeneffekt besitzt. Sehr oft, wenn Routinen angepaßt werden, ändern sich auch die Einsprungadressen der nachfolgenden Unterprogramme, da fast nie die Zahl der Bytes gleich bleibt. Das heißt, daß auf jedem Computer die Betriebssystemroutinen andere Adressen haben und alle Programme, die auf diese Routinen zurückgreifen, dementsprechend umgeschrieben werden müssen. Mit den Befehlscodes aber wird der Befehlssatz der Maschinensprache praktisch um neue Operationen erweitert (und das für alle Computer gleich). Das bedeutet für die Softwarehersteller, daß sie ihre Programme nicht für jeden neuen Rechner neuschreiben müssen.

2.2. IN FREMDSPRACHEN GANZ GROß: INTERPRETER

Der englische Name dieses Maschinenspracheprogrammes im ROM verrät schon viel über dessen Aufgabe. Der *Übersetzer* (so die deutsche Bedeutung von "Interpreter") setzt BASIC-Befehle in Aktionen des Prozessors um. Der Z-80 kann nämlich von Haus aus nur seine spezielle (für jeden Mikroprozessortyp verschiedene) Maschinensprache verarbeiten. BASIC-Befehle sind für ihn zunächst nur mehr oder minder zusammenhängende Folgen von Zeichen. Kommt zu diesen Zeichen noch ein ENTER hinzu, so beginnt der BASIC-Interpreter seine erste Übersetzungsphase (für Science-Fiction-Fans: Alarmstufe 1)!
Jetzt hat der Prozessor alle Hände... äh... Busse voll zu tun, um die Buchstaben und Ziffern als Zeilennummern, BASIC-Befehle und Daten zu erkennen. Befehle erhalten eine spezielle Codenummer, anhand der sich der Interpreter später bei der Abarbeitung das richtige Unterprogramm "herauspicken" kann.

Bis jetzt ist die Zeichenfolge lediglich umcodiert worden, im Programmspeicher steht sie aber noch nicht. Dazu muß vielleicht auch erst einmal Platz geschaffen werden (falls die Zeile nachträglich eingefügt wird). Dann wird sie endlich abgespeichert.
Alle bis hierhin beschriebenen Vorgänge laufen in der für den Programmierer kaum wahrnehmbaren Zeitspanne vom Druck auf die ENTER-Taste bis zum Wiedererscheinen des Cursors in der nächsten Zeile ab.

Der zweite Teil der *Übersetzung* wird mit dem RUN-Befehl gestartet. Der Interpreter holt sich dann nacheinander die BASIC-Befehle und Daten aus dem Speicher und arbeitet sie ab. Findet er beispielsweise einen PRINT-Befehl, so wird das Maschinenunterprogramm PRINT gestartet, welches die Variablen liest, das das Betriebssystem veranlaßt, Zeichen auszugeben usw.
Da die Befehle alle spezielle Codes erhalten haben, braucht nach RUN nicht mehr die Zeichenfolge P R I N T erkannt, sondern nur noch der richtige Unterprogrammzeiger herausgesucht werden. So wird sehr viel Zeit eingespart.

2.3. UNTERBRECHUNGEN MIT SYSTEM

Was zwischen Menschen als unhöflich gilt, gehört unter Computern zum guten Ton. Denn gerade der *Interrupt* ermöglicht viele großartige Anwendungen, die ohne dieses Detail nie machbar wären.
Sowohl BASIC-Interpreter als auch Betriebssystem sind Maschinenprogramme. Beide werden mit dem Einschalten des Rechners gestartet und laufen so lange weiter, bis ein anderes Programm in Maschinensprache aufgerufen wird. Geschieht letzteres durch einen CALL-Befehl, so kehrt der Rechner nach Beendigung der Maschinenroutine zum BASIC zurück.

Wie Sie von der BASIC-Programmierung her wissen, kann ein Computer (Multiprozessorsysteme ausgenommen) immer nur ein Programm zur gleichen Zeit ausführen. Interpreter und Betriebssystem sind aber zwei getrennte Programme, die zur Erledigung bestimmter Aufgaben simultan ablaufen müssen. Wie wird dieses Problem gemeistert?
Die einfachste Möglichkeit, zwei Programme fast gleichzeitig ablaufen zu lassen, ist der *gegenseitige Aufruf*. Immer wenn das BASIC mit einem Teil seiner Arbeit fertig ist, schaltet es das Betriebssystem ein und umgekehrt. Dies geschieht zum Beispiel, wenn auf Peripheriegeräte zugegriffen werden soll. Das BASIC stellt lediglich die Informationen zur Verfügung, die das Betriebssystem dann zum Gerät schickt. Dies beinhaltet aber auch, daß z.B. die Tastatur nur dann abgefragt wird, wenn das Betriebssystem gerade läuft. Nun soll aber während des Programmlaufs zumindest die ESC-Taste eine sinnvolle Wirkung (BREAK) zeigen. Um dieses Problem zu lösen, erfanden die Computerhersteller den INTERRUPT (engl. Unterbrechung). Jede $^1/_{50}$ Sekunde unterbricht der Prozessor das gerade laufende Maschinenprogramm (ob BASIC, Betriebssystem oder eigene Routine) und springt in die Unterprogramme für *Tastaturabfrage* und ähnliche Dinge. "Entdeckt" der Rechner dabei einen Druck auf die ESC-Taste, so wird das gerade laufende BASIC-Programm abgebrochen. Sollte eine andere Taste gedrückt worden sein, so wird dies im Tastaturpuffer gespeichert.

Für den Anwender scheint es, als ob die Tastatur ständig abgefragt würde, da selbst der schnellste Tipper wohl kaum mehr als 15 Zeichen in der Sekunde eingeben kann. Für den Mikroprozessor dagegen erscheint die Zeit zwischen zwei Interrupts ewig lange, da er mit einem Takt von ca. 4 000 000 Schlägen pro Sekunde läuft und ein Maschinenbefehl im Durchschnitt 8 bis 10 solche Schläge zur Ausführung benötigt. Der Prozessor kann also Tausende von Instruktionen durchführen, ehe ein Interrupt ihn aus seiner Arbeit reißt. Nach der Tastaturabfrage macht der Prozessor an der Stelle weiter, an der er vor dem Interrupt aufgehört hat.

Während einiger Befehle wird der Interrupt ausgeschaltet, so z.B. teilweise bei Cassettenoperationen. Das ist auch daran zu erkennen, daß bei Verwendung von alternierenden Farben diese aufhören zu blinken. Auch die Wechselfarben werden über Interrupts gesteuert. Dabei wird bei jedem Interrupt einfach eine andere Farbe eingeschaltet.

Auch im BASIC können Sie Unterbrechungen einsetzen, denn mit den Befehlen AFTER und EVERY können Sie eigene Interruptroutinen als BASIC-Unterprogramme erstellen. Dies funktioniert übrigens vom Prinzip her genau wie in der Maschinensprache.
Trifft der Interpreter auf ein AFTER oder EVERY, so wird ein *TIMER* gestartet, der nach einer bestimmten Zeit wie ein Wecker Alarm bzw. einen Interrupt auslöst. Der Interpreter läßt dann alles stehen und liegen und führt die Interruptroutine aus.

Ein *Interruptsignal* kann auch von anderen Bauteilen des Computers erzeugt werden, um z.B. anzuzeigen, daß von einer Schnittstelle Daten übernommen werden sollen. Das BASIC reagiert aber nur auf TIMER-Signale.

Eine ganz spezielle Art von Unterbrechung stellt der *RESET* dar. Hier wird der Z-80 veranlaßt, genau wie beim Einschalten des Rechners mit dem Programm ab Speicherzelle 0 neu zu starten, egal was bis hierhin geschah. Ab Zelle 0

beginnt aber nichts anderes als die Initialisierungsroutine, die den Speicher löscht und andere wichtige Grundeinstellungen vornimmt. Außer durch SHIFT-CTRL-ESC können Sie dies vom BASIC aus auch mit CALL 0 auslösen - doch Vorsicht! Wichtige Daten sollten Sie zuvor auf Cassette abspeichern.

2.4. NICHT NUR FÜR STARPROGRAMMIERER: SPEZIALBEFEHLE DES BASIC

Stellen Sie sich folgende Situation vor: Sie finden in einer der zahlreichen Computerzeitschriften ein Superprogramm zum Eintippen. Sie haben inzwischen das 20 K Listing eingetippt, doch der Probelauf endete vorzeitig mit einem ERROR.
Es hilft nichts, Sie müssen die Funktionsweise des Programms verstehen, um den Fehler zu beseitigen, wenn Sie nicht jeden Buchstaben im Listing einzeln vergleichen wollen. Wenn da nur nicht diese blöden POKE-Befehle wären! Die benutzt doch kein normaler Programmierer! Es ist also an der Zeit, das Pseudo-Geheimnis um solche Instruktionen zu lüften.

2.4.1. PEEK & POKE

Nehmen wir zuerst den POKE-Befehl. Seine Syntax dürfte bekannt sein: POKE Adresse, Byte. Die Adresse darf zwischen 0 und 65535 liegen, das Byte zwischen 0 und 255. Die Aufgabe dieses Befehls ist es, das Byte unter der angegebenen Adresse abzuspeichern. Dies kann vielen Zwecken dienen, je nach Adresse kann man damit den Bildschirm füllen, einen Maschinenbefehl ablegen oder anderes mehr. Damit können wir dem Computer ganz schön ins Handwerk pfuschen, denn sowohl Betriebssystem als auch Interpreter müssen sich zwangsläufig irgendwo bestimmte Daten "merken". Wie diese Daten aussehen, erfahren wir durch PEEK. Auch hier dürfte die Syntax

hinlänglich bekannt sein: PRINT PEEK (Adresse) gibt das unter der angegebenen Adresse abgespeicherte Byte aus.
Wichtig ist, daß PEEK eine ***FUNKTION*** ist und deshalb nur innerhalb einer Zuweisung (A=PEEK...) oder eines anderen Ausdrucks stehen darf.
Gemeinsam haben beide Befehle die Eigenart, daß sich der Zweck nach der Adresse richtet, auf die zugegriffen werden soll. Es empfiehlt sich daher, bei solchen Befehlen im Speicherbelegungsplan nachzusehen, in welchem Bereich gearbeitet wird. Meist läßt sich daraus die Funktion entnehmen.

2.4.2. CALL

Kommen wir nun zu dem Befehl, der eigentlich nur für Maschinenprogrammierer interessant ist: CALL Adresse. Er dient zum Aufruf von Programmen in Maschinensprache.
Beim CALL-Befehl gibt die Adresse das Byte an, mit dem die Ausführung des Programmes beginnen soll. Nach Beendigung der Maschinenroutine kehrt der Interpreter wie aus einem Unterprogramm in das BASIC-Programm zurück.
Außerdem könnten noch Daten an den CALL-Befehl angehängt werden, die pfiffige Maschinenprogramme dann auswerten.

2.4.3. EIN KLEINER AUSFLUG IN DIE BINÄRARITHMETIK

Zu den beschriebenen Befehlen gesellen sich noch einige, die Sie bestimmt schon kennen, aber deren Vielseitigkeit Ihnen bisher verborgen blieb. Als erstes sind hier *AND*, *OR*, *XOR* und *NOT* zu nennen. Bisher haben Sie sie immer nur in IF-THEN-Konstruktionen verwendet, z.B. in dieser Form:
IF A=0 AND B=0 THEN 100
Eigentlich sind sie aber für die logische Verknüpfung von Variablen und Zahlen gedacht. Dazu muß man wissen, daß der

Rechner auch *Vergleiche* wie Zahlen behandelt. Probieren Sie einmal folgende Befehle:

```
PRINT (1=2)
PRINT (1=1)
```

Ein Vergleich mit dem Ergebnis "wahr" liefert eine -1, ein "falsch" eine 0. Im Binärsystem sieht eine -1 so aus: 1111 1111. Wenn man das am weitesten links stehende Bit nicht als Vorzeichen interpretiert, so ergibt die gleiche Kombination 255. Was aber haben die BASIC-Befehle damit zu tun?

Eine IF-THEN-Konstruktion wird immer dann verlassen, wenn das Ergebnis des Terms gleich 0 ist. Es wäre also auch folgende Befehlsfolge denkbar:

```
IF 3*A THEN 110
```

Die Ergebnisse der einzelnen Vergleiche werden einfach miteinander verknüpft, und das Ergebnis daraus bestimmt den weiteren Programmablauf. Um die Wirkungsweise der einzelnen Verknüpfungen zu verstehen, machen wir jetzt einen kleinen Ausflug in die *Binärarithmetik*.

AND, OR, XOR und NOT sind sogenannte *BOOLESCHE OPERATIONEN*, die der Verknüpfung von logischen Zuständen dienen. Und wie Sie wissen, können logische Zustände mit Bits sehr einfach dargestellt werden (0 für "falsch", 1 für "wahr").

Jeweils zwei Bits werden miteinander verknüpft. Was dabei herauskommt, geben die Tabellen an.

AND	0	1
0	0	0
1	0	1

OR	0	1
0	0	1
1	1	1

XOR	0	1
0	0	1
1	1	0

Sie sehen, daß das Ergebnis auf jeden Fall dann 1 ist, wenn beide Eingangsbits 1 sind. Man kann beide Funktionen wörtlich übersetzen. Bei AND ist das Ergebnis dann 1, wenn Bit 1 ***UND*** Bit 2 auf 1 sind, bei OR, wenn Bit 1 ***ODER*** Bit 2 auf 1 sind.

Bei *Exklusiv-Oder* (XOR) wird das Ergebnis immer dann 1, wenn entweder der eine oder der andere Operand auf 1 sind.

Anders verhält es sich mit NOT. Diese Funktion invertiert einfach das Eingangsbit.

NOT	0	1
	1	0

So weit so gut. Leider bleibt für uns unbedarfte BASIC-Programmierer noch ein Problem. Im BASIC nützen uns einzelne Bits wenig. Dort haben wir es mit Dezimalzahlen zu tun. Um zu berechnen, was der Ausdruck 45 AND 123 ergibt, müssen wir folgendermaßen vorgehen:

1. Zahl ins Dualsystem umwandeln
Wie das geht, wissen Sie aus dem CPC-Handbuch

Die Zahlen 45 und 123 sehen im Binärsystem so aus:
45 = 00101101
123 = 01111011

2. Dualzahlen bitweise verknüpfen
In unserem Beispiel 45 AND 123 ergibt das:

```
   00101101
AND 01111011
------------
   00101001
```

3. Ergebnis ins Dezimalsystem umwandeln
00101001 = 41

Das könnten Sie natürlich auch einfacher haben, indem Sie einfach PRINT 45 AND 123 eintippen. Aber so hat man ungleich mehr Einblick.

Mit Recht stellen Sie jetzt die Frage, wozu das gut sein soll. Neben der Verknüpfung von Vergleichen werden diese Befehle oft zum Beeinflussen einzelner Bits benutzt. Durch eine AND-Verknüpfung mit 254 wird auf jeden Fall das am weitesten rechts stehende Bit gelöscht, durch eine OR-Verknüpfung mit 1 wird es auf jeden Fall wieder gesetzt. Probieren Sie es mit beliebigen Zahlen aus!

2.4.4. VERBINDUNG NACH DRAUßEN: PORT-BEFEHLE

Über die Stecker an der Rückseite Ihres CPC können Sie den Computer mit Peripheriegeräten kommunizieren lassen. Damit dies auch vom BASIC aus möglich ist, gibt es Spezialbefehle. *INP* (Port) holt ein Byte von dem *Port*, der in Klammern angegeben wurde. Die Portnummer ist dabei nicht mit Speicheradressen identisch. Vielmehr gibt es dafür ein eigenes Adressierungssystem, das erst in Verbindung mit detaillierten Maschinensprachekenntnissen verständlich wird. Trotzdem möchte ich die dazugehörigen Befehle erklären, da Sie dann zumindest teilweise wissen, was dahintersteht.
Mit *OUT* Port, Byte kann ein Byte zum Port ausgegeben werden. Wie Sie sehen, entsprechen diese beiden Befehle weitgehend PEEK und POKE.
Jetzt bleibt nur noch ein geheimnisvoller Befehl: *WAIT* Port, X, Y.
Er hat eine Aufgabe, bei der sich jedem Prozessor die Bits im Speicher umdrehen. Er soll nämlich warten. Und das mag ein Computer überhaupt nicht. Dies geschieht durch fortlaufende Verknüpfung von Bytes. Kommt der Interpreter zu einem WAIT-Befehl, so liest er zunächst ein Byte vom angegebenen Port. Diese Zahl wird EXKLUSIV-ODER mit der Zahl Y verknüpft.
Das Ergebnis der ersten Verknüpfung wird nun noch AND-verknüpft mit der Zahl X. Sollte dieses Ergebnis 0 sein, so wiederholt der Interpreter die ganze Prozedur, andernfalls macht er mit dem nächsten Befehl weiter.
Es gibt aber auch noch eine zweite Variante des WAIT-Befehls, bei der das Y-Argument nicht angegeben wird. Hier wartet der Interpreter, bis das Byte vom angegebenen Port AND X ungleich 0 wird.

2.5. BEFEHLE, DIE NICHT IM HANDBUCH STEHEN!

Was man sonst nur bei anderen Computerherstellern erlebte, ist auch beim CPC wieder passiert. Da hat ein Programmierer mit viel Liebe und Aufwand einen Interpreter geschrieben, ihn nach langen Sitzungen vor einem schwarzen Kasten mit Tastatur und Bildschirm endlich von Fehlern befreit, und dann vergißt ein Handbuchautor, einen der Befehle zu beschreiben.

Der Befehl heißt *MOD*. MOD kommt von "modulo", einem Begriff, der den Mathematikern unter Ihnen sehr geläufig sein wird. Eine Modulo-Funktion liefert die Reste aus einer Division. Hier ein paar Beispiele:

```
10 / 4 = 2,5 oder 2 Rest 2
10 MOD 4 = 2
11 MOD 4 = 3 (11 / 4 = 2 Rest 3)
```

Sie können das Befehlswort MOD also für das Divisionszeichen "/" einsetzen und erhalten so den Rest der Division. Allerdings funktioniert MOD nur bei Zahlen und Variablen des Typs INTEGER (-32768 bis +32767).
Mit dem MOD-Befehl kann z.B. der Algorithmus des Euklid (siehe Kap. 14) sehr einfach programmiert werden.
Das Gegenstück zum MOD-Befehl ist die INTEGER-DIVISION. Sie wird durch den umgekehrten Schrägstrich repräsentiert und liefert aus der Division 11 durch 4 das Ergebnis 2. Auch die INTEGER-Division ist nur für Integerwerte gedacht.

In den Vorankündigungen zum CPC war immer davon die Rede, daß *zusätzliche ROMs* angeschlossen werden können, deren Programme (z.B. Spiele, Programmiersprachen, Businessoftware) mit einem BASIC-Befehl gestartet werden können. Dieser Befehl ist im Handbuch nicht verzeichnet. Nichts desto trotz existiert er. Im Gegensatz zu anderen Befehlen besteht dieser allerdings nur aus einem Zeichen, das Sie ereichen können, wenn Sie den Klammeraffen (das kleine a mit dem Kringel darum) zusammen mit SHIFT drücken.

Auf dem Bildschirm erscheint ein senkrechter Strich (den Sie auch für Grafiken benutzen können). Dieser Strich teilt dem Computer mit, daß ein anderer Speicherbereich angesprochen werden soll. Er weiß aber noch nicht, welches ROM. Daher enthalten diese ein sogenanntes *Label*, eine Art Namen, anhand dessen das Betriebssystem das ROM erkennt. Da Sie in Ihrem CPC ohne Erweiterungen nur das BASIC-ROM haben, ist der einzige Name, den Sie benutzen dürfen, das Wort "BASIC". Versuchen Sie dies, und das BASIC startet ganz neu mit der Meldung "BASIC 1.0". Dabei werden alle Daten gelöscht.
Ein ROM-Bereich kann auch mehrere LABELs enthalten, die dann sozusagen als zusätzliche Befehle arbeiten. So enthält auch der Floppy-Controller (siehe Kap. 10.1.) eine solche Befehlssatzerweiterung.

Das BASIC des CPC ist übrigens auch ungewöhnlich mitteilsam. Es ist nämlich eine Funktion vorhanden, die die Adresse, ab der eine Variable gespeichert ist, angibt. Für einen BASIC-Programmierer ist das zwar nicht weiter interessant, wenn Sie jedoch auf Maschinensprache umsteigen wollen, so können Sie das vielleicht nutzbringend anwenden.
Die Funktion ist über den Klammeraffen zu erreichen, PRINT &a ergibt also die Adresse der Variablen a (sofern diese schon existiert, sonst erscheint IMPROPER ARGUMENT)!

Eine weitere Merkwürdigkeit stellt die Funktion *DEC$ (x,y)* dar. Sie wird nur ein einziges Mal erwähnt, und zwar als verwandter Befehl zu BIN$. Im ROM gibt es diese Funktion wirklich, wie sich durch eine Auflistung der Befehlsworttabelle feststellen ließ. Doch scheint es dazu kein Unterprogramm innerhalb des Interpreters zu geben, so daß als einziges Ergebnis ein "SYNTAX ERROR" erscheint. Vielleicht hatte der Programmierer ursprünglich vor, diese Funktion als Gegenstück zu BIN$ und HEX$ einzubauen, kam dann aber auf die (bessere) Idee mit den Prefixen & und &X.

3. DER SPEICHER

Der CPC-464 gehört in Punkto Speicherkapazität zu den Spitzenkönnern unter den Homecomputern. Neben 64 K RAM stehen 32 K ROM zu Ihren Diensten. Außerdem läßt sich der Speicherbereich mit speziellen Zubehörteilen immens erweitern. Wie das funktioniert und was man damit alles anfangen kann, sollen die folgenden Kapitel zeigen.

3.1. SPEICHER SCHÜTZEN

Für das BASIC stehen - wie weiter oben bereits erwähnt - ca. 42,5 Kilobyte freier Speicher zur Verfügung. Die Maschinenprogrammierer unter Ihnen (und solche, die es werden wollen) haben davon allerdings herzlich wenig. Denn der BASIC-Interpreter mag Maschinenprogramme überhaupt nicht. Er macht sehr freizügigen Gebrauch von seinem Speicher. Wird eine neue BASIC-Zeile oder Variable erzeugt, so überschreibt er einfach den (für ihn) freien Bereich nach dem Programm mit den neuen Daten. Dabei werden aber nicht etwa die Speicherplätze nach und nach von unten nach oben gefüllt. Am Anfang des *BASIC-Speichers* stehen die Programmzeilen, die Strings werden an das Ende des reservierten Bereichs gelegt. Dazwischen fristen die anderen Variablen ihr Leben.
Irgendwann "stoßen" die Bereiche in der Mitte des Speichers zusammen; dann gibt der CPC einen Memory-Full-Error aus. Man kann also nie genau sagen, welche Bytes im Speicher frei sind und dies auch während eines BASIC-Programmablaufes bleiben.

Für Maschinenprogramme muß daher eine bestimmte Zahl von Bytes reserviert werden. Dazu dient das *MEMORY*-Kommando, dessen Funktionsweise sehr einfach ist. Das Ende des BASIC-Speichers wird durch einen Zeiger angegeben, an dem sich der Interpreter bei seiner Arbeit orientiert. Das

MEMORY-Kommando tut nichts weiter, als diesen Zeiger unseren Wünschen entsprechend zu verändern. Wird er auf eine niedrigere Adresse gesetzt, so erreicht das BASIC das Speicherende schon eher.
Den Zeiger finden Sie übrigens in den Speicherzellen &AE7B und &AE7C. Sie können ihn statt mit MEMORY auch durch POKE-Befehle ändern, falls Sie Anhänger der Methode "Warum einfach, wenn es auch kompliziert geht?" sind.

Der Befehl "? HIMEM " erfüllt die umgekehrte Aufgabe. Er nennt uns den gegenwärtigen Stand des Pointers. Nach dem Einschalten des Rechners ist er auf 43903 gesetzt. Diese Zahl markiert das Ende des BASIC-Speicherbereichs, der bei Byte 368 beginnt.
Das MEMORY-Kommando wird nur bei Werten zwischen 368 und 43903 ausgeführt, andernfalls wird ein Memory-Full-Error ausgegeben. Dies geschieht, um das Überschreiben von Betriebssystemdaten zu verhindern. Außerdem verhindert MEMORY auch das ungewollte Vernichten von BASIC-Programmen. Soll die Speichergrenze derart weit heruntergesetzt werden, daß dadurch ein bereits bestehendes Programm zerstört würde, so wird ebenfalls ein Memory-Full-Error gemeldet. Diese Sperre läßt sich (wenig sinnvoll) dadurch umgehen, daß der Pointer mittels POKE geändert wird.

Wenn Sie sich 256 Bytes für ein Maschinenprogramm reservieren möchten, so geben Sie einfach MEMORY 43647 ein. Jetzt können Sie ab Speicherzelle 43648 Ihre Maschinenprogramme ablegen (MEMORY und HIMEM beziehen sich auf das letzte Byte des BASIC-Speichers)!

Zusammenfassung: Speicher schützen

Mit dem MEMORY-Befehl kann die Obergrenze des BASIC-Speichers herunter- und wieder heraufgesetzt werden. Der BASIC-Speicher bewegt sich zwischen den Adressen 368 und 43903.
Mit HIMEM kann die gegenwärtige Obergrenze abgefragt werden.

3.2. WIE FUNKTIONIERT DAS BANKSWITCHING?

Wie Sie aus den vorhergegangenen Kapiteln wissen, liegen in einigen Bereichen des Speichers RAM und ROM nebeneinander. Der Prozessor kann aber immer nur einen der beiden Speicherteile ansprechen. Er muß also zwischen RAM und ROM umschalten können. In einer speziellen *Logikschaltung* werden die Speicheradressen, die auf dem Adressbus ankommen, analysiert und dann die entsprechenden Speicherbausteine angesprochen.
Diese Schaltung ist von außen beeinflußbar, das heißt, der Prozessor kann über Ein- und Ausgabebefehle (ähnlich INP und OUT) bestimmen, wie die Adressen dekodiert werden. Mal werden nur die RAM-Bausteine angesprochen, in anderen Fällen wird der Adressbus vom RAM abgekoppelt und an das ROM geschaltet. Außerdem können beide ROM-Bereiche getrennt geschaltet werden. So ist es möglich, daß nur das Betriebssystem aktiv ist, während auf das Video-RAM zugegriffen wird. Die Logikschaltung funktioniert also wie eine Weiche.

Auch vom BASIC aus können Sie auf das ROM umschalten; das Ergebnis ist allerdings in den meisten Fällen ein Aufhängen des Rechners. Versuchen Sie es. Mit OUT &7F82,&82 können Sie beide ROM-Bereiche einschalten.

Wie Sie gesehen haben, ist das Umschalten der Speicherbereiche vom BASIC aus nicht sehr empfehlenswert. Das ist auch kein Wunder, denn mit dem Bankswitching schieben Sie dem Z-80 ein Kuckucksei in Form eines veränderten Speichers unter. In vielen Fällen arbeitet der Interpreter im RAM. Schalten Sie jetzt aufs ROM um, so findet der Prozessor sein eigentliches Programm nicht mehr vor. Die Folge ist ein kapitaler Absturz.

3.3. ROM AUSLESEN

Oft kann es nützlich oder sogar notwendig sein, Daten aus dem ROM zu lesen, z.B. den Zeichensatz (der übrigens im Bereich von &3800 bis &3FFF liegt). Vom BASIC aus ist das nicht möglich. Die PEEK-Funktion bezieht sich nur auf das RAM, und die Umschaltung von RAM auf ROM endet meistens mit dem Verlust des mit soviel Mühe erstellten Programmes. Rein theoretisch wäre es möglich, ein BASIC-Programm zum Auslesen des ROMs zu schreiben, doch der CPC reagiert darauf sehr allergisch. Trotzdem werde ich dieses Programm auflisten, ...

a. da es das Prinzip verdeutlicht und
b. das ersatzweise mitgelieferte Maschinenprogramm nicht mehr erklärt werden muß, da es (fast) eine genaue Übersetzung darstellt.

Das BASIC-Program lautet:

```
1 OUT &7F82, &82
2 A= PEEK (X): REM x= gewünschte Adresse
```

Das Äquivalent in Maschinensprache sieht so aus:

```
1 DATA &01,&82,&7F,&ED,&49
2 DATA &1A,&32,&7F,&AB,&C9
3 MEMORY &AB6F: FOR i=&AB70 TO &AB79
4 READ a: POKE i,a: NEXT: END
5 REM x=gewünschte Adresse
6 CALL &AB70, x
7 a= PEEK(&AB7F): RETURN
```

Zur Nutzung der Maschinenroutine ist zu sagen, daß die Zeilen 1 bis 4 das Programm initialisieren, d.h. hier werden die Maschinenbefehle in den Speicher gebracht (durch die POKE-Schleife). Dieser Teil wird nur einmal am Anfang durchgeführt, dann steht das Maschinenprogramm im Speicher und kann ausgeführt werden. Nur die Werte in den DATA-Zeilen stellen die eigentlichen Befehle dar (10 Bytes).

Wenn Sie jetzt ein Byte aus dem ROM lesen wollen, so müssen Sie der Maschinenroutine dessen Adresse mitteilen. Die Adresse wird in der Variablen x abgelegt und dann mittels GOSUB 6 der zweite Programmteil gestartet.
Zeile 6 übernimmt den Aufruf durch CALL &AB7O,x. Da Maschinenspracheprogramme nicht so ohne weiteres Daten an das BASIC übergeben können, helfen wir uns hier damit, daß der gesuchte Wert aus dem ROM an einer speziellen Stelle im Speicher abgelegt wird, von wo wir ihn mit PEEK (&AB7F) abholen können. Dies geschieht in Zeile 7.
Selbstverständlich können Sie die Variablennamen beliebig ändern, ebenso die Zeilennummern.

3.4. SPEICHERERWEITERUNGEN

In der Werbung finden Sie oft den Hinweis darauf, daß der Speicher des CPC durch zusätzliche Platinen erweitert werden kann. Das können z.B. ROMs mit darin gespeicherten Programmen wie Textverarbeitung o.ä. sein, es ist aber auch möglich, zusätzlichen RAM-Speicherplatz anzuschließen.

Wichtig ist aber, daß auch diese Speicherkarten über die Logikschaltung an den Adressbus angeschlossen werden. Es ist daher möglich, durch Bankswitching diese Speicherbereiche wie das eingebaute ROM anzusprechen. Für den Z-8O macht das kaum einen Unterschied, er muß nur die Zahlen im OUT-Befehl entsprechend ändern.
Auch das Floppy-Laufwerk hat ein solches Erweiterungs-ROM, in dem die Befehle für die Bedienung gespeichert sind.

4. TRICKS FÜR DEN BILDSCHIRM

Der CPC gehört auch bei den Grafikfähigkeiten zu den Spitzenkönnern. Die eingebauten BASIC-Befehle erlauben schon eine weitgehende Nutzung, doch es gibt noch versteckte Möglichkeiten, die in den folgenden Abschnitten aufgezeigt werden.

4.1. BILDSCHIRMSTEUERUNG DURCH CHR$-BEFEHLE

Ist Ihnen die Tabelle mit den Steuerzeichen im Kapitel 9 des Handbuchs schon aufgefallen? Alle diese Zeichen können Sie wie zusätzliche Befehle zur Bildschirmsteuerung nutzen, denn damit lassen sich direkt einige Betriebssystemunterprogramme ansprechen. Auch der BASIC-Interpreter hat nicht etwa eigene Unterprogramme für solche Funktionen, er übergibt dem Betriebssystem einfach das entsprechende Steuerzeichen.

Unten sind übrigens nur solche Steuerzeichen aufgeführt, die sich auch wirklich sinnvoll einsetzen lassen. Was nützt Ihnen ein zweiter LOCATE-Befehl, der zudem noch umständlicher zu handhaben ist, als die BASIC-Anweisung?
Gemeinsam ist allen Steuerzeichen, daß sie über PRINT CHR$(X) aufgerufen werden können. Der PRINT-Befehl sorgt dafür, daß die Zeichen bei der richtigen Betriebssystemroutine landen, die CHR$-Funktion hilft uns lediglich dabei, den Code als Zahl angeben zu können, sonst müßte hier ein Grafikzeichen stehen. Letzteres funktioniert aber auch nicht immer, weil das Betriebssystem Grafik- und Steuerzeichen verschieden behandelt!

Mit dem Steuerzeichen Nr.1 (SOH) können Sie andere *Steuerzeichen sichtbar machen*. Wie Sie wissen, werden Zeichen, deren ASCII-Code kleiner als 32 ist, nicht ausgegeben, sondern als Steuerzeichen an das Betriebssystem übergeben. Setzen Sie dagegen dem Code ein CHR$(1) voran, so

wird ein Zeichenmuster angezeigt; z.B für LF (Line Feed oder Zeilenvorschub) ein Pfeil nach unten. Probieren Sie einmal
PRINT CHR$(1)CHR$(10)
SOH bezieht sich nur auf den nachfolgenden Code. Sie müssen also für jedes auszugebende Zeichen ein neues SOH einschieben.

Auch das Steuerzeichen mit der Nummer 2 hat einen sehr interessanten Effekt. Mit STX (so der Name dafür) läßt sich der *Cursor ausschalten*. Das funktioniert aber nur während des Programmlaufs. Arbeitet das BASIC im Direktmodus, so schaltet es nach jeder Ausgabe von "Ready" den Cursor wieder ein. So wie unten funktioniert es aber:

```
10 PRINT CHR$(2)
20 INPUT "TEXT"; A$
30 PRINT CHR$(3)
```

Wie Sie sehen, erscheint beim INPUT-Befehl nicht mehr der gewohnte Cursor.
Der PRINT-Befehl in Zeile 30 bewirkt genau das Gegenteil; hiermit wird der Cursor wieder eingeschaltet (was wir uns im Beispiel eigentlich sparen könnten, weil das nach dem Programmende automatisch geschieht).

Das Steuerzeichen Nummer 7 kennen Sie schon aus dem Handbuch, deshalb gehe ich auf diesen piepsenden Zeitgenossen nicht näher ein.

Kommen wir nun zu den Zeichen, die den *Cursor* auf dem Bildschirm *bewegen*. CHR$(8) wirkt wie die Taste mit dem Linkspfeil, CHR$(9) bewegt den Cursor um eine Stelle nach rechts, CHR$(10) nach unten und CHR$(11) nach oben. Das kann für Fälle sehr nützlich sein, in denen Sie potenzierte oder indizierte Werte ausgeben müssen. Mit den beschriebenen Codes können Sie den Cursor in eine andere Zeile bewegen, dort die Potenz oder den Index drucken lassen und kehren dann auf die gleiche Weise wieder in die alte Zeile zurück.
Auf den ersten Blick überflüssig erscheint Steuerzeichen 12. Damit kann der *Bildschirm* wie durch CLS *gelöscht* werden.

Ganz neue Möglichkeiten tun sich jedoch auf, wenn man eines oder mehrere Steuerzeichen in normale BASIC-Strings einbaut. Probieren Sie einmal dieses:

```
A$= CHR$(12) + "UEBERSCHRIFT"
```

Jedesmal, wenn Sie jetzt den A$ ausgeben lassen (einfach durch PRINT) so wird zunächst der Bildschirm gelöscht, dann erst erscheint der Text. Grundsätzlich lassen sich alle Steuerzeichen auf die oben gezeigte Art in Strings einbauen.

Ein weiteres Steuerzeichen für den Cursor ist CR (CHR$(13)). Damit kann der Cursor an den Anfang der Zeile zurückgesetzt werden. CR bedeutet *Carriage Return*, was auf deutsch nichts anderes als Wagenrücklauf heißt. Diese Bezeichnung stammt noch aus der Zeit, als Computer über Fernschreiber als Terminals bedient wurden. Mit CR konnte der Rechner den Schreibkopf des Fernschreibers an den linken Rand der Zeile bewegen.
Auch die Funktion von CHR$(16) ist Ihnen bereits bekannt. Damit können Sie die CLR-Taste vom BASIC aus simulieren und das *Zeichen unter dem Cursor löschen*. Probieren Sie es selbst:

```
10 CLS
20 LOCATE 20,10
30 PRINT "LUECKENTEXT";:
40 WHILE INKEY$ = "": WEND
50 PRINT CHR$(8)CHR$(8)CHR($8)CHR$(8)CHR$(8)CHR$(16)
```

Das Programm braucht wohl nicht erklärt zu werden, starten Sie es einfach, drücken Sie eine beliebige Taste, und sehen Sie, was die Steuerzeichen aus Zeile 40 bewirken.
Auf den ersten Blick mag das nicht sehr sinnvoll erscheinen, doch bleibt es Ihnen überlassen, von den neuen Möglichkeiten Gebrauch zu machen und "vernünftige" Anwendungen zu finden.

Zum *Löschen von ganzen Bildschirmteilen* gedacht sind die Steuerzeichen 17 bis 20. Ersetzen Sie, um ein Anschauungsstück zu haben, in Zeile 40 das letzte

Steuerzeichen durch CHR$(17). Wenn Sie das Programm jetzt starten, werden alle Zeichen vom Zeilenanfang bis zur Cursorposition gelöscht, was sich durch das Verschwinden des Wortes LUECKEN bemerkbar macht. Ersetzen Sie das CHR$(17) durch CHR$(18), so geht es genau umgekehrt. Jetzt werden die Zeichen NTEXT und der Rest der Zeile (die bei uns leer war) gelöscht.
Die Steuercodes 19 und 20 wirken ähnlich, nur löscht Steuerzeichen 19 alle Zeichen vom Beginn des Fensters bis zur Cursorposition. CHR$(20) löscht alles vom Cursor bis zum Ende des Fensters (rechte untere Ecke). Dies können Sie leicht testen, indem Sie in unserem Beispielprogramm weitere PRINT-Befehle einfügen, die auch andere Zeilen beschreiben.

Nun aber weg von der Cursorsteuerung. Das nächste Steuerzeichen stellt ein ganz besonderes Bonbon dar. Mit PRINT CHR$(21) können Sie den *Textbildschirm abschalten*, was nichts anderes bedeutet, als daß alle Bildschirmausgaben, die nicht von Grafikbefehlen stammen, unterdrückt werden. Das läßt sich für Passwordeingaben benutzen, wie in untenstehendem Programm:

```
10 PRINT "Bitte Kennwort eingeben!"CHR$(21)
20 INPUT A$
30 IF A$ = "KENNWORT" THEN GOTO 1000 ELSE NEW
1000 PRINT CHR$(6)
...
```

Nach der Ausführung der Zeile 10 werden die Textausgaben unterdrückt, d.h. auch die Zeichen, die Sie über die Tastatur eingeben (siehe Zeile 20) erscheinen nicht auf dem Bildschirm, also kann auch niemand unbefugt Ihre Kennworte lesen. Ansonsten funktioniert der INPUT-Befehl ganz normal. Zeile 1000 hebt die Wirkung des Steuerzeichens 21 auf.

Auch das Zeichen SYN (Code 22) hat einen interessanten Effekt. Folgt ihm eine 1, so wird der *Transparentmodus* eingeschaltet. Im Transparentmodus wird die Bildschirmstelle, auf die ein Zeichen ausgegeben werden soll, nicht wie üblich vorher gelöscht. Das alte Zeichen

bleibt stehen, das neue wird einfach darüber geschrieben. Das können Sie zum Beispiel zum Unterstreichen am Bildschirm benutzen (was bei Computern nur in den seltensten Fällen möglich ist), indem Sie nach dem Drucken des Textes den Cursor zurückbewegen, den Transparentmodus durch "PRINT CHR$(22)CHR$(1);" einschalten und dann Unterstreichungszeichen PRINTen (erreichbar durch SHIFT und 8). PRINT CHR$(22)CHR$(0) schaltet wieder zurück.

Sehr nützlich ist auch Steuerzeichen 24. Es tauscht einfach PAPER- und PEN-Farbe aus. Das Ergebnis ist die *inverse Darstellung* von Zeichen, d.h. Buchstaben, die bisher gelb auf blauem Grund (oder hell auf dunkel) ausgegeben wurden, erscheinen jetzt blau auf gelbem Hintergrund (dunkel auf hell), was unseren üblichen Gepflogenheiten mehr entspricht (oder schreiben Sie mit weißem Stift auf schwarzem Papier?).

Das letzte Steuerzeichen ist wieder für die Cursorsteuerung zuständig. Mit PRINT CHR$(30) wird der Cursor in die linke obere Ecke des Bildschirms gesetzt. Diese Funktion nennt man auch *HOME*.

Ein kleiner Wermutstropfen ergibt sich allerdings auch bei der Anwendung der Steuerzeichen. Wird der PRINT-Befehl durch TAG auf den Grafikcursor umgeleitet, so werden nur noch Grafikzeichen ausgegeben, die Funktionen der Codes sind bis TAGOFF abgeschaltet.

Zusammenfassung: CHR$-Codes

Code(s)	Funktion
1	Drucken von Steuerzeichen
2/3	Textcursor aus/ein
7	Beep
8	Cursor ein Zeichen zurück
9	" " " vor
10	" eine Zeile abwärts
11	" " " aufwärts
12	Bildschirm löschen

```
13      Cursor auf Zeilenanfang
16      wie CLR-Taste
17      Zeile bis Cursor löschen
18        "   ab    "       "
19      Schirm bis Cursor löschen
20        "   ab    "       "
21/6    Textschirm aus/ein
22      Transparentmodus ein/aus
24      Inverse
30      HOME
```

4.2. VIDEO-RAM "VON INNEN"

Was wissen Sie über das Video-RAM? Bis jetzt noch nicht sehr viel, vermute ich. Das soll sich mit den folgenden Zeilen ändern. Aus den vorhergegangenen Kapiteln wissen Sie bereits, daß die Bytes im Video-RAM die Punkte auf dem Bildschirm repräsentieren. Als erste Frage stellt sich hier: "Wie sind die Punkte im Speicher angeordnet?" Um dies beantworten zu können, machen wir ein kleines Experiment. Vorher sollten Sie den CPC einmal kurz aus- und wieder einschalten, damit alle internen Daten unseren Wünschen entsprechen. Dann geben Sie bitte ein:

```
MODE 2
FOR I = 49152 TO 65535: POKE I,255: NEXT
```

Mit dieser Befehlsfolge werden alle Bits des Video-RAMs auf 1 gesetzt und damit alle Punkte des Bildschirms zum Leuchten gebracht. Nach dem Druck auf die RETURN-Taste sehen Sie, wie zunächst die obersten Punkte aller Textzeilen gesetzt werden, dann alle Punkte darunter usw. bis der ganze Bildschirm eine Fläche darstellt.
Denken Sie sich nun alle 25 Textzeilen zu einer einzigen aneinandergelegt, so erhalten Sie insgesamt 8 Punktzeilen (da jedes Zeichen im Modus 2 aus 8 x 8 Punkten besteht). Teilen wir 16 Kilobytes (so groß ist der Bildschirmspeicher)

durch 8, dann ergibt das 2 K oder 2048 Bytes für jede Zeile. Legen Sie alle 8 Punktzeilen aneinander, haben Sie die interne Organisation des Bildschirms im Speicher vor sich. Damit ist aber noch nicht klar, wie ein einzelner Punkt dargestellt wird.
Wenn - wie im Modus 2 - nur zwei Farben (Punkt leuchtet oder bleibt dunkel) unterschieden werden müssen, reicht ein Bit pro Punkt (0 für dunkel, 1 für hell). Bei 640 x 200 Punkten sind das genau 128000 Bits oder 16000 Bytes.
Denn 16 Kilobytes sind 16384 Bytes, also 384 mehr, als unser Bildschirm eigentlich benötigt. Dafür gibt es zwei Erklärungen. Zum einen ist es für den Prozessor und den TV-Chip einfacher, mit vollen Kilobyte statt mit Zwischenwerten zu rechnen (Sie benutzen ja auch lieber ganze als gebrochene Zahlen). Außerdem braucht der Prozessor für einige Operationen wie horizontales Scrolling (darauf kommen wir noch später) Reserve-Bytes. Daher sind am Ende jedes 2 K-Blocks 48 Bytes unbenutzt.

Im Modus 1 haben Sie 4 Farben zur Verfügung, Deshalb werden pro Punkt zwei Bits gebraucht. Deren mögliche Kombinationen sind 00, 01, 10, 11. Weil aber jetzt doppelt so viele Bits gebraucht werden, kann nur die Hälfte der Punkte dargestellt werden, diese dafür doppelt so breit. So werden auch für die 320 x 200 Punkte nur 16000 Bytes gebraucht.

Im Modus 0 gibt es 16 Farben. Um 16 verschiedene Punktsorten unterscheiden zu können, brauchen wir jeweils ein halbes Byte. Da wiederum die Anzahl der Bits verdoppelt wurde, muß auch die Punktanzahl halbiert werden. Es bleiben also "nur" 160 x 200 Punkte.

Aber auch jetzt ist das Geheimnis um das Video-RAM noch nicht ganz gelöst. Im Modus 2 ist es ganz klar, welche Bits für welche Punkte stehen. Das erste Byte des Video-RAMs bestimmt das Aussehen der ersten 8 Punkte. Mit POKE 49152,x können Sie ein beliebiges Bitmuster einspeichern, das auch genauso auf dem Bildschirm erscheint (solange der CPC im Modus 2 ist). Versuchen Sie es mit den Werten 1, 2, 4, 8,

16, 32, 64, 128, 240, 15, 170 und halten Sie sich dabei jeweils die Binärdarstellung dieser Zahlen vor Augen. Sie werden feststellen, daß die Punkte genau der Binärzahl entsprechen.

Im Modus 1 braucht jeder Punkt zwei Bits. Es wäre das naheliegendste, wenn jeweils benachbarte Bits die Farbe für einen Punkt ergäben. Doch leider ist es nicht so. Zunächst unterscheidet der Chip, der das TV-Signal erzeugt, die linke und rechte Hälfte des Bytes im Video-RAM (ein halbes Byte heißt auch Nibble). Schreiben Sie jetzt das rechte Nibble unter das linke, so wie im Beispiel (hier sind nur die Bitnummern aufgeführt, Sie sollten die Werte der einzelnen Bits notieren):

```
7 6 5 4
3 2 1 0
```

Jeweils untereinander liegende Bits ergeben die Farbe für einen Punkt an. Mit dem Byte 11110000 hätten alle 4 Punkte die Farbe 01 (es wird von unten nach oben gelesen), bei 11111010 hätten der erste und der dritte Punkt die Farbe 3 (binär 11), der zweite und vierte dagegen Farbe 1. Prüfen Sie auch dies wieder mit POKE-Befehlen nach.

Auch Modus 0 wird ähnlich organisiert. Schreiben Sie diesmal die Bits wie im nächsten Beispiel untereinander:

```
7 6
3 2
5 4
1 0
```

Wie Sie wissen, stellt ein Byte in diesem Modus zwei Punkte dar. Lesen Sie die Bitkombinationen wieder von unten nach oben, und Sie erhalten die Farbnummer des Punktes. Welche Farben dann erscheinen, können Sie der Tabelle in Ihrem Handbuch entnehmen (sofern Sie die Farben nicht mittels INK-Befehl geändert haben).

4.3. GRAFIK IM VERBORGENEN

Bei der Programmierung von Grafiken, Texten und ähnlichem möchte man oft das Bild erst komplett aufbauen, bevor es sichtbar wird. Oder Sie möchten zwei völlig unabhängige Grafiken darstellen. Dazu gibt es zwei Möglichkeiten. Wenn die Darstellungen sich nicht überschneiden, so genügt es, eine von beiden in dunkler Farbe auszugeben (mit INK 2,0) und erst, wenn alles komplett aufgebaut ist, die Farbe auf hell zu schalten (z.B. INK 2,24). Gleichzeitig sollte dann das andere Bild in gleicher Weise abgedunkelt werden.

Dieses Verfahren funktioniert aber nur, wenn sich beide Bilder nicht überschneiden, und wenn sich der CPC in einem anderen Modus als 2 befindet (im Modus 2 können Sie keine Reservefarbe zum Umschalten einsetzen).

Einfacher kann es sein, den CPC anzuweisen, einen *zweiten Bildschirmspeicher* einzusetzen. Dieses Verfahren kostet uns allerdings sehr viel BASIC-Speicher. Der Bildschirmspeicher darf logischerweise nur in solche Speicherbereiche verlegt werden, die nicht schon anderweitig vom Betriebssystem benutzt werden. Außerdem kann das Video-RAM nur in Schritten von 16 K verschoben werden, also an die Anfangsadressen 0, 16384, 32768, und 49152. Letztere Möglichkeit ist beim Einschalten des Rechners schon gewählt worden. In den Bereichen von 0 - 16383 und 32768 bis 49151 haben sowohl Betriebssystem als auch BASIC-Interpreter wichtige Daten gespeichert. Es bleibt also nur der Bereich ab 16384.
Leider heißt das, daß wir den BASIC-Speicher mittels MEMORY 16383 auf weniger als 16 K begrenzen müssen. Diese Methode eignet sich deshalb nur für verhältnismäßig kleine Programme.

Mit CALL &BC06,&40 kann der Bildschirmspeicher nach 16384 verlegt werden,
CALL &BC06,&C0 bringt den Ausgangszustand zurück.

Alle Grafik- und sonstigen Ausgabebefehle beziehen sich

immer nur auf den Bildschirmspeicher, der gerade eingeschaltet ist, Sie können also ganz normal arbeiten.
Aber auch das läßt sich ändern. In einigen Fällen kann es notwendig werden, im noch verborgenen Bild zu arbeiten, während das andere sichtbar bleibt. Glücklicherweise gibt es eine Speicherzelle im RAM, in der sich das Betriebssystem die Bildschirmadresse merkt. Soll ein Text oder ein Grafikpunkt ausgegeben werden, so hält sich der Rechner nicht an die wirkliche Adresse, sondern an den Wert aus der erwähnten Speicherzelle. So können wir dem Betriebssystem noch nicht existente Bildschirmspeicher vorgaukeln. Die Speicherzelle hat die Adresse &B1CB. Solange dort der gleiche Wert steht, wie er auch dem CALL-Befehl angehängt wird, läuft die Bildschirmausgabe normal.
CALL &BC06, &40: POKE &B1CB, &C0 schaltet also das Video-RAM ab Adresse 16384 ein, die Ausgaben laufen aber weiterhin im alten Speicher ab.
Der POKE-Befehl kann natürlich auch alleine gegeben werden, wichtig ist, daß nur die Werte &40 oder &C0 gePOKEd werden, weil es sonst zum Aufhängen des Rechners kommen kann.

Leider müssen Sie noch auf eine Besonderheit des CPC Rücksicht nehmen. Wenn der Bildschirm gescrollt werden soll, wird nicht etwa der Inhalt des Video-RAM Byte für Byte im Speicher verschoben. Vielmehr ändert sich für den TV-Baustein der Anfang des Video-RAMs, es beginnt statt bei 49152 z.B. bei 53248 (um einen willkürlichen Wert zu nennen). Durch eine Adresslogik werden die "hinten herausfallenden" Bytes wieder bei 49152 angehängt, so daß sich für uns auf den ersten Blick nichts ändert. POKEn wir aber in den Bildschirm, so sehen Sie, daß die gewünschten Punkte nicht oben links, sondern irgendwo anders erscheinen. Für den Rechner (und damit auch für Sie) hat diese Methode den Vorteil, daß das *Scrolling* viel schneller abläuft.
Wegen dieser eigenwilligen Technik empfiehlt es sich, ganz zu Beginn des BASIC-Programms in jedem Bildschirmspeicher ein MODE-Kommando zu geben. Damit werden die Pointer wieder zurückgesetzt. Es reicht nicht, mit CLS oder CLG zu löschen. Im weiteren Verlauf Ihres Programms sollten Sie dann sorgfältig darauf achten, daß keine Zeilen gescrollt werden.

Zusammenfassung: Bildschirm verlegen

CALL &BC06,&40 schaltet Bereich 16384 bis 32767 als neues Video-RAM (vorher mit MEMORY schützen).
CALL &BC06,&C0 schaltet wieder zurück.
In der Speicherzelle &B1CB merkt sich das Betriebssystem, welches Video-RAM eingeschaltet ist.

4.4. BILDSCHIRME ABSPEICHERN

Wie Sie wissen, gibt es im CPC-BASIC einen speziellen Befehl zum SAVEn von verschiedenen Speicherbereichen. Damit läßt sich auch der Bildschirmspeicher auf Cassette sichern und wieder zurückholen. Die Befehle dafür lauten:

```
SAVE "!TV",B,49152,16384
LOAD "!TV",49152
```

Natürlich können Sie den Filenamen beliebig ändern, doch sollten Sie nie das Ausrufezeichen weglassen, weil sonst die Meldungen des Betriebssystems den Bildschirminhalt zerstören.

Es ist auch möglich, nur Teilbereiche abzuspeichern. Dazu muß für jede Punktzeile die Anfangs- und Endadresse im Speicher berechnet werden. Mit den Kenntnissen aus Kapitel 4.2. dürfte Ihnen das keine besonders großen Schwierigkeiten bereiten. Jede Zeile wird dann mit einem eigenen SAVE-Kommando auf das Band gebracht und kann per LOAD zurückgeholt werden. Das funktioniert natürlich nur, wenn noch nicht gescrollt wurde (das Problem kennen Sie ja bereits aus Kapitel 4.3.).

4.5. SCROLLING

Das Phänomen "Scrolling" hat jeder Besitzer eines Computers schon einmal bemerkt. Ist der Cursor am unteren Bildschirmrand angekommen, so wird der gesamte Bildschirminhalt flugs nach oben geschoben, um Platz für eine neue Zeile zu erhalten. Dabei gibt es gleich zwei Dinge, die für den BASIC-Programmierer interessant sein können. Einmal ist das die Art und Weise, wie das Scrolling erzeugt wird, zum anderen ist das Bildschirmrollen nicht nur auf die Aufwärts-Richtung beschränkt. Damit lassen sich zum Teil sehr interessante Effekte erzielen.

Analog zum Normalfall kann der bestehende Text nach unten gerollt werden, wenn in der obersten Bildschirmzeile das Steuerzeichen "Cursor nach oben" (CHR$(11)) ausgegeben wird. Danach können Sie die entstandene Zeile einfach mit PRINT füllen. Aber auch in vertikaler Richtung kann der gesamte Bildschirminhalt auf einen Schlag verschoben werden. Dazu sind nur zwei OUT-Befehle notwendig, mit denen der *Offset* für den Anfang des Video-RAMs geändert wird. Wie Sie aus den vorherigen Kapiteln bereits wissen, bewirkt ein Scrolling kein Umschichten des Bildschirmspeicherinhalts, sondern die Anfangsadresse wird einfach um die Länge einer Bildschirmzeile verschoben. Diese Verschiebung nennt man Offset. Glücklicherweise kann dieser Offset auch in kleineren Schritten als eine Zeile geändert werden. Die kleinstmögliche Änderung beträgt 2 Bytes, das entspricht je nach Bildschirmmodus einem halben, ganzen oder zwei Zeichen. So ist auch ein *seitliches Scrolling* machbar.

Der 6845-Videocontroller besitzt diverse Register, in denen dieser Offset gespeichert wird. In die Register kann durch OUT-Befehle geschrieben werden, ein Lesen durch INP ist nicht möglich. Die Befehle dazu lauten:

```
OUT &BC00,13: OUT &BD00, offset
```

Im Normalfall (also wenn noch kein Scrolling aufgetreten

ist) hat der Offset den Wert 0. Durch Vergrößern dieser Zahl wird nach links geschoben, verringern scrollt nach rechts. Wollen Sie von der Mitte aus nach links und rechts verschieben, so empfiehlt es sich, vor dem ersten PRINT-Befehl den Bildschirm zu löschen und dann mittels OUT &BC00,13: OUT &BD00,20 alles schon einmal in die Mitte zu scrollen.

Eine ganze Zeile weiterschieben können Sie durch eine Schleife wie die untenstehende:

```
OUT &BC00,13: FOR i= 0 TO 40: OUT &BD00,i: NEXT
```

Der erste OUT-Befehl dient nur zum Anwählen des richtigen 6845-Registers. Deshalb muß er innerhalb der Schleife nicht ständig wiederholt werden. Da aber auch das Betriebssystem Register im Video-Controller benutzt, kann es sein, daß die Registernummer nicht mehr stimmt. Sie kommen also nicht um das OUT &BC00,13 vor jedem Do-it-yourself-Scrollen herum!

Zusammenfassung: Scrolling

Scrolling nach unten kann durch Ausgabe von CHR$(11) in der obersten Bildschirmzeile erreicht werden.
Für seitliches Rollen wird die Befehlsfolge
OUT &BC00, 13: OUT &BD00, offset
benutzt, wobei offset die Byteanzahl geteilt durch 2 darstellt, um die verschoben werden soll.

4.6. "SCROLLING" EINMAL ANDERS

In den folgenden Zeilen werde ich Ihnen eine Eigenschaft des CPC beschreiben, die Besitzer anderer Computer schlichtweg als "Hammer" bezeichnen. Neben dem normalen Scrolling, bei dem einfach der Bildschirmspeicher verändert wird, gibt es auch noch eine andere Methode, bei der man den Monitor ohne

großes Aufhebens dazu bringen kann, das gesamte Bild (inklusive Rahmen!!!) in alle vier Himmelsrichtungen zu verschieben (an dieser Stelle ist ein ehrfurchtsvolles "Wow!" durchaus verständlich, die Konstrukteure haben eben ganze Arbeit geleistet).

Auch hier spielen wieder die 6845-Register eine gewichtige Rolle. Hier wird aber nicht der Bildschirmspeicher beeinflußt, sondern das *Video-Signal* für den Monitor. Dieses Video-Signal enthält neben der Bildinformation für den Monitor auch *Synchronisationssignale*, die das Ende einer Zeile bzw. des Bildes anzeigen. Wird dieses Signal verspätet abgeschickt, so kann der Monitor die Bildinformation nicht mehr an der richtigen Stelle auf den Schirm bringen. Das Ergebnis ist ein verschobenes Bild.

Der 6845 bietet die Möglichkeit, den Zeitpunkt der Synchronisationssignale über OUT-Befehle direkt zu bestimmen. Vor einer Verschiebung in horizontaler Richtung muß OUT &BC00,2 ausgeführt werden. Mit einem zweiten Befehl (OUT &BD00,x) wird der Synchronisationszeitpunkt eingestellt. Im Normalfall hat x den Wert 46. Sie können für x jede Zahl zwischen 0 und 63 einsetzen, bei größeren Werten meckert der Bursche und hängt sich kurzerhand auf. Aus diesem Schmollwinkel holt ihn nur noch sanftes Aus- und Einschalten wieder hervor.
Werden die Zahlen kleiner, so wird nach rechts geschoben, bei größeren entsprechend nach links. Die komplette Befehlsfolge für eine Verschiebung um eine Stelle nach links lautet also:

```
OUT &BC00,2: OUT &BD00,47
```

Genauso geht man bei der vertikalen Verschiebung vor. Hier lauten die Befehle:

```
OUT &BC00,7: OUT &BD00,x
```

X darf im Bereich von 0 bis 38 liegen, der Normalwert ist 30. Wird x kleiner, so verschiebt sich das Bild nach unten.

Mit diesen Tricks kann man natürlich sehr schöne Effekte erzielen, z.B. ist es möglich, bei einem Spiel "Strafpunkte" durch das Wegwandern des Bildes anzuzeigen. Natürlich lassen sich auch beide Verschiebungsmöglichkeiten miteinander kombinieren.

```
Zusammenfassung: Bildverschiebung

Horizontale Verschiebung:
   OUT &BC00,2: OUT &BD00,x
Vertikale Verschiebung:
   OUT &BC00,7: OUT &BD00,y
```

4.7. NOCH EINMAL: CURSORSTEUERUNG

Der INKEY$ hat gegenüber dem INPUT-Befehl den entscheidenden Nachteil, daß kein Cursor auf dem Bildschirm erscheint. Aber zum Glück kann der *Cursor* auch vom BASIC aus in jeder beliebigen Programmsituation *ein- und ausgeschaltet* werden. Dazu werden einfach zwei Betriebssystemroutinen aufgerufen. Hier ein Beispielprogramm, aus dem die Vorgehensweise ersichtlich wird:

```
10 CALL &BB81: REM Cursor einschalten
20 WHILE INKEY$="": WEND: REM auf Tastendruck warten
30 CALL &BB84: REM Cursor ausschalten
40 ... weiteres Programm
```

Das Ganze funktioniert nicht im Direktmodus, weil das Betriebssystem bei jeder Ausgabe von "Ready" wieder selbst die Kontrolle über den Cursor übernimmt.

```
Zusammenfassung: Cursor ein-/ ausschalten

Cursor sichtbar: CALL &BB81
Cursor unsichtbar: CALL &BB84
```

5. GRAFIK

Möchten Sie mehr als nur Linien und Punkte auf dem Bildschirm erscheinen lassen, so läßt Sie der CPC zunächst ziemlich allein. Der BASIC-Interpreter stellt in recht unüblicher Weise keine Funktionen für Kreise, Rechtecke u.ä. zur Verfügung. Zum Glück sind diese Funktionen durch sehr einfache Unterprogramme zu ersetzen.

5.1. DAS GRAFIKSTEUERZEICHEN

Im Kapitel 4.1. habe ich Ihnen bewußt ein Steuerzeichen vorenthalten, das mit der hochauflösenden Grafik zu tun hat. CHR$(23) schaltet die verschiedenen *Grafikfarbstiftmodi* ein. Im Normalfall (d.h. wenn Sie den CHR$(23) noch nicht benutzt haben), werden die Punkte aus Grafikbefehlen einfach am Bildschirm gesetzt, ganz egal, wie es dort vorher aussah. Der CPC beherrscht aber noch andere Techniken. Wenn Sie vor dem Setzen eines Punktes den Befehl PRINT CHR$(23) CHR$(1) gegeben haben, dann werden die neuen Punkte mit den alten XOR-verknüpft. Die XOR-Verknüpfung hat den interessanten Effekt, daß ihr Ergebnis nur dann 1 wird, wenn die beiden Eingangsbits verschieden waren. Mit anderen Worten: XOR setzt nur dann einen Punkt, wenn alter Bildschirmpunkt und neuer verschieden sind. Hatten Sie eine Linie vom Punkt (0,0) zum Punkt (100,100) gezogen, und tun Sie dies ein zweites Mal im *XOR-Modus*, so wird die Linie gelöscht, da die Punkte, aus denen sich die Linie zusammensetzt, und die Punkte, aus denen die zweite konstruiert wird, jeweils gleich sind. Nehmen wir für einen sichtbaren Punkt den Wert 1, so werden immer 1 und 1 XOR-verknüpft - das Ergebnis daraus ist 0, also werden die Punkte gelöscht. Soll dagegen eine Linie an Stellen gesetzt werden, wo vorher noch nichts stand, so werden die Punkte sichtbar, weil Ursprungswert und neuer Punkt verschieden sind.
Hierzu ein Beispielprogramm:

```
10 MODE 2: PRINT CHR$(23)CHR$(1): REM XOR-Modus
20 FOR I = 100 TO 200 STEP 2
30 MOVE 10,I: DRAW 100,I: NEXT: REM Kasten zeichnen
40 FOR J = 1 TO 2
50 WHILE INKEY$ = "": WEND
60 FOR I= 130 TO 180 STEP 2
70 MOVE 60,I: DRAW 150,I: NEXT I,J: REM zweiter Kasten
```

Als erstes werden Sie sich über den STEP 2 gewundert haben. Dieser Befehl ist wichtig, weil Sie in senkrechter Richtung nur 200 verschiedene Punkte erzeugen können, die Koordinaten dafür aber im Bereich von 0 bis 399 liegen. Daher erzeugen DRAW 100,100 und DRAW 100,101 die gleiche Linie. Gerade das ist aber im XOR-Modus zu vermeiden, weil zwei übereinander gesetzte Linien sich löschen.

Den Rest des Programms können Sie sich selbst plausibel machen. Es werden insgesamt drei Kästen gezeichnet, ein großer und zwei kleine, wobei der letzte seinen Vorgänger löscht. Sehen Sie es sich selbst an.

Es gibt noch zwei andere Modi, nämlich *AND* und OR. Beide funktionieren wie der XOR-Modus, es wird jedoch eine andere Verknüpfung gebraucht. Bei AND wird nur dann ein Punkt gesetzt, wenn auch vorher schon einer da war. Das läßt sich in den Mehrfarbenmodi ausnutzen, um Farben zu wechseln. Setzen Sie Punkte in einer anderen Farbe über alte Punkte, so erscheint diese neue Farbe nur dort, wo vorher andere Farben als 0 gesetzt waren. Dieser Modus läßt sich durch PRINT CHR$(23)CHR$(2) einschalten.

Der *OR-Modus* (PRINT CHR$(23)CHR$(3)) entspricht dem Transparentmodus für Texte. Neue Punkte werden einfach über die alten gesetzt. Auch wenn ein neuer Punkt die Farbe 0 (schwarz) hat, wird der alte nicht gelöscht!

```
Zusammenfassung: Grafikmodi

Normal: CHR$(23)CHR$(0)
XOR   : CHR$(23)CHR$(1)
AND   : CHR$(23)CHR$(2)
OR    : CHR$(23)CHR$(3)
```

5.2. KASTEN UND RECHTECK

In den folgenden Abschnitten möchte ich Ihnen eine Reihe von Unterprogrammen vorstellen, die Sie wie zusätzliche Grafikbefehle nutzen können. Wir beginnen mit den einfachsten Figuren: Kasten und Rechteck.

Ein *Kasten* soll nach unserer Definition einfach eine von 4 Linien in Rechteckform umschlossene Fläche sein. Wir müssen also nur diese vier Linien mittels DRAW zeichnen lassen, und schon ist der Kasten fertig. Dies leistet das folgende Unterprogramm:

```
65500 REM Kasten: x1,y1,x2,y2,f
65501 MOVE x1,y1: DRAW x1,y2,f
65502 DRAW x2,y2,f: DRAW x2,y1,f
65503 DRAW x1,y1,f: RETURN
```

Dieses und alle folgenden Unterprogramme sind so konstruiert, daß sie mit MERGE an das Ende Ihrer Programme angehängt werden können. Außerdem werden Variablen benutzt, um die erforderlichen Parameter zu übergeben.

Um das Aussehen des Kastens festzulegen, müssen nur die linke obere und die rechte untere Ecke angegeben werden. Diese Koordinaten sollen vor dem Unterprogrammaufruf in den Variablen X1 und Y1 für den linken oberen Punkt und in X2 und Y2 für den anderen Punkt stehen. Außerdem müssen Sie die Farbstiftnummer, die benutzt werden soll, in F festlegen.

Die erste Zeile unseres Unterprogramms setzt den Grafikcursor auf den Anfangspunkt unseres Kastens und zieht

die erste Linie. Mit den nächsten drei DRAW-Befehlen werden dann die restlichen Strecken gezeichnet. RETURN veranlaßt den Rücksprung ins Hauptprogramm.
Ein typischer Aufruf dieser Routine lautet:
x1= 100: y1= 200: x2= 200: y2= 100: f= 1: GOSUB 65500

Das nächste Programm soll ein *Rechteck* zeichnen, das sich vom Kasten nur darin unterscheidet, daß auch die Fläche zwischen den Linien ausgefüllt ist. Dazu setzen wir einfach viele Linien direkt untereinander, bis das Rechteck die gewünschte Größe hat. Hier das Listing:

```
65505 REM Rechteck: x1,y1,x2,y2,f
65506 FOR ii= y1 TO y2 STEP 2*SGN(y2-y1)
65507 MOVE x1,ii: DRAW x2,ii,f
65508 NEXT ii: RETURN
```

Auch hier müssen wieder nur zwei Punkte angegeben werden; dies geschieht genau wie beim Kasten. Im FOR-Befehl finden Sie auch wieder den STEP 2, der für den XOR-Modus wichtig ist.
Außerdem muß die Schrittweite mit -1 multipliziert werden, wenn y2 kleiner als y1 ist. Dies leistet SGN(y2-y1).
In Zeile 65507 wird zunächst der Grafikcursor an die linke Seite des Rechtecks zurückgefahren und dann eine Linie nach rechts gezeichnet. Mit jedem neuen Durchlauf der FOR-NEXT-Schleife wird diese Linie um einen Punkt nach unten versetzt, so daß nach und nach eine ausgefüllte Fläche entsteht.

5.3. ALLERLEI MIT SINUS UND COSINUS

Die nächsten beiden Routinen sind Ihnen bereits in etwas einfacherer Form aus dem CPC-Handbuch bekannt. Doch aus den auf die Dauer langweiligen Kreisen und Scheiben läßt sich viel mehr machen.

Bleiben wir zunächst noch bei den Kreisroutinen. In beiden Fällen muß der Mittelpunkt (x1,y1) und der Radius (r1) sowie die Farbe (f) angegeben werden. Nach dem Aufruf des Unterprogramms "umläuft" eine FOR-NEXT-Schleife den ganzen Umfang des *Kreises* (0 - 360 Grad) und berechnet jeweils mittels Sinus- und Cosinusfunktionen den Abstand des zu setzenden Punktes vom Mittelpunkt. Eine *Scheibe* erhält man, wenn nicht PLOTR verwendet wird, sondern jeweils eine Linie von der Mitte zum Rand gezogen wird.

Wird der Radius sehr groß gewählt, so kann es vorkommen, daß innerhalb der Scheibe einzelne Punkte nicht in der gewünschten Farbe aufleuchten, sondern dunkel bleiben. Dies läßt sich verhindern, wenn an das FOR-NEXT-Kommando ein STEP 0.5 angehängt wird (notfalls noch kleinere Schrittweite nehmen).
Hier die beiden Routinen:

```
65510 REM Kreis: x1,y1,r1,f
65511 DEG: r2= r1: FOR ii = 0 TO 359
65512 xx= COS(ii)*r1: yy= SIN(ii)*r2
65513 MOVE x1,y1: PLOTR xx,yy,f
65514 NEXT ii: RETURN

65515 REM Scheibe: x1,y1,r1,f
65516 DEG: r2= r1: FOR ii = 0 TO 359
65517 xx= COS(ii)*r1: yy= SIN(ii)*r2
65518 MOVE x1,y1: DRAWR xx,yy,f
65519 NEXT ii: RETURN
```

Typische Aufrufe sind:

```
x1= 320: y1= 200: r1= 100: f= 1: GOSUB 65510
```

```
x1= 100: y1= 100: r1= 30: f= 1: GOSUB 65515
```

Wußten Sie, daß es mit nur geringen Änderungen möglich ist, auch *Ellipsen* zu zeichnen? Bestimmt sind Ihnen schon die zwei Radiusvariablen in Zeile 65512 aufgefallen, die auf den ersten Blick unnötig erscheinen. Wenn R1 und R2 verschiedene Werte haben, so erhalten Sie eine Ellipse. Versuchen Sie es! R1 ist der Radius in X-Richtung, R2 gibt den Abstand in Y-Richtung an.
Um das ganze leichter handhaben zu können, fügen Sie bitte folgende Zeilen an:

```
65520 REM Ellipse: x1,y1,r1,r2,f
65521 DEG: FOR ii = 0 TO 359
65522 GOTO 65512
```

Durch das "GOTO 65512" in Zeile 65522 wird das Unterprogramm für den Kreis mitbenutzt, da es bis auf die ersten beiden Zeilen identisch ist. So kann Speicherplatz (und Tipparbeit) gespart werden.

Mit einer kleinen Abwandlung des Scheibenprogramms können wir *Sterne* mit beliebiger Strahlanzahl erzeugen. Im Grunde genommen ist auch die Scheibe eine Art Stern, da zu jedem der 360 Punkte auf dem Kreisrand ein Strahl vom Mittelpunkt gezogen wird (siehe Abb.). Weil die Strahlen aber so dicht beieinander liegen, erscheint es für uns wie eine Fläche. Wir müssen also nur noch die Anzahl der Linien reduzieren. Das können wir mit einem STEP-Kommando erreichen, es wird also z.B nur noch jeder 10. Strahl gezeichnet. Um die Strahlen gleichmäßig zu verteilen, wird die Zahl 360 durch deren Anzahl geteilt und das Ergebnis als STEP eingesetzt.

Auch in Zeile 65527 finden Sie wieder ein Spar-GOTO. Damit auch wirklich alle Strahlen erscheinen, muß die FOR-NEXT-Schleife bis 360 verlängert werden.

Auch für den Stern muß nur der Mittelpunkt, der Radius und die Farbe in den bekannten Variablen hinterlegt werden. Außerdem beherbergt die Variable EC die gewünschte Anzahl

Strahlen. Hier wieder das Listing und ein Beispielaufruf:

```
65525 REM Stern: x1,y1,r1,ec,f
65526 DEG: r2= r1: FOR ii = 0 TO 360 STEP 360/ec
65527 GOTO 65517

x1= 100: y1= 100: r1= 30: ec= 12: f= 1: GOSUB 65525
```

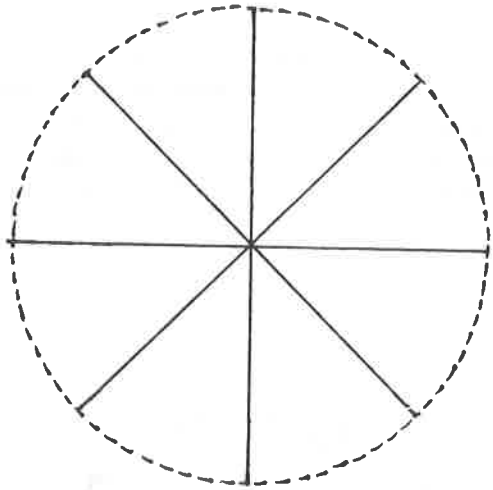


Abb. 2. Stern

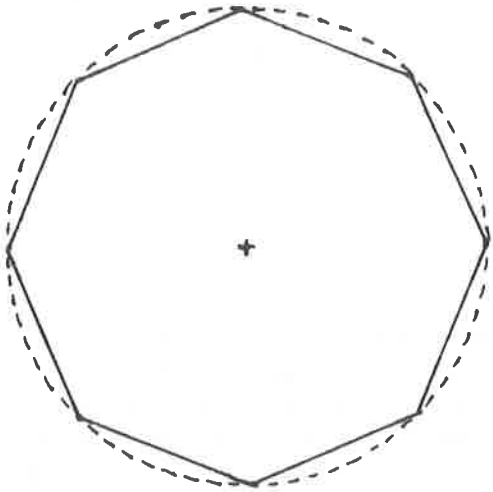
Das letzte Unterprogramm zeichnet beliebige Mehrecke (oder neudeutsch *Polygone*). Zum Funktionsprinzip sollten Sie sich noch einmal die Sterne aus dem letzten Unterprogramm ansehen. Verbinden Sie alle äußeren Strahlpunkte auf dem gedachten Kreisumfang durch Geraden, so erhalten Sie ein Mehreck. Dies macht sich unsere Routine zunutze. Wie bei den

Sternen werden in mehr oder minder großen Abständen Punkte auf dem Kreis berechnet und dann eine Linie zum letzten Punkt gezeichnet. Außerdem muß der Grafikcursor sozusagen "auf die Spur" gesetzt werden, weil sonst höchst undekorativ eine Linie vom letzten Cursorstandort zu Polygon gezogen würde. Auch hier muß die Schleife wieder verlängert werden, diesmal bis 370.

```
65530 REM Polygon: x1,y1,r1,ec,f
65531 DEG: MOVE x+r1,y: FOR ii = 0 TO 370 STEP 360/ec
65532 xx= COS(ii)*r1: yy= SIN(ii)*r1
65533 DRAW x1+xx,y1+yy,f: NEXT ii: RETURN
```

Wenn Sie möchten, können Sie auch "eirige" Sterne und Polygone erzeugen, indem einfach ein zweiter Radius eingeführt wird. Sie können auch Halbkreise oder andere Kreisausschnitte zeichnen, wenn die FOR-NEXT-Schleife z.B. "nur" von 180 bis 360 läuft. Ihrer Phantasie sind keine Grenzen gesetzt.

Abb. 3. Polygon



5.4. WOZU PIXELTEST?

Auf den ersten Blick erscheinen die Befehle *TEST* und *TESTR* relativ sinnlos. Wenn wir nur wissen möchten, welche Farbe ein Punkt hat, so brauchen wir doch nur auf den Bildschirm zu sehen. Es gibt aber auch Anwendungen, die ohne TEST nicht denkbar sind.

In diesem Zusammenhang fallen mir die Commodore-*Sprites* und die *Shapes* des Apple-Computers ein. Für den Anfänger, der in der Computerwelt noch nicht so sehr zuhause ist, sei erklärt, daß es sich in beiden Fällen um Figuren handelt, die der Programmierer wie einen Buchstaben vordefinieren kann und die dann mit einem einzigen Befehl auf den Bildschirm gebracht werden. So etwas ähnliches können wir auch für den CPC in BASIC programmieren.
Unser Programm soll zwei Funktionen beeinhalten. Zum einen sollen beliebige Bereiche des Bildschirms an andere Stellen kopiert werden. Das Prinzip dieses Unterprogramms ist recht einfach. Wie bei den Kasten- und Rechteckroutinen aus 5.1. werden zwei Punkte angegeben, die den zu kopierenden Bereich eingrenzen. Dann werden mittels zweier FOR-NEXT-Schleifen alle Punkte des Rechtecks "abgefahren" und das Ergebnis der TEST-Befehle als Farbstiftnummer für einen neu zu setzenden Punkt benutzt. War das TEST-Ergebnis z.B. 0, so wird ein Punkt mit der Farbe 0 gesetzt, was im Endeffekt bedeutet, daß ein dunkler Punkt erscheint. Auf diese Weise wird der ganze Bereich Pixel für Pixel kopiert.

Die zweite Routine soll eine Figur auf den Bildschirm bringen, die vorher noch nirgends zu sehen war, sie wird also nicht kopiert, sondern neu erstellt. Dazu muß natürlich irgendwo festgehalten sein, wie die Figur aussehen soll. Im BASIC-Programm geht das am besten mit DATA-Zeilen. Für jede Punktzeile, die unser Unterprogramm auf den Bildschirm bringen soll, muß ein String vorhanden sein, in dem die Farben der zu setzenden Pixel stehen. Die Länge der Strings gibt jeweils an, wieviele Punkte in einer Zeile gesetzt werden sollen. Für jeden Punkt ist ein Zeichen im String

abgelegt. Dabei soll das Zeichen eine Hexziffer darstellen, die für den Farbstift mit der gleichen Nummer steht. Außerdem muß die Routine noch wissen, wann die Figur fertig ist. Dazu wird einfach ein weiterer String mit der Länge 0 angehängt. Trifft das Unterprogramm auf diesen Leerstring, so wird die Ausführung beendet.
Schließlich brauchen wir nur noch die Position, an die die Figur gesetzt werden soll. Die notwendigen Koordinaten werden einfach in Variablen übergeben.

```
999 REM Bereiche kopieren: x1,y1,x2,y2,xs,ys
1000 FOR ii= y1 TO y2 STEP -2: FOR jj= x1 TO x2
1010 PLOT xs+jj-x1, ys+ii-y1, TEST (jj,ii)
1020 NEXT jj,ii: RETURN

1049 REM Bereich auf Bildschirm plotten: xs,ys
1050 zz= 0
1060 READ aa$: ll= LEN(aa$): IF ll=0 THEN RETURN
1070 FOR ii= 0 TO ll-1: aa= VAL("&"+MID$(aa$,ii,1))
1080 PLOT xs+ii*m, ys-zz*2,aa: NEXT ii: zz= zz+1: GOTO 1060
```

Das erste Unterprogramm braucht in x1, y1, x2 und y2 die Koordinaten des zu kopierenden Bereichs, xs und ys geben die linke obere Ecke des Zielbereichs an.

Bei der zweiten Routine müssen Sie nicht so viele Werte angeben. In xs und ys übergeben Sie die Koordinaten der linken oberen Ecke des Ziels auf dem Bildschirm. Im PLOT-Befehl sollten Sie sich die Verdopplung von zz genauer ansehen, diese Variable gibt die Nummer der gerade durchlaufenen Zeile an. Geschähe dies nicht, so würden zwei unserer DATA-Zeilen übereinandergeplottet (der CPC verdoppelt die Y-Koordinaten). Dies müssen Sie aber auch bei der Angabe der Höhe der Figur berücksichtigen. Also: Für jede zu plottende Punktzeile zwei Punkte in Y-Richtung weiterzählen. Ähnliches gilt für die X-Richtung. Hier wird mit der Variablen m multipliziert, die je nach Bildschirmmodus eine andere Zahl enthält. Bei Modus 0 muß m den Wert 4 enthalten, da hier 4 X-Koordinaten den gleichen Punkt ansprechen, bei Modus 1 ist es 2, für Modus 2 muß der Wert eine 1 sein.

Damit die Benutzung dieses Unterprogrammes klar wird, ist unten noch ein Beispielprogramm angegeben.

```
100 MODE 2
110 xs= 320: ys= 200: m= 1: GOSUB 1050: END
120 DATA "10000100001100010000010000011111"
130 DATA "10000100010010010000010000010001"
140 DATA "10000100100001010000010000010001"
150 DATA "10000100100001010000010000010001"
160 DATA "11111100111111010000010000010001"
170 DATA "11000100110001011000011000011001"
180 DATA "11000100110001011000011000011001"
190 DATA "11000100110001011000011000011001"
200 DATA "11000100110001011111101111101111"
210 DATA ""
```

Diese Zeilen müssen Sie in das obige Listing einfügen.
In den DATA-Zeilen ist ein Punktmuster für das Wort HALLO angegeben. Wie Sie leicht erkennen können, repräsentiert eine 1 einen gesetzten Punkt, 0 erzeugt ein leeres Pixel. Wenn Sie in anderen Modi arbeiten, können Sie auch mehrere Farben gleichzeitig benutzen, z.B. "f" für den Farbstift 15.

5.5. KOORDINATENSYSTEME

Dieser Abschnitt ist vor allem für die Mathematiker und Schüler unter Ihnen interessant. Erstere mögen mir verzeihen, wenn ich doch so einfache Dinge wie Koordinatensysteme noch einmal völlig unfachmännisch erkläre, letztere haben dagegen vielleicht Probleme mit dem noch unbekannten Thema.

Jedesmal, wenn Sie eine Funktion grafisch ausgeben möchten, müssen Sie die Funktionswerte in Bildschirmkoordinaten umrechnen. Dabei kann Ihnen das *ORIGIN*-Kommando einen Teil der Arbeit abnehmen. Lassen Sie uns das am Beispiel einer Sinuskurve durchgehen.

Wie Sie wissen, liegen die Funktionswerte beim Sinus zwischen -1 und +1. Daher sollte die X-Achse genau in der Mitte des Bildschirms liegen.

Die Y-Achse wird für trigonometrische Funktionen meist an den linken Rand gesetzt. Damit ergeben sich für den Nullpunkt unseres Koordinatensystems die Bildschirmkoordinaten (0,199). Geben wir jetzt ORIGIN 0,199 ein, so beziehen sich alle weiteren Grafikbefehle nur noch auf diesen neuen Nullpunkt. PLOT 100,-1 setzt also einen Punkt direkt unter die X-Achse.

Haben Sie einmal mit ORIGIN-Kommandos herumexperimentiert und die dazugehörigen Werte vergessen, so können Sie letztere durch folgende Zeile wieder auf den Bildschirm bringen:

```
PRINT UNT(PEEK(&B328)+256*PEEK(&B329))
```

Damit erhalten Sie den Wert für die X-Richtung, für die Y-Achse brauchen nur die Adressen in &B32A und &B32B geändert werden.

Würden Sie jetzt einfach die Werte aus der Sinusfunktion in

die Grafikbefehle einsetzen (z.B. per PLOT x,SIN(x)), so wäre Ihre Sinuskurve nur 1 Pixel hoch. Deshalb sieht der richtige Befehl so aus:

```
PLOT x,SIN(x)*199
```

Selbstverständlich können Sie auch die X-Koordinaten mit einem solchen Faktor multiplizieren, wenn dieser Bereich nicht mit den Grafikkoordinaten übereinstimmt.

Unten finden Sie ein Unterprogramm, das anhand Ihrer Angaben zum X- und Y-Bereich ein *Koordinatenkreuz* auf den Bildschirm zeichnet.

```
1000 INPUT "X-Minimum"; XA
1010 INPUT "X-Maximum"; XE
1020 INPUT "Y-Minimum"; YA
1030 INPUT "Y-Maximum"; YE
1040 CLG
1050 MX= (XE-XA)/639: X= -XA/MX
1060 MY= (YE-YA)/399: Y= -YA/MY
1070 ORIGIN X,Y
1080 MOVE XA/MX,0: DRAW XE/MX,0: REM Y-Achse
1090 MOVE 0,YA/MY: DRAW 0,YE/MY: REM X-Achse
1100 RETURN
```

Die Aufgabe der ersten 5 Zeilen dürfte klar sein; deshalb gehe ich darauf nicht näher ein.
In den nächsten beiden Zeilen werden die Maßstäbe (MX,MY) für X- und Y-Achse sowie die Koordinaten für den Nullpunkt berechnet (X,Y).
Der Maßstab errechnet sich aus der Differenz von Minimum und Maximum des jeweiligen Bereichs geteilt durch die Anzahl der möglichen Punkte. Die Differenz gibt an, wieviele Punkte gewünscht werden; diese werden durch die Division auf die möglichen Punkte projeziert.
Immer, wenn aus einem Funktionswert oder einem X-Wert eine Bildschirmkoordinate berechnet werden soll, muß nur durch den zugehörigen Maßstab geteilt werden. So geschieht dies auch bei der Berechnung der neuen Koordinaten des

Nullpunkts.

Zeile 1080 zieht dann noch die Y-Achse, Zeile 1090 ist für die X-Achse zuständig. Auch hier finden Sie wieder den Maßstab.

Sie können dieses Programm noch verfeinern, indem Sie mittels TAG und PRINT Bezeichnungen und Achsenbeschriftungen ausgeben. Dazu müssen natürlich auch noch die Einteilungsstriche gePLOTtet werden.

5.6. 3D-GRAFIKEN

Darstellungen dreidimensionaler Funktionen sind zweifellos die interessanteste Art der Grafiken, leider sind sie auch die aufwendigsten. Trotzdem möchte ich Ihnen mit einem kleinen Programm auch diese Technik näherbringen.

Wenden wir uns zunächst dem *Koordinatensystem* zu. Im Gegensatz zur normalen Funktionsdarstellung haben wir 3 Achsen: X, Y, Z.
Die Z-Achse muß notwendigerweise schräg in den Raum hineinlaufen (siehe Bild), den Winkel (und damit die Perspektive) können wir selbst bestimmen.

Da wir im PLOT-Kommando keine 3 Koordinaten angeben können, müssen diese auf 2 Dimensionen umgerechnet werden. Das nennt man eine *Projektion*; wir beschäftigen uns mit der einfachen parallelen Projektion, bei der es keinen Fluchtpunkt gibt. In der Wirklichkeit parallel verlaufende Linien werden dabei auch -wie der Name schon sagt- parallel projeziert und laufen nicht in der Ferne zusammen.

Wie Sie im Bild sehen, muß ein Punkt je nach z-Koordinate mehr oder weniger weit nach rechts oben verschoben werden. Auf der z-Achse sind zwei Punkte mit den Koordinaten (0,0,za) und (0,0,zb) eingezeichnet. Wie Sie sehen, können

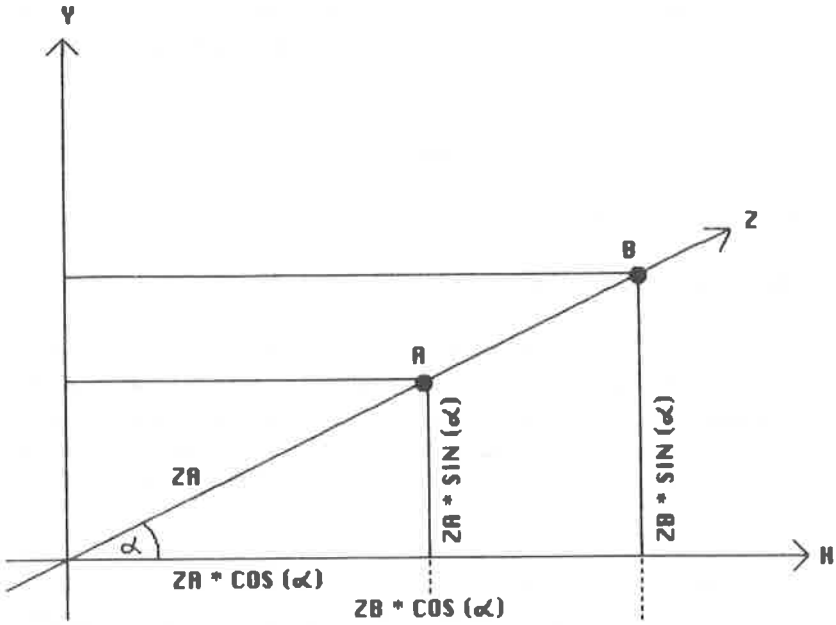


Abb. 4. Koordinatensystem für 3D-Grafiken

die Projektionskoordinaten durch Addition von za*cos(a) und zb*sin(a) berechnet werden. Die Projektionsformeln lauten also:

x= 0 + za*cos(a) y= 0 + za*sin(a)

oder in der allgemeinen Form:

x= xa + za*cos(a) y= ya + za*sin(a)

Weil diese 2D-Koordinaten aber nicht immer mit den Bildschirmkoordinaten übereinstimmen, führen wir noch Streckungsfaktoren (SX, SY) ein (siehe Programmlisting, Zeile 100). Der Koordinatenursprung wird durch ORIGIN 320,200 verlegt, damit das Bild in der Mitte des Schirms erscheint.

Da von den drei Originalkoordinaten nur eine berechnet wird und die anderen vorgegeben sind, brauchen wir im Programm zwei verschachtelte FOR-NEXT-Schleifen. Außerdem empfiehlt es sich aus Gründen, die ich weiter unten noch erläutere, die Funktion von hinten nach vorne zu PLOTten.

Im untenstehenden Programm werden alle Werte, die beim Experimentieren oft verändert werden müssen, über INPUT-Befehle eingelesen. Außerdem arbeitet das Programm im Modus 2, da es nicht auf möglichst viele Farben sondern auf höchstmögliche Auflösung ankommt. Tippen Sie das Programm ein und starten Sie es, wobei Sie folgende Werte eingeben: w= 45, sx= 0.5, sy= 0.5, xs= 2, zs= 20

```
10 INPUT "Winkel";w: DEG: z1=cos(w): z2=sin(w)
20 INPUT "Streckung-x";sx
30 INPUT "Streckung-y";sy
40 INPUT "Schrittweite-x";xs
50 INPUT "Schrittweite-z";zs
60 MODE 2: ORIGIN 320,200
70 FOR z= 180 TO -180 STEP -zs
80 FOR x= -180 TO 180 STEP xs
90 y= sin(x)*cos(z)*100: REM Funktionsterm
```

```
100 bx= sx*(x+z*z1): by= sy*(y+z*z2)
110 PLOT bx,by,1
120 NEXT x,z
```

Vielleicht ist Ihnen beim Programmlauf schon etwas aufgefallen. Es gibt es einige Punkte, die eigentlich von optisch davorliegenden Bildteilen verdeckt sein sollten. Dieses Problem heißt *Hinterschneidung*. Es läßt sich lösen, wenn alle Punkte, die unter dem gerade gesetzten liegen, gelöscht werden. Ändern Sie deshalb Zeile 110 in

```
110 PLOT bx,by,1: MOVE bx,by-2: DRAW bx,-200,0
```

Lassen Sie mich das an einem Beispiel erklären. Gehen wir davon aus, daß unsere Funktion eine einzige völlig ebene Fläche ergibt. Dann verläuft diese Fläche schräg von unten nach oben in den Raum hinein. Der vorderste Punkt ist demnach der am tiefsten liegende, er kann deshalb keine anderen verdecken. Nehmen wir jetzt an, die y-Koordinaten der vordersten Punktzeile werden um 1 erhöht. Dann verdeckt sie genau die darüberliegende Zeile. Erhöhen wir jetzt die Koordinaten noch weiter, so werden weitere Zeilen verdeckt. Daraus folgt, daß alle Punkte unter der jeweils untersten Zeile gelöscht werden müssen. Das gilt auch dann, wenn die unterste Zeile eine Kurve o.ä. darstellen soll. Am besten ist es, wenn Sie sich dieses Löschen beim Entstehen der Grafik genau ansehen, so werden Sie es sofort verstehen.

Ein paar Erläuterungen noch zu den verschiedenen Programmzeilen. Zeile 10 berechnet die Cosinus- und Sinuswerte des Winkels w. Da diese beiden Werte immer gleich bleiben, können Sie einfach als Konstanten eingesetzt werden. Das spart viel Rechenzeit. Die Zahlen in den FOR-NEXT-Schleifen (Zeilen 70 und 80) können Sie verändern, um größere oder kleinere Ausschnitte aus der Funktion darzustellen.
In Zeile 90 ist Ihnen wahrscheinlich das "*100" aufgefallen. Da die Funktionswerte immer kleiner als 1 bleiben, müssen diese Werte für sich auch noch einmal gestreckt werden.

Probieren Sie es jetzt mit verschiedenen Schrittweiten, Streckungsfaktoren und Funktionen. Viel Spaß!

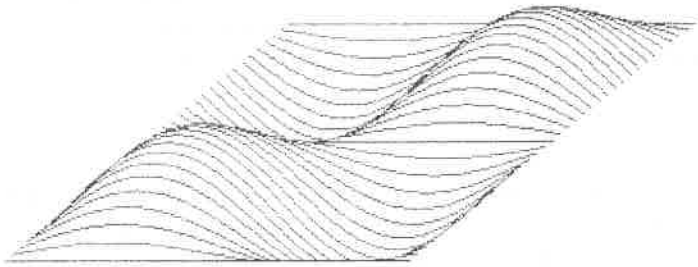


Abb. 5. y = sin(x) * sin(z) * 140

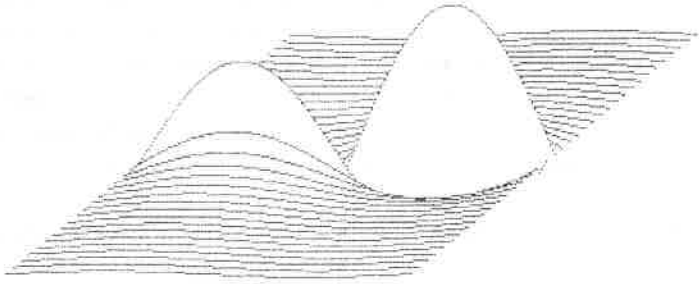


Abb. 6. y = sin(x) * 2 / (z / 8 + 0.5) * 80

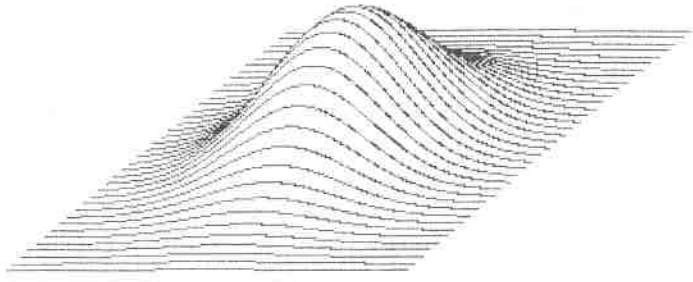


Abb. 7.

```
y = exp(-(x*x+z*z)/2)
    /sqr(2*π)*300
```

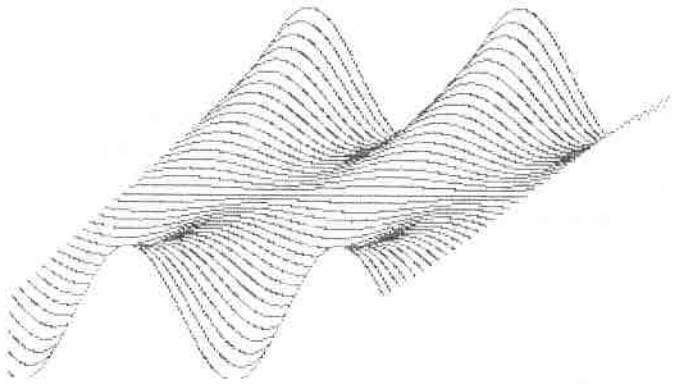


Abb. 8.

```
y = sin(x/30)*(exp(z/90)
    -1/exp(z/90))*10
```

6. GRAFIKANWENDUNGEN

Mit den Hilfsmitteln aus dem letzten Kapitel lassen sich schon sehr hübsche Grafiken aufbauen. Anwendungen mit echtem Nutzwert waren aber noch nicht dabei. Das möchte ich jetzt ändern. Wenn Sie zu den Anwendern gehören, die den CPC auch beruflich einsetzen wollen, wird Sie Kapitel 6.1. besonders interessieren. Für den Hobbyanwender und Künstler ist das Malprogramm aus Kapitel 6.2. gedacht.

6.1. DIVERSE DIAGRAMME

Wohl fast jeder Computerbesitzer hat sich beim Betrachten der vielfältigen Hochrechnungen am Wahlabend schon gewünscht, auch solche Grafiken zu erzeugen. Oder möchten Sie Verkaufszahlen und ähnliche Werte in *Balkendiagrammen* darstellen? Das läßt sich auch auf dem CPC sehr leicht machen. Im Grunde genommen ist ein Balken nichts anderes als ein Rechteck, das wir schon in 5.1. programmiert haben. Wir müssen also nur noch unsere Werte in Koordinaten umsetzen. Die Umrechnungen der tatsächlichen Werte in Bildschirmkoordinaten habe ich schon in Kapitel 5.5. erklärt; ich werde diese Methode auch hier wieder anwenden.

Für den X-Bereich brauchen Sie nicht mehr Minimum und Maximum angeben, sondern nur noch die Anzahl der auszugebenden Balken in XE. Diese Zahl wird mit 10 multipliziert, um die Punkte für die Balkenbreite zu reservieren (ein Balken soll ja breiter als 1 Pixel sein; siehe Zeile 60040).

Im Array W (0...XE) müssen die einzelnen Werte stehen. Das Maximum dieser Zahlen wird in der ersten FOR-NEXT-Schleife berechnet. Da die Numerierung von Array-Elementen bei 0 anfängt, wir Menschenkinder aber gemeinhin bei 1 zu zählen beginnen, muß von der Anzahl XE 1 abgezogen werden.

Bei den Maßstab-Formeln fallen Ihnen sicher die verringerten Punktkonstanten auf und die dadurch verkleinerte Fläche für unser Diagramm. Diese sind nötig, um ein wenig Platz für Beschriftungen zu lassen.

Da wir davon ausgehen, daß Verkaufswerte u.ä. nie negativ werden, können wir auf die Minima in beiden Bereichen (XA und YA) verzichten.

Zeile 60040 löscht den Bildschirm und setzt den Koordinatenursprung auf den Punkt (50,30). Damit bleibt links und unterhalb der Balken Platz für Beschriftungen.In Zeile 60050 wird die Begrenzung des Balkenfeldes gezeichnet; das funktioniert fast genau wie beim Koordinatenkreuz.
In den Zeilen 60060 und 60070 finden Sie das Rechteckprogramm in etwas abgewandelter Form. Da wir senkrechte Balken zeichnen möchten, ist es vorteilhafter und schneller, auch die Linien für die Rechtecke senkrecht zu DRAWen.
Außerdem habe ich einige Variablen direkt durch arithmetische Ausdrücke ersetzt. Dazu ist noch zu sagen, daß die Konstante 10 die Anzahl von Maßstabseinheiten angibt, die für einen Balken plus Zwischenraum zur Verfügung stehen. Die tatsächliche Balkenbreite ist durch "+8" in Zeile 60070 (wiederum in Maßstabseinheiten, nicht als Pixelzahl) angegeben. Wenn Sie größere oder kleinere Zwischenräume möchten, brauchen Sie nur diesen Wert zu verändern.

```
60000 REM Balkendiagramme: xe, w(0...xe-1),f
60010 xe= xe-1: ye= 0
60020 FOR ii= 0 TO xe: ye= MAX(ye,w(ii)): NEXT
60030 mx= (xe+1)*10/584: my= ze/364
60040 CLG: ORIGIN 50,30
60050 MOVE -5,0: DRAW -5, ye/my+5: MOVE -5,0: DRAW
(xe+1)*10/mx+5, 0
60060 FOR ii= 0 to xe: for jj= ii*10/mx TO (ii*10+8)/mx
60070 MOVE jj,0: DRAW jj, w(ii)/my, f: NEXT jj, ii
60080 RETURN
```

Die zweite Diagrammart erscheint viel komplizierter, doch auch hier können wir wieder auf alte Unterprogramme zurückgreifen und sie abwandeln. Gemeint sind die *Tortendiagramme*, bei denen den darzustellenden Werten ein mehr oder minder großes Kreissegment zugewiesen wird.

Die Vorgehensweise bei der Erstellung solcher Diagramme ist sehr einfach. Nacheinander werden für jede Zahl Kreisausschnitte gezeichnet, alle Ausschnitte zusammen ergeben eine volle Scheibe. Um die Segmente voneinander unterscheiden zu können, werden verschiedene Farben benutzt.

Sie haben sich vielleicht schon gefragt, wie ein solches Kreissegment programmiert wird, da wir bisher nur mit ganzen Kreisen und Scheiben gearbeitet haben. Für die folgende Erläuterung blättern Sie bitte noch einmal zum Kapitel 5.3. zurück. Im Listing in Zeile 65516 sehen Sie die Befehlsfolge "FOR i= 0 TO 359". Wie ich dort schon erwähnte, wird damit der Kreisumfang von 0° bis 359° abgefahren. Wenn wir statt dessen FOR i= 180 TO 270 eingeben, so erhalten wir einen Viertelkreis. Sie sehen, daß der Anfangs- und Endpunkt unseres Kreisbogens frei gewählt werden darf.

Es ergibt sich aber noch ein anderes Problem. Aufmerksame Beobachter unter den geneigten Lesern haben sicher schon festgestellt, daß unsere ursprüngliche Routine den Kreis mathematisch korrekt von "Osten" aus gegen den Uhrzeigersinn zeichnet. Unsere Lesegewohnheit ist aber genau umgekehrt. Dies läßt sich beheben, wenn wir die SIN- und die COS-Funktion für die Koordinaten vertauschen. Damit beginnt der Umlauf bei "12 Uhr" (also an der obersten Position) und bewegt sich leserichtig im Uhrzeigersinn.
Hier das Listing:

```
60100 REM Tortendiagramme:
x1,y1,r,xe,w(0...xe-1),f(0...xe-1)
60110 wb= 0: DEG: FOR ii= 0 TO xe-1
60120 wa= wb: wb= wb+w(ii)*3.6: s(ii)= (wb-wa)/2
60130 FOR jj= wa TO wb
60140 xx= SIN(jj)*r: yy= COS(jj)*r
```

```
60150 MOVE x1,y1: DRAWR xx,yy,f(ii): PLOTR 0,0,1
60160 NEXT jj,ii: RETURN
```

Auch dieses Unterprogramm erwartet verschiedene Parameter in Übergabevariablen (siehe Liste in 60100). X1 und x2 geben den Mittelpunkt des Kreises an, r den Radius. Auch die Funktion von xe ist die gleiche geblieben. Die einzelnen Werte, die die Größe der zugehörigen Kreissegmente angeben, sind als Prozentzahlen im Array w(0...xe-1) gespeichert. Soll ein Feld 20% des Kreises bedecken, so muß in der zugehörigen Variable eine 20 stehen.
Außerdem kann für jeden Ausschnitt angegeben werden, welche Farbe er erhalten soll (f(ii)).

Auch dieses Programm besteht aus zwei verschachtelten Schleifen. Die äußere sorgt dafür, daß alle Werte berücksichtigt werden, die innere geht alle Winkel des anstehenden Segments durch.

In Zeile 60120 werden Anfangs- und Endwinkel der Ausschnitte berechnet. Außerdem leisten wir uns noch ein wenig Luxus. In dem Array s(0...xe-1) (das Sie vor Gebrauch des Unterprogramms DIMensionieren müssen) wird jeweils der Winkel gespeichert, der das Segment halbiert. Wenn Sie die einzelnen Ausschnitte beschriften wollen, so können Sie den Grafikcursor auf diesen Winkel setzen und mittels TAG einen Text an der richtigen Stelle ausgeben. Auch hier müssen die Koordinaten mit SIN und COS berechnet werden, doch sollten Sie den Radius vergrößern, damit die Schrift den Kreis nicht berührt.

Innerhalb der Scheibenroutine ist nur das PLOTR-Kommando neu. Es zeichnet einen Rand um das Diagramm für den Fall, daß die Segmentfarbe schwarz ist. Geschieht dies nicht, so kann der Eindruck eines Sterns entstehen.
Den Rest des Programmes kennen Sie bereits, deshalb verzichte ich auf eine Erklärung.

Beachten sollten Sie noch, daß die Summe der Werte im Array w genau 100 ergeben muß, sonst kann es sein, daß entweder

nur ein Teilkreis gezeichnet wird, oder der Rechner "verschlört" ein Segment.

6.2. DAS PROGRAMM FÜR DIE "KÜNSTLER"

Das folgende Programm ist für Leute gedacht, die viel mit Grafiken experimentieren und vorgefertigte Bilder in eigenen Programmen einsetzen wollen. Es ist viel zu mühselig, ganze Bilder mit PLOT- und DRAW-Befehlen aufzubauen; einfacher ist es, mit den Cursortasten auf dem Bildschirm umherwandern und nach Belieben Punkte setzen und löschen zu können. Das leistet untenstehendes Programm:

```
10 MEMORY 16383: CALL &BC06,&40: MODE 2: x= 320: y= 200:m= 1
20 a$="":WHILE a$ = "" :a$= INKEY$:WEND:  PLOT x,y,f
30 IF a$=CHR$(240) AND y<398 THEN y=y+2
40 IF a$=CHR$(243) AND x<639 THEN x=x+1
50 IF a$=CHR$(241) AND y>1 THEN y=y-2
60 IF a$=CHR$(242) AND x>0 THEN x=x-1
70 IF a$=CHR$(224) THEN f = 1:BORDER 24: m = 0
80 IF a$=CHR$(13) THEN f= 0: BORDER 24,0: m = 0
90 IF a$=" " THEN m = 1: BORDER 0
100 IF a$="s" THEN CALL &BC06,&C0: MODE 2: INPUT "filename"; f$: SAVE f$,b,16384
,16384: CALL &BC06,&40: GOTO 20
110 IF a$="l" THEN CALL &BC06,&C0: MODE 2: INPUT "filename"; f$: LOAD f$,16384:
CALL &BC06,&40: GOTO 20
120 IF a$="c" THEN CLG: GOTO 20
130 IF a$="x" THEN CALL &BC06,&C0: END
140 IF m= 1 THEN f=TEST(x,y)
150 PLOT x,y,1: GOTO 20
```

Für die Grafik wird der zweite Bildschirmspeicher ab 16384 benutzt, damit ist gewährleistet, daß etwaige Texteingaben das mühsam erstellte Bild nicht zerstören.

Beim Zeichnen gibt es drei Modi, die sich durch

unterschiedliche Rahmenfarben bemerkbar machen. Den ersten Modus möchte ich "Cursor-Modus" nennen, weil Sie hiermit einen Grafik-Cursor über den Bildschirm bewegen können, ohne die darunter liegenden Punkte zu ändern. Der Rahmen ist dabei dunkel. Ist der Rahmen hell, so wird der Punkt unter dem Grafikcursor gesetzt, blinkt der Rahmen, so wird gelöscht. Jedem dieser Modi ist eine Taste zugeordnet, durch die er aufgerufen wird. Für den Cursor-Modus ist es die Leertaste, Punkte werden gesetzt nach Druck auf COPY und ENTER bringt das Programm in den Löschmodus.

Aber das ist noch nicht alles. Mit der S-Taste kann das Bild auf Cassette abgespeichert werden. Dazu wird auf den Originalbildschirmspeicher zurückgeschaltet, wo dann der Filename eingeben werden kann. Dann wird geSAVEd (siehe Zeile 100). Ähnlich funktioniert das Laden eines bereits fertigen Bildes mit L (Zeile 110). In beiden Zeilen befinden sich MODE-Kommandos, mit denen der Bildschirm gelöscht wird und die Scrollingeffekte beim Umschalten vermieden werden.

Die C-Taste löscht den Bildschirm. Schließlich gibt es noch X, damit Sie das Programm ohne Bildverlust beenden können. Wollen Sie dann noch etwas ändern und die alte Grafik zurückholen, tippen Sie einfach CALL &BC06,&40: GOTO 20 ein.

Nun noch nähere Erläuterungen zu den einzelnen Programmzeilen. Zeile 10 initialisiert den Bildschirm und setzt die Startkoordinaten. In der nächsten Zeile wird ein Zeichen von der Tastatur geholt und das neue Pixel (Ergebnis des vorherigen Schleifendurchlaufs) gesetzt. Dann wird der Tastendruck in verschiedenen IF-THEN-Konstruktionen ausgewertet.

Die Variable f gibt die Farbe für den PLOT-Befehl aus Zeile 20 an. Ist der Cursormodus eingeschaltet (m=1), dann wird f nicht starr vorgegeben, sondern von der Farbe des ursprünglichen Pixels festgelegt. Auf diese Weise bleibt das Bild unverändert erhalten. Der PLOT-Befehl ist hinter das INKEY$ verlagert, um den Grafik-Cursor so lange wie möglich auf dem Bildschirm zu halten. Der Cursor selbst wird durch

den PLOT-Befehl in Zeile 150 erzeugt.

Natürlich stellt dieses kleine Programm keine Super-Software dar, das würde auch den Rahmen dieses Buches sprengen. Vielmehr soll es als Grundstock für eigene Entwicklungen und als kleines Hilfsmittel für "zwischendurch" dienen.
Übrigens können Sie Bilder, die mit dem Programm abgespeichert wurden, durch LOAD "!Name",49152 in den normalen Bildschirmspeicher holen.

7. *INTERRUPTPROGRAMMIERUNG*

Die Interruptprogrammierung ist eine der herausragendsten Eigenschaften des CPC-BASICs. Leider wird im Handbuch weder die Funktionsweise des Interrupts beschrieben, noch sind die dazugehörigen Befehle sehr ausführlich erklärt. Dem soll hier ein wenig abgeholfen werden.

7.1. *WIE FUNKTIONIERT EIN BASIC-INTERRUPT?*

Grundsätzlich unterscheiden sich Maschinensprache- und BASIC-Interrupt nicht. In beiden Fällen liefert ein Baustein (meist der TIMER) ein *Interruptsignal*, das das gerade laufende Programm unterbricht (aber erst nachdem der zur Zeit der Interruptanforderung anstehende Befehl abgearbeitet wurde) und einen Sprung in ein spezielles Unterprogramm auslöst. So weit, so gut. Beim Z-80 läuft dies alles auf Hardwarebasis ab, das bedeutet, daß nicht Programme, sondern spezielle Schaltungen diese Arbeit übernehmen. Im BASIC dagegen sind ausgeklügelte Maschinenprogramme für die korrekte Ausführung zuständig; außerdem kann eine BASIC-Unterbrechung nur durch einen der 4 Timer (0-3), nicht aber durch einen anderen Baustein (z.B. eine Schnittstelle) ausgelöst werden.
Liegt ein solches Signal vor, so prüft das BASIC zunächst, ob vielleicht noch ein Befehl beendet werden muß. Es ist daher nicht möglich, einen INPUT oder ähnlich zeitaufwendige Anweisungen zu unterbrechen.
Es könnte auch sein, daß die Interruptroutine durch einen *DI*-Befehl (Disable Interrupt) gesperrt worden ist. In den beiden letzten Fällen wird der BASIC-Interpreter die Interruptanforderung zwischenspeichern und dann nach der Freigabe nachholen.
Schließlich stellt er fest, welche Routine für den gerade abgelaufenen Timer zuständig ist und startet sie wie ein normales Unterprogramm.

Das sind noch nicht alle Aufgaben, die der CPC im Zusammenhang mit Interrupts wahrnehmen muß. Wenn Sie mit dem INK-Befehl wechselnde Farben definiert haben, so wird dieser *Farbwechsel* durch einen speziellen Timer ausgelöst. Sie können diesen Timer mittels SPEED INK stellen. Immer wenn die Farbe gewechselt werden soll, ändert das Betriebssystem einfach ein Byte in dem Chip, der für die Farberzeugung zuständig ist.
Außerdem muß die *Tastatur abgefragt* werden. Auch dies geschieht in inzwischen altbewährter Weise durch einen Interrupt, der alle $^1/_{50}$ Sekunde den Schnittstellenbaustein veranlaßt, den Code der gerade gedrückten Taste an den Prozessor zu schicken.
In Ihrem CPC ist also ganz schön was los!

7.2. DIE INTERRUPTBEFEHLE

Nach soviel Theorie wollen wir jetzt die einzelnen Interruptbefehle genauer unter die Lupe nehmen. Zunächst sind hier die Befehle zum Einschalten des BASIC-Interrupts zu nennen: *AFTER* und *EVERY*.
Falls Sie des Englischen nicht so ganz mächtig sind, hier noch die Übersetzung der beiden Begriffe. AFTER entstammt nicht etwa der Fäkaliensprache (wie Ähnlichkeiten aus dem deutschen Sprachgebrauch vielleicht vermuten lassen), sondern bedeutet schlicht und einfach "nach". EVERY heißt soviel wie "jede(r)".
Beide Befehle bewirken fast dasselbe, auch die Syntax ist gleich:

```
AFTER X, Y, GOSUB Z
EVERY X, Y, GOSUB Z
```

Sowohl AFTER als auch EVERY stellen einen Timer wie einen Wecker auf eine bestimmte Zeit ein. Im Unterschied zu unserem lärmenden Weckwerkzeug muß aber nur angegeben

werden, in wievielen 50stel Sekunden der Interrupt ausgelöst werden soll. Diesen Wert können wir im Parameter X angeben. Ein AFTER 200... wird also in 200/50 = 4 Sekunden einen Interrupt auslösen.
Damit sind wir auch schon beim einzigen Unterschied zwischen AFTER und EVERY. Ein AFTER-Befehl bewirkt nur einen einzigen Interrupt, während EVERY den Timer auf Dauerbetrieb einstellt, d.h. immer wenn die vorgegebene Zeit verstrichen ist, wird ein neuer Zyklus gestartet. So kann z.B. alle 10 Sekunden ein Meßwert über die Tastatur abgefragt werden, während fast gleichzeitig dazu Berechnungen ausgeführt werden.

Der zweite Parameter (Y) gibt an, welcher der 4 Timer benutzt werden soll. Je höher die Nummer, desto mehr Bedeutung hat der Interrupt. Timer 3 kann Timer 2 unterbrechen, nicht aber umgekehrt.
Der Rest des Befehls ist eigentlich selbsterklärend, hier wird nur noch angegeben, welche Zeile im Falle des Interrupts angesprungen werden soll.

Die beiden Befehle DI und EI sind schnell erklärt. DI ist die Abkürzung für Disable Interrupt, was nichts anderes bedeutet, als daß ab jetzt keine weiteren Interrupts mehr berücksichtigt werden. Auch ein Timer höherer *Priorität* kann jetzt nichts mehr ausrichten. Der Interpreter beschränkt sich lediglich darauf, die Anforderungen für einen späteren Zeitpunkt abzuspeichern. Dieser Zeitpunkt ist gekommen, wenn ein *RETURN* oder der *EI*-Befehl ausgeführt werden. EI steht für Enable Interrupt und bewirkt (wie auch RETURN), daß alle bisher gesperrten Interrupts nachgeholt werden.
DI und EI werden vor allem bei Routinen eingesetzt, die mit Grafikbefehlen arbeiten, um zu verhindern, daß der Grafikcursor durch einen zweiten Interrupt ungewollt verändert wird.

Zum besseren Verständnis der Prioritäten und Interruptsperren hier ein kleines Beispielprogramm. Es enthält zwei Interruptroutinen; das Hauptprogramm (Zeile 30) besteht aus einer Endlosschleife.

```
1 CLS: PRINT t;"   Sekunden"
2 LOCATE 6,5: PRINT "Sekunden"
10 EVERY 50, 0 GOSUB 100
20 EVERY 50, 1 GOSUB 200
30 GOTO 30
100 t= t+1: LOCATE 1,1: PRINT t
101 WHILE INKEY$ ="" : WEND
102 RETURN
200 x= x+1: LOCATE 1,5: PRINT x
201 RETURN
```

Beide Interruptroutinen setzen jede volle Sekunde einen Zähler um 1 weiter. Die Routine ab Zeile 200 hat größere Priorität, weil sie vom Timer 1 ausgelöst wird. Das Unterprogramm ab 100 braucht dagegen länger zur Ausführung, weil hier erst ein Tastendruck abgewartet wird (Zeile 101). Lassen Sie das Programm laufen, und Sie sehen, daß die erste Routine ständig von der zweiten unterbrochen wird (der obere Sekundenzähler verändert sich erst nach einem Tastendruck, der untere läßt sich dagegen davon nicht beeinflussen).
Fügen Sie jetzt in Zeile 100 vor "t= t+1" das Befehlswort DI ein. Damit werden für die Dauer der ersten Routine alle weiteren Interrupts gesperrt.
Wenn Sie das Programm jetzt wieder starten, so wird auch der zweite Sekundenzähler erst nach Tastendruck aktualisiert, da die zweite Interruptroutine auf die Freigabe durch das RETURN am Ende der ersten Routine warten muß.

Der letzte Befehl im Zusammenhang mit den Interrupts heißt *REMAIN*. Er führt ein Doppelleben, denn er hat zwei mögliche Formen:
a. REMAIN (Y)
b. PRINT REMAIN (Y) oder A= REMAIN (Y) usw.

Auf jeden Fall bewirkt er, daß der Timer Y zurückgesetzt wird, d.h. er kann keine Interrupts mehr auslösen.
Wird REMAIN als Funktion eingesetzt (Form b), so gibt sie zusätzlich noch an, wieviele $^1/_{50}$-Sekunden bis zur nächsten Unterbrechung noch verblieben wären.

7.3. IDEEN FÜR DIE INTERRUPTPROGRAMMIERUNG

Das am häufigsten genannte Beispiel für ein Interruptprogramm ist die Uhr, die irgendwo am Bildschirm ständig eingeblendet wird. Dazu läßt man einfach jede volle Sekunde (EVERY 50...) ein Unterprogramm aufrufen, das jedesmal die Sekundenzahl um 1 erhöht und ggf. auch Stunden und Minuten aktualisiert. Der Haken dabei ist, daß die Uhr nach jeder Cassettenoperation nachgeht, da diese Befehle den Timer anhalten.

Aber wie wäre es mit einem Ratespiel, bei dem es nicht nur auf die richtige Antwort, sondern auch auf die Geschwindigkeit ankommt? Der Ablauf könnte folgender sein:
1. Sie stellen dem Kandidaten "draußen an der Tastatur" eine Frage.
2. Sie geben mehrere mögliche Antworten vor, die alle eine Kennzahl erhalten. Diese Kennzahl kann über INKEY$ abgefragt werden.
3. Über einen AFTER-Befehl wird nach einer bestimmten Zeit (z.B. 10 Sekunden) die INKEY$-Schleife abgebrochen. Der Kandidat erhält dann je nach Antwort und Zeit Plus- oder Minuspunkte.

Punkt 2 läßt sich übrigens nicht mit einem INPUT programmieren, da ein Interrupt niemals einen laufenden Befehl unterbricht - also auch nicht einen INPUT.

Eine weitere Anwendung könnte darin bestehen, zwei Zeichen auf dem Bildschirm periodisch auszutauschen. So können recht einfach Bewegungen dargestellt werden. Das erste Zeichen zeigt beispielsweise einen PAC-MAN mit offenem Mund, im zweiten ist der Mund zu. Wenn durch eine Interruptroutine regelmäßig beide Zeichen ausgetauscht werden, entsteht der Eindruck von Kaubewegungen.

Ebenso könnte mittels Interrupt z.B. ein Auto mit stets gleichbleibender Geschwindigkeit über den Bildschirm bewegt werden.

Anwendungen dieser Art gibt es viele. Es bleibt Ihrer Phantasie überlassen, was Sie daraus machen!

8. SOUND

Von vielen Fachleuten wird der CPC-464 als Supercomputer bezeichnet. Auch die Geräuscherzeugung kann man zu den gelungenen Details zählen.
Weil ich der Meinung bin, daß man die Soundprogrammierung selbst hören muß, um sie zu verstehen, finden Sie hier keine weitere Erläuterung. Ich möchte Sie statt dessen auf die Erklärungen im Handbuch verweisen und nur die Anwendungen beschreiben, die sich daraus ergeben.

8.1. MINI-SYNTHESIZER

Die *ENT*- und *ENV*-Anweisungen bieten dem Programmierer eine unbegrenzte Zahl vom Möglichkeiten, einen Ton zu verändern und sogar Instrumente zu simulieren. Denn durch diese beiden Befehle lassen sich alle notwendigen Faktoren außer der Klangfarbe beeinflußen. Unten finden Sie deshalb ein Programm, mit dem alle Parameter durch Tastendruck verändert werden können. Das ermöglicht ein schnelles Experimentieren mit den Hüllkurven.

```
10 MODE 2: WINDOW #1,2,46,2,7:REM menue- & arbeitswindow
20 WINDOW #2,1,40,9,23: REM env-fenster
30 WINDOW #3,42,80,9,23: REM ent-fenster
40 MOVE 0,272:DRAW 639,272
50 MOVE 0,32: DRAW 639,32
60 MOVE 320,32: DRAW 320,272
70 MOVE 368,272: DRAW 368,399
80 LOCATE 49,2: PRINT "Mini-Synthesizer  Version 1.0"
90 LOCATE 49,4: PRINT CHR$(164)" 1985  Data-Becker GmbH"
100 LOCATE 49,6: PRINT "Autor: Hans Joachim Liesert"
110 LOCATE 2,25: PRINT "1 = Schrittzahl              2 = Schrittgroesse
    3 = Pausenlaenge"
```

```
120 LOCATE #2,3,2: PRINT #2,"ENvelope Volume"CHR$(10)
130 PRINT #2,"          1         2         3"CHR$(10)
140 PRINT #2,"    1     0         0         0"CHR$(10)
150 PRINT #2,"    2     0         0         0"CHR$(10)
160 PRINT #2,"    3     0         0         0"CHR$(10)
170 PRINT #2,"    4     0         0         0"CHR$(10)
180 PRINT #2,"    5     0         0         0"
190 LOCATE #3,3,2: PRINT #3,"ENvelope Tone"CHR$(10)
200 PRINT #3,"          1         2         3"CHR$(10)
210 PRINT #3,"    1     0         0         0"CHR$(10)
220 PRINT #3,"    2     0         0         0"CHR$(10)
230 PRINT #3,"    3     0         0         0"CHR$(10)
240 PRINT #3,"    4     0         0         0"CHR$(10)
250 PRINT #3,"    5     0         0         0"
260 DATA "q","2","w","3","e","r","5","t","6","y","7","u","i"
270 DIM T$(12): FOR i = 0 TO 12: READ t$(i): NEXT
280 DIM v(5,3),t(5,3)
290 SPEED KEY 255,10: ENV 1: ENT 1
300 LOCATE 1,1: c= 1
1000 CLS #1: PRINT #1,"Hauptmenue"CHR$(10)
1010 PRINT #1,"Leertaste = Melodie spielen"
1020 PRINT #1,"V         = ENV aendern"
1030 PRINT #1,"T         = ENT aendern"
1040 a$="":WHILE a$="":a$=INKEY$:WEND
1050 IF a$=" " THEN 1100
1060 IF a$="v" THEN 1200
1070 IF a$="t" THEN 1300
1080 GOTO 1040
1100 CLS#1: PRINT #1,"Klaviatur"CHR$(10)
1110 PRINT#1," 2 3   5 6 7"
1120 PRINT#1,"q w e r t y u i"CHR$(10)
1125 PRINT#1,"Leertaste = Menue"
1130 a$="":WHILE a$="":a$=INKEY$:WEND
1140 IF a$= " " THEN 1000
1150 FOR i = 0 TO 12 : IF a$<>t$(i) THEN NEXT i : GOTO 1130
1160 per = ROUND(125000/440/2^(i/12))
1165 c= c*2: IF c=8 THEN c=1
1170 SOUND c,per,0,0,1,1
1180 GOTO 1130
1200 CLS#1: PRINT#1,"ENV aendern"CHR$(10)
1210 PRINT#1,"Fuer Ende 0 eingeben"CHR$(10)
1220 INPUT #1,"Welches Element (Zeile,Spalte)";z,s
1230 IF z*s = 0 THEN 1000
1240 INPUT #1,"Wert";v(z,s)
1250 LOCATE #2,s*10,4+2*z: PRINT #2,v(z,s);"    "
1260 ENV 1,v(1,1),v(1,2),v(1,3),v(2,1),v(2,2),v(2,3),v(3,1),v(3,2),v(3,3),v(4,1,
,v(4,2),v(4,3),v(5,1),v(5,2),v(5,3)
1270 GOTO 1200
1300 CLS#1: PRINT#1,"ENT aendern"CHR$(10)
1310 PRINT#1,"Fuer Ende 0 eingeben"CHR$(10)
1320 INPUT #1,"Welches Element (Zeile,Spalte)";z,s
1330 IF z*s = 0 THEN 1000
1340 INPUT #1,"Wert";t(z,s)
1350 LOCATE #3,s*10,4+2*z: PRINT #3,t(z,s);"    "
1360 ENT -1,t(1,1),t(1,2),t(1,3),t(2,1),t(2,2),t(2,3),t(3,1),t(3,2),t(3,3),t(4,1
),t(4,2),t(4,3),t(5,1),t(5,2),t(5,3)
1370 GOTO 1300
```

Nach dem Programmstart erscheint die Bildschirmmaske mit dem Hauptmenue (links oben). Der gewünschte Programmteil kann einfach durch Tastendruck angewählt werden. Solange alle Parameter auf 0 sind, macht eine gespielte Melodie wenig Sinn. Daher sollte zumindest eine Lautstärkenhüllkurve eingegeben werden. Dazu muß jeweils das zu verändernde Element und dessen neuer Wert angegeben werden. Sofort wird der Inhalt der unten stehenden Matrix geändert. Alle Parameter werden direkt an die Hüllkurvenkommandos übergeben, Sie können sie deshalb auch sehr einfach in eigene Programme übertragen.

Auch zu diesem relativ umfangreichen Programm noch ein paar Anmerkungen:
Die Zeilen 10 bis 270 richten die Bildschirmmaske und die drei Windows ein. Zwei der Windows werden für die Ausgabe der ENV- und ENT-Parameter verwendet, im verbleibenden Fenster werden alle Eingaben und Bedienungshinweise abgewickelt. In 260 wird die Tastaturbelegung für die einzelnen Töne in das Array t$ eingelesen. Durch Vergleich kann jeder Taste der Feldindex als "Tonnummer" zugewiesen werden. Diese Nummer wird zur Berechnung der Tonperiode benutzt.

In Zeile 280 werden die beiden Parameterarrays für ENV und ENT dimensioniert. Zeile 290 schaltet schließlich die Repeatfunktion ab und löscht vorherige Hüllkurven.

Bei Zeile 1000 beginnt das eigentliche Programm. Zunächst wird das Menue ausgegeben. In den darauf folgenden Zeilen wird ein Zeichen von der Tastatur geholt (a$), aufgrund dessen dann die verschiedenen Teilprogramme angesprungen werden.

Die Zeilen 1100 bis 1180 beherbergen das Teilprogramm "Melodie spielen". Interessant ist hier vor allem die FOR-NEXT-Schleife in 1150. Sie wird nur dann fortgesetzt, wenn keine Übereinstimmung zwischen a$ und t$(i) gefunden

wurde. Sonst wird aus i (= Tasten- bzw. Tonnummer) die Tonperiode (per) für das SOUND-Kommando berechnet.

In Zeile 1200 beginnt der Änderungsteil für die ENvelope-Volume. Hier gibt es eigentlich nichts Ungewöhnliches, dank der Window-Technik können die benötigten Daten einfach per INPUT eingelesen werden. Der letzte Teil (ab 1300) dient zum Ändern der ENT-Parameter und ist mit dem ENV-Teil fast identisch.

8.2. WIE WIRD EIN TON "GEPLANT"?

Wie Sie aus dem Handbuch des CPC schon erfahren haben, können mit den *Hüllkurven* die Klangcharakteristika von verschiedenen Instrumenten nachgeahmt werden. Um dies zu erreichen, sind neben viel Geduld beim Experimentieren auch etwas Wissen über die Hüllkurven "echter" Töne nötig. So läßt sich aus dem Gedächtnis feststellen, daß der Ton einer Trompete genau so schnell aufhört wie er beginnt, während eine Glocke einen schnellen Anschlag hat und sehr lange nachklingt. Ein Klavier hat einen sehr abrupten Anschlag und klingt ebenfalls nach, es unterscheidet sich aber von der Glocke in der Tonart, es klingt etwas komplexer. Wir wollen in den folgenden Zeilen versuchen, verschiedene Instrumente durch Veränderung der Hüllkurvenparameter zu simulieren.

Eine Glasglocke hat einen sehr reinen, d.h. in der Frequenz unveränderten Ton. Deshalb wird die Tonhöhe durch die ENT-Parameter nicht verändert. Die Lautstärke muß dagegen sofort auf die höchste Stufe heraufschnellen und dann ganz langsam wieder auf 0 gehen. Die beiden Parametermatrizen sollten daher so aussehen:

1	15	1	1	0	1
1	0	1	1	0	1
1	0	1	1	0	1
12	-1	8	1	0	1
2	-1	20	1	0	1

oder im ENV-Kommando so:

```
ENV 1, 1,15,1, 1,0,1, 1,0,1, 12,-1,8, 2,-1,20
```

Eine Glocke aus Metall klingt etwas schnarrend. Das können wir beim CPC erreichen, indem wir durch die TON-Hüllkurve die Frequenz ständig auf- und abschwellen lassen. Das Kommando dazu lautet:

```
ENT -1, 1,1,3, 1,-1,3, 1,0,1, 1,1,3, 1,-1,3
```

Leider hat die Tonerzeugung Ihres Rechners einen Nachteil. Es gibt keine Möglichkeit, die Klangfarbe zu ändern. Im Gegensatz zu anderen Computern kann der CPC Ihnen keine dumpfen Flötentöne blasen, sondern nur helle Klänge. Deshalb gibt es für das eine oder andere Instrument (insbesondere Holzblasinstrumente) keine Nachahmungsmöglichkeit. Auch der typische metallene Klang von Trompeten wird Ihrem Rechner kaum zu entlocken sein (auf jeden Fall nicht vom BASIC aus). Trotzdem seien hier noch einige Hüllkurven beschrieben, die wenigstens starke Ähnlichkeiten erkennen lassen.

Eine Harmonika hat, bedingt durch Bau- und Spielweise, einen relativ langsamen Lautstärkeanstieg und auch einen langen Ausklang. Außerdem klingt der Ton nicht sehr rein, er schnarrt ein wenig. Mit

```
ENV 1, 7,2,1, 1,1,1, 1,0,1, 1,0,1, 15,-1,1
ENT -1, 1,0,3, 1,-1,1, 1,0,2, 1,0,1, 1,1,1
```

kann dies erreicht werden.

Alle Instrumente, die die Töne durch Schwingungen von Saiten erzeugen, haben eine charakteristische Lautstärkenhüllkurve. Nach dem sehr schnellen Anschlag schwillt der Ton zunächst etwas ab, um dann langsam auszuklingen (das Kommando dazu lautet ENV 1, 1,15,1, 1,-3,2, 1,0,1, 1,0,1, 12,-1,4). Mit verschiedenen Ton-Hüllkurven können jetzt verschiedene

Saiteninstrumente nachgeahmt werden. Einen klavierähnlichen Ton erhält man mit

```
ENT -1, 1,1,3, 1,-1,3, 1,0,3, 1,1,3, 1,1,3, 1,-1,3
```

Den etwas verzerrten Klang eines Banjo kann man (zugegebenermaßen mehr schlecht als recht) mit

```
ENT -1, 1,2,1, 1,0,2, 1,0,2, 1,-2,1, 1,0,4
```

nachahmen.

Das ist nur ein kleiner Ausschnitt der Möglichkeiten. Schließlich kann man ja auch andere Dinge als Musikinstrumente nachahmen. Wie wäre es mit einem Schlagzeug, das durch kurze Geräusche mit hartem Anschlag und kurzer Ausklangzeit programmiert werden kann. Oder Sie versuchen sich mit Phantasietönen. Ihrer Kreativität sind keine Grenzen gesetzt.

```
Zusammenfassung: Hüllkurven

Glocke:          ENV 1, 1,15,1, 1,0,1, 1,0,1, 12,-1,8, 2,-1,20
Metallglocke:    ENT -1, 1,1,3, 1,-1,3, 1,0,1, 1,1,3, 1,-1,3
Harmonika:       ENV 1, 7,2,1, 1,1,1, 1,0,1, 1,0,1, 15,-1,1
                 ENT -1, 1,0,3, 1,-1,1, 1,0,2, 1,0,1, 1,1,1
Saiteninstr.:    ENV 1, 1,15,1, 1,-3,2, 1,0,1, 1,0,1, 12,-1,4
Klavier:         ENT -1, 1,1,3, 1,-1,3, 1,0,3, 1,1,3, 1,-1,3
Banjo:           ENT -1, 1,2,1, 1,0,2, 1,0,2, 1,-2,1, 1,0,4
```

9. BASIC UND BETRIEBSSYSTEM

Die Routinen des Interpreters und des Betriebssystems können sehr nützlich sein. Einen Teil davon haben Sie bereits kennengelernt.

9.1. WIE WERDEN BASIC-ZEILEN GESPEICHERT?

Sie haben sich sicher schon gefragt, wie eine BASIC-Zeile im Speicher abgelegt wird. Wie Sie bereits aus Kapitel 2 wissen, erhalten alle Befehlsworte wie PRINT, SIN, SAVE usw. einen speziellen Code. Diesen Code nennt man auch *TOKEN*.
Mit dieser Methode wird enorm viel Speicherplatz gespart (für PRINT statt 5 Buchstabencodes nur 1 TOKEN-Byte). Außerdem muß der Interpreter dann während des Programmlaufs keine Buchstabenfolgen analysieren.

Variablennamen und Texte werden dagegen als ASCII-Codes gespeichert. Verknüpfungen wie *, / aber auch AND usw. sowie Relationszeichen ("=" etc.) haben ebenfalls einen TOKEN-Code.
Zahlen werden in einem sehr komplizierten Format abgelegt, das sich je nach Typ (Integer, Real) und Größe der Zahl ändert.

Innerhalb des Speichers kann man TOKEN daran erkennen, daß sie aus Bytes größer als 127 bestehen (es sei denn, man hat gerade eine Zahl vor sich). Die Variablennamen und Zeichenketten werden alle mit Zeichen definiert, die unter 128 liegen. So braucht der Interpreter nur eine Codetabelle.

Machen wir jetzt einen kleinen Test. Löschen Sie bitte ein evtl. noch vorhandenes Programm mit NEW und tippen Sie dann diese Zeile (Zeichen für Zeichen genauso) ein:

```
100 PRINT "test"
```

Dieses "Programm" wollen wir uns jetzt im Speicher ansehen. Dazu geben Sie bitte im Direktmodus ein:

```
FOR i = 368 TO 383: PRINT PEEK (i),: NEXT
```

Bei Byte 368 beginnt unser BASIC-Speicher, die obige Befehlsfolge gibt uns also die ersten 16 Bytes aus. Auf dem Bildschirm erscheinen diese Werte:

```
13  0  100  0  191  32  34  116  101  115  116  34  0  0  0
0
```

Die ersten 4 Bytes stellen zwei 16-Bit-Zahlen im Pointerformat dar. Die ersten beiden geben die Länge der Zeile (13 Bytes) an. Wenn der Interpreter eine spezielle Zeile sucht (z.B. bei GOTO) so prüft er zunächst, ob die erste Zeilennummer die richtige ist. Hat er das Ziel nicht gefunden, so wird einfach die Zeilenlänge zur gegenwärtigen Adresse addiert. So erreicht der Interpreter die nächste Zeile und kann auch diese prüfen.

In den nächsten beiden Bytes steht die Zeilennummer. Dann folgt das erste TOKEN; die 191 steht für PRINT. 32 und 34 sind die ASCII-Codes für Leer- und Anführungszeichen. Wie nicht schwer zu erraten war, stellen die folgenden vier Bytes die Codes für "test" dar. Mit der 34 für das letzte Anführungszeichen endet unsere Zeile. Weil danach keine weitere Programmzeile gespeichert ist, sind die letzten vier Bytes auf 0.

Diese Struktur können wir ein wenig manipulieren. Findet der Interpreter am Anfang des Programmspeichers eine Zeile mit der Nummer 0, so wird diese nicht gelistet, obwohl sie ganz normal ausgeführt werden kann. Lediglich Sprünge zu dieser Zeile funktionieren nicht, auch wenn Sie nicht gelöscht werden kann, weil das BASIC eine 0 als Zeilennummer nicht akzeptiert. Wollen Sie also die erste Zeile vor *LIST schützen*, so brauchen Sie nur die Bytes 370 und 371 mittels POKE auf 0 setzen. Probieren Sie es.

Bei Zeilen, die nicht am Anfang des BASIC-Textes vorkommen, ist dieses Verfahren nicht möglich. Die Zeilennummer kann zwar auf 0 gebracht werden, doch wird die Zeile weiterhin normal gelistet.

Vielleicht kennen Sie von anderen Computern den RENEW- oder auch OLD-Befehl. Er dient dazu, ein durch NEW gelöschtes Programm zu rekonstruieren. Das funktioniert, weil einige Rechner (z.B. Commodore) bei NEW nicht den Speicher mit Nullen füllen, sondern nur einige Zeiger löschen, so daß für den Interpreter der Eindruck entsteht, es gäbe kein Programm. Unser CPC hält sich leider nicht an diese Spielregeln, daher kann auch im Do-it-yourself-Verfahren ein solcher Befehl nicht eingebaut werden.

Zusammenfassung: Format einer BASIC-Zeile

Alle Befehle und Kommandos werden als TOKEN abgelegt, Texte und Variablennamen im ASCII-Code. Die ersten beiden Bytes geben die Länge der Zeile an, die nächsten beiden die Zeilennummer. Die Zeilennummer kann durch POKE künstlich geändert werden. Dies ermöglicht einen Listschutz für die erste Zeile, wenn die Nummer auf 0 geändert wird.

9.2. GARBAGE COLLECTION

Haben Sie schon einmal von einer Müllsammlung (das ist die Übersetzung von Garbage Collection) gehört? Wenn, dann wohl nur im Zusammenhang mit Computern. Denn was sich hier mehr wie ein naher Verwandter der städtischen Müllabfuhr ankündigt, ist eine sehr segensreiche Einrichtung. Um das zu verstehen, brauchen Sie aber einige Vorkenntnisse.

Wenn der Interpreter mit Stringvariablen arbeitet, so produziert er Müll in rauhen Mengen. Jedesmal, wenn ein

String irgendwie verändert wird (und sei es nur ein Buchstabe), so wird er komplett neu angelegt, während die alte Zeichenkette unverändert bleibt und auch nicht überschrieben wird. Sie steht nur noch nutzlos im Speicher herum. Irgendwann sind alle Bytes belegt, obwohl nur ein Bruchteil des Speicherplatzes wirklich noch gültige Daten enthält. Der Rest ist Müll. Soll jetzt ein weiterer String angelegt werden, muß der Computer erst einmal aufräumen. Diesen Vorgang nennt man Garbage Collection. Und weil er sehr viel Zeit benötigen kann, ist er bei viele Computerfans berühmt-berüchtigt. Ein kleines Beispiel soll Ihnen das verdeutlichen:

```
DIM a$(8000)
FOR i = 0 TO 8000: a$(i)= CHR$(1): NEXT i
```

Mit diesen Befehlen haben wir den Speicher mächtig vollgepackt (fast bis Oberkante Unterlippe). Durch PRINT FRE("") können wir jetzt eine Garbage Collection auslösen (nicht aber durch FRE(0)!).
Der Haken dabei ist: Die Ausführung dieser scheinbar simplen Funktion dauert jetzt mehrere Minuten! Der Interpreter muß nämlich von den 8001 Zeichenketten (die alle identisch sind) 8000 löschen. Es wäre ja auch sinnlos, 8000 gleiche Strings zu speichern, wenn eine Kette als Beschreibung für alle anderen dienen kann.

Um zu verhindern, daß eines Ihrer Programme durch eine solche Garbage Collection zeitweise aufgehängt wird, sollten Sie ab und zu in besonders stringintensiven Programmteilen ein FRE("") einstreuen, etwa in der Form F= FRE(""). Wenn nur kleine Müllmengen beseitigt werden müssen, läuft die Garbage Collection zwar insgesamt nicht viel schneller ab, doch stört Sie dann nicht so sehr.

9.3. ACHTUNG: FEHLER!

Nach der alten Regel "Kein Programm ohne Fehler" hat sich auch im *BASIC-ROM* ein Fehler eingeschlichen, der sich allerdings nicht so ohne weiteres bemerkbar macht. Dummerweise versteckt er sich auch noch in REM-Statements, so daß man ihn nach einem fehlerhaften Programmlauf sehr leicht übersieht.

Wenn in einer REM-Zeile eines der Steuerzeichen "Pfeil rechts" (TAB-Taste) oder "Senkrechter Strich" (SHIFT + Klammeraffe) auftritt, so verhaspelt sich der Interpreter; es treten unberechenbare Erscheinungen auf. Wenn man Glück hat, beschränkt sich die Störung lediglich auf einen Sprung zur Zeile 32511. Sollte diese Zeile nicht existieren, so wird danach die Programmausführung abgebrochen. Dabei kann es aber auch zur Löschung von ganzen Programmteilen kommen, oder diese Zeilen werden unsichtbar gemacht, d.h. man kann sie durch GOTO, GOSUB, RUN oder LIST nicht mehr erreichen, das REM verzweigt aber trotzdem noch zu den alten Zeilen.

Die Auswirkungen des REM-Fehlers sind sehr unterschiedlich und hängen wahrscheinlich auch vom übrigen Programm und der Stellung des REM-Befehls ab. So kann auch schon einmal ein Wechsel des Bildschirmmodus auftreten. Vielleicht werden diese Erscheinungen auch von anderen Steuerzeichen als den oben genannten hervorgerufen.

Möglicherweise läßt sich bei genügender Erforschung des Phänomens ein zweiter LIST-Schutz aufbauen, oder es gibt - wie beim HP-41 - synthetische Befehle; Anweisungen also, die vom Hersteller eigentlich nicht vorgesehen sind. Für entsprechende Hinweise zu diesem Thema wäre ich sehr dankbar.

9.4. UNBEKANNTE SEITEN

Ähnlich den unbekannten Befehlen aus Kapitel 2.5. möchte ich Ihnen hier noch zwei Eigenschaften von BASIC und Betriebssystem vorstellen, die im Handbuch nicht erwähnt wurden.

Die erste Eigenschaft betrifft den Editor, der z.B. für die Cursorsteuerung, COPY-Taste und Zeileneingabe zuständig ist. Wenn Sie schon einige Zeichen einer neuen Zeile eingegeben haben oder eine alte mittels EDIT bearbeiten und eines der Zeichen korrigieren wollen, so müssen Sie zunächst den Cursor an die richtige Stelle fahren. Alle Zeichen, die Sie jetzt eintippen, werden eingefügt und die alten entsprechend weitergeschoben. Das kann sehr lästig sein, wenn man alte Zeichen überschreiben möchte. Dieses *Einfügen* kann aber sehr einfach durch gleichzeitiges Drücken von *CTRL und TAB* abgestellt werden. Erneutes Drücken schaltet wieder ein.

Für die zweite Eigenschaft ist der BASIC-Interpreter zuständig. Wenn Sie ein Array (z.B. A(1)) zum ersten Mal ansprechen, ohne es vorher dimensioniert zu haben, so übernimmt der BASIC-Interpreter automatisch eine sogenannte *Vordimensionierung* mit 11 Elementen. Das ersetzt den Befehl DIM A(10).

Dazu ist allerdings zu sagen, daß das Weglassen des DIM-Befehls nur dann lohnt, wenn das Feld wirklich 11 Elemente enthalten soll. Für jedes dimensionierte Element wird Speicherplatz reserviert, der von anderen Daten nicht genutzt werden kann. Außerdem macht es ein Programm nicht gerade leichter lesbar, wenn plötzlich ein Feld auftritt, das vorher nicht ausdrücklich dimensioniert wurde.

9.5. VON EINEM, DER AUSZOG, DEM BASIC DAS FÜRCHTEN ZU LEHREN

Wie jedes Programm braucht auch der BASIC-Interpreter einige Speicherbytes, in denen er seine internen Daten ablegen kann. Da sich diese Bytes durch POKEs beeinflußen lassen, können wir dem BASIC ganz schön was unterschieben.

Nehmen wir zum Beispiel den im BASIC eingebauten *Programmschutz*. Soll ein geschütztes Programm von der Cassette eingelesen werden, so merkt sich der Interpreter dies in einem speziellen Byte. Ist dann das Programmende erreicht, so wird dieses Byte geprüft und gegebenenfalls ein NEW ausgeführt. Die Speicherzelle &AE45 beherbergt dieses Merkbyte. Wenn es einen anderen Wert als 0 enthält, so wird das im Speicher befindliche Programm geschützt. Durch POKE &AE45,1 kann dieser Schutz also von Hand eingeschaltet werden. POKE &AE45,0 wirkt dementsprechend genau umgekehrt.

Das Byte mit der Adresse &AC00 birgt eine weitere nützliche Eigenschaft. Je nach dessen Inhalt werden bei der Eingabe von Programmzeilen *überflüssige Leerzeichen gelöscht* oder sie bleiben (das ist der Normalfall) erhalten. POKE &AC00,1 schaltet diese sehr nützliche und Speicherplatz sparende Funktion ein.

Im Kapitel 3.1. haben wir die Wirkungsweise des HIMEM-Befehls näher betrachtet. Dabei wird ein Bereich oberhalb des BASIC-Speichers reserviert. Genausogut kann aber der *Anfang des BASIC-Programmes nach oben verlegt* werden. Dafür gibt es zwar keinen eigenen Befehl, aber es eröffnet vielleicht neue Möglichkeiten.
Die Speicherzellen &AE81 und &AE82 enthalten einen Zeiger auf den Programmanfang. Normalerweise zeigt er auf die Adresse 367, also das Byte vor dem Programm. Verändern wir diesen Zeiger, so wird der Speicher selbst nicht verändert, der Interpreter sucht jedoch jetzt an einer anderen Adresse nach seinem Programm. Alle Programmteile, die vor diesem neuen Startpunkt liegen, beachtet das BASIC gar nicht mehr.

Auf diese Weise kann man Programmteile verstecken, wenn man die Adressen weiß, wo die einzelnen Zeilen enden.

Sehr einfach ist es, das ganze *Programm verschwinden zu lassen*. Die Register &AE83 und &AE84 enthalten den Zeiger auf das Programmende. Durch die Befehle

```
POKE &AE81, PEEK(&AE83): POKE &AE82, PEEK(&AE84): NEW
```

wird der Startpunkt für den Interpreter hinter den Programmtext im Speicher verlegt. Der NEW-Befehl ist nötig, um alte Variablen zu löschen, die das BASIC als Programmzeilen mißverstehen könnte; unser Programm wird dadurch nicht beeinflußt, weil die Zeiger schon vorher geändert wurden.
Jetzt kann man ohne weiteres ein zweites Programm in den Speicher laden und bearbeiten, ohne das alte zu beeinflußen. Lediglich die Variablen werden gelöscht.

```
POKE &AE81,111: POKE &AE82,1
```

bringt den alten Zustand wieder zurück.

Solcherart versteckte Programme haben übrigens eine recht interessante Eigenschaft. Werden die Befehle zum Verändern des Pointers während eines Programmlaufes ausgeführt, so beeinflußt dies die Abarbeitung nachfolgender Zeilen fast überhaupt nicht. Nur die Sprungbefehle funktionieren nicht mehr, weil der Interpreter bei diesen Operationen die Zeiger als Orientierungspunkte für die Suche nach einer Zeile benutzt.

Zusammenfassung: BASIC überlisten

&AE45 dient als Merkbyte für Programmschutz.
POKE &AC00,1 schaltet Komprimiermodus ein, d.h. alle überflüssigen Leerzeichen werden gelöscht (Ausschalten durch POKE &AC00,0).
Die Bytes &AE81 und &AE82 zeigen auf das Byte vor dem

Programmanfang, &AE83 und &AE84 auf das Programmende.

9.6. NOCH EIN PAAR TRICKS

Vor allem die Werte von Befehlen wie SPEED xxxx oder ähnlichen werden sehr schnell vergessen, wenn man ein wenig experimentiert. Dann ist meist guter Rat teuer. Zumindest beim *SPEED INK*-Befehl weiß ich aber Abhilfe. Die beiden Parameter dieser Anweisung werden nämlich vom BASIC einfach in den beiden Bytes mit den Adressen &B1D7 und &B1D8 gespeichert, von wo sie das Betriebssystem bei Bedarf (bei jedem Farbwechsel) abholt. Und was das Betriebssystem kann, können wir mittels PEEK schon recht lange...

Sicher haben Sie auch schon von den *selbstdefinierbaren Zeichen* Gebrauch gemacht. Vielleicht ging es Ihnen dabei wie mir - ich ärgerte mich darüber, daß man alle 8 Bytes einer Zeichenmatrix immer wieder neu berechnen muß, auch wenn nur ein einziger Punkt geändert werden soll. Nun, auch hier gibt es wieder Abhilfe. Die selbstdefinierbaren Zeichen CHR$(240) bis CHR$(255) sind im RAM von &AB80 bis &ABFF gespeichert. Für jedes Zeichen sind acht Bytes reserviert, die wir leicht durch PEEK auslesen können. So muß nur noch das Byte mit dem zu ändernden Punkt neu berechnet werden.

Die Adresse eines Zeichens läßt sich mit dieser Formel bestimmen:

Adresse = 43904 + (X-240)*8

X ist dabei die Nummer des gewünschten Zeichens im Bereich zwischen 240 und 255.

Zusammenfassung: Tricks zum Betriebssystem

Die Parameter des SPEED INK-Kommandos können in den Bytes

```
&B1D7 und &B1D8 abgefragt werden.
Die Zeichenmatrizen der selbstdefinierbaren Charakter liegen
im Bereich &AB80 bis &ABFF.
```

10. ZUBEHÖRGERÄTE UND IHRE FUNKTIONSWEISE

Jeder Computer braucht Zubehör (Peripherie), um vernünftig arbeiten zu können. Bei einigen Herstellern, wie z.B. IBM muß man sich jedes Teil buchstäblich zusammensuchen, denn sowohl Tastatur als auch Monitor des IBM-PC werden extra geliefert. Glücklicherweise tut sich Schneider hier besonders hervor, denn beim CPC gibt es sogar noch einen Cassettenrecorder und einen Monitor serienmäßig. Aber auch für den CPC kann man sich noch weiteres Zubehör besorgen.

10.1. DAS DISKETTENLAUFWERK - DER DATENSPRINTER

Gehören Sie auch zu den "Hektikern", denen das Laden von Programmen von Cassetten zu lange dauert? Leiden Sie unter nervösen Zuckungen, wenn der Lautsprecher durch infernalische Geräusche das Laden des sechsundzwanzigsten Blocks verkündet? In diesen Fällen brauchen Sie ein Diskettenlaufwerk. Alte Computerhasen wissen längst, was für eine enorme Arbeitserleichterung selbst eine relativ langsame Floppy-Disk-Station gegenüber einem Cassettenrecorder bringt. Ein kleiner Zeitvergleich soll das belegen. Um ein Programm von 20 K zu laden, braucht der Cassettenrecorder mit Speedload 2 Minuten und 27 Sekunden, die Floppy begnügt sich mit lumpigen 9 Sekunden.

Aber das ist nicht der einzige Vorteil eines Diskettenlaufwerks. Weil auf einer Diskette die verschiedenen Magnetspuren (es gibt derer 40) wie bei einem Plattenspieler durch einfaches Schwenken des Tonarms bzw. Lesekopfs angewählt werden können, ist es möglich, das sogenannte *Direktzugriffsverfahren* anzuwenden. Das bedeutet, daß der Rechner nicht alle Files von Anfang an nach den richtigen Daten durchsuchen muß, sondern Sie können quasi sagen "Hol mir das 3. Byte aus der Adressen-Datei". Der Computer akzeptiert zwar zunächst nur die originalgetreue

BASIC-Übersetzung unserer Anweisung, doch die führt er dann superschnell aus. Auf diese Weise kann die Diskette fast wie ein vergrößerter Speicher benutzt werden.

Ein Diskettenlaufwerk hat zwar noch weitere Vorteile (z.B. gibt es ein *Inhaltsverzeichnis* mit allen auf der Diskette gespeicherten Files), aber auch kleine Nachteile. Zunächst ist es sehr teuer. Außerdem braucht es eine spezielle Ansteuerungselektronik, die *Controller* heißt. Sie wird dem ersten Laufwerk, das Sie kaufen (und das deshalb auch teurer ist), beigelegt. Der Controller kann bis zu zwei Laufwerke gleichzeitig verwalten, deshalb können Sie das zweite Laufwerk an das erste anschließen. Ohne Controller läuft jedoch nichts.
Außerdem enthält das Gehäuse der Kontrollelektronik ein ROM, das den Befehlssatz Ihres CPC um die notwendigen Diskettenbefehle erweitert.

10.2. DER DRUCKER

Einer der vielen Vorzüge Ihres Computers ist die eingebaute Druckerschnittstelle. Im Gegensatz zur Floppy braucht Sie keine Erweiterungssoftware und keinen Controller, all das ist schon eingebaut.
Außerdem handelt es sich um eine sogenannte *Centronics-Schnittstelle*, was bedeutet, daß Sie den Interfaces des Druckerherstellers Centronics angepaßt ist. Die meisten Hersteller haben diese Schnittstelle übernommen. Daher können Sie den überwiegenden Teil aller auf dem Markt befindlichen Drucker direkt anschließen. Aber auch hier gibt es wieder den berühmten Wermutstrofen. Denn der CPC überträgt nur die untersten 7 Bits der 8-Bit-ASCII-Codes und "klaut" Ihnen damit glatt die Hälfte der darstellbaren Zeichen. Das schadet Ihnen weniger als Sie denken, da die Codes 0 bis 127 alle wichtigen Zeichen enthalten, und die verlorenen Codes 128 bis 255 meist mit Graphikzeichen belegt sind.

Oft kommt es vor, daß der ASCII-Code des Druckers nicht mit dem Ihres Computers übereinstimmt. In diesem Fall müssen Sie sich eine *Anpassungstabelle* programmieren. Das klingt schwieriger als Sie denken. Tragen Sie einfach die Codes des Druckers, die den Zeichen des CPC entsprechen, der Reihe nach in ein Array (0 bis 127) von Integer-Variablen ein. Sollen jetzt Daten ausgedruckt werden, so müssen Sie die Zeichenketten Buchstabe für Buchstabe abschicken. Das geht mit PRINT#8, CHR$(X). X ist der ASCII-Code des gewünschten Zeichens. Sollen jetzt die korrigierten Codes übertragen werden, so setzen Sie statt X einfach den Ausdruck ARRAY (X) ein. Dann wird nicht der Computercode gedruckt, sondern der Code, der statt dessen im ARRAY steht.

10.3. DER JOYSTICK

Wie viele Anwender werden auch Sie wahrscheinlich irgendwann einmal versucht haben, ein Programm mit Joystickabfrage zu schreiben. In den meisten Fällen haben Sie dabei die verschiedenen Positionen durch IF-THEN-Konstruktionen ausgewertet, etwa so:

```
IF JOY(0)= 1 THEN 100
IF JOY(0)= 2 THEN 200
IF JOY(0)= 4 THEN 300
...
```

Diese Methode ist sehr langsam und vergleichsweise umständlich. Besser geht es mit diesem Programm (Erklärung folgt unten):

```
10 A = LOG (JOY(0)) / LOG (2)
20 ON A GOTO 100,200,300...
```

Zeile 20 verzweigt je nach Wert der Variablen A in verschiedene Programmteile. Würden wir der Variablen einfach

den JOY-Wert zuweisen, so würde das ON-Kommando nicht wunschgemäß arbeiten, da JOY keine aufeinanderfolgenden Werte für die einzelnen Richtungen liefert (wie 1, 2, 3 usw.), sondern deren Zweier-Potenzen (1, 2, 4, 8, 16, 32). Daher wird von diesen Potenzen der Logarithmus zur Basis 2 berechnet, was dann die gewünschten Nummern erzeugt.

Die Funktionsweise eines Joysticks ist sehr einfach. Er besteht einfach aus 5 oder 6 mehr oder weniger aufwendigen Tastern. Einer bzw. zwei werden für den Feuerknopf benutzt, die anderen sind unter dem Steuerknüppel in den vier verschiedenen Bewegungsrichtungen angebracht. Je nach Stellung des Knüppels wird dann der entsprechende Taster betätigt - der CPC registriert das dann und gibt einen entsprechenden Wert aus.
Natürlich gibt es auch unter den verschiedenen Joysticks auf dem Markt Unterschiede. Einfache und billige Exemplare arbeiten mit einfachen Folienkontakten (ehemaligen ZX-81-Besitzern sicher noch in unguter Erinnerung), aufwendigere Verwandte dagegen besitzen Mikroschalter, die sich meist mit einem kleinen Klick bemerkbar machen.
Beim Kauf sollte darauf geachtet werden, daß der Joystick möglichst abgerundete Kanten besitzt. Andernfalls können beim Spiel sehr schnell Ermüdungserscheinungen auftreten. Im übrigen passen alle Atari-kompatiblen Joysticks auch für den Schneider-Computer.

11. EINIGES ÜBER SCHNITTSTELLEN

Auf der Rückseite Ihres CPC befinden sich diverse Anschlüsse und Stecker. Wenn Sie sich bis jetzt gefragt haben, wie all das funktioniert und was eigentlich dahintersteckt, dann sollten Sie dieses Kapitel lesen.

11.1. KLEINE SCHNITTSTELLENINVENTUR

Bevor wir uns mit den einzelnen Anschlüssen auseinandersetzen, sollten wir vielleicht erst einmal den Begriff "Schnittstelle" (oder neudeutsch *Interface*) näher betrachten.

Bei einer Schnittstelle handelt es sich nicht immer nur um eine Verbindung zur Peripherie, an den Anschlußbuchsen ist der Computer tatsächlich "aufgeschnitten", hier kann man fast in den Rechner eingreifen. Das beste Beispiel dafür bietet der *Expansion Connector*, der auf dem Gehäuse nur ganz lapidar mit "Floppy-Disc" beschriftet ist. Hier handelt es sich nämlich wirklich um einen Erweiterungsanschluß, der nicht nur die notwendige Verbindung zur Floppy herstellt, sondern auch für den Anschluß zusätzlicher *ROM-Module* vorgesehen ist. ROMs sind aber Speicherbausteine und gehören somit zum unmittelbaren Zugriffsbereich des Prozessors. Dementsprechend besteht der Expansion-Connector auch "nur" aus Adress-, Daten- und Steuerbus des Z-80 plus einiger zusätzlicher Steuersignale.

Daneben finden Sie den *Druckeranschluß*. Er ist der Centronics-Norm angepaßt. Auch der Drucker-Port ist mit dem Prozessor-Datenbus verbunden, nicht jedoch mit dem Adress- und dem Steuerbus. Dafür kommen jedoch einige Steuerleitungen vom Portbaustein 8255.

Leider haben die Entwickler des CPC ausgerechnet bei dieser

Einrichtung gespart. Statt der normal üblichen 8-Bit-Übertragung wurde beim CPC einfach das achte Bit weggelassen (aber wo gibt es schon einen perfekten Computer?).

Der *Joystickanschluß* ist nur eine Erweiterung der Tastatur. Er ist mit verschiedenen Bausteinen im Rechner verbunden (darauf kommen wir später noch).

Auch die Anschlüsse für Monitor und Stereoverstärker sind eigentlich Schnittstellen. Sie arbeiten aber nicht direkt mit dem Prozessor zusammen.

11.2. WIE FUNKTIONIERT EINE SCHNITTSTELLE?

Die Funktionsweise einer Schnittstelle ist bei allen Computern grundsätzlich gleich. Die zu übertragende Information wird vom Prozessor an den *Schnittstellenbaustein* geliefert, der dann seinerseits das Peripheriegerät anspricht und die Daten überträgt. Dazu kann es notwendig werden, beide Geräte zu synchronisieren. In diesem Fall werden über spezielle Steuerleitungen Impulse ausgetauscht, die die Bereitschaft zur Übertragung signalisieren.

Vor einer Datenübertragung muß der Prozessor außerdem angeben, ob Daten empfangen oder gesendet werden sollen. Je nach gewünschtem Modus wird dann der Port auf Ein- oder Ausgabe geschaltet.

In einigen Fällen wird kein spezieller Schnittstellenbaustein eingesetzt, so auch im CPC. Dann übernimmt der Prozessor alle notwendigen Operationen selbst (was natürlich etwas langsamer abläuft). Die Hardware beschränkt sich hier lediglich auf das Anpassen unterschiedlicher Spannungspegel.

Die für den Betrieb einer Schnittstelle notwendigen

Programme sind alle schon im ROM gespeichert und lassen sich durch BASIC-Befehle anwenden; außerdem ist eine effiziente Schnittstellenprogrammierung (mit wenigen Ausnahmen) nur in Maschinensprache möglich.

11.3. IHRE PERSÖNLICHE SCHNITTSTELLE

Gehören Sie auch zu den unverbesserlichen Hardwarespezialisten, die bei Computern vor allem an wilde Drahtverhaue, heiße Lötkolben und diverse Basteleien denken? Bei den Vertretern dieser Spezies des homo electronicus äußert sich häufig der Wunsch, Daten von eigenen Geräten (z.B. Thermometer o.ä.) einzulesen. Zu diesem Zweck stehen theoretisch das Centronics-Interface und der Expansion-Connector zur Verfügung. Beide sind aber in der Praxis für andere Zwecke vorgesehen.

Das gilt zwar auch für den *Joystickanschluß*, doch der wird meist nur bei Spielen benutzt. Deshalb kann er für seriöse Anwendungen auch gut zweckentfremdet werden.

Der *User-Port* (so steht es schwarz auf schwarz auf dem Gehäuse) kann vom BASIC aus sehr gut durch die Funktionen JOY und INKEY abgefragt werden. Damit sind alle Voraussetzungen für einen Kontakt mit der Außenwelt gegeben.

An die Pins 1 bis 7 (siehe Handbuch, Anhang V) können jeweils zwei Schalter angeschlossen werden. Der erste Schalter jedes Pins wird mit COMMON (Stift 8), der zweite mit COM 2 (Stift 9) verbunden. Mit den beiden letztgenannten Pins unterscheidet der Rechner zwischen Joystick 0 und 1. Beim Schließen eines Schalters werden ein Eingang (1 bis 7) an eines der Commonsignale gelegt. Für den CPC ist dies das Gleiche wie eine Tasten- oder Joystickbetätigung, Sie können das durch die erwähnten Befehle feststellen.

An die Eingänge des Joystickports können Sie beliebige

Schalter anschließen. Was für Geräte dahinterstecken (Relais, Transistoren etc.), bleibt Ihrem Einfallsreichtum überlassen.

11.4. TASTATURABFRAGE

Der Vollständigkeit halber möchte ich Ihnen in diesem Abschnitt erklären, wie die Tastaturabfrage funktioniert. Wie Sie durch Nachzählen leicht feststellen können, hat der CPC 73 Tasten (SHIFT nur einmal gerechnet), die alle in regelmäßigen Abständen ($^1/_{50}$ Sekunde) geprüft werden müssen. Zu diesem Zweck ist die Tastatur elektrisch in 10 Spalten aufgeteilt, die der Z-80 einzeln einschalten kann. Dazu muß nur die Nummer der Spalte an den Portbaustein 8255 übergeben werden.

Ist eine Taste der gewählten Spalte gedrückt, so wird ein Bit auf 0 gesetzt, sonst bleibt es auf 1. Pro Spalte entstehen so bis zu 8 einzelne Bits. Auf diese Weise kann ein ganzes Byte zusammengestellt werden, das der Soundchip (jawohl, der Soundchip) über den 8255 an den Z-80 zurückgibt. Der Prozessor kann also einfach durch Auswertung der gelöschten Bits feststellen, welche Tasten in der Spalte gedrückt sind. Der auf den ersten Blick umständlich erscheinende Weg über den Soundchip wurde benutzt, weil dieser Baustein schon von Haus aus mit einem zusätzlichen Port ausgerüstet ist und die anderen, schneller erreichbaren Ports nicht mit der relativ langsamen Tastaturabfrage belastet werden sollten.

Das Prinzip der Abfrage über Spalten wird übrigens in fast allen Computern mit mehr oder weniger großen Abwandlungen eingesetzt.

12. CASSETTENRECORDER UND TASTATUR

Die Bedienung des Cassettenrecorders ist im CPC-Handbuch leider etwas stiefmütterlich behandelt worden. Diesem Mangel soll jetzt ein wenig abgeholfen werden. Und auch in der Tastaturabfrage stecken vielleicht noch unbekannte Möglichkeiten.

12.1. WIE BAUT MAN DATEIEN AUF?

Im CPC-Handbuch haben Sie sicherlich schon die Beschreibung des Filetyps "ASCII" gelesen. Leider wurde durch einen genialen Glücksgriff verschwiegen, daß damit nicht nur Listings für Textverarbeitungsprogramme erzeugt werden können (per SAVE "name",A). Der CPC ist in der Lage, auch String- und Arithmetikvariablen auf Band zu speichern. Außerdem möchte ich Ihnen zeigen, wie die ASCII-Listings vom BASIC aus eingelesen werden können.

Der Name "ASCII-File" kommt von der Tatsache, daß alle Daten als einfache Folge von ASCII-Codes (oder Bytes) abgespeichert werden. Der einzige Unterschied zum Programmfile besteht darin, daß nicht einfach ein Speicherausschnitt auf Band kopiert wird, sondern die einzelnen Einträge durch ein CHR$(13) abgeschlossen sind. Dabei macht es keinen Unterschied, ob die Daten aus Variablen oder sonst woher stammen.

Das Beschreiben und Lesen dieser Files ähnelt stark der Bildschirmausgabe; dazu gibt es die Befehle PRINT#9 und INPUT#9. Das kommt nicht von ungefähr, denn auch bei der Bildschirmausgabe wird CHR$(13) zur Trennung einzelner Informationen benutzt. Das Steuerzeichen hat dort jedoch die Aufgabe, dem Betriebssystem mitzuteilen, daß der Cursor in die nächste Zeile bewegt werden muß.

Nehmen wir jetzt spaßeshalber an, Sie hätten in Ihrer unermüdlichen Datensammelwut, der Sie als echter Computerfan nun einmal erlegen sind, ganze zwei Variablen mit Daten gefüllt, die auch morgen noch erhalten sein sollen. In diesem Fall helfen Ihnen die untenstehenden Programmbeispiele:

```
10 REM Datei anlegen
20 OPENOUT "TEST"
30 PRINT #9, a$
40 PRINT #9, b
50 CLOSEOUT
```

```
10 REM Datei wieder einlesen
20 OPENIN "TEST"
30 INPUT #9, a$
40 INPUT #9, b
50 PRINT a$,b
60 CLOSEIN
```

Das erste Programm schreibt die beiden Variablen a$ und b auf die Cassette. Mit dem zweiten Programm können die Inhalte wieder zurückgeholt werden. Dabei sind die Variablennamen nicht wichtig, Sie könnten ebenso x$ und y benutzen. Im Gegensatz dazu muß aber der Variablentyp stimmen, sonst könnte es zu einem TYPE-MISMATCH-Error kommen.

Beim Abspeichern von *ASCII-Listings* werden auf ähnliche Weise Zeichenketten erzeugt, die das BASIC als Strings lesen kann. Als Beispiel ist wieder ein Programm angegeben:

```
10 INPUT "Wie viele Zeilen?"; a: a=a-1
20 DIM a$(a)
30 OPENIN "filename"
40 FOR i = 0 TO a
50 INPUT #9, a$(i)
60 NEXT
70 CLOSEIN
```

Nach dem Ablauf dieses Programmes stehen die Zeilen eines ASCII-Listings im Stringarray, wo Sie sie dann weiterverarbeiten können. Leider hat das ganze System einen kleinen Haken. Das BASIC benutzt auch Kommata zur Trennung von Strings. Daher werden alle Zeilen, in denen ein Komma steht (und das sind ziemlich viele!) an der betreffenden Stelle getrennt, und ein zweiter String wird mit der Restzeile belegt. Dies können Sie jedoch durch geschickte Programmierung wieder wettmachen. Die geschickte Programmierung besteht einfach darin, daß Sie das Befehlswort INPUT durch *LINE INPUT* ersetzen. Diese spezielle Form erkennt nur ein CHR$(13) als Zeilenendemarkierung an.

Im obigen Beispiel mußten Sie per Hand eingeben, wie viele Zeilen das Listing enthält. Sollten Sie sich dabei geirrt haben, so könnte ein "EOF met" auftreten, wenn das Programm versucht, mehr Daten zu lesen, als in dem File vorhanden sind. Dieser Fehler läßt sich vermeiden. Im BASIC gibt es eine Funktion, die das Ende einer Datei anzeigt, mit Namen *EOF* (End Of File). PRINT EOF ergibt den Wert 0, wenn noch Daten vorhanden sind, am Fileende ist das Ergebnis -1. Unser Programm läßt sich deshalb so ändern:

```
10, 20, 30 und 70 wie oben
40 WHILE NOT(EOF)
50 LINE INPUT #9, a$(i): i=i+1
60 WEND
```

Durch die WHILE-WEND-Konstruktion wird nur solange vom Band gelesen, wie noch Daten vorhanden sind.
Leider kann jetzt noch ein SUBSCRIPT OUT OF RANGE-Error auftreten, wenn das Listing mehr Zeilen umfaßt, als Strings dimensioniert wurden.

Beim Aufbau eigener Dateien können Sie sich damit behelfen, daß Sie bei der Anlage des Files zuerst die Anzahl der Daten als normale Variable auf Band schreiben und dann erst die eigentlichen Daten folgen. Wenn die Datei später wieder eingelesen wird, so liest man zuerst die Datenanzahl, dimensioniert damit die Felder und holt dann den Rest

herein.

Eine sehr exotische Anwendung ermöglichen 32 Speicherstellen im Bereich &B807 bis &B816 und &B84C bis &B85B. Hier werden die Namen der INPUT- (B807) und der OUTPUT-Files (B84C) vom Betriebssystem als ASCII-Ketten abgespeichert. Durch die folgende Zeile können Sie den letzten OUTPUT-*Filenamen auslesen*:

```
FOR i = &B84C TO &B85B: PRINT CHR$(PEEK(i));: NEXT
```

Wie wäre es, wenn Sie Ihre Programme mit einer Testroutine ausstatten, die verhindert, daß der Name verändert wird? Dazu müßte nur einmal am Anfang des Programms auf die gezeigte Weise der Filename ausgelesen werden. War er falsch, so lassen Sie das Programm durch NEW löschen, vielleicht noch von einem bissigen Kommentar begleitet.

Und wenn Sie wissen möchten, welche *Schreibgeschwindigkeit* gerade eingestellt ist, brauchen Sie nur PRINT PEEK(&B8D1) einzutippen. Erscheint als Ergebnis eine 6, so wird langsam gespeichert, bei 12 ist *SPEED WRITE 1* gegeben worden.

Zusammenfassung: Dateien auf Cassette

Mit LINE-INPUT können auch Strings, die ein Komma enthalten, gelesen werden. Die Funktion EOF zeigt das Dateiende an. Mittels der beiden genannten Befehle können leicht ASCII-Listings in BASIC-Arrays geholt und bearbeitet werden. Der Name des letzten Files kann aus den Speicherzellen &B807-&B816 bzw. &B84C-&B85B ausgelesen werden. Das Byte &B8D1 gibt die Schreibgeschwindigkeit an.

12.2. *INKEY IN EINEM ANDEREN LICHT*

Spätestens wenn Sie Ihr erstes BASIC-Spiel schreiben,

brauchen Sie eine Form der Tastaturabfrage, bei der auch mehrere gleichzeitige Tastendrücke registriert werden. Glücklicherweise gibt es hierfür eine entsprechende BASIC-Funktion. INKEY(X) (wohlgemerkt: ohne das "$") gibt an, ob die Taste x gerade gedrückt ist. Da diese Funktion unabhängig vom Tastaturpuffer arbeitet, wird nicht das erste Zeichen im Puffer zurückgegeben, sondern nur der elektrische Tastenkontakt getestet. Als Beispiel für die Anwendung mag das folgende Listing dienen:

```
10 CLS
20 IF INKEY(69) = 0 THEN LOCATE 1,5: PRINT "Taste a
gedrueckt"
30 IF INKEY(36) = 0 THEN LOCATE 1,10: PRINT "Taste 1
gedrueckt"
40 LOCATE 1,5: PRINT SPACE$(20): REM Zeile 5 löschen
50 LOCATE 1,10: PRINT SPACE$(20): REM Zeile 10 löschen
60 GOTO 20
```

Starten Sie das Programm und drücken Sie dann einmal die Tasten A und L gleichzeitig. Beide Meldungen erscheinen untereinander auf dem Bildschirm. Bei einer entsprechenden Programmierung mit INKEY$ wäre das nicht möglich, weil damit nur ein Zeichen geholt werden kann.

Auch zur Tastaturabfrage sei noch ein kleiner Trick angemerkt. In vielen Programmen werden spezielle Schleifen eingebaut, die den Programmlauf so lange aufhalten sollen, bis es der Anwender durch Tastendruck wieder freigibt. Eine solche Schleife sieht meistens so aus:

```
WHILE INKEY$="": WEND
```

Die gleiche Arbeit übernimmt eine Maschinenroutine im ROM, die durch CALL &BB18 aufgerufen werden kann.

Zusammenfassung: Tastaturabfrage

Das INKEY-Kommando eignet sich auch zur gleichzeitigen

```
Abfrage mehrerer Tasten.
CALL &BB18 wartet, bis eine beliebige Taste gedrückt wurde.
```

13. EINFÜHRUNG IN DIE Z-80-MASCHINENSPRACHE

In vielen Publikationen finden Sie immer wieder Progamme zum Abtippen. Sehr oft sind diese Programme in Maschinensprache geschrieben, einer Sprache, die dem Neuling wie ein Buch mit 7 Siegeln erscheint. Zugegeben, die Maschinensprache ist nicht so leicht zu erlernen wie BASIC, doch dafür ist sie sehr viel schneller und bietet dem erfahrenen Programmierer ungleich mehr Möglichkeiten. Daher möchte ich Ihnen hier die Grundzüge der maschinennahen Programmierung erläutern. Nach der Lektüre dieses Kapitels sind Sie in der Lage, die grundsätzliche Funktionsweise von Maschinenprogrammen zu verstehen und selbst zu entscheiden, ob Sie sich weiter mit dieser Sprache beschäftigen wollen. Sollte Ihnen die Maschinensprache nicht gefallen, so ist das auch kein Beinbruch. Das erworbene Wissen läßt sich auch für andere Aufgaben einsetzen, und schließlich sind PASCAL oder LOGO ja auch keine schlechten Wege, einem Computer etwas beizubringen.

13.1. WAS IST MASCHINENSPRACHE ÜBERHAUPT?

Wie Sie sicher wissen, stellt die Maschinensprache die einzige Möglichkeit dar, den Prozessor ohne Umweg über einen Compiler oder Interpreter direkt zu programmieren. Daher ermöglicht diese Sprache auch so immens hohe Geschwindigkeiten.
Die Maschinensprache umfaßt verschiedene Befehle, aus denen sich alle komplexeren Operationen des BASICs oder anderer Sprachen zusammensetzen lassen. Man kann die Maschinenbefehle grob in drei Gruppen einteilen. Für BASIC-Programmierer am einfachsten zu verstehen sind die Sprungbefehle, mit denen das Programm ähnlich GOTO und GOSUB im Speicher umherspringen kann. Andere Befehle bewirken Datenmanipulationen (z.B. Additionen, Verknüpfungen etc.). Die letzte Gruppe umfaßt die Operationen, die Daten von

einem Ort zum anderen innerhalb des Speichers bewegen.
Grundsätzlich gilt, daß es für Mikroprozessoren keine Variablen gibt. Er kennt nur die normalen Speicherzellen und interne Register. Für die Unterscheidung zwischen Daten und Programmbytes muß der Programmierer selbst sorgen. Im allgemeinen können Datenmanipulationen nur in den internen Registern ablaufen.
Ein Maschinenbefehl besteht immer aus einem sogenannten Operationscode (oder *Opcode*), der sozusagen die "Nummer" des Befehls angibt. Dieser Opcode kann bis zu drei Bytes umfassen. Außerdem können dem Befehl noch bis zu zwei Bytes an Daten folgen. Rein theoretisch haben die Z-80-Befehle also eine Länge von bis zu 5 Bytes. Praktisch sind es aber nur 4, da 3-Byte-Opcodes nur bei Befehlen mit höchstens 1 Datenbyte vorkommen.

13.2. DER TAKT

Alle Bauteile des Computers richten sich nach einem kleinen unscheinbaren Quarz, der den Takt (4 Megahertz = 4000000 Schläge oder Zyklen pro Sekunde) vorgibt. Dies ist nötig, um die verschiedenen ICs zu synchronisieren. Geschähe dies nicht, so könnte es z.B. passieren, daß ein Speicherbaustein Daten zum Prozessor schickt, obwohl dieser noch gar nicht zur Übernahme bereit ist. Auch ein noch so schneller Mikroprozessor braucht immer noch ein wenig Zeit zur Verarbeitung der Daten.

13.3. DER AUFBAU DES Z-80

Jeder Mikroprozessor besitzt *interne Register*, in denen die Operationen durchgeführt werden. Das wichtigste Register ist der sogenannte *Akkumulator*. In ihm laufen die meisten arithmetischen und logischen Verknüpfungen ab. Der

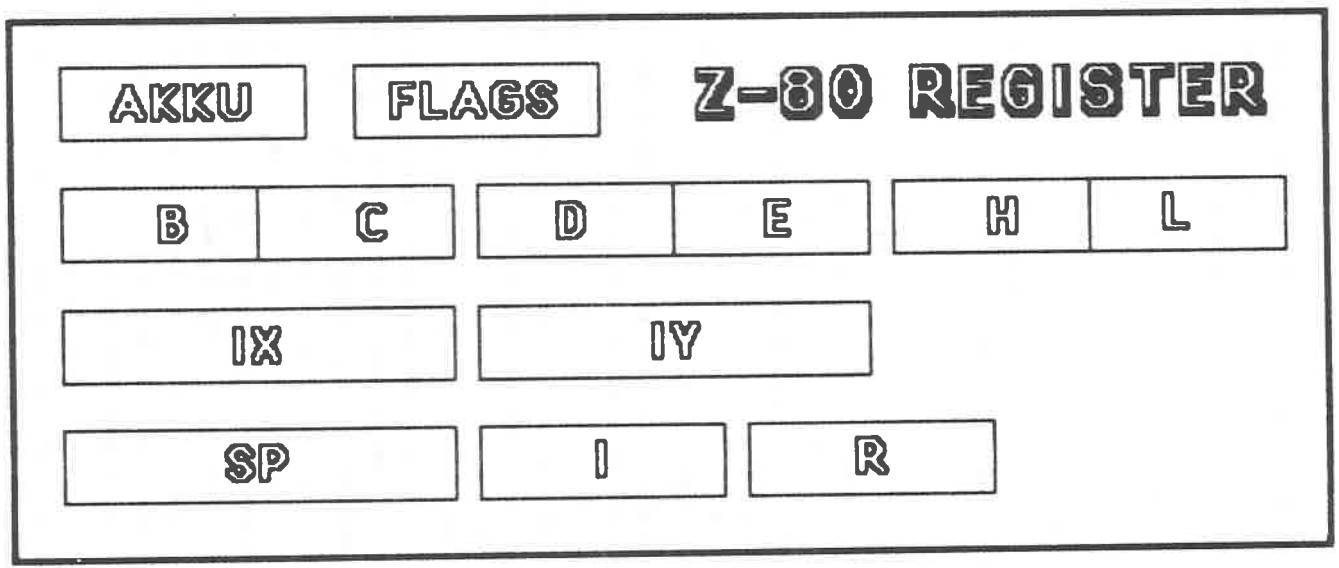


Abb. 9.

Akkumulator (kurz Akku oder nur A) ist ein 8-Bit-Register, er kann also 1 Byte aufnehmen und bearbeiten (eigentlich bearbeitet der Akku selbst nichts, die Ergebnisse werden nur in diesem Register abgelegt). Die meisten Arithmetikbefehle brauchen zwei Operanden (z.B. die Addition von zwei Zahlen). Der erste Operand steht vor dem Befehlsausführung schon im Akku, der zweite stammt aus einem anderen Register im Prozessor oder aus dem Speicher. Nach der Addition wird das Ergebnis wieder im Akku gespeichert.
Ein anderes Register mit Namen F speichert verschiedene Flags, die bestimmte Zustände des Prozessors wiederspiegeln. Anhand dieser Flags kann zum Beispiel festgestellt werden, ob der Akkuinhalt 0 ist.
Aber das sind noch nicht alle Register. Es gibt 6 weitere 8-Bit-Register mit einer besonderen Eigenschaft. Je zwei dieser Speicherzellen bilden zusammen ein 16-Bit-Register. Das Paar HL ist dabei schon fast ein 16-Bit-Akkumulator, d.h. es übernimmt die gleichen Aufgaben wie der normale Akku, dies aber jetzt für 16 statt 8 Bits. Somit wird die Bearbeitung größerer Zahlen vereinfacht.
Weitere Registerpaare sind BC und DE. Damit aber noch nicht genug. IX und IY sind zwei Indexregister (mit je 16-Bit), die als Zeiger auf bestimmte Zellen im Speicher weisen. Mittels dieser Zeiger können auf einfache Weise ganze Gruppen von Daten manipuliert werden. Wie das funktioniert, erkläre ich später.
Das 16-Bit-Register SP hat eine Spezialaufgabe. Es zeigt immer auf das oberste Element des Stapels (SP bedeutet auch Stack-Pointer). Immer wenn etwas auf den Stapel gelegt oder davon weggenommen wurde, aktualisiert der Z-80 den Zeiger, so daß er auf die neue Position zeigt.
Schließlich gibt es noch die Register I und R, die speziellen Zwecken (Hardwaresteuerung) vorbehalten sind.
Als ob das noch nicht genug wäre, gibt es zu jedem der Register A bis L noch ein Zweitregister, das mit dem Ursprungsregister vertauscht werden kann. Man kann aber immer nur mit einem Registersatz gleichzeitig arbeiten. Daher dienen die Zweitregister meist als kleiner Zwischenspeicher. In den Befehlen werden die Reserveregister mit einem nachgestellten Apostroph gekennzeichnet (in diesem

Buch aus technischen Gründen mit Anführungszeichen).
Damit kennen Sie bereits die für die Maschinensprache zur Verfügung stehenden Register im Z-80 (siehe Abb. 1). Im nächsten Abschnitt wird der Ablauf eines einzigen Maschinenbefehls im Prozessor erläutert.

13.4. DIE FUNKTIONSWEISE DES Z-80

Nehmen wir einmal an, im Speicher ihres Computers stehe ein Maschinenprogramm, das nur darauf wartet, ausgeführt zu werden. Natürlich muß sich der Mikroprozessor irgendwo merken, wo das Programm eigentlich steht. Dazu gibt es ein spezielles 16-Bit-Register im Z-80, genannt *Programmzähler* (engl. Program Counter = PC). In ihm ist die Adresse des Befehls gespeichert, der als nächster zur Ausführung kommt. Soll der Befehl jetzt durchgeführt werden, so holt der Prozessor das Byte aus der angegebenen Speicheradresse. Dieses Byte wird im Prozessor festgehalten und der Programmzähler um 1 erhöht, damit wir die Adresse des nächsten Bytes erhalten. Gleichzeitig wird der Opcode (um den handelt es sich nämlich bei dem Byte) dekodiert, d.h. der Z-80 stellt fest, welcher der vielen Befehle da eigentlich im Speicher steht. Einige Befehle haben einen Opcode, der mehrere Bytes lang ist (sonst könnten ja nur 256 Befehle erkannt werden). In diesem Fall werden die benötigten Bytes einfach nacheinander genau wie das erste aus dem Speicher geholt (der PC zeigt ja immer auf die aktuelle Speicherzelle, weil er nach jedem Byte erhöht wird). Ebenso kann es vorkommen, daß noch ein oder zwei Bytes Daten folgen. Auch diese werden eingelesen, müssen jedoch nicht dekodiert werden. Sie gelangen statt dessen in bestimmte Register (womit der Befehl schon beendet sein könnte), oder werden zur Bearbeitung irgendwo im Prozessor bereitgehalten.
Müssen die Daten noch irgendwie verändert werden (z.B durch Addition o.ä.), so findet diese Operation jetzt statt und das Ergebnis wird wieder abgespeichert (z.B im Akku). Damit

ist der Befehl beendet, der nächste kann gestartet werden.

Alle diese Vorgänge laufen natürlich nicht in nullkommanichts ab - auch Strom benötigt eben ein wenig Zeit zum Fließen. Im Normalfall benötigt jeder der Arbeitsschritte wie "Opcode holen", "Opcode decodieren", "Befehl ausführen" und "Ergebnis speichern" einen Taktzyklus. Deshalb faßt man 4 Takte auch oft zu einem sog. Maschinenzyklus zusammen. Komplexere Befehle können mehrere dieser Maschinenzyklen zur Ausführung benötigen.

Zusammenfassung: Der PC zeigt immer auf die Speicherzelle, die das nächste zu bearbeitende Byte enthält. Nacheinander werden Opcode(s) und Daten eingelesen. Der Opcode wird dekodiert, dann der Befehl ausgeführt.

13.5. DAS HEXADEZIMALSYSTEM

Wann immer Sie sich mit Maschinensprache beschäftigen, werden Sie auf die Zahlendarstellung im Hexadezimalsystem treffen. Dieses System besitzt im Gegensatz zu unserem herkömmlichen Dezimalsystem 16 Ziffern (0-9 und A-F für die Werte 10 bis 15). Es wird so häufig benutzt, weil die Umwandlung von Binär- in Hexzahlen sehr einfach ist. Ein weiterer Vorteil des Hexadezimalsystems ist es, daß eine Hexziffer genau ein halbes Byte darstellt (die größte zweistellige Hexzahl FF entspricht der Binärkombination 1111 1111, dem größtmöglichen Inhalt eines Bytes). Man nimmt daher jeweils ein Halbbyte und wandelt es in eine Hexziffer um. Die Tabelle zeigt die dezimalen und binären Entsprechungen:

Dez.	Hex.	Bin.
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Aus dem Byte 1010 1011_2 wird also die hexadezimale Zahl AB_{16} (da $1010_2 = A_{16}$ und $1011_2 = B_{16}$). Natürlich funktioniert das auch umgekehrt.

Für die Umwandlung von Hexzahlen in Dezimalzahlen werden zunächst alle Ziffern einzeln in das dezimale Äquivalent übersetzt. Die am weitesten rechts stehende Ziffer wird das mit 16^0=1, die zweite mit 16^1=16, die dritte mit 16^2=256 usw. multipliziert. Die erhaltenen Produkte werden dann addiert. Ein Beispiel:

$ABCD_{16}$ (entspricht 10,11,12,13)
= 10*16^3 + 11*16^2 + 12*16^1 + 13*16^0
= 10*4096 + 11*256 + 12*16 + 13*1
= 43981

Für den umgekehrten Weg (dez-hex) können Sie die Dezimalzahl durch 16 teilen und den entstehenden Divisionsrest als Hexziffer notieren. Das Ergebnis wird wieder durch 16 geteilt usw., bis es 0 wird. Auch hier ein Beispiel:

53000 / 16 = 3312 Rest 8 -) 8
3312 / 16 = 207 Rest 0 -) 0

```
  207 / 16 =   12 Rest 15 -) F
   12 / 16 =    0 Rest 12 -) C
=) 53000(10) = CF08(16)
```

Inzwischen gibt es Taschenrechner, die eine spezielle Funktion für die Basisumwandlung besitzen. Gute Assembler bzw. Hexmonitore bieten diese Funktion ebenfalls.

Auch Ihr CPC tut sich hier hervor, er besitzt nämlich eine BASIC-Funktion, die es ermöglicht, hexadezimale Zahlen zu bearbeiten. Diesen wird einfach ein & (oder genauer ein &H) vorgestellt, und schon wird die Zahl ins dezimale System umgewandelt.

13.6. BINÄRARITHMETIK

13.6.1. ADDITION

Um es gleich zu Anfang zu sagen: Die binäre *Addition* unterscheidet sich von der dezimalen nur im Zahlensystem, ansonsten funktioniert sie genauso.
Die Summen von zwei Nullen oder einer Null und einer Eins (egal in welcher Reihenfolge) bedürfen keiner Erläuterung, hier wird ganz normal addiert. Wollen wir jedoch 1 + 1 rechnen, so ergibt sich ein Problem. In der dezimalen Entsprechung wäre das Ergebnis eine 2. Die gibt es jedoch im binären System nicht. Also muß (wie beim Überschreiten der 9 im Dezimalsystem) ein *Übertrag* auf die nächste Stelle gemacht werden:

```
  0     0     1     1
+ 0   + 1   + 0   + 1
---   ---   ---   ---
  0     1     1    10
```

Auch ganze Bytes lassen sich sehr einfach verknüpfen. Hier wird einfach jede Stelle für sich addiert (und ein eventueller Übertrag beachtet):

```
  01101101 =  109
+ 00001001 = +  9

     1  1    (Überträge)
----------   ----
  01110110 =  118
```

Zur besseren Übersicht sind hier die Überträge aufgeführt worden.
Sollte es vorkommen, daß zwei Einsen addiert werden müssen und noch ein Übertrag dazukommt (1+1+1=3) so ist das Ergebnis 1 1 (eigentlich klar!)

Versuchen Sie einmal diese Addition:

```
    10010011
  + 11011111

   1  11111    (Überträge)
-------------
   101110010
```

Jetzt haben wir im Ergebnis plötzlich 9 Bits! Das neunte Bit heißt *Carry- oder Übertrags-Bit*. Es zeigt an, daß die Addition von zwei 8-Bit-Zahlen den zulässigen Bereich für ein Byte (0 - 255) überschritten hat, womit wir auch schon bei der 16-Bit-Addition sind. Kein Computer kommt mit nur 8-Bit für die Zahlendarstellung aus, die Zahlen haben meist einen viel größeren Bereich. Tatsache ist aber, daß ein 8-Bit-Mikroprozessor (wie der 6510) immer nur 8 Bits gleichzeitig verarbeiten kann. Besteht eine Zahl z.B. aus zwei Bytes, so muß die Addition nacheinander an beiden durchgeführt werden. Da bis auf den Übertrag die beiden Teile der Zahl völlig unabhängig voneinander addiert werden können, braucht man nur das Carry-Bit, um auch größere Zahlen zu bearbeiten. Es hat die Aufgabe, den Übertrag von der letzten Stelle des ersten Bytes zur ersten Stelle des zweiten Bytes zwischenzuspeichern.
Ein Beispiel:

```
    00110101 10010011
  + 10011011 11011111

     1111111   11111  (Überträge)
  --------------------
    11010001 01110010
```

Den rechten Teil der Addition kennen Sie bereits aus dem vorherigen Beispiel.

13.6.2. SUBTRAKTION

Wenn ein Computer eine Zahl von einer anderen subtrahieren will, so bildet er zunächst das negative Äquivalent dieser Zahl (d.h. er multipliziert mit -1) und addiert es dann. Dies läuft so ab, weil eine Addition und eine Negierung aus elektronischen Grundbausteinen (wie AND, OR, XOR, NOT) zusammengesetzt werden kann, nicht aber eine Subtraktion.
Um eine negative Zahl darzustellen, wird der Zahlenbereich eines Bytes von 0 - 255 nach -127 bis +127 verschoben. Das höchstwertige Bit (Bit 7) dient dann als Vorzeichen. Ist es auf 1, so haben wir eine negative Zahl vor uns, bei 0 ist das Byte positiv. Dabei kann aber zur Negierung einer Zahl nicht einfach Bit 7 gesetzt werden. Ein Beispiel verdeutlicht die Schwierigkeiten:

```
  00000001
+ 10000001
----------
  10000010
```

In Dezimalsystem übertragen würde dies bedeuten, daß 1 + (-1) = -2 ist. Deshalb wird ein anderer Weg gegangen. Ein Byte kann durch Bildung des sogenannten *Zweierkomplements* sehr einfach mit -1 multipliziert werden. Dazu werden alle Bits invertiert und zusätzlich eine 1 addiert.

```
Beispiel:    01011011
invertiert:  10100100
           +        1
           ----------
             10100101
```

Wenn wir nach diesem Schema 1 - 1 im binären System berechnen, so erhalten wir das richtige Ergebnis:

```
  00000001
+ 11111111

 11111111  (Überträge)
----------
 100000000
```

Wie Sie sehen, entsteht scheinbar ein Übertrag. Doch auch hier verhält sich die Subtraktion anders. Wir können es hier einfach ignorieren. Würden wir 16 Bit subtrahieren, so würde unser jetzt überflüssiges Carry-Bit dafür sorgen, daß die Stellen des zweiten Bytes auch auf 0 gesetzt würden. Das ist wichtig, da bei negativen 16-Bit-Zahlen alle 16 Stellen invertiert werden. Als Zwei-Byte-Zahl sähe -1 also so aus: 11111111 11111111. Fehlte das Carry-Bit jetzt, so lautete unser Ergebnis 11111111 00000000. Und das ist falsch!
Zum Glück ist die Programmierung einer Subtraktion nicht so kompliziert. Die Subtraktionsbefehle des Z-80 beinhalten bereits die Bildung des Zweierkomplements

13.6.3. MULTIPLIKATION

Auch wenn Sie es nicht glauben: Die Z-80-Maschinensprache hat nur zwei Rechenbefehle, und zwar für Addition und Subtraktion. Alle anderen Rechenarten werden aus diesen Grundbefehlen zusammengesetzt, meist als Unterprogramm.
Da wir nicht in allen Einzelheiten in die Maschinensprache einsteigen wollen (dazu gibt es bessere und ausführlichere Literatur), stelle ich Ihnen nur den einfachsten Algorithmus zur Multiplikation vor. Er wird von Profis nicht gern benutzt, da er nicht sehr effizient ist. Nun aber zur Sache. Um das Produkt x * n zu berechnen, genügt es, x n-mal zu addieren. Dies funktioniert natürlich nur bei ganzen Zahlen. Für Dezimalbrüche gibt es kompliziertere Verfahren, bei denen Zahlen z.B. Stelle für Stelle und nicht als Ganzes miteinander verknüpft werden, die aber im Prinzip ähnlich funktionieren.
Zum besseren Verständnis noch ein Beispiel:

```
4 * 3 = 4 + 4 + 4 = 12
```

13.6.4. DIVISION

Auch für die Division gibt es ein sehr einfaches Verfahren. Um x durch n zu teilen, wird einfach fortwährend n von x abgezogen. Die Anzahl der möglichen Subtraktionen, bis n größer x wird, gibt das Ergebnis der Division an. Hier ein Beispiel:

```
10 / 3 = ?
10 - 3 = 7    Zählregister= 1
 7 - 3 = 4    Zählregister= 2
 4 - 3 = 1    Zählregister= 3
=) 10 / 3 = 3 Rest 1
```

Diese Methoden sind möglich, weil die Maschinensprache so ungeheuer große Geschwindigkeiten erlaubt. Übrigens arbeitet auch ein Taschenrechner nach diesem Prinzip. Jedesmal, wenn Sie eine Rechentaste drücken, läuft ein kleines Maschinenprogramm (natürlich mit den erwähnten aufwendigen Algorithmen) ab.
Aus den 4 Grundrechenarten lassen sich dann noch höhere Funktionen (z.B. Potenzen, Sinus o.ä.) zusammensetzen. Auf diese Art und Weise kann jede mathematische Operation durch kleinste AND-, OR-, XOR- und NOT-Operationen ausgedrückt werden (da Addition und Subtraktion sich aus letzteren konstruieren lassen).

13.7. WIE FUNKTIONIEREN VERGLEICHE?

Im BASIC stellen Vergleiche nichts Ungewöhnliches dar. Doch wie kann man sie in Maschinensprache erzeugen? Sehen wird uns dazu einmal ein Beispiel an:
A = B (=) A - B = 0
Wie Sie sehen, kann ein Vergleich zwischen 2 Zahlen (hier A

und B) recht einfach umgeformt werden. Für den Computer hat diese Form den Vorteil, daß auf der rechten Seite der Gleichung eine 0 steht. Die 0 ist die einzige Zahl, von der der Mikroprozessor feststellen kann, ob sie gerade im internen Rechenregister (meist Accu genannt) steht oder nicht. Dazu werden einfach alle Bits miteinander ODER-verknüpft - etwa so:

Bit 7 OR Bit 6 OR Bit 5 OR Bit 4 OR Bit 3 OR Bit 2 OR Bit 1 OR Bit 0

Wenn alle 8 Bits des Accus auf 0 waren, so ist das Ergebnis dieser Verknüpfungskette eine 0, in allen anderen Fällen (d.h. wenn mindestens ein Bit auf 1 ist) ist das Ergebnis 1. So kann der Mikroprozessor angeben, ob das Rechenregister (wo fast immer das Ergebnis der letzten Operation steht) gleich oder ungleich 0 ist - voila, die ersten beiden Vergleiche sind erzeugt. Für einen Vergleich A=B oder A ungleich B brauchen wir also nur die beiden Zahlen voneinander zu subtrahieren und dann festzustellen, ob der Inhalt des Accus 0 ist. Dies können Sie mittels des *Z-Flags* (Z steht für Zero). Ist es auf 1, so ist das Ergebnis der letzten Operation 0 gewesen. Ist Z=0, so war das letzte Ergebnis ungleich 0. Das Flag ist also nicht nur auf den Akku beschränkt, andererseits verändern aber auch nicht alle Befehle die Flags. Ob das geschieht, können Sie in der Befehlsliste im Anhang nachlesen. Doch nun zurück zu den Vergleichen.
Bei "größer" und "kleiner" gehen wir fast wie bei "gleich" vor. Nach der Subtraktion sehen wir nach, ob die Zahl im Accu kleiner oder größer 0 ist, erkennbar am Vorzeichenbit:

A größer B (=) A - B größer 0 (erfüllt, wenn Bit 7 = 0)
A kleiner B (=) A - B kleiner 0 (erfüllt, wenn Bit 7 = 1)

Das *Vorzeichenbit* wird von vielen Befehlen in das *S-Flag* übertragen (S bedeutet "sign" = Vorzeichen). Dort kann es mittels spezieller Befehle einfach abgefragt werden.
Ein weiteres Flag heißt P/V (da wir es in diesem Buch kaum

brauchen werden, bezeichne ich es einfach und kürzer mit P). Es hat zwei Funktionen. Zum einen kann es die Parität (gerade oder ungerade; was das ist, brauchen Sie nicht zu wissen) anzeigen, zum anderen meldet es nach einigen arithmetischen Operationen, ob das Vorzeichenbit fehlerhaft verändert wurde (auch das braucht uns hier nicht zu interessieren; wir möchten ja nur in die Maschinensprache hineinriechen).

13.8. DAS ERSTE PROGRAMM

Nachdem Sie die entscheidenden Grundlagen der Maschinenprogrammierung kennengelernt haben, werden wir jetzt mit dem einfachsten Programm anfangen. Entwerfen wir also ein *Additionsprogramm* für zwei 8-Bit-Zahlen.

Zunächst müssen wir dem Programm irgendwie die beiden zu addierenden Zahlen mitteilen. Eine Art INPUT-Befehl gibt es in der Maschinensprache nicht, deshalb behelfen wir uns damit, die beiden Zahlen irgendwo im Speicher abzulegen. Von dort kann sich das Programm dann die Werte selbst holen. Das ist auch schon die erste Aufgabe, die unser Programm erledigen soll. Der Befehl "LD A,(nnnn)" wirkt fast wie ein PEEK, er holt das Byte unter der Adresse nnnn in den Akkumulator.

Von dem zweiten Byte wissen wir ebenfalls, wo es steht. Wir können es aber nicht mit LD B,(nnnn) in den Prozessor holen, diesen Befehl gibt es in der Maschinensprache leider nicht. Es ist aber möglich, das Registerpaar HL als Zeiger auf unser Byte einzusetzen. Dazu bringen wir die Adresse mittels LD HL,nnnn in die gewünschten Register. Beachten Sie, daß der Ausdruck "nnnn" jetzt nicht mehr mit Klammern umschlossen ist. Das zeigt uns an, daß dieser Ausdruck direkt in HL geladen werden soll und nicht als Adresse für den eigentlichen Wert steht.

Mit dem nächsten Befehl sollen die beiden Bytes endlich addiert werden. Er lautet "ADD A,(HL)" und bewirkt, daß der Wert aus dem Akkumulator und das Byte, dessen Adresse in HL gespeichert ist, addiert werden. Das Ergebnis daraus wird wieder im Akku abgelegt. Sollte die Summe aus beiden Zahlen den Wertebereich eines Bytes (0-255) überschreiten, so zeigt der Z-80 das an, indem er das Übertragsbit auf 1 setzt.

Da uns das Ergebnis im Akkumulator herzlich wenig nützt, soll es mit einem weiteren Befehl in eine Speicherzelle befördert werden, von wo es mittels PEEK gelesen werden

kann. Dies erledigt "LD (nnnn),A". Dieser Befehl funktioniert genau wie "LD A,(nnnn)", nur in umgekehrter Richtung.

Schließlich beendet RET das Unterprogramm (genau wie RETURN in BASIC). Dazu müssen Sie wissen, daß der Interpreter *Maschinenprogrammaufrufe* durch CALL wie Unterprogramme behandelt; der letzte Befehl in einer Maschinenspracheroutine muß also immer RET sein, sonst hängt sich der Rechner auf.

Wir haben zwar bisher ganz schön drauflos programmiert, uns aber dabei nicht um die Speicherzellen gekümmert, in denen das stattfinden soll. In unserem Beispiel können Sie die Adressen fast beliebig wählen, nur sollten Sie sicherstellen, daß diese nicht vom Computer schon anderweitig verplant werden (das können Sie mit MEMORY verhindern). Mein Vorschlag: Sie begrenzen den BASIC-Speicher auf &AAFF. So können Sie alle Bytes von &AB00 bis &AB7F nutzen. Unser Programm setzen wir ab &AB00 in den Speicher, die zu addierenden Werte und das Ergebnis legen wir am besten ans Ende des (ehemaligen) BASIC-Bereichs. Dann sieht unser Programm so aus:

Adresse	Befehl	Kommentar
AB00	LD A,(AB7F)	Erste Zahl aus AB7F laden
AB03	LD HL,AB7E	Zweite Zahl steht in AB7E
AB06	ADD A,(HL)	Akku und zweite Zahl addieren
AB07	LD (AB7D),A	Ergebnis abspeichern
AB0A	RET	Programmende

An den Adressen sehen Sie, daß die Befehle unterschiedlich viele Bytes belegen. Befehle, die eine Adressenangabe beinhalten, benötigen mindestens 3 Bytes, andere wie z.B. "ADD A,(HL)" sind Einbytebefehle.

Bevor Sie die Bytes in den Speicher POKEn können, müssen Sie wissen, wie diese aussehen. In der Tabelle mit den Z-80-Befehlen können Sie für jeden Befehl nachschlagen welchen Opcode er hat. Für "LD A,(nnnn)" ist es 3A plus zwei

Adressbytes. Dabei ist zu beachten, daß das Lowbyte immer vor dem Highbyte der Adresse steht. Der komplette Befehl lautet also in Zahlen ausgedrückt:

```
3A 7F AB
```

Hier die Codes für den Rest des Programms:

```
21 7E AB  (LD HL,AB7E)
86        (ADD A,(HL))
32 7D AB  (LD (AB7D),A)
C9        (RET)
```

Diese Werte müssen in die entsprechenden Speicherzellen gePOKEd werden. Diese Aufgabe übernimmt das folgende BASIC-Programm:

```
10 MEMORY &AAFF
20 FOR I = &AB00 TO &AB0A: READ A: POKE I,A: NEXT
30 DATA &3A,&7F,&AB,&21,&7E,&AB,&86,&32,&7D,&AB,&C9
```

Die Maschinenroutine kann mit CALL &AB00 gestartet werden. Vorher müssen noch die beiden zu addierenden Zahlen mittels POKE &AB7F, Z1 und POKE &AB7E, Z2 eingespeichert werden. Nach dem CALL meldet Ihnen PRINT PEEK(&AB7D) das Ergebnis. Damit Sie auf einfache Weise verschiedene Werte ausprobieren können, hängen Sie bitte folgende Zeilen an das obige BASIC-Programm an:

```
40 INPUT "Zahl 1, Zahl 2"; Z1,Z2
50 POKE &AB7F, Z1: POKE &AB7E, Z2
60 CALL &AB00: PRINT PEEK(&AB7D): GOTO 40
```

Die *Subtraktion* wird genauso programmiert, Sie müssen lediglich den ADD-Befehl durch SUB (HL) ersetzen. In der DATA-Zeile ändert sich dadurch der 7. Wert von &86 in &96

13.9. WIE WIRD EINE SCHLEIFE PROGRAMMIERT?

Wollen Sie im BASIC einen Vorgang mehrfach wiederholen, so haben Sie zwei Möglichkeiten, das zu programmieren: FOR-NEXT und WHILE-WEND. Eine dritte Möglichkeit ist nicht so komfortabel und wird deshalb auch nicht oft benutzt. Es ist ohne weiteres machbar, am Ende eines jeden Durchlaufs eine Zählvariable um 1 zu erhöhen und mittels IF-THEN an den Anfang zurückzuspringen, wenn ein weiterer Durchlauf nötig sein sollte. So unkomfortabel dieses Verfahren auch erscheint, es ist die einzige Möglichkeit, die uns die Maschinensprache für solche Zwecke bietet. Statt einer Variable benutzen wir ein Prozessorregister und die Schleifenlänge wird von oben nach unten heruntergezählt, der Rest funktioniert genauso. Das untenstehende Programm tut nichts anderes als 255 Durchläufe lang zu warten (da die Schleife leer ist). Ich habe keinen BASIC-Lader angegeben, da die Wirkung nicht sichtbar wird. Die Maschinensprache arbeitet so schnell, daß die Schleife in Bruchteilen einer Sekunde abgearbeitet wird. Hier das Programm:

```
AB00  LD B,FF      Schleifenlänge laden
AB02  DEC B        B um 1 vermindern
AB03  JP NZ,AB02   Springe, wenn ungleich 0
AB06  RET          sonst Programmende
```

Der erste Befehl bringt die Schleifenlänge in das Register B. Diese Zahl ist fest in das Programm eingebaut, sie steht also direkt nach dem Opcode. Sie werden sich vielleicht fragen, warum ausgerechnet Register B für den Zähler gebraucht wird. Nun, der Akku ist für Rechenoperationen prädestiniert, und auf ähnliche Weise gibt es spezielle Zählbefehle für das B-Register.

Der DEC-Befehl (Dekrement = um 1 vermindern) bewirkt, daß eine 1 von B subtrahiert wird. Sollte das Ergebnis daraus 0 sein, so wird das Zero-Flag auf 1 gesetzt, ansonsten bleibt es 0. Dieses Verhalten nutzt der nächste Schritt aus. JP NZ springt nur dann zur angegebenen Adresse, wenn das Z-Flag auf 0 ist, der Rücksprung findet also nur statt, wenn das

Schleifenende noch nicht erreicht ist. Ist Z auf 1, so macht der Z-80 mit dem nächsten Befehl weiter. Der heißt bei uns RET und beendet das Programm.

13.10. WEITERE ARITHMETIKROUTINEN

13.10.1 16-BIT-ADDITION

Mit 8-Bit-Zahlen können - wie schon gesagt - nur Zahlen bis 255 bearbeitet werden. Bei 16 Bits sind es schon Zahlen aus dem Bereich bis 32767. Der Z-80 gehört zu den wenigen 8-Bit-Prozessoren, die auch Befehle für die 16-Bit-Arithmetik liefern, auch besitzt er die dazu notwendigen 16-Bit-Register.

Für eine 16-Bit-Addition sollte die erste Zahl ins Registerpaar DE geladen werden. Mit "LD DE,(nnnn)" wird das Byte nnnn in Register D gebracht, E erhält den Wert von der Speicherzelle mit der Adresse nnnn+1. Auch hier wird also das "Pointerformat" Low-High benutzt. Der gleiche Befehl existiert auch für HL.
Der Additionsbefehl für 16 Bits lautet "ADD HL,DE". Wie Sie richtig vermuten, werden damit die beiden Zahlen aus DE und HL addiert und das Ergebnis in HL abgelegt. Von da wird es mittels "LD (nnnn),HL" in den Speicher gebracht.
Das ganze Programm sieht so aus:

```
AB00 LD DE,(AB7E)
AB04 LD HL,(AB7C)
AB07 ADD HL,DE
AB08 LD (AB7A),HL
AB0B RET
```

Beachten Sie, daß jeder der LD-Befehle zwei Bytes gleichzeitig bearbeitet. Das Ladeprogramm lautet:

```
10 DATA &ED,&5B,&7E,&AB
11 DATA &2A,&7C,&AB
12 DATA &19
13 DATA &22,&7A,&AB
14 DATA &C9
20 FOR i = &AB00 to &AB0B: READ a: POKE i,a: NEXT
30 INPUT "Zahl 1, Zahl 2"; z1,z2
40 POKE &AB7E, z1 AND 255: POKE &AB7F, INT(z1/256)
50 POKE &AB7C, z2 AND 255: POKE &AB7D, INT(z2/256)
60 CALL &AB00: PRINT PEEK(&AB7A) + 256 * PEEK(&AB7B)
70 GOTO 50
```

Auch hier können Sie wieder mit verschiedenen Werten experimentieren.

13.10.2. MULTIPLIKATION

Um zwei Zahlen zu multiplizieren, muß man mehrfach addieren (das haben wir schon in Kap. 13.6.3. festgestellt). In BASIC würde dieser Algorithmus so umgesetzt werden:

```
10 INPUT "Zahl 1, Zahl 2"; z1,z2: e=0
20 FOR i = 1 TO z1
30 e = e + z2: NEXT
40 PRINT "Ergebnis ";e
```

Wie Sie sehen, enthält das Programm im wesentlichen nur eine Schleife und eine Addition. Beides haben wir schon in Maschinensprache programmiert.

Die Multiplikation von zwei 8-Bit-Zahlen hat eine 16-Bit-Zahl als Ergebnis. Daher empfiehlt es sich, die 16-Bit-Additions-Routine entsprechend abzuwandeln.
Innerhalb der Schleife muß nur der Additionsbefehl stehen. Alle anderen Befehle dienen nur der Voreinstellung von Registern, bzw. dem Abspeichern des Ergebnisses. Mit dem ersten Befehl (siehe Listing) wird Zahl 1 in den Akku

geladen. Sie soll eigentlich in Register B landen, da aber B nicht direkt aus dem Speicher geladen werden kann, holen wir die Zahl zunächst in den Akku und bringen sie dann mit einem zweiten Befehl nach B. Damit steht die Schleifenlänge fest. Die andere 8-Bit-Zahl wird immer wieder zum Registerpaar HL addiert. Darum gelangt sie (ebenfalls über den Umweg) in Register E.
Da ADD HL,DE immer auch das D-Register mitaddiert, muß es vorher auf 0 gesetzt werden. Auch HL muß vor Beginn der Schleife 0 sein. Das übernehmen die nächsten beiden Befehle.

Dann beginnt die Schleife mit dem ADD-Befehl. Da das Registerpaar HL nur vom Additionsbefehl verändert wird, enthält es immer das letzte Zwischenergebnis.
Den Rest des Maschinenprogramms können Sie sich wahrscheinlich selbst erklären. DEC B und JP NZ,AB0D markieren das Schleifenende, LD (AB7C),HL speichert das Ergebnis ab und RET beendet das Programm. Hier das Listing:

```
AB00 LD A,(AB7F)   Zahl 1
AB03 LD B,A        nach B
AB04 LD A,(AB7E)   Zahl 2
AB07 LD E,A        nach E
AB08 LD D,00       D löschen
AB0A LD HL,0000    HL löschen
AB0D ADD HL,DE     addieren
AB0E DEC B         dekrementieren
AB0F JP NZ,AB0D    Rücksprung, wenn nicht 0
AB12 LD (AB7C),HL  Ergebnis speichern
AB15 RET           Rückkehr nach BASIC
```

Natürlich gibt es auch zu diesem Programm wieder ein BASIC-Ladeprogramm:

```
10 DATA &3A,&7F,&AB
11 DATA &47
12 DATA &3a,&7E,&AB
13 DATA &5F
14 DATA &16,&00
15 DATA &21,&00,&00
```

```
16 DATA &19,&05
17 DATA &C2,&0D,&AB
18 DATA &22,&7C,&AB
19 DATA &C9
20 FOR i= &AB00 TO &AB15: READ a: POKE i,a: NEXT
30 INPUT "Zahl 1, Zahl 2"; z1,z2
40 POKE &AB7F, z1: POKE &AB7E, z2
50 CALL &AB00: PRINT PEEK(&AB7C)+256*PEEK(&AB7D)
60 GOTO 30
```

13.11. NÜTZLICHE MASCHINENROUTINEN

In diesem Abschnitt möchte ich das Angenehme mit dem Nützlichen verbinden und Ihnen einige Maschinenroutinen vorstellen, die nicht nur dem Erlernen der Maschinensprache dienen, sondern auch einen praktischen Nutzwert haben.

Beginnen wir mit etwas Destruktivem, dem *Löschen von ganzen Speicherbereichen.* Im BASIC können Sie das mit einer FOR-NEXT-Schleife und POKE-Befehlen lösen, doch das dauert relativ lange. Diese Aufgabe können Sie auch der sehr viel schnelleren Maschinensprache übertragen. Allerdings muß hier die Einschränkung gelten, daß maximal 256 Bytes in einem Arbeitsgang gelöscht werden können (es sei denn, die Routine wäre so aufgebaut, daß Sie sie nicht verstehen).

Ähnlich dem SAVE-Befehl für bestimmte Speicherbereiche müssen bei unserem Programm nur die Anfangsadresse und die Zahl der zu löschenden Bytes abgegeben werden.
HL dient als Zeiger auf die jeweils zu löschenden Bytes und wird in jedem Schleifendurchlauf inkrementiert (um 1 erhöht). Register B enthält wieder in altbewährter Weise die Schleifenlänge (und damit die Anzahl der zu löschenden Bytes). In der Schleife wird zunächst das Byte, auf das HL zeigt, mit 0 geladen. Dann werden noch die beiden Register aktualisiert und nötigenfalls springt JP NZ,(xxxx) wieder an den Schleifenanfang.

Damit wäre unser Programm eigentlich schon zuende, doch wir bauen noch ein wenig Luxus ein. Am Ende der Schleife enthält HL die Adresse des Bytes, das als nächstes gelöscht würde, wäre die Schleife noch nicht abgearbeitet. Diese Adresse wird zurück in den Speicher an die Stelle gebracht, wo der erste Befehl sich die Anfangsadresse abholt. Wollen Sie jetzt weiterlöschen, so brauchen Sie nicht mehr mühselig die Startadresse einPOKEn, sondern können sich auf den CALL-Befehl beschränken.

Ein ähnliches Verfahren können Sie für die Schleifenlänge

anwenden. Da die Speicherzelle für diese Zahl vom Programm nicht verändert wird und der Wert auch immer gleich bleiben kann, entfällt dieser POKE ebenfalls nach dem ersten Aufruf. Wollen Sie also in 10er-Schritten 100 Bytes löschen, so benutzen Sie einfach diese Befehlsfolge:

```
POKE &AB7F, 10: POKE &AB7D, Lowbyte: POKE &AB7E, Highbyte
FOR i = 1 TO 10: CALL &AB00: NEXT
```

So ist das zwar nicht sehr sinnvoll, denn das ließe sich mit einem einzigen Aufruf und Schleifenlänge 100 besser lösen. Doch wenn Sie weitere Befehle innerhalb der FOR-NEXT-Schleife einfügen (z.B. Warten auf Tastendruck), läßt sich daraus unter anderem ein schrittweises Löschen des Bildschirmspeichers konstruieren. Außerdem ist so ein Löschen von langen Speicher-Blöcken machbar.

Zur Schleifenlänge ist noch ein weiterer Punkt anzumerken. Wenn eine Routine mit der Schleifenlänge 0 aufgerufen wird, so werden 256 Bytes gelöscht, weil vor dem ersten JP (also vor der ersten möglichen Beendigung) B dekrementiert wird. 0 minus 1 ergibt für den Z-80 (und alle anderen Mikroprozessoren) 255 (sehen Sie sich mal die Binärdarstellung an und addieren Sie dann 255 und 1). Deshalb ist die Schleife jetzt noch nicht zuende, es werden noch weitere 255 Bytes gelöscht.

Das Programm sieht dann so aus:

```
AB00 LD HL,(AB7D)
AB03 LD A,(AB7F)
AB06 LD B,A
AB07 LD (HL),0
AB09 INC HL
AB0A DEC B
AB0B JP NZ,AB07
AB0E LD (AB7D),HL
AB11 RET
```

```
10 DATA &2A,&7D,&AB
11 DATA &3A,&7F,&AB
12 DATA &47
13 DATA &36,&00
14 DATA &23
15 DATA &05
16 DATA &C2,&07,&AB
17 DATA &22,&7D,&AB
18 DATA &C9
20 FOR i = &AB00 TO &AB11: READ a: POKE i,a: NEXT
30 INPUT "Startadresse";a
40 POKE &AB7D, a-INT(a/256): POKE &AB7E, INT(a/256)
50 INPUT "Länge";b
60 POKE &AB7F, b
70 CALL &AB00: GOTO 20
```

Die zweite Routine dient zum *Kopieren von Speicherblöcken*. Da wir einen speziellen Z-80-Befehl verwenden können, ist es kürzer als das vorherige Programm, und die Beschränkung auf 256 Bytes entfällt.

Der Spezialbefehl heißt LDIR. Er kopiert ohne weiteres Zutun unsererseits ganze Speicherblöcke beliebiger Länge. Diese Länge muß vorher im Registerpaar BC gespeichert sein. Außerdem muß der Z-80 noch die Anfangsadressen des zu kopierenden Bereichs in HL und des Zielbereichs in DE vorfinden.
Dann kann LDIR in Aktion treten. Er kopiert das Byte, das durch HL adressiert wird, an die Adresse aus DE. Dann werden HL und DE inkrementiert, sie zeigen also jeweils auf das nächste Byte. Außerdem dekrementiert der Z-80 noch BC. Solange BC ungleich 0 ist, wird der LDIR-Befehl automatisch wiederholt. Ein einziger Befehl ersetzt also eine ganze Schleife!

Alles weitere bedarf dann wohl keiner Erläuterung mehr. Dieses Programm wurde übrigens für einen etwas anderen Speicherbereich geschrieben. Daher überschneidet es sich nicht mit der Löschroutine und kann gleichzeitig mit Ihr

benutzt werden.

```
AB20 LD HL,(AB6E)
AB23 LD DE,(AB6C)
AB27 LD BC,(AB6A)
AB2B LDIR
AB2D RET
```

```
10 DATA &2A,&6E,&AB
11 DATA &ED,&5B,&6C,&AB
12 DATA &ED,&4B,&6A,&AB
13 DATA &ED,&B0
14 DATA &C9
20 FOR i = &AB20 TO &AB2D: READ a: POKE i,a: NEXT
30 INPUT "Quellbereich";a
40 POKE &AB6E, a-INT(a/256): POKE &AB6F, INT(a/256)
50 INPUT "Zielbereich";b
60 POKE &AB6C, b-INT(b/256): POKE &AB6D, INT(b/256)
70 INPUT "Länge";c
80 POKE &AB6A, c-INT(c/256): POKE &AB6B, INT(c/256)
90 CALL &AB20: GOTO 30
```

13.12. DIE ADRESSIERUNGSMÖGLICHKEITEN

Wenn Sie sich die Befehlsliste einmal angesehen haben, so ist Ihnen sicher aufgefallen, daß der Operand nicht nur aus Registern wie A, B, C usw. besteht, sondern auch Klammern und ähnliches vorkommen. Bei einem Befehl wie ADD A,B ist es eigentlich klar, was er bewirkt; hier werden die Register A(kku) und B addiert. Was aber bedeutet ADD A,(HL) ?

Allgemein sollten Sie sich merken, daß ein Ausdruck in Klammern die Adresse eines Bytes im Speicher darstellt und nicht den Operand selbst. "(nn)" bedeutet also "Inhalt der Speicherzelle nnnn" (in der Kürzelschreibweise stellt ein Buchstabe ein Byte dar; nn steht also für zwei Bytes = 16 Bits = 4 Hexziffern). Diese Methode nennt man auch *absolute Adressierung.* Fehlen die Klammern, so werden die Werte direkt als Operand übernommen. ADD A,n addiert also das Byte nn (das direkt nach dem Opcode im Speicher steht, quasi eine Konstante) zum Akku. Diese Adressierungsart heißt *direkt.*
Eine weitere Methode ist die *indirekte Adressierung.* Indirekt bedeutet, daß der Operand nicht die Adresse der zu bearbeitenden Daten angibt, sondern den Ort, wo diese steht. Der Befehl ADD A,(HL) arbeitet also in der Weise, daß der Prozessor zuerst "nachsieht", welcher Wert im Register HL gespeichert ist. Dieser Wert dient als Zeiger auf die Speicherzelle, in der das Byte gespeichert ist, das dann schließlich zum Akku addiert wird. "Ganz schön umständlich" werden Sie sagen. "Ganz schön pfiffig" sagten die Z-80-Entwickler. Mittels dieser indirekten Adressierung können nämlich ganze Blöcke von Bytes wie ein Array angesprochen werden.
Eine Spielart der gerade erklärten Methode stellt die *indirekt - indizierte Adressierung* dar. Der Operand "(IX+d)" bedeutet in diesem Fall, daß zum Wert des Registers IX noch die Konstante dd addiert wird, und erst das Ergebnis dieser Addition die endgültige Adresse nennt.
Die letzte im Bunde ist die *relative Adressierung*, die nur bei einer bestimmten Sorte von Sprüngen zu finden ist. Ähnlich der indizierten Adressierung folgt dem Opcode eine

Konstante. Diese wird zum Programmzähler addiert. Ist das Bit 7 des Bytes auf 1, so wird es als negative Zahl behandelt (was übrigens auch für die indirekt-indizierte Adressierung gilt). Das Ergebnis ist dann ein Rückwärtssprung. Weil der Programmzähler nach dem Einlesen der Konstanten schon auf den nächsten Befehl zeigt, kann um maximal 129 Schritte vorwärts und um maximal 127 Schritte rückwärts gesprungen werden.

13.13. DIE BEFEHLE DES Z-80

Wenn Sie sich den Anhang mit den Z-80-Opcodes ansehen (dort sind alle Befehle mit Ihren Codes und den beeinflußten Flags aufgeführt), so werden Sie feststellen, daß es mehrere Gruppen von Befehlen gibt, die sich nur durch den Operanden (also die Adressierungsart) unterscheiden. Diese Gruppen werden im folgenden beschrieben. Sie müssen nicht alle Beschreibungen auswendig lernen; sie sind nur zum schnellen Nachschlagen und als Überblick über die großen Möglichkeiten der Maschinensprache gedacht. Wenn Sie später in Maschinensprache programmieren, können Sie die Opcodetabelle zum Nachschlagen benutzen.

Da die Befehle für verschiedene Operanden immer gleich funktionieren, werden letztere in den Beschreibungen durch Platzhalter (z.B. x, n usw.) vertreten.

ADC A,X

Der Operand x wird zum Akku addiert. Zusätzlich wird auch noch ein evtl. vorhandener Übertrag aus dem Carrybit beachtet. Das Ergebnis wird wieder im Akku abgelegt, ein neuer Übertrag im Carrybit gespeichert. Nach der Ausführung sind die Flags entsprechend dem Akkuinhalt gesetzt.

ADC HL,X

Dieser Befehl funktioniert wie der vorherige, nur bewirkt er eine 16-Bit-Addition. Dazu werden der Operand (jetzt ein 16-Bit-Register) und das Carry-Bit zum Registerpaar HL addiert. Das Ergebnis wird in HL gespeichert, die Flags entsprechend gesetzt.

ADD A,X

Der Operand wird zum Akku addiert und das Ergebnis dort wieder abgelegt. Das Carry-Bit wird nicht beachtet, ist aber nach der Addition wie alle anderen Flags entsprechend dem Ergebnis gesetzt.

ADD HL,X

wie ADC HL,x, das Carry-Bit geht jedoch nicht in die Addition ein.

ADD IX,X

Der Operand wird zum Indexregister IX addiert, das Ergebnis dort wieder abgespeichert und das Carry-Bit aktualisiert. Die anderen Flags werden jedoch nicht verändert.

ADD IY,X

wie ADD IX,x, jedoch für Indexregister IY

AND X

Der Operand und der Akku werden UND-verknüpft, das Ergebnis wird wieder im Akku gespeichert. Anhand des Ergebnisses werden die Flags aktualisiert. Das Carry-Bit wird 0 (AND liefert keinen Übertrag).

BIT N,X

Das Bit n des Operanden x wird getestet, d.h. es wird komplementiert und in das Z-Flag übertragen. War das gewählte Bit auf 1, so ist Z=0, war das Bit auf 0, so wird Z=1. Die Flags S und P erhalten zufällige Werte!

CALL NN

Das Unterprogramm ab Adresse nnnn wird aufgerufen (ähnlich GOSUB).

CALL BEDINGUNG,NN

Das Unterprogramm nnnn wird nur dann angesprungen, wenn die Bedingung erfüllt ist. Als Bedingung kann jedes der vier Flags S, Z, P und C(arry) getestet werden. Je nach Zustand wird dann verzweigt oder mit dem nächsten Befehl im Programm fortgefahren.

CCF

Komplementiert das Carry-Bit (1-)0, 0-)1).

CP X

Der Operand x wird mit dem Akku verglichen, d.h. er wird subtrahiert, das Ergebnis jedoch nicht abgespeichert, so daß der Akkuinhalt erhalten bleibt. War der Operand kleiner als der Akku, so wird S=1, war er größer oder gleich, so wird S=0, waren beide Werte gleich, so ist Z=1, bei Ungleichheit Z=0.

CPD

HL dient als Zeiger auf eine Speicherstelle, die mit dem Akku verglichen wird. Wenn beide Werte gleich waren, so ist Z=1, wenn der Akku größer oder gleich war, ist S=0, bei S=1 war der Akku kleiner als das Byte aus dem Speicher. Dann werden HL und BC dekrementiert (um 1 vermindert). Ist der Inhalt von BC=0, so wird das P-Flag auf 0 gebracht, sonst ist es auf 1. Daher kann BC als Bytezähler dienen.

CPDR

Wie CPD, die Ausführung wird jedoch automatisch solange wiederholt, bis Gleichheit festgestellt wurde (Z=1) oder BC=0 (P-Flag auf 0) ist.

CPI

Wie CPD, HL wird jedoch nicht dekrementiert, sondern inkrementiert (um 1 erhöht).

CPIR

Wie CPDR, HL wird jedoch inkrementiert.

CPL

Die Bits des Akkus werden invertiert (aus 0 wird 1 und umgekehrt).

DAA

Nach einer arithmetischen Operation (ADC, ADD, DEC, INC, NEG, SBC, SUB) werden evtl. vorhandene falsche BCD-Ziffern im Akku und die Flags entsprechend korrigiert.

DEC X

Die angegebenen Operanden werden um 1 vermindert und

zurückgespeichert. Außer bei den Registern BC, DE, HL, IX, IY und SP werden dann noch die Flags dem Ergebnis entsprechend gesetzt.

DI

Nach Ausführung dieses Befehls ignoriert der Z-80 alle Interruptanforderungen (maskierbare Interrupts).

DJNZ E

B wird um 1 vermindert. Falls der Inhalt von B ungleich 0 ist, wird der relative Sprungabstand e zur Adresse des nachfolgenden Befehls addiert und dann das Programm an der erhaltenen neuen Adresse fortgesetzt.

EI

Nach diesem Befehl sind die maskierbaren Interrupts wieder freigegeben, d.h. nach einer Interruptanforderung verzweigt der Prozessor in spezielle Routinen.

EX X,Y

Die beiden angegebenen Operanden werden miteinander vertauscht. X kann auch für (SP) stehen, dann wird der zweite Operand mit dem obersten Stapelelement vertauscht.

EXX

Die Register BC, DE und HL werden mit den Zweitregistern vertauscht.

HALT

Der Z-80 hält die Programmausführung solange an, bis ein Interrupt angefordert wird.

IM N

Interruptmodus auswählen.
Modus 0: Der Baustein, der den Interrupt anfordert, muß auf dem Datenbus einen Befehl zur Ausführung bereithalten.
Modus 1: Der Z-80 springt zur Adresse 0038 (hexadezimal) und führt dort eine Interruptbehandlungsroutine aus.
Ihr CPC arbeitet ebenfalls in diesem Modus. Sie sollten den Interruptmodus auch niemals ändern, weil Ihr Rechner dann

nicht mehr richig arbeitet.
Modus 2: Der Baustein liefert die untere Hälfte einer Adresse, deren obere Hälfte im Register I gespeichert ist. Unter der angegebenen Adresse ist ein Zeiger auf den Start der Interruptroutine abgelegt.

IN X,(C)
Register C gibt den Port an, von dem ein Byte in das Register unter x gelesen wird.

IN A,(N)
Der Akku wird mit einem Byte von Port n geladen.

INC X
Wie DEC, jedoch wird der Operand um 1 erhöht.

IND
Das Register HL dient als Zeiger auf eine Speicherstelle, in die ein Byte vom Port gelesen wird. Register C gibt die Nummer des Ports an. Außerdem werden HL und B dekrementiert. Sollte B=0 sein, so wird das Z-Flag auf 1 gesetzt. Die übrigen Flags (außer Carry) erhalten zufällige Werte.

INDR
Wie IND, der Befehl wird jedoch solange wiederholt, bis B=0 ist.

INI
wie IND, HL wird jedoch inkrementiert.

INIR
wie INDR, HL wird jedoch inkrementiert.

JP NN
das Programm springt zur Adresse nnnn (ähnlich GOTO).

JP BEDINGUNG,NN
springt, wenn die Bedingung erfüllt ist (für Bed. siehe CALL).

JP X

Springt zur Adresse, die im Operand x (HL, IX, IY) gespeichert ist.

JR N UND JR BEDINGUNG,N

Wie die entsprechenden JP-Befehle. Es wird jedoch relativ gesprungen, d.h. das dem Opcode folgende Byte (n) wird zum PC addiert.

LD X,Y

Diese Gruppe umfaßt die weitaus meisten Befehle. Alle folgen jedoch einem Prinzip. Der Inhalt des zweiten Operands wird in den ersten Operand übertragen. Das kann z.B. so aussehen, daß der Inhalt des Akkus in die Speicherzelle nnnn geladen wird, genauso gut geht es auch umgekehrt. Ebenso gibt es Ladebefehle für die 16-Bit-Register. Soll ein 16-Bit-Register in den Speicher übertragen werden, so wird das Low-Byte in nnnn abgelegt, das High-Byte in nnnn+1. Dieses Format wird in der Computertechnik, obwohl zunächst ein wenig ungewohnt, immer benutzt (merke: zuerst Low, dann High!).

LDD

HL und DE dienen als Zeiger auf Bytes im Speicher. Der Inhalt der von HL adressierten Zelle wird in die von DE adressierte übertragen. Dann werden HL, DE und BC dekrementiert. Sollte BC=0 sein, so wird das P-Flag 0. Auf diese Weise kann BC als Zähler benutzt werden.

LDDR

Wie LDD, der Befehl wird jedoch automatisch solange wiederholt, bis BC=0 ist.

LDI

Wie LDD, die Register HL und DE werden jedoch inkrementiert.

LDIR

Wie LDDR, HL und DE werden jedoch inkrementiert.

NEG

Das Byte aus dem Akku wird negiert, d.h. es wird mit -1 multipliziert. Das Ergebnis wird im Akku abgelegt und die Flags aktualisiert. P ist 1, wenn der Akkuinhalt vorher 80 (hexadezimal) war, C ist 1, wenn der Akku den Inhalt 0 hatte.

NOP

Dieser Befehl bewirkt nichts. Er dient zur Erzeugung kleiner Wartezeiten.

OR X

Der Akku und der Operand werden ODER-verknüpft. Das Ergebnis wird in den Akku zurückgespeichert und die Flags werden aktualisiert. Dabei wird das Carry-Bit auf 0 gesetzt, da bei der ODER-Verknüpfung kein Übertrag auftreten kann.

OTDR

HL dient als Zeiger auf eine Speicherzelle, deren Inhalt zum Port (C) ausgegeben wird. Außerdem werden HL und B dekrementiert. Dies wird solange wiederholt, bis B=0 ist. Außer dem Carry-Bit und Z (das auf 1 gesetzt wird) erhalten alle Flags zufällige Werte.

OTIR

wie OTDR, HL wird jedoch inkrementiert.

OUT (C).X

Der Operand x wird an einen Port ausgegeben, wobei das Register C dessen Nummer angibt.

OUT (N).A

Der Akku wird an Port n ausgegeben.

OUTD

HL dient als Zeiger auf eine Speicherzelle, deren Inhalt zum Port (C) ausgegeben wird. Außerdem werden HL und B dekrementiert. Sollte B=0 werden, so ist das Z=Flag auf 1 gesetzt. Die übrigen Flags (außer Carry) erhalten zufällige Werte.

OUTI
wie OUTD, HL wird jedoch inkrementiert.

POP X
Das Registerpaar aus dem Operand wird vom Stapel geladen und der Stapelzeiger aktualisiert. AF steht dabei für Akku & Flags.

PUSH X
Das Registerpaar aus dem Operand wird auf den Stapel gebracht und der Stapelzeiger aktualisiert. AF steht dabei für Akku & Flags.

RES N,X
Das Bit n des Operanden x wird gelöscht.

RET UND RET BEDINGUNG
Diese Befehle bewirken einen Rücksprung aus einem Unterprogramm (ähnlich RETURN in BASIC). Zusätzlich kann eine Bedingung angegeben sein, d.h. der Rücksprung erfolgt nur, wenn ein bestimmtes Flag einen bestimmten Wert hat.

RETI
Bewirkt Rücksprung aus Interruptroutine (ähnlich RET).

RETN
Bewirkt Rücksprung aus NMI-Routine.

RL X UND RLA
Der Operand wird nach links rotiert, wobei das Carry-Bit in Bit 0 und das Bit 7 wiederum in das Carry-Bit geschoben wird (siehe Abb.). Die RL-Befehle verändern alle Flags.
Der RLA-Befehl (nicht RL A) stellt hier eine kleine Ausnahme dar. Er arbeitet wie RL A, beeinflußt jedoch nur das Carry-Flag.

Abb. 9. RL

RLC X UND RLCA

Der Operand wird nach links rotiert, zusätzlich wird das Bit 7 ins Carry übertragen (siehe Abb.).
Außer RLCA verändern die RLC-Befehle alle Flags.

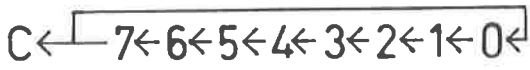


Abb. 10. RLC

RLD

Dieser Befehl führt eine BCD-Rotation durch, d.h. die linke Hälfte der Speicherzelle, deren Adresse in HL abgelegt ist, wird in die rechte Hälfte des Akkus übertragen, die rechte Hälfte der Speicherzelle gelangt in die linke, und die ursprüngliche rechte Hälfte des Akkus gelangt in die rechte Hälfte der Speicherzelle (siehe Abb.). Die Flags S, Z und P werden beeinflußt.

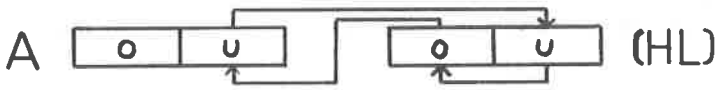


Abb. 13 RLD

RR X UND RRA

Diese Befehle funktionieren wie RL x und RLA, nur wird der Operand nach rechts rotiert.

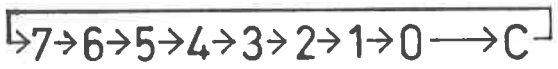


Abb. 11. RR

RRC X UND RRCA

Wie RLC x und RLCA, es wird jedoch nach rechts rotiert.

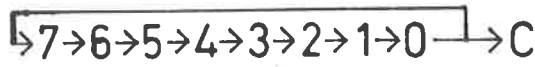


Abb. 12. RRC

RRD

Wie RLD, jedoch Rechtsrotation. Das untere Akku-Nibble (1 Nibble ist ein halbes Byte) wird ins obere Nibble der durch HL adressierten Speicherstelle geschoben, das obere Speicherzellen-Nibble gelangt in das untere und untere wird in die untere Akku-Hälfte verschoben.

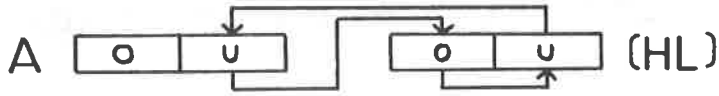


Abb. 14. RRD

RST N

Bei der (fest vorgegebenen) Adresse n wird ein Unterprogramm gestartet.

SBC A,X

Der Operand x wird vom Akku subtrahiert, zusätzlich wird auch ein evtl. vorhandener Übertrag aus dem Carry-Bit beachtet. Das Ergebnis wird wieder im Akku abgelegt, ein neuer Übertrag im Carry-Bit gespeichert. Nach der Ausführung sind die Flags entsprechend dem Akkuinhalt gesetzt.

SBC HL,X

Dieser Befehl funktioniert wie der letzte, nur bewirkt er

eine 16-Bit-Addition. Dann werden der Operand (jetzt ein 16-Bit-Register) und das Carry-Bit vom Registerpaar HL subtrahiert. Das Ergebnis wird wieder in HL abgelegt, die Flags entsprechend gesetzt.

SCF
Das Carry-Bit wird auf 1 gesetzt.

SET N,X
Das Bit n des Operanden x wird auf 1 gesetzt.

SLA X
Der Operand wird nach links geschoben, Bit 0 wird mit einer 0 aufgefüllt, Bit 7 gelangt ins Carry-Bit (siehe Abb.). Die Flags werden entsprechend dem Ergebnis gesetzt.

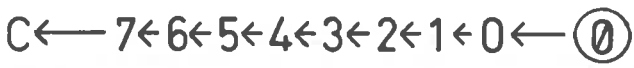


Abb. 15. SLA

SRA X
Der Operand wird nach rechts geschoben, Bit 7 (das Vorzeichen) bleibt erhalten. Bit 0 gelangt ins Carry-Bit (siehe Abb.). Alle Flags werden beeinflußt.

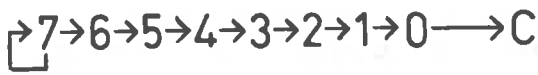


Abb. 16. SRA

SRL X
Dre Operand wird nach rechts geschoben, das Bit 7 mit 0 aufgefüllt. Bit 0 gelangt ins Carry (siehe Abb.). Alle Flags werden beeinflußt.

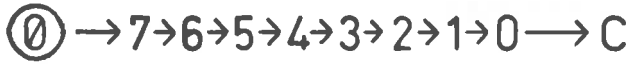


Abb. 17. SRL

SUB X

Der Operand wird vom Akku subtrahiert, die Flag entsprechend dem Ergebnis gesetzt und letzteres wieder i Akku gespeichert.

XOR X

Der Operand wird mit dem Akku exklusiv-oder-verknüpft. Da Ergebnis wird wieder im Akku abgelegt. Die Flags werde aktualisiert, wobei das Carry-Bit gelöscht wird (d Exklusiv-oder-Verknüpfung erzeugt keinen Übertrag).

13.14. Z-80-OPCODES

In der nachfolgenden Tabelle sind alle verfügbaren Befehle des Z-80 mit Opcodes aufgeführt. Außerdem gibt die Spalte "Flags" alle Flags an, die durch den Befehl verändert werden. Bei einzelnen Befehlen kann es allerdings vorkommen, daß nach der Ausführung verschiedene Flags zufällig gesetzt sind. Diese Fälle entnehmen Sie bitte den ausführlichen Beschreibungen.

Mnemonic	Opcode	Flags	Beschreibung
ADC A,A	8F	S Z P C	Addiert Carry und A zu A
ADC A,B	88	S Z P C	... zu B
ADC A,C	89	S Z P C	... zu C
ADC A,D	8A	S Z P C	... zu D
ADC A,E	8B	S Z P C	... zu E
ADC A,H	8C	S Z P C	... zu H
ADC A,L	8D	S Z P C	... zu L
ADC A,n	CEnn	S Z P C	... zum folgenden Byte
ADC A,(HL)	8E	S Z P C	... zur Adresse in HL
ADC A,(IX+d)	DD8Edd	S Z P C	... zur Adresse IX+d
ADC A,(IY+d)	FD8Edd	S Z P C	... zur Adresse IY+d
ADC HL,BC	ED4A	S Z P C	Addiert Carry und HL zu BC
ADC HL,DE	ED5A	S Z P C	... zu DE
ADC HL,HL	ED6A	S Z P C	... zu HL
ADC HL,SP	ED7A	S Z P C	... zu SP
ADD A,A	87	S Z P C	Addiert Akku zu Akku
ADD A,B	80	S Z P C	... zu B
ADD A,C	81	S Z P C	... zu C
ADD A,D	82	S Z P C	... zu D
ADD A,E	83	S Z P C	... zu E
ADD A,H	84	S Z P C	... zu H
ADD A,L	85	S Z P C	... zu L
ADD A,n	C6nn	S Z P C	... zum folgenden Byte
ADD A,(HL)	86	S Z P C	... zur Adresse in HL
ADD A,(IX+d)	DD86dd	S Z P C	... zur Adresse IX+d
ADD A,(IY+d)	FD86dd	S Z P C	... zur Adresse IY+d

```
ADD HL,BC       09              C   Addiert HL zu BC
ADD HL,DE       19              C     ... zu DE
ADD HL,HL       29              C     ... zu HL
ADD HL,SP       39              C     ... zu SP
ADD IX,BC       DD09            C   Addiert IX zu BC
ADD IX,DE       DD19            C     ... zu DE
ADD IX,HL       DD29            C     ... zu HL
ADD IX,SP       DD39            C     ... zu SP
ADD IY,BC       FD09            C   Addiert IY zu BC
ADD IY,DE       FD19            C     ... zu DE
ADD IY,HL       FD29            C     ... zu HL
ADD IY,SP       FD39            C     ... zu SP
AND A           A7       S Z P C=0 AND-verknüpft   Akku   mit
Akku
AND B           A0       S Z P C=0   ... mit B
AND C           A1       S Z P C=0   ... mit C
AND D           A2       S Z P C=0   ... mit D
AND E           A3       S Z P C=0   ... mit E
AND H           A4       S Z P C=0   ... mit H
AND L           A5       S Z P C=0   ... mit L
AND n           E6nn     S Z P C=0   ... mit folgenden Byte
AND (HL)        96       S Z P C=0   ... mit Adresse in HL
AND (IX+d)      DD96dd   S Z P C=0   ... mit Adresse IX+d
AND (IY+d)      FD96dd   S Z P C=0   ... mit Adresse IY+d
BIT 0,A         CB47       Z        Testet Bit 0 des Akkus
BIT 0,B         CB40       Z        Testet Bit 0 von B
BIT 0,C         CB41       Z          ... von C
BIT 0,D         CB42       Z          ... von D
BIT 0,E         CB43       Z          ... von E
BIT 0,H         CB44       Z          ... von H
BIT 0,L         CB45       Z          ... von L
BIT 0,(HL)      CB46       Z          ... der Adresse in HL
BIT 0,(IX+d)    DDCBdd46   Z          ... der Adresse IX+d
BIT 0,(IY+d)    FDCBdd46   Z          ... der Adresse IY+d
BIT 1,A         CB4F       Z        Testet Bit 1 des Akkus
BIT 1,B         CB48       Z        Testet Bit 1 von B
BIT 1,C         CB49       Z          ... von C
BIT 1,D         CB4A       Z          ... von D
BIT 1,E         CB4B       Z          ... von E
BIT 1,H         CB4C       Z          ... von H
```

BIT 1,L	CB4D	Z	... von L
BIT 1,(HL)	CB4E	Z	... der Adresse in HL
BIT 1,(IX+d)	DDCBdd4E	Z	... der Adresse IX+d
BIT 1,(IY+d)	FDCBdd4E	Z	... der Adresse IY+d
BIT 2,A	CB57	Z	Testet Bit 2 des Akkus
BIT 2,B	CB50	Z	Testet Bit 2 von B
BIT 2,C	CB51	Z	... von C
BIT 2,D	CB52	Z	... von D
BIT 2,E	CB53	Z	... von E
BIT 2,H	CB54	Z	... von H
BIT 2,L	CB55	Z	... von L
BIT 2,(HL)	CB56	Z	... der Adresse in HL
BIT 2,(IX+d)	DDCBdd56	Z	... der Adresse IX+d
BIT 2,(IY+d)	FDCBdd56	Z	... der Adresse IY+d
BIT 3,A	CB5F	Z	Testet Bit 3 des Akkus
BIT 3,B	CB58	Z	Testet Bit 3 von B
BIT 3,C	CB59	Z	... von C
BIT 3,D	CB5A	Z	... von D
BIT 3,E	CB5B	Z	... von E
BIT 3,H	CB5C	Z	... von H
BIT 3,L	CB5D	Z	... von L
BIT 3,(HL)	CB5E	Z	... der Adresse in HL
BIT 3,(IX+d)	DDCBdd5E	Z	... der Adresse IX+d
BIT 3,(IY+d)	FDCBdd5E	Z	... der Adresse IY+d
BIT 4,A	CB67	Z	Testet Bit 4 des Akkus
BIT 4,B	CB60	Z	Testet Bit 4 von B
BIT 4,C	CB61	Z	... von C
BIT 4,D	CB62	Z	... von D
BIT 4,E	CB63	Z	... von E
BIT 4,H	CB64	Z	... von H
BIT 4,L	CB65	Z	... von L
BIT 4,(HL)	CB66	Z	... der Adresse in HL
BIT 4,(IX+d)	DDCBdd66	Z	... der Adresse IX+d
BIT 4,(IY+d)	FDCBdd66	Z	... der Adresse IY+d
BIT 5,A	CB6F	Z	Testet Bit 5 des Akkus
BIT 5,B	CB68	Z	Testet Bit 5 von B
BIT 5,C	CB69	Z	... von C
BIT 5,D	CB6A	Z	... von D
BIT 5,E	CB6B	Z	... von E
BIT 5,H	CB6C	Z	... von H

BIT 5,L	CB6D	Z	... von L
BIT 5,(HL)	CB6E	Z	... der Adresse in HL
BIT 5,(IX+d)	DDCBdd6E	Z	... der Adresse IX+d
BIT 5,(IY+d)	FDCBdd6E	Z	... der Adresse IY+d
BIT 6,A	CB77	Z	Testet Bit 6 des Akkus
BIT 6,B	CB70	Z	Testet Bit 6 von B
BIT 6,C	CB71	Z	... von C
BIT 6,D	CB72	Z	... von D
BIT 6,E	CB73	Z	... von E
BIT 6,H	CB74	Z	... von H
BIT 6,L	CB75	Z	... von L
BIT 6,(HL)	CB76	Z	... der Adresse in HL
BIT 6,(IX+d)	DDCBdd76	Z	... der Adresse IX+d
BIT 6,(IY+d)	FDCBdd76	Z	... der Adresse IY+d
BIT 7,A	CB7F	Z	Testet Bit 7 des Akkus
BIT 7,B	CB78	Z	Testet Bit 7 von B
BIT 7,C	CB79	Z	... von C
BIT 7,D	CB7A	Z	... von D
BIT 7,E	CB7B	Z	... von E
BIT 7,H	CB7C	Z	... von H
BIT 7,L	CB7D	Z	... von L
BIT 7,(HL)	CB7E	Z	... der Adresse in HL
BIT 7,(IX+d)	DDCBdd7E	Z	... der Adresse IX+d
BIT 7,(IY+d)	FDCBdd7E	Z	... der Adresse IY+d
CALL nn	CDnnnn		Ruft Unterprogramm nnnn auf
CALL C,nn	DCnnnn		... wenn Carry=1
CALL M,nn	FCnnnn		... wenn S=1 (minus)
CALL NC,nn	D4nnnn		... wenn Carry=0
CALL NZ,nn	C4nnnn		... wenn Z=0
CALL P,nn	F4nnnn		... wenn S=0 (plus)
CALL PE,nn	ECnnnn		... wenn P=1
CALL PO,nn	E4nnnn		... wenn P=0
CALL Z,nn	CCnnnn		... wenn Z=1
CCF	3F	C	Komplementiert Carry-Flag
CP A	BF	S Z P C	Vergleicht A mit A
CP B	B8	S Z P C	... mit B
CP C	B9	S Z P C	... mit C
CP D	BA	S Z P C	... mit D
CP E	BB	S Z P C	... mit E

CP H	BC	S Z P C	... mit H
CP L	BD	S Z P C	... mit L
CP n	FEnn	S Z P C	... mit folgendem Byte
CP (HL)	BE	S Z P C	... mit Adresse in HL
CP (IX+d)	DDBEdd	S Z P C	... mit Adresse IX+d
CP (IY+d)	FDBEdd	S Z P C	... mit Adresse IY+d
CPD	EDA9	S Z P	Vergleicht & dekrementiert
CPDR	EDB9	S Z P	Blockvergleich & Dekrement
CPI	EDA1	S Z P	Vergleicht & inkrementiert
CPIR	EDB1	S Z P	Blockvergleich & Inkrement
CPL	2F		Komplementiert Akku
DAA	27	S Z P C	BCD-Anpassung des Akkus
DEC A	3D	S Z P	Dekrementiert A
DEC B	05	S Z P	... B
DEC BC	0B		... BC
DEC C	0D	S Z P	... C
DEC D	15	S Z P	... D
DEC DE	1B		... DE
DEC E	1D	S Z P	... E
DEC H	25	S Z P	... H
DEC HL	2B		... HL
DEC IX	DD2B		... IX
DEC IY	FD2B		... IY
DEC L	2D	S Z P	... L
DEC SP	3B		... SP
DEC (HL)	35	S Z P	... Adresse in HL
DEC (IX+d)	DD35dd	S Z P	... Adresse IX+d
DEC (IY+d)	FD35dd	S Z P	... Adresse IY+d
DI	F3		Sperrt Interrupt
DJNZ e	10ee		Dekrement & Sprung bei ()0
EI	FB		Interrupt freigeben
EX AF,AF"	08		Tauscht A/Flags m.
Zweitre.			
EX DE,HL	EB		Vertauscht DE und HL
EX (SP),HL	E3		Vertauscht HL m.
Stapelbyte			
EX (SP),IX	DDE3		... IX
EX (SP),IY	FDE3		... IY
EXX	D9		... BC,DE,HL m.
Zweitreg.			

HALT	76		Stoppt Z-80 bis Interrupt
IM 0	ED46		Interruptmodus 0
IM 1	ED56		Interruptmodus 1
IM 2	ED5E		Interruptmodus 2
IN A,(C)	ED78	S Z P	Liest Byte vom Port C in A
IN A,(n)	DBnn		... vom Port n
IN B,(C)	ED40	S Z P	... in B
IN C,(C)	ED48	S Z P	... in C
IN D,(C)	ED50	S Z P	... in D
IN E,(C)	ED58	S Z P	... in E
IN H,(C)	ED60	S Z P	... in H
IN L,(C)	ED68	S Z P	... in L
INC A	3C	S Z P	Inkrementiert A
INC B	04	S Z P	... B
INC BC	03		... BC
INC C	0C	S Z P	... C
INC D	14	S Z P	... D
INC DE	13		... DE
INC E	1C	S Z P	... E
INC H	24	S Z P	... H
INC HL	23		... HL
INC IX	DD23		... IX
INC IY	FD23		... IY
INC L	2C	S Z P	... L
INC SP	33		... SP
INC (HL)	34	S Z P	... Adresse in HL
INC (IX+d)	DD34dd	S Z P	... Adresse IX+d
INC (IY+d)	FD34dd	S Z P	... Adresse IY+d
IND	EDAA	Z	Einlesen mit Dekrement
INDR	EDBA	Z=1	Blockeinlesen m. Dekrement
INI	EDA2	Z	Einlesen mit Inkrement
INIR	EDB2	Z=1	Blockeinlesen m. Inkrement
JP C,nn	DAnnnn		Springt nach nn, wenn C=1
JP M,nn	FAnnnn		... nach nn, wenn S=1
JP NC,nn	D2nnnn		... nach nn, wenn C=0
JP nn	C3nnnn		... nach nn
JP NZ,nn	C2nnnn		... nach nn, wenn Z=0
JP P,nn	F2nnnn		... nach nn, wenn S=0
JP PE,nn	EAnnnn		... nach nn, wenn P=1
JP PO,nn	E2nnnn		... nach nn, wenn P=0

```
JP Z,nn         CAnnnn                ... nach nn, wenn Z=1
JP (HL)         E9                    ... zur Adresse in HL
JP (IX)         DDE9                  ... zur Adresse in IX
JP (IY)         FDE9                  ... zur Adresse in IY
JR C,e          38ee                Springt relativ, wenn C=1
JR e            18ee                  ... zu PC+e
JR NC,e         30ee                  ... wenn C=0
JR NZ,e         20ee                  ... wenn Z=0
JR Z,e          28ee                  ... wenn Z=1
LD A,A          7F                  Lädt Akku mit Akku
LD A,B          78                    ... mit B
LD A,C          79                    ... mit C
LD A,D          7A                    ... mit D
LD A,E          7B                    ... mit E
LD A,H          7C                    ... mit H
LD A,I          ED57      S Z P       ... mit I
LD A,L          7D                    ... mit L
LD A,n          3Enn                  ...   m.   nachfolgendem
Byte
LD A,R          ED5F      S Z P       ... mit R
LD A,(BC)       0A                    ... aus Adresse in BC
LD A,(DE)       1A                    ... aus Adresse in DE
LD A,(HL)       7E                    ... aus Adresse in HL
LD A,(IX+d)     DD7Edd                ... aus Adresse IX+d
LD A,(IY+d)     FD7Edd                ... aus Adresse IY+d
LD A,(nn)       3Annnn                ... aus Adresse nn
LD B,A          47                  Lädt B mit Akku
LD B,B          40                    ... mit B
LD B,C          41                    ... mit C
LD B,D          42                    ... mit D
LD B,E          43                    ... mit E
LD B,H          44                    ... mit H
LD B,L          45                    ... mit L
LD B,n          06nn                  ...   m.   nachfolgendem
Byte
LD B,(HL)       46                    ... aus Adresse in HL
LD B,(IX+d)     DD46dd                ... aus Adresse IX+d
LD B,(IY+d)     FD46dd                ... aus Adresse IY+d
LD BC,nn        01nnnn              Lädt   BC   mit   folgenden
Bytes
```

```
LD BC,(nn)      ED4Bnnnn          ... mit nnnn und nnnn+1
LD C,A          4F                Lädt C mit Akku
LD C,B          48                  ... mit B
LD C,C          49                  ... mit C
LD C,D          4A                  ... mit D
LD C,E          4B                  ... mit E
LD C,H          4C                  ... mit H
LD C,L          4D                  ... mit L
LD C,n          0Enn                ...   m.   nachfolgendem
Byte
LD C,(HL)       4E                  ... aus Adresse in HL
LD C,(IX+d)     DD4Edd              ... aus Adresse IX+d
LD C,(IY+d)     FD4Edd              ... aus Adresse IY+d
LD D,A          57                Lädt D mit Akku
LD D,B          50                  ... mit B
LD D,C          51                  ... mit C
LD D,D          52                  ... mit D
LD D,E          53                  ... mit E
LD D,H          54                  ... mit H
LD D,L          55                  ... mit L
LD D,n          16nn                ...   m.   nachfolgendem
Byte
LD D,(HL)       56                  ... aus Adresse in HL
LD D,(IX+d)     DD56dd              ... aus Adresse IX+d
LD D,(IY+d)     FD56dd              ... aus Adresse IY+d
LD DE,nn        11nnnn            Lädt   DE   mit  folgenden
Bytes
LD DE,(nn)      ED5Bnnnn            ... mit nnnn und nnnn+1
LD E,A          5F                Lädt E mit Akku
LD E,B          58                  ... mit B
LD E,C          59                  ... mit C
LD E,D          5A                  ... mit D
LD E,E          5B                  ... mit E
LD E,H          5C                  ... mit H
LD E,L          5D                  ... mit L
LD E,n          1Enn                ...   m.   nachfolgendem
Byte
LD E,(HL)       5E                  ... aus Adresse in HL
LD E,(IX+d)     DD5Edd              ... aus Adresse IX+d
LD E,(IY+d)     FD5Edd              ... aus Adresse IY+d
```

```
LD H,A          67              Lädt H mit Akku
LD H,B          60                ... mit B
LD H,C          61                ... mit C
LD H,D          62                ... mit D
LD H,E          63                ... mit E
LD H,H          64                ... mit H
LD H,L          65                ... mit L
LD H,n          26nn              ...   m.   nachfolgendem
Byte
LD H,(HL)       66                ... aus Adresse in HL
LD H,(IX+d)     DD66dd            ... aus Adresse IX+d
LD H,(IY+d)     FD66dd            ... aus Adresse IY+d
LD HL,nn        21nnnn          Lädt   HL   mit  folgenden
Bytes
LD HL,(nn)      2Annnn            ... aus nnnn und nnnn+1
LD I,A          ED47            Lädt I mit Akku
LD IX,nn        DD21nnnn        Lädt   IX   mit  folgenden
Bytes
LD IX,(nn)      DD2Annnn          ... aus nnnn und nnnn+1
LD IY,nn        FD21nnnn        Lädt   IY   mit  folgenden
Bytes
LD IY,(nn)      FD2Annnn          ... aus nnnn und nnnn+1
LD L,A          6F              Lädt L mit Akku
LD L,B          68                ... mit B
LD L,C          69                ... mit C
LD L,D          6A                ... mit D
LD L,E          6B                ... mit E
LD L,H          6C                ... mit H
LD L,L          6D                ... mit L
LD L,n          2Enn              ...   m.   nachfolgendem
Byte
LD L,(HL)       6E                ... aus Adresse in HL
LD L,(IX+d)     DD6Edd            ... aus Adresse IX+d
LD L,(IY+d)     FD6Edd            ... aus Adresse IY+d
LD R,A          ED4F            Lädt R mit Akku
LD SP,HL        F9              Lädt SP mit HL
LD SP,IX        DDF9              ... mit IX
LD SP,IY        FDF9              ... mit IY
LD SP,nn        31nnnn            ... mit folgenden Bytes
LD SP,(nn)      ED7Bnnnn          ... aus nnnn und nnnn+1
```

LD (BC),A	02	Lädt Adresse aus BC m. Akku
LD (DE),A	12	Lädt Adresse aus DE m. Akku
LD (HL),A	77	Lädt Adresse aus HL m. Akku
LD (HL),B	70	... mit B
LD (HL),C	71	... mit C
LD (HL),D	72	... mit D
LD (HL),E	73	... mit E
LD (HL),H	74	... mit H
LD (HL),L	75	... mit L
LD (HL),n	36nn	... m. nachfolgendem Byte
LD (IX+d),A	DD77dd	Lädt Adresse IX+d mit Akku
LD (IX+d),B	DD70dd	... mit B
LD (IX+d),C	DD71dd	... mit C
LD (IX+d),D	DD72dd	... mit D
LD (IX+d),E	DD73dd	... mit E
LD (IX+d),H	DD74dd	... mit H
LD (IX+d),L	DD75dd	... mit L
LD (IX+d),n	DD36ddnn	... m. nachfolgendem Byte
LD (IY+d),A	FD77dd	Lädt Adresse IY+d mit Akku
LD (IY+d),B	FD70dd	... mit B
LD (IY+d),C	FD71dd	... mit C
LD (IY+d),D	FD72dd	... mit D
LD (IY+d),E	FD73dd	... mit E
LD (IY+d),H	FD74dd	... mit H
LD (IY+d),L	FD75dd	... mit L
LD (IY+d),n	FD36ddnn	... m. nachfolgendem Byte
LD (nn),A	32nnnn	Lädt Zelle nnnn mit Akku
LD (nn),BC	ED43nnnn	... mit C und nnnn+1 m. B
LD (nn),DE	ED53nnnn	... mit E und nnnn+1 m. D
LD (nn),HL	22nnnn	... mit L und nnnn+1 m. H
LD (nn),IX	DD22nnnn	... und nnnn+1 mit IX

LD (nn),IY	FD22nnnn		... und nnnn+1 mit IY
LD (nn),SP	ED73nnnn		... und nnnn+1 mit SP
LDD	EDA8	P	Laden und Dekrementieren
LDDR	EDB8	P=0	Blockladen und Dekrement
LDI	EDA0	P	Laden und Inkrementieren
LDIR	EDB0	P=0	Blockladen und Inkrement
NEG	ED44	S Z P C	Negiert Akku
NOP	00		Wartet (Nulloperation)
OR A	B7	S Z P C	ODER-verknüpft Akku m. Akku
OR B	B0	S Z P C	... mit B
OR C	B1	S Z P C	... mit C
OR D	B2	S Z P C	... mit D
OR E	B3	S Z P C	... mit E
OR H	B4	S Z P C	... mit H
OR L	B5	S Z P C	... mit L
OR n	F6nn	S Z P C	... nachfolgendem Byte
OR (HL)	B6	S Z P C	... mit Adresse in HL
OR (IX+d)	DDB6dd	S Z P C	... mit Adresse IX+d
OR (IY+d)	FDB6dd	S Z P C	... mit Adresse IY+d
OTDR	EDBB	S Z P	Blockausgabe und Dekrement
OTIR	EDB3	S Z P	Blockausgabe und Inkrement
OUT (C),A	ED79		Gibt an Port C Akku aus
OUT (C),B	ED41		... B aus
OUT (C),C	ED49		... C aus
OUT (C),D	ED51		... D aus
OUT (C),E	ED59		... E aus
OUT (C),H	ED61		... H aus
OUT (C),L	ED69		... L aus
OUT (n),A	D3nn		Gibt Akku an Port n aus
OUTD	EDAB	S Z P	Ausgabe und Dekrementieren
OUTI	EDA3	S Z P	Ausgabe und Inkrementieren
POP AF	F1		Holt Akku & Flags vom Stack
POP BC	C1		Holt BC vom Stapel
POP DE	D1		Holt DE vom Stapel
POP HL	E1		Holt HL vom Stapel
POP IX	DDE1		Holt IX vom Stapel
POP IY	FDE1		Holt IY vom Stapel
PUSH AF	F5		Legt Akku & Flags auf

Stack		
PUSH BC	C5	Legt BC auf Stapel
PUSH DE	D5	Legt DE auf Stapel
PUSH HL	E5	Legt HL auf Stapel
PUSH IX	DDE5	Legt IX auf Stapel
PUSH IY	FDE5	Legt IY auf Stapel
RES 0,A	CB87	Löscht Bit 0 des Akkus
RES 0,B	CB80	Löscht Bit 0 von B
RES 0,C	CB81	... von C
RES 0,D	CB82	... von D
RES 0,E	CB83	... von E
RES 0,H	CB84	... von H
RES 0,L	CB85	... von L
RES 0,(HL)	CB86	... der Adresse in HL
RES 0,(IX+d)	DDCBdd86	... der Adresse IX+d
RES 0,(IY+d)	FDCBdd86	... der Adresse IY+d
RES 1,A	CB8F	Löscht Bit 1 des Akkus
RES 1,B	CB88	Löscht Bit 1 von B
RES 1,C	CB89	... von C
RES 1,D	CB8A	... von D
RES 1,E	CB8B	... von E
RES 1,H	CB8C	... von H
RES 1,L	CB8D	... von L
RES 1,(HL)	CB8E	... der Adresse in HL
RES 1,(IX+d)	DDCBdd8E	... der Adresse IX+d
RES 1,(IY+d)	FDCBdd8E	... der Adresse IY+d
RES 2,A	CB97	Löscht Bit 2 des Akkus
RES 2,B	CB90	Löscht Bit 2 von B
RES 2,C	CB91	... von C
RES 2,D	CB92	... von D
RES 2,E	CB93	... von E
RES 2,H	CB94	... von H
RES 2,L	CB95	... von L
RES 2,(HL)	CB96	... der Adresse in HL
RES 2,(IX+d)	DDCBdd96	... der Adresse IX+d
RES 2,(IY+d)	FDCBdd96	... der Adresse IY+d
RES 3,A	CB9F	Löscht Bit 3 des Akkus
RES 3,B	CB98	Löscht Bit 3 von B
RES 3,C	CB99	... von C
RES 3,D	CB9A	... von D

RES 3,E	CB9B	... von E
RES 3,H	CB9C	... von H
RES 3,L	CB9D	... von L
RES 3,(HL)	CB9E	... der Adresse in HL
RES 3,(IX+d)	DDCBdd9E	... der Adresse IX+d
RES 3,(IY+d)	FDCBdd9E	... der Adresse IY+d
RES 4,A	CBA7	Löscht Bit 4 des Akkus
RES 4,B	CBA0	Löscht Bit 4 von B
RES 4,C	CBA1	... von C
RES 4,D	CBA2	... von D
RES 4,E	CBA3	... von E
RES 4,H	CBA4	... von H
RES 4,L	CBA5	... von L
RES 4,(HL)	CBA6	... der Adresse in HL
RES 4,(IX+d)	DDCBddA6	... der Adresse IX+d
RES 4,(IY+d)	FDCBddA6	... der Adresse IY+d
RES 5,A	CBAF	Löscht Bit 5 des Akkus
RES 5,B	CBA8	Löscht Bit 5 von B
RES 5,C	CBA9	... von C
RES 5,D	CBAA	... von D
RES 5,E	CBAB	... von E
RES 5,H	CBAC	... von H
RES 5,L	CBAD	... von L
RES 5,(HL)	CBAE	... der Adresse in HL
RES 5,(IX+d)	DDCBddAE	... der Adresse IX+d
RES 5,(IY+d)	FDCBddAE	... der Adresse IY+d
RES 6,A	CBB7	Löscht Bit 6 des Akkus
RES 6,B	CBB0	Löscht Bit 6 von B
RES 6,C	CBB1	... von C
RES 6,D	CBB2	... von D
RES 6,E	CBB3	... von E
RES 6,H	CBB4	... von H
RES 6,L	CBB5	... von L
RES 6,(HL)	CBB6	... der Adresse in HL
RES 6,(IX+d)	DDCBddB6	... der Adresse IX+d
RES 6,(IY+d)	FDCBddB6	... der Adresse IY+d
RES 7,A	CBBF	Löscht Bit 7 des Akkus
RES 7,B	CBB8	Löscht Bit 7 von B
RES 7,C	CBB9	... von C
RES 7,D	CBBA	... von D

RES 7,E	CBBB					... von E
RES 7,H	CBBC					... von H
RES 7,L	CBBD					... von L
RES 7,(HL)	CBBE					... der Adresse in HL
RES 7,(IX+d)	DDCBddBE					... der Adresse IX+d
RES 7,(IY+d)	FDCBddBE					... der Adresse IY+d
RET	C9					Rücksprung aus Unterpgm.
RET C	D8					Rücksprung, wenn C=1
RET M	F8					... wenn S=1
RET NC	DO					... wenn C=0
RET NZ	CO					... wenn Z=0
RET P	FO					... wenn S=0
RET PE	E8					... wenn P=1
RET PO	EO					... wenn P=0
RET Z	C8					... wenn Z=1
RETI	ED4D					Rücksprung aus Interrupt
RETN	ED45					... aus NMI-Routine
RL A	CB17	S	Z	P	C	Linksrotation v. Carry & A
RL B	CB10	S	Z	P	C	... und B
RL C	CB11	S	Z	P	C	... und C
RL D	CB12	S	Z	P	C	... und D
RL E	CB13	S	Z	P	C	... und E
RL H	CB14	S	Z	P	C	... und H
RL L	CB15	S	Z	P	C	... und L
RL (HL)	CB16	S	Z	P	C	... und Adresse in HL
RL (IX+d)	DDCBdd16	S	Z	P	C	... und Adresse IX+d
RL (IY+d)	FDCBdd16	S	Z	P	C	... und Adresse IY+d
RLA	17				C	... und Akku
RLC A	CB07	S	Z	P	C	Linksrotation des Akkus
RLC B	CB00	S	Z	P	C	... von B
RLC C	CB01	S	Z	P	C	... von C
RLC D	CB02	S	Z	P	C	... von D
RLC E	CB03	S	Z	P	C	... von E
RLC H	CB04	S	Z	P	C	... von H
RLC L	CB05	S	Z	P	C	... von L
RLC (HL)	CB06	S	Z	P	C	... der Adresse in HL
RLC (IX+d)	DDCBdd06	S	Z	P	C	... der Adresse IX+d
RLC (IY+d)	FDCBdd06	S	Z	P	C	... der Adresse IY+d
RLCA	07				C	... des Akkus
RLD	ED6F	S	Z	P		Dezimalrotation links

```
RR A            CB1F      S Z P C   Rechtsrotation  v. Carry &
A
RR B            CB18      S Z P C     ... und B
RR C            CB19      S Z P C     ... und C
RR D            CB1A      S Z P C     ... und D
RR E            CB1B      S Z P C     ... und E
RR H            CB1C      S Z P C     ... und H
RR L            CB1D      S Z P C     ... und L
RR (HL)         CB1E      S Z P C     ... und Adresse in HL
RR (IX+d)       DDCBdd1E  S Z P C     ... und Adresse IX+d
RR (IY+d)       FDCBdd1E  S Z P C     ... und Adresse IY+d
RRA             1F              C     ... und Akku
RRC A           CB0F      S Z P C   Rechtsrotation des Akkus
RRC B           CB08      S Z P C     ... von B
RRC C           CB09      S Z P C     ... von C
RRC D           CB0A      S Z P C     ... von D
RRC E           CB0B      S Z P C     ... von E
RRC H           CB0C      S Z P C     ... von H
RRC L           CB0D      S Z P C     ... von L
RRC (HL)        CB0E      S Z P C     ... der Adresse in HL
RRC (IX+d)      DDCBdd0E  S Z P C     ... der Adresse IX+d
RRC (IY+d)      FDCBdd0E  S Z P C     ... der Adresse IY+d
RRCA            0F              C     ... des Akkus
RRD             ED67      S Z P     Dezimalrotation rechts
RST 00          C7                  Uproaufruf bei 00
RST 08          CF                    ... bei 08
RST 10          D7                    ... bei 10
RST 18          DF                    ... bei 18
RST 20          E7                    ... bei 20
RST 28          EF                    ... bei 28
RST 30          F7                    ... bei 30
RST 38          FF                    ... bei 38
SBC A,A         9F        S Z P C   Subtrahiert  Carry & A von
A
SBC A,B         98        S Z P C     ... B von Akku
SBC A,C         99        S Z P C     ... C von Akku
SBC A,D         9A        S Z P C     ... D von Akku
SBC A,E         9B        S Z P C     ... E von Akku
SBC A,H         9C        S Z P C     ... H von Akku
SBC A,L         9D        S Z P C     ... L von Akku
```

```
SBC A,n         DEnn      S Z P C      ... folgendes Byte von A
SBC A,(HL)      9E        S Z P C      ... Adresse in HL von A
SBC A,(IX+d)    DD9Edd    S Z P C      ...   Adresse  IX+d  von
Akku
SBC A,(IY+d)    FD9Edd    S Z P C      ...   Adresse  IY+d  von
Akku
SBC HL,BC       ED42      S Z P C   Subtrahiert C & BC von HL
SBC HL,DE       ED52      S Z P C     ... DE von HL
SBC HL,HL       ED62      S Z P C     ... HL von HL
SBC HL,SP       ED72      S Z P C     ... SP von HL
SCF             37              C=1 Setzt Carry-Bit auf 1
SET 0,A         CBC7                Setzt Bit 0 des Akkus
SET 0,B         CBC0                Setzt Bit 0 von B
SET 0,C         CBC1                  ... von C
SET 0,D         CBC2                  ... von D
SET 0,E         CBC3                  ... von E
SET 0,H         CBC4                  ... von H
SET 0,L         CBC5                  ... von L
SET 0,(HL)      CBC6                  ... der Adresse in HL
SET 0,(IX+d)    DDCBddC6              ... der Adresse IX+d
SET 0,(IY+d)    FDCBddC6              ... der Adresse IY+d
SET 1,A         CBCF                Setzt Bit 1 des Akkus
SET 1,B         CBC8                Setzt Bit 1 von B
SET 1,C         CBC9                  ... von C
SET 1,D         CBCA                  ... von D
SET 1,E         CBCB                  ... von E
SET 1,H         CBCC                  ... von H
SET 1,L         CBCD                  ... von L
SET 1,(HL)      CBCE                  ... der Adresse in HL
SET 1,(IX+d)    DDCBddCE              ... der Adresse IX+d
SET 1,(IY+d)    FDCBddCE              ... der Adresse IY+d
SET 2,A         CBD7                Setzt Bit 2 des Akkus
SET 2,B         CBD0                Setzt Bit 2 von B
SET 2,C         CBD1                  ... von C
SET 2,D         CBD2                  ... von D
SET 2,E         CBD3                  ... von E
SET 2,H         CBD4                  ... von H
SET 2,L         CBD5                  ... von L
SET 2,(HL)      CBD6                  ... der Adresse in HL
SET 2,(IX+d)    DDCBddD6              ... der Adresse IX+d
```

SET 2,(IY+d)	FDCBddD6	... der Adresse IY+d
SET 3,A	CBDF	Setzt Bit 3 des Akkus
SET 3,B	CBD8	Setzt Bit 3 von B
SET 3,C	CBD9	... von C
SET 3,D	CBDA	... von D
SET 3,E	CBDB	... von E
SET 3,H	CBDC	... von H
SET 3,L	CBDD	... von L
SET 3,(HL)	CBDE	... der Adresse in HL
SET 3,(IX+d)	DDCBddDE	... der Adresse IX+d
SET 3,(IY+d)	FDCBddDE	... der Adresse IY+d
SET 4,A	CBE7	Setzt Bit 4 des Akkus
SET 4,B	CBE0	Setzt Bit 4 von B
SET 4,C	CBE1	... von C
SET 4,D	CBE2	... von D
SET 4,E	CBE3	... von E
SET 4,H	CBE4	... von H
SET 4,L	CBE5	... von L
SET 4,(HL)	CBE6	... der Adresse in HL
SET 4,(IX+d)	DDCBddE6	... der Adresse IX+d
SET 4,(IY+d)	FDCBddE6	... der Adresse IY+d
SET 5,A	CBEF	Setzt Bit 5 des Akkus
SET 5,B	CBE8	Setzt Bit 5 von B
SET 5,C	CBE9	... von C
SET 5,D	CBEA	... von D
SET 5,E	CBEB	... von E
SET 5,H	CBEC	... von H
SET 5,L	CBED	... von L
SET 5,(HL)	CBEE	... der Adresse in HL
SET 5,(IX+d)	DDCBddEE	... der Adresse IX+d
SET 5,(IY+d)	FDCBddEE	... der Adresse IY+d
SET 6,A	CBF7	Setzt Bit 6 des Akkus
SET 6,B	CBF0	Setzt Bit 6 von B
SET 6,C	CBF1	... von C
SET 6,D	CBF2	... von D
SET 6,E	CBF3	... von E
SET 6,H	CBF4	... von H
SET 6,L	CBF5	... von L
SET 6,(HL)	CBF6	... der Adresse in HL
SET 6,(IX+d)	DDCBddF6	... der Adresse IX+d

SET 6,(IY+d)	FDCBddF6		... der Adresse IY+d
SET 7,A	CBFF		Setzt Bit 7 des Akkus
SET 7,B	CBF8		Setzt Bit 7 von B
SET 7,C	CBF9		... von C
SET 7,D	CBFA		... von D
SET 7,E	CBFB		... von E
SET 7,H	CBFC		... von H
SET 7,L	CBFD		... von L
SET 7,(HL)	CBFE		... der Adresse in HL
SET 7,(IX+d)	DDCBddFE		... der Adresse IX+d
SET 7,(IY+d)	FDCBddFE		... der Adresse IY+d
SLA A	CB27	S Z P C	Schiebt Carry & Akku links
SLA B	CB20	S Z P C	... und B links
SLA C	CB21	S Z P C	... und C links
SLA D	CB22	S Z P C	... und D links
SLA E	CB23	S Z P C	... und E links
SLA H	CB24	S Z P C	... und H links
SLA L	CB25	S Z P C	... und L links
SLA (HL)	CB26	S Z P C	... & Adresse in HL links
SLA (IX+d)	DDCBdd26	S Z P C	... & Adresse IX+d links
SLA (IY+d)	FDCBdd26	S Z P C	... & Adresse IY+d links
SRA A	CB2F	S Z P C	... und Akku rechts
SRA B	CB28	S Z P C	... und B rechts
SRA C	CB29	S Z P C	... und C rechts
SRA D	CB2A	S Z P C	... und D rechts
SRA E	CB2B	S Z P C	... und E rechts
SRA H	CB2C	S Z P C	... und H rechts
SRA L	CB2D	S Z P C	... und L rechts
SRA (HL)	CB2E	S Z P C	... & Adresse HL rechts
SRA (IX+d)	DDCBdd2E	S Z P C	... & Adresse IX+d rechts
SRA (IY+d)	FDCBdd2E	S Z P C	... & Adresse IY+d rechts
SRL A	CB3F	S Z P C	... und Akku rechts
SRL B	CB38	S Z P C	... und B rechts
SRL C	CB39	S Z P C	... und C rechts
SRL D	CB3A	S Z P C	... und D rechts
SRL E	CB3B	S Z P C	... und E rechts
SRL H	CB3C	S Z P C	... und H rechts

```
SRL L          CB3D      S Z P C      ... und L rechts
SRL (HL)       CB3E      S Z P C      ... & Adresse HL rechts
SRL (IX+d)     DDCBdd3E  S Z P C      ...   &   Adresse   IX+d
rechts
SRL (IY+d)     FDCBdd3E  S Z P C      ...   &   Adresse   IY+d
rechts
SUB A          97        S Z P C   Subtrahiert Akku vom Akku
SUB B          90        S Z P C     ... B vom Akku
SUB C          91        S Z P C     ... C vom Akku
SUB D          92        S Z P C     ... D vom Akku
SUB E          93        S Z P C     ... E vom Akku
SUB H          94        S Z P C     ... H vom Akku
SUB L          95        S Z P C     ... L vom Akku
SUB n          D6nn      S Z P C     ... nachfolgendes Byte
SUB (HL)       96        S Z P C     ... Adresse in HL
SUB (IX+d)     DD96dd    S Z P C     ...   Adresse  IX+d  vom
Akku
SUB (IY+d)     FD96dd    S Z P C     ...   Adresse  IY+d  vom
Akku
XOR A          AF        S Z P C=0 Exklusiv-oder-verknüpft A
XOR B          A8        S Z P C=0   ... und B          und A
XOR C          A9        S Z P C=0   ... und C
XOR D          AA        S Z P C=0   ... und D
XOR E          AB        S Z P C=0   ... und E
XOR H          AC        S Z P C=0   ... und H
XOR L          AD        S Z P C=0   ... und L
XOR n          EEnn      S Z P C=0   ...   m.   nachfolgendem
Byte
XOR (HL)       AE        S Z P C=0   ... & Adresse HL
XOR (IX+d)     DDAEdd    S Z P C=0   ... & Adresse IX+d
XOR (IY+d)     FDAEdd    S Z P C=0   ... & Adresse IY+d
```

14. TRICKS UND FORMELN IN BASIC

Mit den folgenden Tricks und Formeln können Sie viele knifflige Aufgaben bei der Programmierung leichter lösen, lästige Errors können umgangen werden, und Sie sparen viel Zeit. Beginnen wir mit mathematischen Formeln.

Vielleicht haben Sie schon festgestellt, daß es im BASIC des CPC (wie auch anderer Computer) nur Befehle für den natürlichen (LOG) und den Logarithmus zur BASIS 10 (LOG10) gibt. Zum Glück gibt es eine sehr einfache Formel, mit der *Logarithmen* zur beliebigen Basis n berechnet werden können. Sie lautet:

$LOG_n(x) = LOG(x)/LOG(n)$

Auch bei den Umkehrungen für die trigonometrischen Funktionen sind BASIC-Interpreter meistens nicht sehr gut bestückt. So fehlen *ARCUSSINUS* und *ARCUSCOSINUS*. Beide lassen sich aber aus dem ARCUSTANGENS (ATN) berechnen:

```
ARCSIN= ATN(x/SQR(1-x*x))
ARCCOS= -ATN(x/SQR(1-x*x))+PI/2
```

Um Brüche zu kürzen, müssen Sie den größten gemeinsamen Teiler zweier Zahlen berechnen. Sehr häufig wird dafür der *Algorithmus des Euklid* benutzt, der den Vorteil hat, daß er sehr schnell arbeitet.
Die Variablen P und Q enthalten die beiden Zahlen, deren größter gemeinsamer Teiler berechnet werden soll. Dann wird der Rest aus der Division P/Q der Variablen R zugewiesen. P erhält den alten Wert von Q, Q wird der Wert von R zugewiesen. Dieser Vorgang wird solange wiederholt, bis der Rest 0 wird. In diesem Fall enthält P den größten gemeinsamen Teiler.
Diese doch etwas trockene Erklärung sehen wir uns am besten noch einmal an einem Beispiel an:

P	Q	R	
56	21	14	Anfangswerte
21	14	7	1. Durchlauf
14	7	0	2. Durchlauf
7	0		P enthält GGT

Das Listing ist erfreulich (oder erschreckend?) kurz:

```
10 p= Zahl 1: q= Zahl 2: r=1
20 WHILE r ungleich 0: r= p MOD q: p=q: q=r: WEND
```

Das R=1 ist wichtig, weil sonst die Bedingung nach WHILE sofort erfüllt wäre, bevor die Schleife überhaupt ein einziges Mal durchlaufen wurde.

Wenn Sie einmal von einer Zahl wissen möchten, wieviele *Stellen* sie *vor dem Komma* hat, so sollten Sie sie in folgende Formel einsetzen:

```
S= INT(LOG10(ABS(x))+1)
```

Die ABS-Funktion erlaubt Ihnen, auch negative Zahlen einzusetzen, da LOG10 nur für positive Werte definiert ist.

Die nächste Formel liefert Ihnen eine *Zufallszahl* aus dem Bereich von a bis b, nicht (wie sonst üblich) aus dem Bereich 0 bis b:

```
x= INT (RND(1)*(b-a+1)+a)
```

Auch der nächste Tip bezieht sich auf die oft geheimnisvolle Welt der Mathematik, wenn auch nur noch sehr entfernt.

Die *Basisumwandlung* von Hex- und Binärzahlen mittels & und &x hat leider einen kleinen Schönheitsfehler. Werden die Zahlen größer als &7FFF bzw. ist das höchstwertige Bit (Nr. 15) auf 1, so erhalten Sie einen negativen Wert. Sollte dies auftreten, addieren Sie bitte 65536 dazu, und es erscheint das "echte Ergebnis".

Viele Textverarbeitungsprogramme zeichnen sich dadurch aus,

daß ein eingegebener Text verschieden formatiert werden kann. Zu diesen Möglichkeiten gehört oft auch die *zentrierte Ausgabe*. Das läßt sich auch durch eine einfache BASIC-Formel erreichen:

```
PRINT TAB ((Zeilenlänge - LEN(TEXT$))/2); TEXT$
```

Die Zeilenlänge ist je nach Modus verschieden, TEXT$ enthält die auszugebende Zeile.

Der letzte Trick bezieht sich auf Strings. Dabei geht es um eine unangenehme Eigenschaft des CPC, die er mit anderen BASIC-Interpretern teilt. Wird versucht, den ASCII- Wert eines leeren Strings (z.B. ASC("")) auszugeben, so reagiert Ihr Computer mit einem lapidaren "Improper Argument". Das läßt sich so vermeiden:

```
PRINT ASC(a$+CHR$(0))
```

Jetzt stimmt die Sache wieder.

I. SPEICHERBELEGUNGSPLAN

Auf den folgenden Seiten sind die Funktionen aller Speicherbytes (soweit bekannt) aufgelistet. Natürlich können sich dabei Fehler einschleichen. Deshalb sind Experimente im Speicher mittels PEEK und POKE mit ein wenig Vorsicht zu genießen. Nichts desto trotz sollten Sie sich nicht von eigenen Nachforschungen abhalten lassen - aber vorher Ihr Programm sichern!

0000 - 3FFF	Betriebssystem-ROM
0000 - 003F	Kopie des ROMs für Bankswitching
0040 - 013F	Eingabepuffer & Arbeitsbereich
0170 - AB7F	BASIC-Speicher
3800 - 3FFF	Zeichengenerator im ROM
AB80 - ABFF	Zeichengenerator f. selbstdefinierbare Zeichen
AC00	Flag für Leerzeichen löschen
AC01 - AC03	Erweiterungssprung für READY-Modus
AC04 - AC06	" " ERROR-Behandlung
AC07 - AC09	" " Befehlsausführung
AC0A - AC0C	" " Funktionsberechnung
AC0D - AC0F	" " Konstante holen
AC10 - AC12	" " BASIC-Zeileneingabe
AC13 - AC15	" " LIST
AC16 - AC18	" " Ziffernumwandlung
AC19 - AC1B	" " Operatoren
AC1C	Flag für AUTO
AC1D - AC1E	AUTO-Zeilennummer
AC1F - AC20	Schrittweite für AUTO
AC21	letzte Streamnummer
AC22	Eingabekanal
AC23	letzte Positon des Druckers
AC24	WIDTH
AC25	letzte Position auf Kassette

```
AC26          Flag für FOR-NEXT
AC27 - AC2B   Zwischenspeicher für FOR
AC2C - AC2D   Adresse für NEXT
AC2E - AC2F      "      "  WEND
AC34 - AC35      "      "  ON-BREAK
AC38 - AC43   Ton-Schlange 0
AC44 - AC4F    "      "     1
AC50 - AC5B    "      "     2
AC5C - AC6D   Bereich für Interrupt 0
AC6E - AC7F      "      "       "     1
AC80 - AC91      "      "       "     2
AC92 - ACA3      "      "       "     3
ACA4 - ADA3   Eingabepuffer
ADA6 - ADA7   Adresse des ERRORs
ADA8 - ADA9   BASIC-Programmzeiger nach ERROR
ADAA          Nummer des ERRORs
ADAB - ADAC   BASIC-Programmzeiger nach Interrupt
ADAD - ADAE   Adresse der unterbrochenen Zeile
ADAF - ADB0   Adresse für ON-ERROR
ADB1          Flag: ON-ERROR aktiv
ADB2          Kanal-Status
ADB3          ENT
ADB4          ENV
ADB5 - ADB6   Periode
ADB7          Rausch-Periode
ADB8          Lautstärke
ADB9 - ADBA   Länge
ADBB - ADBC   ENV und ENT
ADCB - ADCF   Zwischenspeicher f. Fließkommaarithmetik
ADD0 - AE03   Tabelle für Skalar-Variablen
AE04 - AE05   FN-Tabelle
AE06 - AE0B   Array-Tabelle
AE0C - AE25   vordefinierte Variablentypen A bis Z
AE2D          Trennzeichen für INPUT
AE2E - AE2F   Adresse für READ
AE30 - AE31   Adresse für DATA
AE32 - AE33   Speicher für Pointer in BASIC-Stack
AE34 - AE35   aktuelle Befehlsadresse
AE36 - AE37   aktuelle Programmzeilenadresse
AE38          Flag für TRACE
```

AE3F - AE40	Startadresse für LOAD-Befehl
AE41	Flag für CHAIN MERGE
AE42	Filetyp
AE43 - AE44	Länge der Datei
AE45	Programmschutzflag
AE46 - AE78	Arbeitsbereich f. Umwandlung in ASCII
AE72 - AE73	CALL-Adresse
AE74	Bankswitchingmodus f. CALL
AE75 - AE76	HL-Register für CALL
AE77 - AE78	SP-Register für CALL
AE79	Tabulatorweite
AE7B - AE7C	HIMEM-Zeiger
AE7D - AE7E	Zeiger auf Ende des freien RAM-Bereichs
AE7F - AE80	" " Start des freien RAM-Bereichs
AE81 - AE82	" " BASIC-Programmstart
AE83 - AE84	" " Programmende
AE85 - AE86	" " Variablenstart
AE87 - AE88	" " Arraystart
AE89 - AE8A	" " Arrayende
AE8B - B08A	Stack für BASIC (FOR, GOSUB etc.)
B08B - B08C	BASIC-Stackpointer
B08D - B08E	Zeiger auf Stringstart
B08F - B090	Zeiger auf Stringende
B09A - B09B	Zeiger für Stringstack
B09C - B0B9	Stringstack
B0BA - B0BC	Stringdescriptor
B0C1	Variablentyp
B0C2 - B0C3	diverse Adressen
B100 - B1AB	Arbeitsbereich für Steuerung des Betriebssys.
B1C8	laufender Bildschirmmodus
B1CA	Bildschirmoffset
B1CB	Bildschirmadresse
B1CC - B1D6	verschiedene Zwecke
B1D7 - B1D8	Blinkzeiten
B1D9 - B20B	verschiedene Register f. blinkende Farben
B20C	aktuelles Window
B20D - B276	Parameter für Windows
B285 - B286	Cursorposition (Zeile, Spalte)
B287 - B28B	verschiedene Register für Windows
B28C - B327	" " " Bildschirm

B328 - B329	ORIGIN-X
B32A - B32B	ORIGIN-Y
B32C - B4DD	verschiedene Register für Grafikausgabe
B4DE - B550	" " " Tastaturabfrage
B551 - B7FF	" " " Sound
B800 - B8DC	" " " Cassette
B807 - B816	Name des INPUT-Files
B84C - B85B	Name des OUTPUT-Files
B8D1	Schreibgeschwindigkeit
B8DD	Insert-Flag
B8E4 - B8E7	Zufallszahl
B8E8 - B8F6	Zwischenspeicher für Fließkommaarithmetik
B8F7	Flag für DEG bzw. RAD
BF00 - BFFF	Prozessor-Stack
C000 - FFFF	Video-RAM
C000 - FFFF	Interpreter-ROM
C000 - FFFF	Erweiterungs-ROM

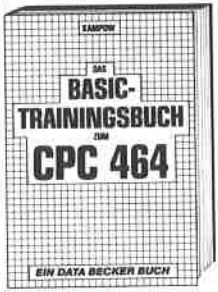
II. STICHWORTVERZEICHNIS

Z

Bücher zum SCHNEIDER CPC 464

Wer sich für den Schneider CPC 464 entschieden hat, der findet hier den optimalen Einstieg. Neben den wichtigsten Hinweisen zu Handhabung und Anschlußmöglichkeiten gibt es auch erste Hilfen für eigene Programme. Zahlreiche Abbildungen und Bildschirmfotos ergänzen den Text. Das ideale Buch also für jeden, der mit dem CPC 464 das Computern beginnen will.

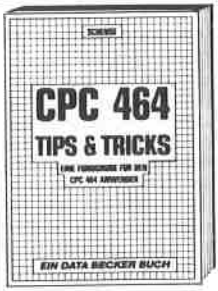
Sczcepanowski
CPC 464 für Einsteiger
206 Seiten, DM 29,-
ISBN 3-89011-037-1



Kampow
Das BASIC-Trainingsbuch zum CPC 464

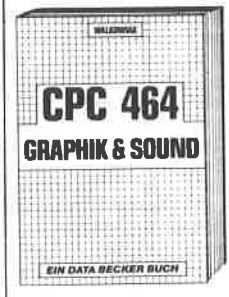
CPC 464 BASIC? Kein Poblem! Mit diesem Trainingsbuch lernen Sie von Grund auf nicht nur die einzelnen Befehle und ihre Anwendungen, sondern auch einen richtig sauberen Programmierstil. Von der Problemanalyse über den Flußplan bis zum fertigen Programm. Dazu viele Übungsaufgaben mit Lösungen und zahlreichen Beispielen. Schlichtweg unentbehrlich.

285 Seiten, DM 39,-
ISBN 3-89011-038-X



Englisch/Germer/
Scheuse/Thrun
CPC 464 Tips & Tricks
Eine Fundgrube für den CPC-464-Anwender
263 Seiten, DM 39,-
ISBN 3-89011-039-8

Rund um den CPC 464 viele Anregungen und wichtige Hilfen. Von Hardwareaufbau, Betriebssystem, BASIC-Tokens, Zeichnen mit dem Joystick, Anwendungen der Windowtechnologie und sehr vielen interessanten Programmen wie einer umfangreichen Dateiverwaltung, Soundeditor, komfortablem Zeichengenerator bis zu kompletten Listings spannender Spiele bietet das Buch eine Fülle von Möglichkeiten.



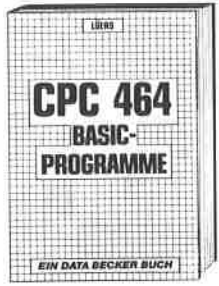
Walkowiak
CPC 464 Graphik & Sound
220 Seiten, DM 39,-
ISBN 3-89011-050-9

In diesem erstklassigen Buch wird gezeigt, wie man die außergewöhnlichen Grafik- und Soundmöglichkeiten des CPC 464 nutzt. Natürlich mit vielen interessanten Beispielen und nützlichen Hilfsprogrammen. Aus dem Inhalt: Grundlagen der Grafikprogrammierung, Sprites, Shapes und Strings, mehrfarbige Darstellungen, Koordinatentransformation, Verschiebungen, Drehungen, Rotation, 3-D-Funktionsplotter, CAD, Synthesizer, Miniorgel, Hüllkurven und vieles mehr.

Walkowiak
Adventures - und wie man sie auf dem CPC 464 programmiert
320 Seiten, DM 39,-
ISBN 3-89011-088-6

Ein faszinierender Führer in die phantastische Welt der Abenteuer-Spiele. Hier wird gezeigt, wie Adventures funktionieren, wie man sie erfolgreich spielt und wie man eigene Adventures auf dem CPC 464 programmiert. Der Clou des Buches ist neben vielen fertigen Adventures (bis hin zum trickreichen Grafikadventure!) ein kompletter ADVENTURE-GENERATOR, mit dem das Selberprogrammieren packender Adventures zum Kinderspiel wird.

Spitzenprogramme vom Disassembler bis zum Sporttabellenprogramm - mit spannenden Superspielen und kompletten Anwendungsprogrammen: mit Hexdump, Grafik- und Soundeditor, deutsche Umlaute, Mathematikzeichensatz, ausführliche Fehlermeldungen, Variablenreferenzliste, Kalender, Disassembler, Langspielplattenverwaltung Texteditor, Codeknacker, Zahlsystemumrechner.



Lüers
CPC 464 BASIC-Programme
185 Seiten, DM 39,-
ISBN 3-89011-049-5

DAS STEHT DRIN:

PEEKs, POKEs und CALLs werden zum Anlaß genommen eine wirklich leichtverständliche Einführung in Betriebssystem und Maschinensprache des CPC zu geben. Daß sich dabei viele interessante Programmier- und Anwendungsmöglichkeiten des Schneider-Computers ergeben, ist ein praktischer Nebeneffekt.

Aus dem Inhalt:

- Hardwareaufbau des CPC
- Betriebssystem und Interpreter
- PEEK & POKE - CALL
- Binärarithmetik
- Speicher schützen
- Bankswitching - ROM auslesen
- Video-RAM - Grafik - Scrolling
- BASIC-Interrupt
- Speicherung von BASIC-Zeilen
- Garbage Collection
- Aufbau und Funktionsweise des Z80
- Adressierungsmöglichkeiten
- Nützliche Maschinenroutinen

UND GESCHRIEBEN HAT DIESES BUCH:

Hans Joachim Liesert ist EDV-Fachbuchautor, der es versteht auch komplizierte Sachverhalte verständlich darzustellen.

ISBN 3-89011-092-4