Carsten Straush

# CPC 464 – Programmieren in Maschinensprache

Speicheraufbau
Z 80-Befehlssatz
Wichtige ROM-Routinen
Disassembler
Monitor

Markt & Technik Verlag

CIP-Kurztitelaufnahme der Deutschen Bibliothek

**Straush, Carsten:**
CPC 464, Programmieren in Maschinensprache :
Speicheraufbau, Z-80-Befehlssatz, wichtige ROM-Routinen, Disassembler, Monitor / Carsten Straush. —
Haar bei München : Markt-und-Technik-Verlag, 1985.
ISBN 3-89090-166-2



15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
89 88 87 86 85

ISBN 3-89090-166-2


Einbandgestaltung: Grafikdesign Heinz Rauner
Druck: Schoder, Gersthofen
Printed in Germany



# Vorwort

Der CPC, als Personalcomputer angepriesen, hat sich in kürzester Zeit zum Freakcomputer gemausert. Vielfältige Änderungsmöglichkeiten und klare Strukturen im Betriebssystem bieten geradezu ideale Voraussetzungen, durch direkte Eingriffe neue Freiräume für den Programmierer zu schaffen; vorausgesetzt, man weiß wie.

Dazu ist jedoch eine enge Kenntnis der inneren Abläufe der Maschine nicht nur nützlich, sondern auch nötig, um Fehler zu vermeiden und alle Chancen wirklich nutzen zu können. Da die vom Hersteller allgemein verfügbaren Informationen hier nicht ausreichen, ist man auf Ausprobieren und Experimentieren angewiesen. Anwendungsbezogene Hintergrundinformationen sind Mangelware.

Um hier ein wenig Abhilfe zu schaffen, habe ich dieses Buch geschrieben. Es verbindet Informationen über den Ablauf interner Prozesse beim CPC mit einer ganzen Palette von Anwendungsmöglichkeiten und Tricks. Die meisten Eingriffe sind dabei bereits via BASIC möglich. Wo es dann nicht mehr weitergeht, kann man mit dem mitgelieferten Monitor und Disassembler in die Maschinensprache durchstarten. Eine ausführliche Darstellung der Z-80-Maschinensprache schafft die Voraussetzung zum Weiterforschen und Experimentieren. Viel Spaß dabei.

Carsten Straush

# Inhaltsverzeichnis

**Einleitung** 11

**1 Grundsätzliches** 15

1.1 Datenformate 15
1.2 Zahlensysteme 16
1.2.2 Konvertieren zwischen Binärsystem und Dezimalsystem 18
1.2.2 Konvertierung zwischen Dezimal- und Hexadezimalsystem 21
1.2.3 Konvertierung zwischen Hexadezimal- und Dualsystem 23
1.3 Codes 24
1.3.1 Der ASCII-Code 25
1.3.2 Die Codierung des BASIC-Programms 28
1.3.3 Die Ablage der Variablen im Speicher 35

**2 Speicherabfrage und -veränderung** 49

2.1 Übersicht über den Speicheraufbau 49
2.1.1 Die Restarts 53
2.2 Einfache Routinen für die Speicheranalyse 55
2.2.1 Die Ablage von Maschinenprogrammen 56
2.2.2 Das Umschalten zwischen ROM und RAM 58
2.3 Ein Monitor für den CPC 66
2.3.1 Anforderungen an den Monitor 66
2.3.2 Das Programm 67

| | | |
|---|---|---|
| **3** | **Die Systemhardware** | 79 |
| 3.1 | Übersicht über das Gesamtsystem | 80 |
| 3.1.1 | Die Aufgaben des Prozessors im System | 83 |
| 3.2 | Der MEMORY-Bereich | 87 |
| 3.2.1 | Die Bildschirmdarstellung | 89 |
| 3.2.2 | Die Freischaltung der Speicherbausteine | 98 |
| 3.3 | Die IO-Bereich | 100 |
| 3.3.1 | Der Schnittstellenbaustein 8255 | 101 |
| 3.3.2 | Der Soundchip | 106 |
| **4** | **Das Herzstück des Systems: der Z80 A** | 113 |
| 4.1 | Die 16-Bit-Register | 114 |
| 4.1.1 | Die Adreßregister | 114 |
| 4.2 | Die Universalregisterpaare BC, DE und HL | 119 |
| 4.3 | Der Akkumulator und die Flags | 120 |
| 4.4 | Spezialregister (I,R) | 124 |
| 4.5 | Eine Möglichkeit zur Registerkontrolle: 'GOMACH' | 125 |
| **5** | **Der Befehlssatz des Z80-Prozessors** | 137 |
| 5.1 | Befehlsarten und Adressierungstechniken | 138 |
| 5.2 | Die Analyse der Befehle beim Z80 | 143 |
| 5.3 | Datentransfer | 146 |
| 5.3.1 | Interner Datentranfer | 147 |
| 5.3.2 | Externer Datentransfer | 153 |
| 5.4 | Arithmetik- und Logikbefehle | 166 |
| 5.4.1 | Die Arithmetikbefehle | 167 |
| 5.4.2 | Die Logikbefehle | 169 |
| 5.4.3 | Verschiebebefehle | 178 |
| 5.4.4 | Kommandos für die Bitmanipulation | 185 |
| 5.4.5 | Sonderbefehle für den Akkumulator und die FLAGs | 187 |
| 5.5 | Sprungbefehle | 190 |
| 5.6 | Systemsteuerbefehle | 203 |
| **6** | **Ein Disassembler für den CPC** | 207 |
| 6.1 | Die Disassemblerprozedur | 210 |





| | | |
|---|---|---|
| **7** | **Die Unterteilung des Firmware-Speichers** | 227 |
| **8** | **Systemfunktionen und Jumpblockbenutzung** | 235 |
| 8.1 | Tastaturabfrage und Dateneingabe über Tastatur | 235 |
| 8.2 | Die Kassettenspeicherung | 243 |
| 8.3 | Die Druckersteuerung | 257 |
| 8.4 | Grafikroutinen | 259 |
| **9** | **Anhang: Codes beim CPC 464** | 265 |
| Index | | 272 |
| Übersicht weiterer Markt&Technik-Bücher | | 277 |

# Einleitung

Im Vergleich zu anderen Homecomputern weist der Schneider CPC 464 eine ganze Reihe von Besonderheiten auf. Nicht nur, daß er bei einem sehr günstigen Preis-/Leistungsverhältnis die Eigenschaften eines Personalcomputers zum Preis eines Homecomputers bietet; auch bei der Konzeption der Hardware und der Software wurden neue Wege beschritten.

Die Hauptkennzeichen des Systems sind Kompaktheit, Integration, jedoch bei gleichzeitig stärkeren Änderungsmöglichkeiten als bei anderen Homecomputern üblich. Dies sieht auf den ersten Blick wie ein Widerspruch aus, der sich jedoch relativ leicht lösen läßt. Der CPC arbeitet mit einer Reihe von hochintegrierten Bausteinen, die unter dem Tastaturgehäuse verborgen liegen. So ist zum Beispiel ein einziger schwarzer Chip, halb so groß wie eine Streichholzschachtel, für die komplette Sounderzeugung zuständig, und auch in den anderen Bereichen ist Integration Trumpf: wenige Bausteine leisten viel.

Der wohl wichtigste Unterschied zu anderen Computern besteht nun darin, daß die Schnittstellen zwischen den einzelnen Bausteinen und auch zwischen Betriebssystemteilen variabel gestaltet sind. Sie können vom Benutzer und natürlich auch vom System selbst verändert werden. So überprüft der CPC zum Beispiel beim Einschalten, welche Arten von Speichern er zur Verfügung hat und übergibt dann an denjenigen mit der höchsten Priorität. Die Belegung von Speicherbereichen ist ebenfalls sehr variabel gestaltet:

Der Bildschirmspeicher, der ein Viertel des veränderbaren Speichers, des RAM, einnimmt, kann an vier verschiedenen Stellen liegen. Die Kenntnis dieser Möglichkeiten eröffnet dem Benutzer, besonders wenn er an der Grenze zwischen BASIC und Maschinensprache operiert, teilweise ungeahn-

te Freiräume. So kann er die Speicheraufteilung seinen Wünschen anpassen, seine Programme so aufbauen, daß nicht unnötig Platz verbraucht wird, oder umgekehrt: versteckte Speicherbereiche noch zur Ablage herangezogen werden können.

Noch weitere Möglichkeiten ergeben sich, wenn man Routinen des Betriebssystems mit heranzieht, um neue Systemfunktionen zu erhalten. So ist es zum Beispiel mit BASIC nicht möglich zu überprüfen, ob ein Drucker angeschlossen beziehungsweise eingeschaltet ist. Auch eine Überprüfungsmöglichkeit, ob ein auf Band gespeicherter Text oder ein Programm wirklich eine 1:1-Kopie des aktuellen Speicherinhaltes ist, fehlt. Der als Ersatz eingebaute Befehl CAT überprüft nämlich nur, ob ein Text oder ein Programm ladbar ist, nicht jedoch, ob es auch inhaltlich mit dem momentanen Speicherinhalt übereinstimmt. Ein echtes VERIFY, wie bei anderen Homecomputern, existiert beim CPC nicht. Die Kenntnis des Speicheraufbaus ermöglicht es auch, sehr viel schneller, direkter und programmplatzsparender die Tastaturbelegung oder den Zeichensatz zu ändern. Auch die Vergrößerung oder Verkleinerung von bestimmten Speicherbereichen, das Schaffen eines geschützten Speicherbereichs, in dem man dann Maschinenprogramme und Daten sicher lagern kann, bieten weitere interessante Anwendungsmöglichkeiten. Es lohnt sich also, sich mit diesen Änderungsmöglichkeiten zu befassen und das Innenleben der Maschine etwas näher zu erforschen.

Da wir dabei nicht nur an der Oberfläche kratzen, sondern auch in die Tiefe des Systems vorstoßen wollen, müssen wir zunächst ein wenig Systematik betreiben, um bei den fantastischen Möglichkeiten der Maschine nicht ins Schlingern zu geraten. Im **ersten Kapitel** geht es denn auch zuerst um Grundsätzliches. Die verschiedenen Datenformate, die beim CPC verwendet werden, werden erläutert. Wir gehen etwas näher auf die drei in Folge benötigten Zahlensysteme Dezimal-, Dual- und Hexadezimal-System ein und zeigen auf, wie diese schon mit BASIC-Befehlen miteinander konvertiert werden können. Das bei anderen Computern notwendige lästige Umrechnen der einzelnen Werte ineinander von Hand entfällt hier. Im dritten Teil wird es dann um die verschiedenen Codes und Codierungen gehen, mit denen der CPC arbeitet. Wir werden uns damit beschäftigen, wie Texte, Daten und Programme abgespeichert werden.

Im **zweiten Kapitel** schaffen wir dann die Voraussetzungen für einen näheren Einblick in die Maschine. Wir entwickeln einige Routinen und kleinere BASIC-Programme, mit denen es möglich ist, die gerade besprochenen Codes und Zahlenwerte ineinander umzuwandeln und auf dem Bildschirm



darzustellen, um uns Speicherbereiche auszudrucken und diese wahlweise als Texte oder Zahlen darzustellen. Gegen Ende entwickeln wir dann aus diesen Routinen ein vollständiges Programm zum Lesen und Verändern von Datenbereich, als Hilfsmittel zur Maschinenspracheprogrammierung: einen Monitor.

Mit den grundsätzlichen Kenntnissen aus Kapitel 1 und den Softwarehilfen sind wir nun in der Lage, uns mit dem Inneren des CPC näher zu beschäftigen. Wir lernen die einzelnen Bausteine, ihre Funktionen und ihr Zusammenwirken anhand einiger Beispiele und Anwendungen kennen. Ausgehend von einer Groborientierung über den Aufbau des Systems von der Hardwareseite gehen wir dann auf das Herzstück des Systems, den Z80-A-Prozessor näher ein. Wir schildern den Aufbau der CPU, die Funktionen und Eigenschaften der verschiedenen Register, den Ablauf von Maschinenprogrammen und gehen auf die verschiedenen Adressierungsarten und den Ablauf von Datentransferoperationen näher ein.

Nach diesem Überblick wenden wir uns in **Kapitel 5** den einzelnen Befehlen gruppenweise zu. Wir beschreiben den kompletten Befehlssatz des Z80 anhand ausgewählter praktischer Anwendungen im Rahmen des Systems. So lesen wir Daten aus den geschützten ROM-Bereichen aus, verschieben Speicherbereiche, zum Beispiel den Grafikbildschirm im Speicher, erklären, wie mit den Arithmetikbefehlen Bits verändert und geprüft werden können, und geben einige Tips und Tricks für die Unterprogramm-Programmierung in Maschinensprache weiter. Auch werden wir auf die Spezialbefehle zur Blockverschiebung, zur Unterbrechungsbehandlung und zur Ein-/Ausgabesteuerung näher eingehen. Aus der Kenntnis des Befehlssatzes entwickeln wir dann in **Kapitel 6** ein Programm für die Übersetzung von Maschinencode aus dem Speicher, einen Disassembler, und verbinden diesen mit dem Monitor aus Kapitel 2 zu einer komplexen Softwarehilfe für die Maschinenspracheprogrammierung.

Noch anwendungsbezogener gestalten sich die letzten beiden Kapitel dieses Buches. Im **Kapitel** 7 analysieren wir die Speicherstruktur, wie sie bei Benutzung des normalen BASIC vom Betriebssystem hergestellt wird. Wir nehmen eine Feintrennung zwischen den einzelnen Speicherbereichen vor und zeigen Änderungsmöglichkeiten im Speicheraufbau auf.

**Kapitel 8** beschäftigt sich dann mit dem Ablauf bestimmter Systemfunktionen im Rahmen des BASIC: Tastaturabfrage und übersetzung, Abspeicherung von Programmen und Daten auf Kassette, Datenverkehr mit externen Geräten, wie dem Drucker und Routinen, die beim Arbeiten mit der Gra-

fik nützlich sind, wie die Ausgabe von Zeichen in Maschinenprogrammen, Cursorführung und Farb- und Bildschirmhandling. Dabei verwenden wir eine ganze Reihe von Systemroutinen aus den verschiedenen Sprungtabellen. Im Gegensatz zu anderen Homecomputern stellt uns nämlich der CPC eine ganze Reihe von nützlichen Unterprogrammen in Sprungtabellen zur Verfügung, die wir gut in eigene Maschinenprogramme einbauen können und die fast jedes normal auftauchende Problem lösen helfen. Durch den Einbau derartiger Routinen in eigene Maschinenprogramme schafft man eine Art Zwischensprache, die von der Mächtigkeit der damit ausführbaren Befehle auf einer Ebene zwischen der eigentlichen Maschinensprache und dem Hochsprachenbasic liegt. Einige weiterführende Tips und Tricks und Schliche bei der Kombination von einzelnen Routinen bilden dann das Ende unserer Forschungsreise durch das Innere unseres Computers.



# 1 Grundsätzliches

## 1.1 Datenformate

Zusammengefaßt betrachtet können Computer als Informationsverarbeitungsmaschinen charakterisiert werden; auch unser CPC macht hier keine Ausnahme. Informationen werden in Form von Daten in die Maschine hineingegeben, zum Beispiel durch die Tastatur oder von Band, werden gegebenenfalls zwischengespeichert und dann irgendwann mit anderen Informationen verknüpft, werden wiederum zwischengespeichert und führen schließlich zu einer Ausgabe oder einer ähnlichen Reaktion des Rechners. Die Informationen, mit denen die Maschine arbeitet, können dabei in verschiedenen Einheiten eingelesen oder auch verarbeitet werden, je nach Verwendungszweck und Bestimmung der einzelnen Daten.

Die kleinste Informationseinheit ist das Bit. Es stellt das informationsmäßige Gegenstück für eine Ja/Nein-Entscheidung dar. Ein Bit kann gesetzt oder nicht gesetzt sein und damit die Zustände 1 beziehungsweise 0 annehmen. Zur Abspeicherung benötigt es genau einen Speicherplatz. Der CPC ist stärker als andere Computer bitorientiert. Daher nehmen auch bitorientierte Operationen wie das Setzen, Rücksetzen oder Testen von Bits und Verknüpfungen, die auf die Änderung von einem oder mehreren Bits abzielen, in den festen Maschinenprogrammen des Rechners, dem Betriebssystem, breiten Raum ein. Im Rahmen der Tastaturdekodierung ist zum Beispiel genau ein Bit dafür zuständig, ob eine bestimmte Taste auf Wiederholung des Zeichens bei längerem Drücken (Repeat) oder nur einmalige Ausgabe des Zeichens (kein Repeat) geschaltet ist. Auch sind die auf dem Bildschirm darstellbaren Zeichen Bit für Bit in einer Matrix aus 8 x 8 Bildpunkten, beziehungsweise 8 x 8 Bit, festgelegt. Ein Bit ist dafür zuständig, ob ein Speicherbereich an- oder abgeschaltet ist, und auch die Befehlsanalyse greift auf den Zustand bestimmter Bits zurück.

8 Bit parallel ergeben ein **Byte**. Ein Byte kann $2^8$ = 256 verschiedene Zustände annehmen, was reicht, um die Zahlen von 0-255 oder 256 verschiedene Befehle oder Textzeichen zu codieren. Dies läßt sich relativ leicht nach folgendem Schema errechnen:

Ein Bit kann zwei Zustände annehmen, nämlich 0 und 1. Nimmt man 2 Bit parallel, so ergeben sich 4 Kombinationsmöglichkeiten, nämlich 00, 01, 10 und 11. Bei 3 Bit parallel erhält man $2^3$ = 8, bei 4 Bit parallel $2^4$ = 16 Kombinationsmöglichkeiten und so weiter. Bei 8 Bit ergeben sich folglich $2^8$ = 256 Kombinationen.

Das Byte ist die dominante Informationsgröße für Homecomputer. Es wird beim CPC für alle Arten der Datenspeicherung und des Datentransportes benutzt. Auch ist das gesamte System von den Bausteinen her darauf angelegt jeweils 8 Bit und damit ein Byte gleichzeitig zu verarbeiten. Ist man bei Verwendung von nur einem Byte noch relativ eingeschränkt, so bietet der Rückgriff auf ein **Doppelbyte oder Adreßwort** mit 16 Bit bereits 65536 Kombinationsmöglichkeiten. Doppelbytes werden beim CPC im wesentlichen für das Rechnen mit ganzen Zahlen, den Integers, und bei der Adressierung von Speicherplätzen benutzt.

Neben diesen Größen, die mehrere Bits zu einer neuen Einheit zusammenfassen, gibt es noch einige Einheiten, die auf dem Byte aufbauen. Im Rahmen des Systems von Bedeutung sind das **K** mit 1024 Byte und das Doppelte hiervon, ein **Block von 2K** für die Abspeicherung von Programmen, Texten und Daten auf Kassette. Gibt der CPC also aus, daß er 5 Blöcke gespeichert hat, so ist die abgelegte Datei zwischen 8K = 4 Blöcke x 2K und 10K = 5 Blöcke a 2K lang. Die genaue Bestimmung der abgespeicherten Datenmenge hängt davon ab, zu wieviel Prozent der letzte Block mit Informationen beschrieben wurde.

## 1.2 Zahlensysteme

Bei unserer Reise in das Innere des CPC 464 werden wir drei verschiedene Arten der Darstellung von Zahlenwerten, drei ZahlenSysteme benutzen, auf die im Folgenden näher eingegangen werden soll. Am vertrautesten ist uns durch den täglichen Gebrauch das **Dezimalsystem**. Es ist aus 10 Ziffern (0-9) aufgebaut, beim Überschreiten der 9 folgt ein Übertrag in die nächsthöhere Stelle.



Die Basis des Dezimalsystems ist 10, und eine Zahl wie zum Beispiel 104 kann somit auch als

$$1*10^2 + 0*10^1 + 4*10^0 = 104$$

geschrieben werden.

Der Computer benutzt intern ein anderes Zahlensystem, welches auf dem Bit aufgebaut ist, das **Dualsystem**. Es besteht, wie wir schon gesehen haben, nur aus zwei Ziffern, der 0 und der 1, und folglich erfolgt bei 2 der Wechsel in die nächsthöhere Stelle. Die Basis des Systems ist also die 2. Ein Hochzählen würde also hier beginnend mit 00 wie folgt lauten: 00,01,10,11. Um die Zahlen von 0-3 im Dualsystem abzubilden, sind also bereits 2 Stellen notwendig. Die Umrechnung einer Dualzahl in ein anderes System, zum Beispiel in das Dezimalsystem, funktioniert hier analog, wie schon bei der Darstellung des Dezimalsystems gezeigt. Zum Beispiel würde sich die Zahl 1101 als

$$1*2^3 + 1*2^2 + 0*2^1 + 1*2^0 = 1101$$

schreiben lassen, was ausgerechnet den Wert 13 im Dezimalsystem ergeben würde. Das Dual- oder auch Binärsystem benötigt also bei nur 2 zu unterscheidenden Ziffern erheblich mehr Stellen als das Dezimalsystem. Durch die Beschränkung auf nur 2 Ziffern, denen 2 elektrische Zustände entsprechen, ist das Binärsystem für den Computer jedoch relativ einfach verarbeitbar.

Wegen der Vielzahl der Stellen ergeben sich allerdings für den Benutzer Probleme bei der Übersichtlichkeit. Daher hat sich die Verwendung eines weiteren Systems eingebürgert, des Hexadezimalsystems. Die Basis dieses Systems ist 16, was dazu führt, daß jeweils 4 Stellen im Dualsystem zu 1 Stelle im **Hexadezimalsystem** zusammengefaßt werden können. Aufgrund der größeren Basis im Vergleich zum Dezimalsystem werden 6 zusätzliche Ziffern benötigt. Hier greift man auf die Buchstaben A-F zurück. Nach der 9 wird also im Hexadezimalsystem ohne Stellenübertrag A,B,C,D,E,F weitergezählt, bevor der Wechsel in die nächste Stelle erfolgt. So entspricht also 0F im Hexadezimalsystem dem Dezimalwert 15. Die Umrechnung bei mehreren Stellen erfolgt analog zu den beiden oben dargestellten Systemen:

So würde zum Beispiel die Speicherobergrenze für die BASIC-Variablen beim Einschalten (AB7F) nach dieser Methode als

$$10*16^3 + 11*16^2 + 7*16^1 + 15*16^0 = 43903$$

umgerechnet werden. Für die Ziffern A-F wurden hier bereits die entsprechenden Werte im Dezimalsystem eingesetzt.

Glücklicherweise müssen wir uns jedoch mit derart zeitaufwendigen Umformungen vom einem Zahlensystem in ein anderes nicht weiter beschäftigen, da uns das Standard-BASIC des CPC bereits eine Reihe von Befehlen und Zusätzen zur Verfügung stellt, um Zahlensysteme ineinander zu konvertieren. Dies geschieht mit den Befehlen **BIN$** und **HEX$** für die Umwandlung eines Dezimalwertes in das Binär- oder Hexadezimalsystem, beziehungsweise mit vorangestellten Sonderzeichen, die angeben, daß der nachfolgende Wert im Binär- oder Hexadezimalsystem geschrieben wurde in der umgekehrten Richtung. Die Ausgabe im letzten Fall erfolgt dann im Dezimalsystem. Die Umwandlung zwischen Binär- und Hexadezimalsystem oder vom Hexadezimalsystem in der umgekehrten Richtung kann dann auf dem Umweg über das Dezimalsystem relativ problemlos erfolgen. Ein paar Beispiele mögen die Umformungsmöglichkeiten illustrieren.

### 1.2.1 Konvertierung zwischen Binärsystem und Dezimalsystem

Soll ein Wert, der im Dezimalsystem gegeben ist, in das Binärsystem überführt werden, so geht dies relativ einfach mit dem Befehl

**BIN$**

BIN$ erzeugt einen String, der den Wert der Zahl im Dualsystem enthält. So gibt uns zum Beispiel

```
PRINT BIN$(20)
```

die Darstellung des Dezimalwertes 20 im Dualsystem mit 10100 aus. Hierbei tritt gleich ein Ärgernis des CPC-BASIC zutage. Im Unterschied zur normalen Darstellung einer Dualzahl im 8-Bit-Format werden bei BIN$ die links von der höchsten Stelle befindlichen Vornullen weggelassen, es wurden also statt der üblichen 8 Bit hier nur 5 Bit ausgegeben. Da es sich bei BIN$ aber um einen String handelt, ist dieses Problem relativ einfach zu lösen. Lassen wir uns nämlich



```
PRINT RIGHT$("00000000"+BIN$(20),8)
```

ausdrucken, so erhalten wir jetzt unsere Binärdarstellung im gewünschten 8-Stellen-Format. Eine andere Möglichkeit bietet der Befehl selber: BIN$ erlaubt nämlich noch einen zweiten Parameter, der die Anzahl der gewünschten Stellen enthält. Mit

```
PRINT BIN$(20,8)
```

erreichen wir dasselbe. Dies ist insbesondere bei der Analyse des Speichers von Nutzen. Da bei der Abfrage einer Speicheradresse jeweils ein Byte ausgelesen wird, können wir nun den Wert jedes einzelnen Bits einer Speicherstelle direkt ablesen. Ersetzen wir die 20 im oberen Beispiel durch eine PEEK-Abfrage der gewünschten Adresse, zum Beispiel Speicherstelle 0, so erhalten wir durch

```
PRINT RIGHT$("00000000"+BIN$(PEEK(0)),8)
```

den Speicherinhalt in Binärdarstellung. Wir können somit überprüfen, ob und welche Bits gesetzt sind. In unserem Beispiel ist nur das Bit in der letzten Stelle ganz rechts gesetzt. Diese Art der Umformung bietet auch die einzige Möglichkeit, von BASIC aus etwas zu realisieren, was der CPC-Prozessor intern permanent anwendet, das Setzen, Rücksetzen beziehungsweise Testen von einzelnen Bits. Nachdem wir nämlich mit der Erweiterung von RIGHT$ einen formatierten String mit 8 Stellen erzeugt haben, ist es nun kein Problem mehr, ein bestimmtes Bit in diesem String zu setzen, rückzusetzen oder zu testen. Dies läßt sich relativ einfach mit anderen Stringbefehlen realisieren. Zum Beispiel können wir mit dem Befehl

**MID$**

ein beliebiges Bit aus unserem 8-Bit-String herauslösen und dann in einer IF-Abfrage überprüfen. Wollen wir zum Beispiel den Zustand des 5. Bits einer bestimmten Speicherstelle für eine Verzweigung benutzen, so können wir wie folgt vorgehen:

```
10 IF MID$(RIGHT$("00000000"+BIN$(Speicherstelle),8),5,1)="1"
THEN GOTO 100 ELSE 200
```

wobei die Zeilennummern 100 beziehungsweise 200 die Ansprungpunkte für eine Programmfortsetzung für den Fall, daß das Bit gesetzt ist (100) beziehungsweise nicht gesetzt ist (200), darstellen. Als zu überprüfenden

String setzen wir den formatierten String ein, aus dem wir dann ab der 5ten Stelle ein Zeichen herauslösen, um es mit dem gesuchten Wert zu vergleichen. Auch das Setzen beziehungsweise Rücksetzen eines bestimmten Bits ist auf ähnliche Art und Weise möglich. Dazu teilen wir unseren String mit den Befehlen

LEFT$ und MID$

in zwei Teilstrings auf und fügen dazwischen das gewünschte Bit ein. Das zu ersetzende Bit wird dabei ausgelassen. Als Testobjekt haben wir uns jetzt eine Speicherstelle im Systemvariablenbereich ausgesucht. Eine Änderung in der dritten Stelle von links würde dann wie folgt ablaufen (zur Vereinfachung haben wir den formatierten String in der Variablen z$ zwischengespeichert):

```
10 z$=RIGHT$("00000000"+BIN$(PEEK(46140)),8)
20 y$=LEFT$(z$,2)+"1"+MID$(z$,4)
30 POKE 46140,VAL("&x"+y$)
```

Wenn Sie vor dem Programmlauf auf die 3 im abgesetzten Tastenfeld drücken und danach noch einmal, werden Sie eine Änderung feststellen. Während im ersten Fall nur einmal die 3 ausgegeben wurde, wird sie bei längerem Druck nach Lauf des Programms permanent ausgegeben. Der Grund: Sie haben in der Bit-Tabelle, die angibt, ob eine Taste auf Repeat geschaltet wurde, oder nicht, das entsprechende Bit für diese Taste auf Repeat gesetzt. Als Training sollten Sie nun versuchen, dieses wieder rückgängig zu machen, wozu Sie nur eine einzige Stelle im obigen Programm ändern müssen. Auf diese Änderungsmöglichkeiten werden wir bei der Behandlung der Tastaturabfrage in Kapitel 7 und 8 noch intensiver eingehen. Hier ging es nur darum, das Prinzip des Bitänderns aufzuzeigen.

Beim Durchlauf dieses Programms haben wir auch schon die Umkehrung, das heißt die Umformung von Binärwerten zurück in das Dezimalsystem kennengelernt. In Zeile 10 wurde aus dem Dezimalwert der Speicherstelle 46140 ein formatierter Binärstring geformt, in dem wir dann in Zeile 20 das dritte Bit ausgetauscht haben. Zeile 30 beinhaltet die Rückübersetzung



und Speicherung der Dezimalzahl in 46140 durch POKE. Wir haben dazu zu unserem Binärstring das Kürzel &x addiert. Das "&" gibt dabei an, daß der nun folgende Wert in einem anderen als dem Dezimalsystem geschrieben ist. Das "x" ist das Kürzel für das Dualsystem. Gibt man hinter dem "&" keine nähere Spezifikation oder ein "H" ein, so wird der Wert als Hexadezimalzahl interpretiert. So gibt uns also

```
PRINT &x110010
```

den Dezimalwert für die Binärzahl 110010 (50) an. Durch den VAL-Befehl wurde der Binärstring wieder in eine Zahl umgewandelt.

### 1.2.2 Konvertierung zwischen Dezimal- und Hexadezimalsystem

Die Umwandlung zwischen Dezimalsystem und Hexadezimalsystem läuft ähnlich wie bei der Konvertierung zwischen Dual- und Dezimalzahlen ab. Maßgeblich für die Umformung sind hier der Befehl HEX$ - er erzeugt den Hexstring einer Dezimalzahl - und das schon behandelte Kürzel "&" gegebenenfalls mit angehängtem "H". So wird bei

```
PRINT HEX$(120)
```

die Darstellung von dezimal 120 im Hexadezimalsystem (78) ausgegeben. Sie sollten diese Art der Umformung einmal nach unserem oben erklärten Schema überprüfen. Leider funktioniert diese einfache Art der Umformung nur für maximal 4stellige Hexwerte. Lassen wir uns also einen Wert größer als 65535 übersetzen, so erscheint eine Overflow-Fehlermeldung. Etwas problematischer dagegen wird die Rückübersetzung. Geben wir

```
PRINT &78
```

ein, so erhalten wir wieder den Dezimalwert 120, und auch bei Hexwerten von 0000 bis 7FFF erfolgt die richtige Ausgabe problemlos. Bei hexadezimal 8000 "kippt" jedoch die Übersetzung. Wir erhalten nun plötzlich einen negativen Wert in der Ausgabe:

```
-32768.
```

Der Grund dafür liegt darin, daß, wie wir schon im vorigen Kapitel gesehen haben, das Wort mit 16 Bit auch für die Abspeicherung und das Rechnen mit Ganzzahlen, den Integers, benutzt wird.

**Integer-Variable** können im Vergleich zu Real-Variablen schneller verarbeitet werden, sind dabei jedoch auf ganze Zahlen zwischen -32768 und +32767 beschränkt im Gegensatz zu **Gleitkommawerten** zwischen 2.9 E-39 und 1.7 E+38. Eine solche Ganzzahl benötigt zur Abspeicherung nun 2 Byte bzw. 16 Bit. Dies entspricht 4 Hexziffern, da ja je 4 Bit zu einer Hexziffer zusammengefaßt werden können. Bei der Umwandlung von hex in dezimal mit "&" wird nun eine Integer-Variable angelegt und deren Wert danach ausgegeben. Da Hexwerte von mehr als 8000 als negativ interpretiert werden, kommt es zu dem negativen Wert.

Aber natürlich hat auch dieses Problem eine Lösung. Wir addieren nämlich einfach bei negativen Werten 65536 und erhalten so eine fortlaufende Darstellung von 0 bis 65535. Dazu wenden wir als kleinen Trick einen Booleschen Ausdruck an. Der CPC weist nämlich in Abfragen den einzelnen Ausdrücken je nach dem Wahrheitsgehalt der Abfrage für "wahr" eine "-1" und für "falsch" eine "0" zu. So ergibt zum Beispiel die falsche Aussage

```
PRINT 3>4
```

als Antwort "0". Ein richtiger Ausdruck wie

```
PRINT 5>2
```

würde dagegen mit "-1" bewertet. Wir haben nun eine Größe, die in Abhängigkeit von einer Entscheidung, z.B. "Wert kleiner null?", zwei Zustände ("0" und "-1") annehmen kann. Erweitern wir nun unseren Umformungsausdruck

```
PRINT &8000    zu    PRINT &8000-65536*(&8000<0)
```

so erhalten wir nun die richtige Darstellung. Da &8000 negativ ist, wird der Ausdruck &8000<0 "wahr" und nimmt damit den Wert -1 an. Die Multiplikation mit der ebenfalls negativen -65536 leistet dann den Rest. Werte kleiner als hex 8000 werden nun unverändert ausgegeben (der Ausdruck nimmt den Wert 0 für logisch "falsch" an), während für Zahlen größer/gleich hex 8000 die entsprechende Korrektur erfolgt.

Mit dieser Änderung können wir nun beliebige maximal vierstellige Hexzahlen in ihr dezimales Gegenstück überführen. Dazu ersetzen wir einfach die 8000 durch die umzuwandelnde Hexgröße.



### 1.2.3 Konvertierung zwischen Hexadezimal- und Dualsystem

Das Grundprinzip der Umwandlung zwischen Dual- und Hexzahlen hatten wir weiter oben bereits angerissen: Man wandelt die vorhandene Größe zuerst in einen Dezimalwert um und läßt den Computer dann diesen wieder in das andere System rückübersetzen. Ein paar Beispiele mögen die Übersetzung demonstrieren:

```
PRINT HEX$(&x1111000000001111)  ergibt: F00F

PRINT BIN$(&hA8A8)              ergibt: 1010100010101000
```

Diese Beispiele zeigen auch noch einmal den oben schon angesprochenen Zusammenhang zwischen Hex- und Dualwerten. Jeweils vier Binärstellen entsprechen einer Hexstelle. Das erste "A" im unteren Beispiel repräsentiert also die linken vier Stellen in der Binärdarstellung (1010), was wir leicht nach dem oben angesprochenen Schema ($1*2^3 + 0*2^2 + 1*2^1 + 0*2^0$ = dezimal 10 = hexadezimal 0A) wieder überprüfen können. Zur Demonstration dieser Zusammenhänge kann auch das folgende kleine Programm dienen. Es stellt nach Eingabe einer max. 2stelligen Hexzahl diese als Binärwert dar und formt diesen dann noch in eine Dezimalzahl um. Die einzelnen Binärstellen wurden dabei nicht mit Potenzen ($2^5$, $2^6$ usw.) beschriftet, sondern gleich mit deren Dezimalwerten, der besseren Darstellbarkeit halber von oben nach unten zu lesen.

```
10 REM ****************
20 REM ** Transformer **
30 REM ****************
40 CLS:INPUT"Bitte geben Sie den Hexwert ein (max. 2-stellig)";h$
50 IF LEN(h$)>2 THEN PRINT"nur 2-stellig":FOR t=1 TO 1000:NEXT t:
GOTO 40
60 CLS:PRINT"Hexwert:";h$:h$="&"+h4:PRINT" 1"
70 PRINT" 2 6 3 1"
80 PRINT" 8 4 2 6 8 4 2 1"
90 PRINT"-----------------"
100 b$=BIN$(VAL(h$)):b$=RIGHT$("00000000"+b$,8)
110 FOR i=1 TO 8:LOCATE 2*i,8
120 PRINT MID$(b$,i,1):NEXT i
130 PRINT:PRINT"Dezimalwert: ";VAL(h$)
```

Der einzige wirklich neue Punkt an diesem Programm ist Zeile 100. Hier wird der Hexwert h$ mit der VAL-Funktion in sein dezimales Gegenstück verwandelt und daraus dann wieder mit BIN$ ein Binärstring erzeugt. Während wir bis jetzt bei der Umformung hex zu binär Zahlenwerte direkt eingegeben haben, wurde hier jetzt mit Strings gearbeitet (h$), um den Hexwert als Hexzahl abspeichern zu können. Durch das Voranstellen des "&" in Zeile 60 haben wir die Zahlen- und Buchstabenkombination als Hexadezimalausdruck definiert, so daß dann die Umformung erfolgen konnte. Hätten wir den String ohne dieses Kürzel mit VAL bearbeitet, wäre der Wert dieser Funktion wie bei beliebigen anderen Buchstaben 0 gewesen.

Der zweite Teil von Zeile 100 dient nur noch der Vornullenaddition, die wir ja bereits bei der Umwandlung von Dezimalzahlen in Hexwerte benutzt haben. Der nun formatierte Binärstring wird mit der Schleife in Zeile 110, 120 mit jeweils einer Leerstelle - durch die Multiplikation von i mit 2 im LOCATE-Kommando - dargestellt. Zeile 130 gibt dann noch den Dezimalwert aus. Sie sollten das Programm einmal mit verschiedenen Werten durchspielen, um sich intensiv mit den einzelnen Zahlensystemen und ihren Zusammenhängen vertraut zu machen. Da der CPC nämlich eine ganze Reihe von Funktionen bitorientiert durchführt, werden wir um die eine oder andere Umformung nicht herumkommen.

## 1.3 Codes

Nachdem das letzte Kapitel sich noch im wesentlichen mit einem alle Homecomputer betreffenden Problembereich in Zusammenhang mit der Datendarstellung, den verschiedenen Zahlensystemen und ihrer Konvertierung ineinander beschäftigt hat, gehen wir nun auf die Abspeicherung von Daten näher ein.

Jede Größe, mit der der CPC arbeitet, sei es ein Text oder eine von mehreren Variablentypen, sei es ein BASIC-Programm oder die Abspeicherung von Grafiksymbolen, muß im Endeffekt an irgendeiner Stelle im variablen Speicher, dem RAM, als Zahlenwert oder eine Kombination von Zahlenwerten abgelegt sein, da der Prozessor intern nur Zahlen verarbeiten kann.



Dabei kommt es vor, daß Zahlen, genauer 8-Bit-Binärzahlen, zu größeren Zahlenwerten zusammengefaßt werden oder in irgendeiner Form eine Codierung für Nichtzahlen darstellen. Als Beispiel mag hier einmal eine 16-Bit-Adresse dienen: Der Prozessor benötigt, wie wir schon gesehen haben, eine 16-Bit-Adresse, um den gesamten Speicher abfragen zu können. Da eine Speicherstelle nur 8 Bit aufnehmen kann, wird die Adresse Lo-Hi abgespeichert, das heißt, in der Speicherstelle x liegen die unteren 8 Bit, in x+1 die oberen. Ein Beispiel für die Codierung mögen Strings bilden. Hier wird zum Beispiel einem "A" der Code 65 zugeordnet. Ein String "ABC" wird also gespeichert, indem nacheinander die Codes 65, 66 und 67 in aufeinanderfolgende Speicherstellen abgelegt werden. Auf die genaue Ablage wollen wir nun im folgenden näher eingehen.

### 1.3.1 Der ASCII-Code

Der wohl wichtigste Code bei der Speicherung von Daten ist der ASCII-Code. Dieser Code besteht aus 7 Bit und einem weiteren achten Bit, welches als Paritätsbit benutzt wird. Die Parität bestimmt man dabei, indem man die Einsen im Ergebnis zählt. Ist ihre Anzahl gerade, so wird das Paritätsbit auf 1 gesetzt, sonst bleibt es 0. Ein Beispiel:

Wert : 65 = 1000001 binär

Der Wert enthält 2 Bit, die auf 1 gesetzt sind; das Paritätsbit bekommt daher den Wert 1. Der ASCII-Code unter Berücksichtigung des Paritätsbits würde dann also 11000001 binär bzw. 193 dezimal lauten.

Das **Paritätsbit** dient in erster Linie der Kontrolle. Diese Kontrollfunktion hat ihre Ursache darin, daß der ASCII-Code in erster Linie zum Übertragen von Daten zwischen einzelnen Bausteinen oder Geräten entwickelt wurde. Durch die Paritätsprüfung ist es möglich, die richtige Übertragung der Daten zu untersuchen und somit Fehler bei der Übertragung weitgehend auszuschalten. An dieser Stelle sollte noch darauf hingewiesen werden, daß es natürlich auch möglich ist, mit gerader Parität zu arbeiten. Hier würde bei einer ungeraden Anzahl von Einsen das Paritätsbit gesetzt. Dieses Verfahren wird allerdings beim CPC nicht angewandt.

Mit 7 Bit ist es möglich, insgesamt $2^7 = 128$ verschiedene Zeichen zu codieren. Beim ASCII-Code sind diese wie folgt unterteilt:

Auf den Werten von 48-57 liegen die Zahlen. Bei 65-90 finden sich die Großbuchstaben des Alphabetes und jeweils um 32 höher die entsprechenden Kleinbuchstaben. Die restlichen freien Zeichen des Bereiches zwischen 32 und 126 werden von Sonderzeichen eingenommen. Besondere Bedeutung im Rahmen des ASCII-Codes haben die Werte zwischen 0 und 31 und der Wert 127. Diese repräsentieren Funktionen, die mit der Datenübertragung in Zusammenhang stehen. So gibt zum Beispiel der Wert 2 den Anfang des zu übertragenden Textes, der Wert 3 das Ende dieses Textes an. Eine 7 läßt die Klingel im übernehmenden Gerät erklingen etc. Eine genaue Übersicht über die Bedeutung der einzelnen Werte von 0-127 im ASCII-Code findet sich im Anhang 1.

Der ASCII-Code wird beim CPC auf viele Arten benutzt. Zum einen dient er dazu, Texte (Buchstaben und Zahlen) zum Beispiel bei der Abspeicherung von Strings, aber auch innerhalb der Abspeicherung eines BASIC-Programms, sofern es sich nicht um BASIC-Schlüsselwörter handelt, zu codieren. Ein String, zum Beispiel ABCD, ist also im Speicher abgelegt, indem die einzelnen Werte im ASCII-Code nacheinander in aufeinanderfolgende Speicherstellen geschrieben werden.

Definieren wir also bei frisch eingeschaltetem Rechner oder nach

```
CTRL+SHIFT+ESC
```

```
A$="abcd"
```

so finden wir nun in den Speicherstellen 43900-43903, welche wir mit PEEK auslesen können, die Werte 97-100, welche das ASCII-Äquivalent für a, b, c und d darstellen. Falls Sie eine Floppy angeschlossen haben, verschiebt sich der für die Stringabspeicherung nutzbare Bereich nach unten, es gelten dann die Speicherstellen von 42616-42619. Daneben wird der ASCII-Code für die Übertragung von Daten auf Kassette oder die Weitergabe an den Drucker benutzt. Auch sind die Daten bei der Weitergabe an die Routine, welche Zeichen auf dem Bildschirm darstellt, im ASCII-Code verschlüsselt.

Hierbei ist jedoch zu beachten, daß der CPC den ASCII-Code in den einzelnen Anwendungsbereichen teilweise ergänzt (um die Werte zwischen 128 und 255, wo beim CPC im wesentlichen Grafiksymbole liegen) oder auch umdefiniert, speziell im Bereich der Bildschirmausgabe, in den Nummern 0-31. Bei der Ansprache des Bildschirms liegen hier die sogenannten Kontrollzeichen. Gibt man diese mit dem PRINT-Befehl oder ähnlichen Kommandos auf dem Bildschirm aus, so bilden sie kein Zeichen ab, sondern führen eine Steuerfunktion aus. Zum Beispiel: die Bewegung des Cursors in eine beliebige Richtung (mit Werten zwischen 8 und 11), das Löschen des Bildschirms, der Sprung des Cursors auf die nächste Textzeile oder im Zusammenhang mit nachstehenden Parametern auch das Setzen der Schreib- oder Hintergrundfarbe und ähnliche Funktionen. Eine nähere Übersicht über die einzelnen Funktionen findet sich in Kapitel 9 des Bedienerhandbuches. Sofern die einzelnen Kontrollzeichen nicht über die CTRL-Ebene der Tastatur erfaßt werden können, kann man sie mit Hilfe der Funktion CHR$ auf dem Bildschirm ausdrucken beziehungsweise an einen String anfügen. So bewegt zum Beispiel ein



```
PRINT CHR$(11)
```

den Cursor um 1 Textzeile nach oben. Man kann sich jedoch wie im obigen Beispiel auch mit

```
A$=CHR$(11)+"ABCD"
```

einen String definieren, der dann bei der Ausgabe auf dem Bildschirm eine Zeile oberhalb der aktuellen Cursorposition ausgedruckt wird. Durch eine Kombination von Steuerzeichen ist es möglich, einen String zu definieren, der bereits eine erhebliche Umordnung des Bildschirms beinhaltet (zum Beispiel mit den Befehlen CHR$(25) bis CHR$(29). Auch kann man einen String definieren, der sich beim Ausdruck revers darstellt. Dies leistet zum Beispiel der nachfolgende Ausdruck:

```
A$=CHR$(24)+"ABCD"+CHR$(24)
```

Die entsprechenden Kontrollzeichen werden jetzt am Anfang und am Ende des Strings eingefügt und führen beim Ausdruck dann zu dem gewünschten Austausch von Papier- und Schreibstiftfarbe. Jedoch ist dabei darauf zu achten, daß diese Definition des Bereiches von 0-31 im ASCII-Code nur für die Bildschirmansprache gilt, eine Ausgabe dieses Strings auf dem Drucker kann also zu unerwarteten und gegebenenfalls auch völlig unerwünschten Effekten führen, da dieser normalerweise auf Norm-ASCII abgestimmt ist. Während wir mit

```
PRINT CHR$(<Code Nr.>)
```

ein Zeichen oder Steuerzeichen des ASCII-Codes auf dem Bildschirm ausgeben oder an einen String anfügen können, leistet der Befehl ASC() das Umgekehrte. Er stellt uns den Codewert eines Zeichens im ASCII-Code dar.

```
PRINT ASC("A")
```

ergibt also als Ausgabe den Wert 65, das Äquivalent für groß "A".

### 1.3.2 Die Codierung des BASIC-Programms

Die Abspeicherung des BASIC-Quelltextes erfolgt Zeile für Zeile. Dabei können wir zwei Bereiche unterscheiden: den Zeilenkopf mit Zeilennummer und Hinweis auf die nächste Zeile (Verzeigerung) sowie den eigentlichen Inhalt der Programmzeile. Die Verzeigerung nimmt je Programmzeile 4 Bytes in Anspruch. Die ersten beiden Bytes enthalten dabei die Länge der Programmzeile im Format LoHi, das heißt, die Entfernung zur nächsten Zeile errechnet sich, indem man den Wert des zweiten Bytes mit 256 multipliziert und dazu das erste Byte addiert. Analog funktioniert die Abspeicherung der Zeilennummer in den nächsten beiden Bytes.

Nun stellt sich die Frage, warum der CPC überhaupt eine Zeilenlänge von mehr als 255 Zeichen bearbeiten können muß, obwohl die Tastatureingaberoutine doch nur maximal 255 Zeichen zuläßt. Dies hat seine Ursache darin, daß der CPC zum Beispiel für die Trennung von Funktionen oder Zahlenwerten im Text noch einmal besondere Steuerzeichen, die angeben, ob es sich bei den nachfolgenden Werten um Zahlenwerte oder Funktionen etc. handelt, mit abspeichert. Dadurch kann eine Zeile im BASIC-Quelltext, auch wenn sie auf dem Bildschirm nur 240 oder 250 Zeichen einnimmt, dennoch für ihre Abspeicherung mehr als 255 Zeichen benötigen, womit dann eine Verzeigerung über einen größeren Bereich notwendig ist.

Der eigentliche Inhalt einer BASIC-Zeile kann wiederum in drei Gruppen unterteilt werden. Die **erste Gruppe** bilden die **BASIC-Schlüsselwörter**. Um Speicherplatz zu sparen ist jedem BASIC-Befehl ein TOKEN genannter Code zugeordnet. Diese TOKENs liegen oberhalb des vom ASCII-Code benutzten Bereiches, also von 128-255. Da das Schneider-BASIC über mehr als 128 verschiedene Befehle verfügt und somit auch mehr als 128 TOKENs benötigt werden, hat man bei der Abspeicherung der BASIC-Befehle einen kleinen Trick angewandt: Das Zeichen mit der Nummer 255 dient als Erweiterungszeichen. Trifft der CPC also auf diesen Code, so schaut er in der



Funktionstabelle nach und interpretiert dann das nachfolgende Byte als ein TOKEN der Erweiterungstabelle. Eine Übersicht über die Bedeutung der einzelnen Werte innerhalb einer BASIC-Zeile ergibt sich aus der Tabelle in Anhang 1. Unter der Überschrift "TOKEN" findet sich dabei die Bedeutung der Codes im Quelltext, die Spalte Funktionstoken gibt den Sinn bei vorangestelltem Code 255 an.

Die **zweite Gruppe** bilden die **Steuer- und Sonderzeichen**. Sie haben Werte zwischen 0 und 31 und dienen der Strukturierung innerhalb der BASIC-Zeile. So definiert eine Null das Zeilenende, eine Eins gibt das Ende eines Befehls an. Steht in einer BASIC-Zeile eine der Zahlen hex 02, 03, 04 oder 0D, so werden die nachfolgenden Werte als Definition einer Variablen angesehen. Auch die Codes zwischen 0E und 1C dienen der Abspeicherung von Zahlenwerten. Da es relativ häufig vorkommt, daß in BASIC-Programmen nur mit einstelligen Dezimalzahlen gearbeitet wird - z.B. bei Zählschleifen - ist man beim Entwurf des BASIC neue Wege gegangen. Statt immer für jede Zahl eine Realvariable anzulegen, wurde beim CPC eine platzsparende Variante gewählt. Eine Zahl zwischen Null und Neun wird als Konstante (Codes 0E bis 17 hex) abgelegt und benötigt somit für die Abspeicherung auch nur 1 Byte, was überdies auch noch eine schnellere Bearbeitung bei der Ausführung des Programms garantiert. Auch Zahlenwerte kleiner als 256 werden einer Spezialbehandlung unterzogen. Der Code 19 hex bzw. 25 dezimal erlaubt eine verkürzte Speicherung dieser Werte. Das nachfolgende Byte gibt dann den Zahlenwert an. Für größere Ganzzahlen gilt die Integerdarstellung in 2 Byte, wobei je nachdem, ob die Eingabe hex, dezimal oder binär erfolgt ist, verschiedene Codes für die Abspeicherung benutzt werden. Mit hexadezimal 1F wird eine Fließkommagröße gekennzeichnet. Die einzelnen Steuercodes sind, ebenso wie die TOKENs, in der Tabelle in Anhang 1 zu finden. Sie stehen mit Hexwerten kleiner 20 in der Spalte für TOKENs.

Als **dritte Gruppe** tauchen dann die schon behandelten **ASCII-Werte** bei der Definition von Strings oder der Ausgabe von Texten auf. Auch die Variablennamen und einige Sonderzeichen, wie Kommas und Punkte sind im ASCII-Format angegeben.

Wir wollen uns nun einmal anschauen, wie die verschiedenen Zeichengruppen im BASIC-Quelltext verbunden sind. Auf eine Spezialität des CPC-BASICs müssen wir davor allerdings noch eingehen: die direkte Adressierung von Variablen und Ansprungadressen. Weisen wir zum Beispiel der Integervariablen a% den Wert 7 zu, so finden wir dies im BASIC wie folgt abgelegt:

```
2  0  0  225  239  21  0
```

Sie erhalten diese Zahlenfolge, wenn Sie nach dem Einschalten oder <CTRL>+<SHIFT>+<ESC> die Zeile

```
10 a%=7
```

eingeben. Dabei sollten Sie jedoch, wie auch im Folgenden, immer auf die exakte Eingabe der Leerzeichen achten, sonst werden Sie zwischen den Werten, die wir jetzt mit

```
PRINT PEEK(372)  bis  PRINT PEEK(378)
```

auslesen können, den Code 32 (Leerzeichen, Space) finden. Der CPC speichert nämlich im Normalfall jedes eingegebene Leerzeichen mit ab.

Doch zurück zur Interpretation dieser Zahlenfolge. Die 2 definiert den Variablentyp, eine Integervariable. Die nachfolgenden 2 Byte definieren die Speicherstelle, an der sich die Integervariable im Speicher befindet. Es handelt sich hierbei jedoch nicht um eine absolute Adresse, sondern um die relative Position ab Programmende. Alle einfachen Variable sind nach dem BASIC-Quelltext in einem abgeschlossenen Datenbereich, dem Variablenspeicher abgelegt. Ein Zeiger (VARSTART) gibt den Beginn dieses Feldes an. Bei der erstmaligen Benutzung einer Variablen im Programm wird diese im Variablenbereich angelegt und in den beiden Byte im BASIC-Quelltext ihre Position relativ zu VARSTART abgelegt. Damit können Variable ohne langes Suchen sehr schnell wieder aufgefunden werden, was besonders bei Schleifenvariablen, die gegebenenfalls mehrere hundertmal durchlaufen werden, erheblich Zeit spart. Dieselbe Methode findet übrigens auch bei der Abspeicherung von Sprüngen wie GOTO und GOSUB Anwendung.



Hier geben die Werte hex 1D bzw. hex 1E an, ob es sich bei der nachfolgenden Sprungangabe um die Adresse der anzuspringenden Zeile im Speicher (1D) oder die Zeilennummer (1E) handelt. Beim erstmaligen Durchlauf des GOTO-Statements wird die anzuspringende Zeile aufgrund der Zeilennummer gesucht und danach deren Position anstelle der Zeilennummer eingetragen sowie das voranstehende 1E in 1D geändert.

Soweit zur direkten Adressierung. Was bedeuten nun aber die restlichen Werte in der Zeile. Als nächsten Wert bei der Interpretation unserer Zahlenfolge stoßen wir auf die Codierung des Namens. Der Name einer Variablen ist in ASCII abgelegt; zur letzten Stelle des Namens werden dabei 128 als Endekennung addiert. Es wird also das 8te Bit des letzten Buchstabens gesetzt. Da wir nur mit einem Buchstaben gearbeitet haben, erfolgt hier die Addition der 128, beziehungsweise 80 hex, in der ersten Stelle.

```
PRINT CHR$(225-128)
```

gibt uns hier die Bestätigung. Es wird ein kleines "a" ausgegeben. Als nächstes folgt für die Wertzuweisung das Gleichheitszeichen (TOKEN 239) und danach in Kurzform der Wert 7. Die nachfolgende 0 schließt die Zeile ab.

Wir wollen nun einmal unser Wissen über die Ablage vom BASIC anhand eines praktischen Beispiels überprüfen. Wir benutzen dazu ein Programm, das Untersuchungshilfe und -gegenstand in einer "Person" ist.

```
5 a%=2:ab=17:a$="abcd"
10 PRINT @a%,@ab,@a$
20 FOR i=367 TO 477:PRINT i,PEEK(i),HEX$(PEEK(i)):NEXT
```

Wenn wir dieses Programm laufen lassen, so erhalten wir zuerst die Ausgabe von 3 Zahlenwerten - wir kommen später noch darauf zurück - und danach die Ausgabe des gesamten Programms Speicherstelle für Speicherstelle. Sie sollten nun zuerst einmal versuchen, dieses selbst zu übersetzen, bevor Sie die nachfolgende Erklärung lesen. Die notwendigen Informationen finden sich wieder im Anhang 1 sowie in den oben angegebenen Erläuterungen.

| Position | dez | hex | Kommentar |
|---|---|---|---|
| 367 | 0 | 0 | Programmanfang |
| 368 | 32 | 20 | nächste Zeile nach |
| 369 | 0 | 0 | 32 Byte |
| 370 | 5 | 5 | Zeilennummer 5 |
| 371 | 0 | 0 | |
| 372 | 2 | 2 | Integervariable |
| 373 | 5 | 5 | Adresse der Variablen gerechnet ab |
| 374 | 0 | 0 | BASIC-Ende |
| 375 | 225 | E1 | Name "a" und Ende weil Bit 7 gesetzt (+ 80H) |
| 376 | 239 | EF | = |
| 377 | 16 | 10 | Konstante 2 |
| 378 | 1 | 1 | ":" Befehl zu Ende |
| 379 | 13 | D | Variable ohne Kennung |
| 380 | 12 | C | ab Adresse BASIC-Ende + 12 |
| 381 | 0 | 0 | |
| 382 | 97 | 61 | "a" |
| 383 | 226 | E2 | "b" + 80H |
| 384 | 239 | EF | = |
| 385 | 25 | 19 | Ein-Byte-Wert |
| 386 | 17 | 11 | 17 |
| 387 | 1 | 1 | ":" Befehl zu Ende |
| 388 | 3 | 3 | Stringvariable |
| 389 | 21 | 15 | liegt VARSTART + 21 |
| 390 | 0 | 0 | |
| 391 | 225 | E1 | A + 80H |
| 392 | 239 | EF | = |
| 393 | 34 | 22 | "" nötig damit auch ASC <= 32 abgespeichert werden können. |
| 394 | 97 | 61 | a |
| 395 | 98 | 62 | b |
| 396 | 99 | 63 | c |
| 397 | 100 | 64 | d |
| 398 | 34 | 22 | "" |
| 399 | 0 | 0 | Zeile zu Ende |
| 400 | 25 | 19 | nach 25 Byte nächste Zeile |
| 401 | 0 | 0 | |
| 402 | 10 | A | Zeilennummer 10 |
| 403 | 0 | 0 | |



| | | | |
|---|---|---|---|
| 404 | 191 | BF | TOKEN für PRINT |
| 405 | 32 | 20 | Abstand |
| 406 | 64 | 40 | @ |
| 407 | 2 | 2 | Integervariable |
| 408 | 5 | 5 | auf VARSTART + 5 |
| 409 | 0 | 0 | |
| 410 | 225 | E1 | Name "a" + 80H |
| 411 | 44 | 2C | , |
| 412 | 64 | 40 | @ |
| 413 | 13 | D | Variable ohne Kennung |
| 414 | 12 | C | auf VARSTART + 12 |
| 415 | 0 | 0 | |
| 416 | 97 | 61 | Name ab mit b + 80H |
| 417 | 226 | E2 | |
| 418 | 44 | 2C | , |
| 419 | 64 | 40 | @ |
| 420 | 3 | 3 | Stringvariable |
| 421 | 21 | 15 | Zeiger in VARSTART + 21 |
| 422 | 0 | 0 | |
| 423 | 225 | E1 | Name a |
| 424 | 0 | 0 | Zeilenende |
| 425 | 52 | 34 | nach 52 Byte nächste Zeile |
| 426 | 0 | 0 | |
| 427 | 20 | 14 | Zeilennummer 20 |
| 428 | 0 | 0 | |
| 429 | 158 | 9E | FOR |
| 430 | 32 | 20 | Abstand |
| 431 | 13 | D | Variable ohne Kennung |
| 432 | 28 | 1C | in VARSTART + 28 |
| 433 | 0 | 0 | |
| 434 | 233 | E9 | Name "i" + 80H |
| 435 | 239 | EF | = |
| 436 | 26 | 1A | 2 Byte Wert dezimal |
| 437 | 111 | 6F | 1*256+111=367 |
| 438 | 1 | 1 | |
| 439 | 32 | 20 | Abstand |
| 440 | 236 | EC | TO |
| 441 | 32 | 20 | Abstand |
| 442 | 26 | 1A | 2 Byte Wert dezimal |
| 443 | 222 | DE | 1*256+222=478 (hinter Programm) |
| 444 | 1 | 1 | |
| 445 | 1 | 1 | ":" Befehlsende |

| | | | |
|---|---|---|---|
| 446 | 191 | BF | PRINT |
| 447 | 32 | 20 | Abstand |
| 448 | 13 | D | Variable ohne Kennung |
| 449 | 28 | 1C | in VARSTART + 28 |
| 450 | 0 | 0 | |
| 451 | 233 | E9 | i + 80H |
| 452 | 44 | 2C | , |
| 453 | 255 | FF | Funktion |
| 454 | 18 | 12 | PEEK |
| 455 | 40 | 28 | ( |
| 456 | 13 | D | Variable ohne Kennung |
| 457 | 28 | 1C | = Position |
| 458 | 0 | 0 | VARSTART + 28 |
| 459 | 233 | E9 | Name "i" + 80H |
| 460 | 41 | 29 | ) |
| 461 | 44 | 2C | , |
| 462 | 255 | FF | Funktion |
| 463 | 115 | 73 | HEX$ |
| 464 | 40 | 28 | ( |
| 465 | 255 | FF | Funktion |
| 466 | 18 | 12 | PEEK |
| 467 | 40 | 28 | ( |
| 468 | 13 | D | Variable ohne Kennung |
| 469 | 28 | 1C | in Position VARSTART + 28 |
| 470 | 0 | 0 | |
| 471 | 233 | E9 | Name "i" + 80H |
| 472 | 41 | 29 | ) |
| 473 | 41 | 29 | ) |
| 474 | 1 | 1 | Befehlsende |
| 475 | 176 | B0 | NEXT |
| 476 | 0 | 0 | Zeilenende |
| 477 | 0 | 0 | Keine weitere Zeile mehr |
| 478 | 0 | 0 | |

Die symbolische Größe VARSTART bezeichnet dabei den Pointer auf den Anfang der Variablen. Die anderen Größen sind ASCII-Werte, TOKENs oder relative Positionen in bezug auf VARSTART. Als letzte Frage bliebe offen, welche Bedeutung den 3 Zahlen am Anfang der Ausgabe sowie dem "@" im PRINT-Kommando zukommt. Die Antworten darauf gibt das nächste Kapitel.



### 1.3.3 Die Ablage der Variablen im Speicher

Nachdem wir uns bereits intensiv mit der Abspeicherung des BASICs beschäftigt haben, wollen wir nun zur Ablage der Variablen übergehen. Das Grundprinzip der Variablenabspeicherung haben wir bereits bei der Analyse des BASIC-Quelltextes kennengelernt. Jede Variable ist im BASIC-Programm mit dem Variablennamen, dem Variablentyp sowie der Adresse, an der sie sich im Variablenspeicher befindet, definiert. Diese Adresse wird beim erstmaligen Aufruf der Variablen entsprechend gesetzt. Es stellt sich nun die Frage, wie die Variablen im Speicher angeordnet sind und wie der Computer beim erstmaligen Aufruf einer Variablen diese im Speicher erkennt. Zuerst einmal können wir drei Typen von Variablen unterscheiden:

-Integers
-Realvariable
-Strings

Jeder dieser drei Variablentypen ist auf unterschiedliche Art und Weise abgespeichert. Jeder Typ kann dann noch in zwei Formen vorkommen, als einzelne Variable oder als ein Feld von Variablen, ein Array, zum Beispiel A(3,3).

Wenden wir uns zunächst den einfachen Variablentypen zu. Am Anfang unserer Betrachtung müssen wir uns etwas näher mit dem Aufbau des BASIC-Speichers beschäftigen. Der BASIC-Speicher reicht von Speicherstelle 367 bis zur Obergrenze des Speichers, die durch den Pointer definiert wird, normalerweise 43903. Falls eine Floppy angeschlossen ist, verschiebt sich die Obergrenze um ungefähr 1K nach unten. Dieser Bereich steht dem Benutzer für BASIC-Programm, Variablenabspeicherung und Kassettenoperationen zur Verfügung. Die Unterteilung des Benutzerspeichers in die einzelnen Teilbereiche wird dabei durch eine Anzahl von Zeigern vorgenommen. So gibt es einen Zeiger für den Start des BASIC-Programms und für das Ende des BASIC-Programms sowie für Variablenstart, Arraystart und das Ende der Arrays. Daneben befinden sich noch zwei Zeiger für den Beginn der Strings und das Ende der Strings. Jeder dieser Zeiger besteht aus 2 Byte und gibt eine absolute Position im Speicher ein. Die Abspeicherung liegt dabei wie üblich im Format Lo-Hi. Lassen wir uns also

```
PRINT PEEK(&AE81)+256*PEEK(&AE82)
```

ausdrucken, so erhalten wir den Wert des Zeigers PROGSTART: 367. Die Anordnung der einzelnen Zeiger und ihrer Funktion bei der Aufteilung des Benutzerspeichers gibt das Schaubild auf der nächsten Seite wieder. Hierbei geben die Pfeile die Ausdehnungsrichtung der einzelnen Bereiche an. Nebenstehend finden sich die einzelnen Pointer. Die Positionen sind dabei wieder Lo-Hi abgespeichert.

Während BASIC, Variable und Arrays sich von unten nach oben ausdehnen, wachsen die Strings von der Speicherobergrenze in umgekehrter Richtung. Soll nun eine neue Variable definiert werden, wir nehmen einmal eine einfache Variable an, so wird der Arrayblock zwischen ARRAY START und ARRAY END um die benötigten Bytes nach oben geschoben, und in den dann freien Raum wird die neue Variable abgelegt. Die Änderungen in den anderen Bereichen laufen analog ab.

Um jederzeit die aktuelle Position einer gesuchten Variablen im Speicher feststellen zu können, stellt uns das Schneider-BASIC den Befehl @, auch Klammeraffe genannt, zur Verfügung. Jede Variable ist im Speicher des Rechners mit Name und den dazugehörigen Werten abgelegt. Das Kommando @ gibt uns nun an, ab welcher Stelle die eigentliche Variable abgelegt ist, das heißt, er zeigt auf das erste Byte nach dem Ende des Variablennamens.



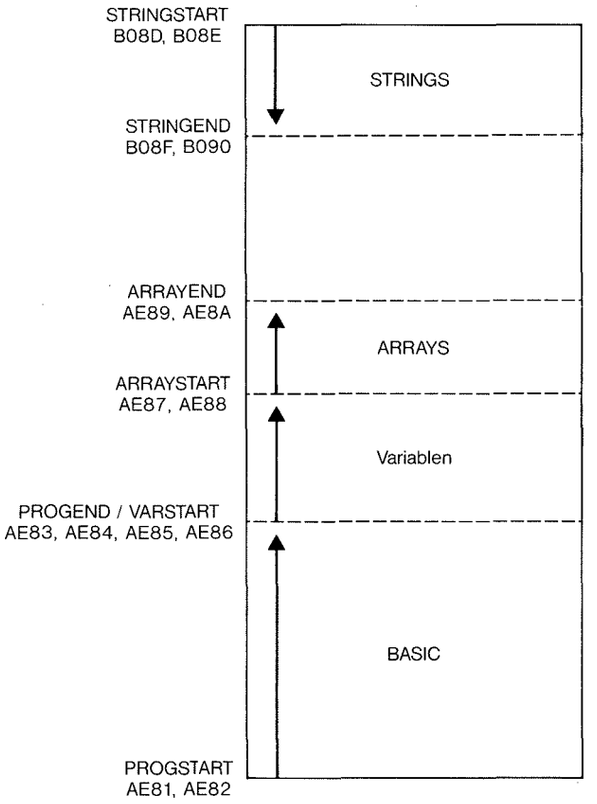



Bild 1.1: Aufbau des Benutzerspeichers mit Pointern

### 1.3.3.1 Die Abspeicherung einfacher Variablen

Schauen wir uns nun die Abspeicherung der Variablen ein wenig näher an. Wir ändern dazu die Zeile 20 in unserem Programm etwas ab. Wir ersetzen die Werte 367 und 478 durch 479 und 550. Die neue FOR-TO-Schleife beginnt also dort, wo die alte geendet hat, im Bereich der Variablen. Nacheinander können wir die einzelnen Variablen erkennen. Sie liegen hintereinander, durch Nullen getrennt. Ihre Reihenfolge ergibt sich dabei aus dem Windhundprinzip: Was zuerst im Programmablauf definiert wurde, liegt zuunterst.

| Position | dez | hex | Kommentar |
|---|---|---|---|
| 479 | 0 | 0 | hierauf steht VARSTART |
| 480 | 0 | 0 | |
| 481 | 193 | C1 | A Name der Variablen |
| 482 | 1 | 1 | % Länge der Mantisse-1 |
| 483 | 2 | 2 | Wert 2 |
| 484 | 0 | 0 | |
| 485 | 1 | 1 | |
| 486 | 0 | 0 | |
| 487 | 65 | 41 | A Name der Variablen |
| 488 | 194 | C2 | B |
| 489 | 4 | 4 | Reallänge Mantisse-1 |
| 490 | 0 | 0 | ma1 |
| 491 | 0 | 0 | ma2 |
| 492 | 0 | 0 | ma3 |
| 493 | 8 | 8 | ma4 |
| 494 | 133 | 85 | exp |
| 495 | 7 | 7 | |
| 496 | 0 | 0 | |
| 497 | 193 | C1 | A |
| 498 | 2 | 2 | $ Stringvariable |
| 499 | 4 | 4 | String ist 4 Zeichen lang |
| 500 | 138 | 8A | Position im Speicher |
| 501 | 1 | 1 | |
| 502 | 0 | 0 | |
| 503 | 0 | 0 | |
| 504 | 201 | C9 | I |
| 505 | 4 | 4 | Realvariable |
| 506 | 0 | 0 | |
| 507 | 0 | 0 | |
| 508 | 0 | 0 | |
| 509 | 126 | 7E | |
| 510 | 137 | 89 | |
| 511 | 0 | 0 | ARRAYSTART |
| 512 | 0 | 0 | |
| 513 | 0 | 0 | |

Nach einer Anfangsnull erhalten wir als ersten interessanten Wert in Speicherstelle 481 eine 193, ein Wert, der uns schon geläufig ist. Er setzt sich aus dem Code für A (65) und der hinzugefügten 128 für das Ende des Namens zusammen. Speicherstelle 482 enthält den Variablentyp Integer. Es folgen zwei Byte für die Angabe des Wertes der Integer, die Abspeicherung ist dabei LOW HIGH. Da wir in dem Programm A% auf 2 gesetzt hatten, finden wir also jetzt hier 2 und 0. Das oberste Byte wird nicht benötigt. Auf die erste dieser beiden Speicherstellen deutet unser Variablenpointer "@" hin. Wir können uns also die jeweilige Position der Variablen durch



```
PRINT @a%
```

ausgeben lassen.

Ähnlich läuft der Vorgang bei unserer nächsten Variablen ab, AB, wir hatten sie auf 17 gesetzt. In den Speicherstellen 487 und 488 finden wir den aus zwei Buchstaben bestehenden Namen. 489 liefert das Typenkürzel, eine 4 für Real. Die nachfolgenden 5 Byte geben nun den Wert der Realvariablen an. Schon bei unseren Betrachtungen zum Thema Konvertierung von Zahlensystemen haben wir uns mit der Abspeicherung von Zahlen im Dualsystem näher auseinandergesetzt. Der CPC speichert Fließkommazahlen in jeweils 5 Byte ab. Die ersten 4 Byte bilden dabei die sogenannte Mantisse, wobei das erste Byte das niederwertigste darstellt. In unserem Fall lauten sie 0,0,0 und 8 und liegen von Speicherstelle 490 bis 493. Dem obersten Bit des vierten Mantissenbyte kommt dabei eine Sonderfunktion zu. Es gibt das Vorzeichen der gespeicherten Zahl an. Eine 0 bedeutet hierbei, daß es sich um eine positive Zahl handelt, bei einer 1 haben wir es mit einem negativen Wert zu tun. Insgesamt steht uns also eine Binärzahl mit 31 Stellen, einem 8stelligen Binärexponenten und einem Bit für das Vorzeichen zur Verfügung.

Die Mantissenbytes 1 bis 3 und auch die unteren Stellen des Mantissenbytes 4 stellen dabei binärgespeicherte Nachkommastellen dar. Da wir auf mehr als 15 Bit zurückgreifen müssen, ist die Umformung leider nicht so einfach durchzuführen, wie wir es bei der Umwandlung der einzelnen Zahlensysteme ineinander durchgeführt haben, und bedarf bereits einiger Gehirnakrobatik. Als ersten Schritt filtern wir das Vorzeichen heraus, indem wir Bit 7 des 4. Mantissenbytes setzen und so den "vorzeichenbereinigten" Mantissenwert erhalten. Als nächsten Schritt müssen wir nun die einzelnen Binärmantissen zu einer Dezimalzahl zusammenfassen. Jedes Mantissenbyte besitzt genau 1/256 der Wertigkeit des nächsthöheren. Das höchste Mantissenbyte besitzt dabei die Wertigkeit 1/256. Je nach dem Inhalt der Mantissenbytes beziehungsweise -bits erhalten wir somit nun Werte zwischen 0.5 und 0.99999999. Diese Zahl kann nun wiederum je nach dem Vorzeichen,

das heißt dem Wert des 8. Bits von Mantisse 4 positiv oder negativ sein. Wir haben somit als Mantisse eine neunstellige Zahl im Bereich ± 1 definiert. Diese Zahl wird nun mit einem 2er-Exponenten multipliziert, um auch größere oder kleinere Werte speichern zu können. Der OFFSET, das heißt der Wert des Exponenten, bei dem keine Änderung des in der Mantisse ausgedrückten Wertes stattfindet, liegt bei 128. Der CPC kann also mit Werten im Bereich von $2^{127}$ bis $2^{-128}$ multiplizieren.

Das nachfolgende kleine 8-Zeilen-Programm gibt Ihnen die Möglichkeit, diese Art der Umformung einmal Schritt für Schritt auszuprobieren und sich experimentell einen Überblick über die Umwandlung der einzelnen Mantissenbytes und ihrer Bedeutung für die Gesamtzahl zu verschaffen. Wenn Sie das Programm mit den in Speicherstelle 490-494 befindlichen Werten 0,0,0, 8 und 133 durchspielen, werden Sie als Ergebnis 17 erhalten, den Wert, auf den wir AB in Zeile 5 unseres Programms definiert hatten.

```
10 REM *****************
20 REM ** Transformer **
30 REM *****************
40 CLS:INPUT"Bitte geben Sie den Hexwert ein (max. 2-stellig)";h$
50 IF LEN(h$)>2 THEN PRINT"nur 2-stellig":FOR t=1 TO 1000:NEXT t:G
OTO 40
60 CLS:PRINT"Hexwert:";h$:h$="&"+h$:PRINT" 1"
70 PRINT" 2 6 3 1"
80 PRINT" 8 4 2 6 8 4 2 1"
90 PRINT"----------------"
100 b$=BIN$(VAL(h$)):b$=RIGHT$("00000000"+b$,8)
110 FOR i=1 TO 8:LOCATE 2*i,8
120 PRINT MID$(b$,i,1):NEXT i
130 PRINT:PRINT"Dezimalwert: ";VAL(h$)
```

Nach soviel Kopfzerbrechen ist die Beschäftigung mit der Abspeicherung der Strings, obwohl auch nicht so einfach wie bei den Integervariablen, doch schon fast eine Entspannung. Bei der Ablage eines Strings müssen zwei Änderungen im Benutzerspeicher vorgenommen werden. Zum einen wird der Inhalt des Strings im Stringbereich, also zwischen STRING START und STRING END, abgelegt. Auf diesen String wird dann ein Zeiger gesetzt, der unter dem Namen der Variablen abgespeichert, ganz normal zwischen den Real- und den Integervariablen im Variablenbereich liegt. Eine Stringvariable besteht somit aus dem Namen der Variablen, darauf folgend eine 2 als Kennzeichen für einen String. Die nächste Speicherstelle enthält die Länge des abgespeicherten Strings, und die darauf folgenden beiden Stellen, wieder im Format Lo-Hi, geben die aktuelle Position des



Strings im Stringspeicher an. Wir wollen dies einmal anhand eines Beispiels durchspielen. Zunächst versetzen wir den Rechner wieder in den jungfräulichen Zustand durch

<CTRL><SHIFT><ESC>

Unser BASIC-Programm ist damit gelöscht, die Variable VARSTART enthält genauso wie PROGEND den Wert 370. Zwischen PROGSTART und VARSTART befinden sich nur die drei Nullen, die das Ende des Programms angeben. Als nächstes definieren wir nun

A$="ABCD".

Schauen wir uns nun unseren Speicher an. Wir finden unsere gesuchte Variable ab Position 372. Das erste Byte enthält wie üblich den Namen der Variablen und hier, da nur mit einem Buchstaben gearbeitet wird, natürlich wieder mit der hinzugefügten 128 erweitert. Auch der nächste Wert stellt keine prinzipielle Neuerung dar. Es handelt sich um eine 2, das Kennzeichen für Strings. Als nächstes folgt nun die Länge des Strings; hier steht eine 4. Die Speicherstellen 375 und 376 geben uns dann die absolute Position unseres Strings mit der Länge von 4 Zeichen im Stringspeicher, das heißt an der Speicherobergrenze an. Lassen wir uns

PRINT PEEK(376+256*PEEK(377))

ausgeben, so erhalten wir den Wert 43900. An dieser Stelle liegt der erste Buchstabe unseres Strings, das große A. Lassen wir uns PEEK von 43900 ausgeben, so erhalten wir eine 65 das ASCII-Äquivalent für klein a. In den nachfolgenden Stellen bis 43903 finden sich die anderen 3 Buchstaben. Wir können uns jetzt auch den Wert von STRINGEND ausgeben lassen. Die Differenz zu STRINGSTART beträgt genau die belegten 4 Byte.

### 1.3.3.2 Die Abspeicherung von Arrays

Bei Arrays haben wir es mit Variablenfeldern zu tun. Das heißt: Unter ein und demselben Variablennamen sind mehrere Variable abgespeichert, die durch Indizierung unterschieden werden. Jede einzelne Variable wird also bezeichnet, indem man den Variablennamen des Feldes angibt und dann in Klammern ein oder mehrere Dimensionsangaben. Mehrdimensionale Felder müssen beim CPC prinzipiell mit dem DIM-Kommando definiert werden, eindimensionale Felder, das heißt Arrays, mit nur einer Indexangabe, in

dem Moment, wo der Index auf Werte größer als 10 läuft. Dies hat seinen Grund darin, daß bei der erstmaligen Verwendung einer indizierten Variablen, zum Beispiel M(8), ein Realfeld angelegt wird, welches dann für den Rest der Arbeitszeit im Speicher verbleibt. Wollte man beliebige eindimensional indizierte Felder ohne den DIM-Befehl aufrufen, so müßte bei dem erstmaligen Aufruf von B(6) z.B. ein eindimensionales Feld von unendlicher Größe definiert werden,bei der nächsten eindimensionalen Variablen wieder usw. Daß dies schon aufgrund der Größe des verfügbaren Speichers natürlich nicht möglich ist, ist sofort einsichtig, und man hat sich deshalb darauf beschränkt, wie auch bei den meisten anderen Homecomputern, beim Aufruf einer eindimensionalen Feldvariablen immer nur ein Feld der Größe 10 zu definieren. Es können also die Indizes von 0 bis 10 ohne vorherige Dimensionierung benutzt werden. Dennoch ist dieses Verfahren nicht immer ratsam, denn schließlich wird, auch wenn man nur 3 indizierte Variable benötigt, ein 10- beziehungsweise genauer gesagt 11-elementiges Feld, denn wir müssen ja die 0 mitzählen, eröffnet, was natürlich Speicherplatz kostet und bei häufiger Anwendung derartiger Schlendrianmethoden auch den Programmablauf erheblich verzögert. Man sollte daher auch kleine eindimensionale Felder grundsätzlich mit dimensionieren. Bei mehrdimensionalen Feldern und größeren eindimensionalen Feldern, das heißt von 11 nach oben, ist dies sowieso unumgänglich.

Schauen wir uns nun einmal die Abspeicherung eines Arrays an. Diese ist eigentlich relativ einfach und eingänglich. Da aber Arrays bei der Anwendung von in Maschinensprache geschriebenen Sortier- und Vergleichsroutinen, zum Beispiel der Bestimmung des Maximalwertes eines Feldes oder der Überprüfung der Existenz eines bestimmten Teiltextes in einem Stringfeld, einen dankbaren Anwendungsbereich bilden, sollten wir uns doch mit ihnen etwas näher auseinandersetzen.

#### 1.3.3.2.1 Stringarrays

Als erstes wollen wir uns einmal den Aufbau eines Stringarrays anschauen. Als Hilfe dazu dient uns wieder ein kleines Untersuchungsprogramm:

```
5 DIM m$(12,12,12)
7 m$(0,0,0)="13 Byte voll":m$(12,12,12)="7 Byte"
8 m$(1,0,0)="1"
10 FOR i= PEEK(&AE87)+256*PEEK(&AE88) TO
PEEK(&AE89)+256*PEEK(&AE8A)
20 PRINT i,PEEK(i),HEX$(PEEK(i)):NEXT
```



Dieses Programm dimensioniert zunächst ein Stringarray der Größe 13 x 13 x 13, das heißt mit Indizes, die von 0-12 laufen dürfen. Danach werden 3 Stringvariable in diesem Feld definiert. In den Zeilen 10 und 20 schließt sich dann die Untersuchungsroutine an. Hierbei lassen wir uns bei der Analyse des Arrays durch den CPC assistieren.

Variablenfelder werden beim CPC in einem Bereich des Benutzerspeichers, der zwischen den Zeigern ARRAY START und ARRAY END liegt, also oberhalb der einfachen Variablen, abgelegt. Über diesen Bereich läuft die Zählschleife in Zeile 10. Zeile 20 gibt dann wie gewohnt den einzelnen Wert der Zählschleife, das heißt die Speicherstelle und den Inhalt der Speicherstelle dezimal und hex aus. Wenn wir das Programm laufen lassen, so erhalten wir die folgende Ausgabe.

**Abspeicherung der ARRAYS**

| Position | dez | hex | Kommentar |
|---|---|---|---|
| 565 | 0 | 00 | |
| 566 | 0 | 00 | |
| 567 | 205 | CD | Name + 128 M |
| 568 | 2 | 02 | Variablentyp "$" |
| 569 | 198 | C6 | Länge des Descriptorfeldes |
| 570 | 25 | 19 | Zeiger auf nächstes Array |
| 571 | 3 | 03 | Dimension |
| 572 | 13 | 0D | 1. Dimension-Variable |
| 573 | 0 | 00 | höherwertiges Byte |
| 574 | 13 | 0D | 2. Dimension-Variable |
| 575 | 0 | 00 | höherwertiges Byte |
| 576 | 13 | 0D | 3. Dimension |
| 577 | 0 | 00 | höherwertiges Byte |
| 578 | 13 | 0D | Länge 1. String m$(0,0,0) |
| 579 | 150 | 96 | Position im Speicher |
| 580 | 1 | 01 | PRG oder Stringspeicher! |
| 581 | 1 | 01 | Länge 2. String 1 m$(1,0,0) |
| 582 | 207 | CF | Position im Speicher |
| 583 | 1 | 01 | |

Sollten Sie für die Speicherstelle, das heißt I, höhere Werte erhalten, so liegt dies daran, daß Sie in dem Programm an irgendeiner Stelle Leerzeichen

eingegeben haben, die, wie wir schon gesehen haben, das Programm verlängern und damit auch zu einer Verschiebung des Arraybereiches nach oben führen. Bei normaler Eingabe des Programms beginnt die Ausgabe ab Speicherstelle 565.

Die Ablage des Feldes läuft ähnlich wie die Abspeicherung einer einfachen Variablen ab. Zunächst folgt der Name, ein Buchstabe; die addierte hex 80 gibt wiederum an, daß der Name zu Ende ist.

Es folgt der Variablentyp, eine 2 für Strings und dann ein etwas ungewöhnlicher Wert: die Länge des Descriptorfeldes. Auf die genaue Bedeutung der einzelnen Werte kommen wir später noch zurück. Es folgt die Angabe der Dimensionen. Wir haben es mit einem dreidimensionalen Feld zu tun, das in jeder Dimension 13 Indizes, daß heißt Zahlen von 0 bis 12, zuläßt.

Nun folgt die Angabe der Position des ersten Strings (Speicherstelle 578). Jeder einzelne String ist wie bei den einfachen Strings mit seiner Länge und Position im Speicher abgelegt. Für die Position im Speicher finden wir hier nun die etwas ungewöhnlichen Werte 150 und 1 (Speicherstelle 579 und 580). Diese Adressen liegen noch im Bereich des BASIC-Programms. Was auf den ersten Blick etwas verwirrend ist, entpuppt sich schnell als ein kluger Schachzug. Da wir ja bereits im Programm die Definition der Variablen abgelegt hatten, ist es eigentlich unnütz, diese noch einmal oben im Stringspeicher anzulegen. Daher befindet sich hier nur die Adresse der Variablen im Programm. Wenn wir uns den BASIC-Quelltext ab Position 406 mit

```
PRINT CHR$(PEEK(406))  PRINT CHR$(PEEK(407))  etc.
```

ausgeben lassen, so finden wir unsere Definition des ersten Strings, nämlich

```
13 Byte voll
```

Die Speicherstelle 406 ergibt sich dabei als 1 * 256 + 150, das heißt, auch hier wurde die Adresse des Strings wieder im Format Lo-Hi abgespeichert. Ab 581-583 folgt der nächste String m$(1,0,0). Diesen hatten wir nur auf ein Zeichen, die 1, definiert, so daß wir hier als Länge eine 1 vorfinden, und 1 x 256 + 207 = 463 gibt uns wiederum die Position im Programm an. Diese Positionsangabe gilt für alle Strings, die erstmalig definiert werden. Wird jedoch der String zum erstenmal wirklich geändert, so folgt dann die Ablage des geänderten Strings im Stringspeicher und natürlich auch eine Korrektur der Positionsangabe bei der Abspeicherung des Arrays.



Wir können dies einmal mit der Eingabe von Zeile 9 ausprobieren.

```
9 m$(0,0,0)=m$(0,0,0)+"A"
```

Diese macht nichts anderes, als daß sie an unseren ersten String ein großes A anfügt. Lassen wir nun unser Programm wieder laufen, so erkennen wir schnell die Änderung; die Länge des Strings wurde auf 14 heraufgesetzt, und in den nachfolgenden beiden Angaben finden wir nun eine Position gekennzeichnet, die im Stringspeicher liegt. Das Ausrechnen der Adresse ergibt 43889. Und natürlich ist auch noch eine andere Änderung eingetreten. Durch das Einfügen der Zeile 9 wurde unser BASIC-Programm länger, so daß der Arraybereich jetzt einige Bytes höher beginnt, womit wir auch bei der Ausgabe der Speicherstellen in der ersten Spalte natürlich höhere Werte erhalten.

Abschließend zu unserer Betrachtung von Stringarrays müssen wir noch auf die Größe des Feldes, das heißt die alten Speicherstellen 569 und 570 eingehen. Sie speichern die gesamte Länge des für die Stringverzeigerung benötigten Feldes. Wie wir schon gesehen haben, benötigt jeder String für die Abspeicherung 3 Byte, seine Länge und die Positionsangabe im Stringspeicher beziehungsweise im BASIC-Quelltext als absolute Adresse. Bei drei Dimensionen und jeweils 13 erlaubten Indizes benötigen wir somit 13 x 13 x 13 x 3 Speicherstellen für die Ablage des Stringzeigers. Addieren wir zu diesen 6591 Byte noch die für die Dimensionsangaben benötigten 7 Byte (Anzahl der Dimensionen und 3 x Elemente je Dimension), so erhalten wir als Ergebnis 6598 Bytes, und dies ist genau jene Zahl, die wir oben in der Länge des Descripterfeldes als 25 * 256 + 198 wiederfinden. Wie sieht das Ganze nun für die anderen beiden Variablentypen, das heißt Integer- und Realvariablen, aus?

#### 1.3.3.2.2 Numerische Arrays

Wir können dies mit wenigen Handgriffen überprüfen. Ändern wir unser kleines Programm wie folgt

```
5 DIM m(12,12,12)
7 m(0,0,0)=999:m(12,12,12)=7
8 m(1,0,0)=1
10 FOR i= PEEK($AE87)+256*PEEK(&AE88) TO
PEEK(&AE89)+256*PEEK(&AE8A)
20 PRINT i,PEEK(i),HEX$(PEEK(i)):NEXT
```

so wird nun ein Fließkommavariablenfeld mit drei Dimensionen und jeweils 13 Elementen angelegt und seine Struktur uns aufgezeigt. Da unser Programm durch die Änderung etwas kürzer geworden ist, beginnt die Abspeicherung der Arrays jetzt schon ab Speicherstelle 543, und wir erhalten die folgenden Ausgaben:

| Position | dez | hex | Kommentar |
|---|---|---|---|
| 543 | 0 | 00 | |
| 544 | 0 | 00 | |
| 545 | 205 | CD | Name "M" + 128 |
| 546 | 4 | 04 | Variablentyp Real |
| 547 | 240 | F0 | Größe des Feldes |
| 548 | 42 | 2A | 10992 Byte |
| 549 | 3 | 03 | Dimensionen |
| 550 | 13 | 0D | 1. Dimension |
| 551 | 0 | 00 | höherwertiges Byte |
| 552 | 13 | 0D | 2. Dimension |
| 553 | 0 | 00 | höherwertiges Byte |
| 554 | 13 | 0D | 3. Dimension |
| 555 | 0 | 00 | höherwertiges Byte |
| 556 | 0 | 00 | 1. Variable m(0,0,0) Mantisse Byte 1 |
| 557 | 0 | 00 | Mantisse 2 |
| 558 | 192 | C0 | Mantisse 3 |
| 559 | 121 | 79 | Mantisse 4 |
| 560 | 138 | 8A | Exponent |
| 561 | 0 | 00 | 2. Variable m(1,0,0) |
| 562 | 0 | 00 | Mantisse 2 |
| 563 | 0 | 00 | Mantisse 3 |
| 564 | 0 | 00 | Mantisse 4 |
| 565 | 129 | 81 | Exponent |

Der Arraykopf ist dem der Stringarrays sehr ähnlich. Unterschiedlich ist natürlich der Variablentyp eines Realvariablenfeldes. Auch die Größe des Feldes hat sich eigentlich erweitert. Dies liegt daran, daß, wie wir ja schon gesehen haben, eine Realvariable 5 Byte für die Abspeicherung benötigt. Rechnen wir ein bißchen:



Abzuspeichernde Variable:

```
13 * 13 * 13 * 5 Byte = 10985 Byte
  + Dimensionierung =     7 Byte
-----------------------------------
                        10992 Byte
```

und damit gerade jene Zahl, die wir auch als 42 * 256 + 240, also die Größe unseres Feldes, bestimmen können. Für den Wert der einzelnen Variablen gilt das schon bei der Abspeicherung der einfachen Realvariablen Gesagte. Das gleiche Prinzip finden wir auch bei der Abspeicherung von Integerarrays wieder. Das Feld ist nun kleiner als bei den Stringvariablen, da ja nur 2 Byte für die Abspeicherung benötigt werden. Als Kennzeichen finden wir die 1 für Integer und nach den Dimensionsangaben dann im 2-Byte-Ersatz Variable auf Variable. Wir können uns dies anschauen, indem wir einfach in unserem Programm hinter jeder Variablen noch das Prozentzeichen für Integervariable einfügen.

# 2 Speicherabfrage und -veränderung

## 2.1 Übersicht über den Speicheraufbau

Im letzten Kapitel haben wir uns bereits ein wenig mit der Aufteilung des variablen Speichers unseres Rechners, des RAM beschäftigt und einige erste Routinen kennengelernt, mit denen es möglich war, Speicherbereiche abzufragen. Wir wollen dieses Wissen nun vertiefen und als Endpunkt ein komplexes Programm erstellen, welches es uns ermöglichen wird, sowohl vielfältige Operationen beim Lesen und Schreiben von Speicherstellen durchzuführen als auch Maschinenspracheprogramme besser einzugeben und zu testen. Zunächst wollen wir etwas näher auf die Aufteilung des Speichers und die Speicherstruktur eingehen. Bei der Beschäftigung mit der Abspeicherung von BASIC-Variablen, -arrays und -strings haben wir es im wesentlichen mit dem Benutzerspeicher des Rechners zu tun gehabt, jenem Teil des variablen Speichers, der dem Benutzer vorbehalten ist. Er liegt, wie wir schon gesehen haben, zwischen Speicherstelle 367 und Speicherstelle 43903, falls man die Kassette angeschlossen hat. Bei angeschlossener Floppy werden vom oberen Teil des Benutzerspeichers noch einige Byte abgezwackt.

Das gesamte RAM umfaßt jedoch erheblich mehr Adressen, nämlich genau 65536 Adressen mit Nummern zwischen 0 und 65535. Dies sind genau $2^{16}$ Speicherstellen, was bedeutet, daß wir mit einer 16-Bit-Adresse, also mit 2 Byte, jede Speicherstelle im RAM ausreichend adressieren können. Nun verfügt der CPC aber nicht nur über variablen Speicher, also RAM, sondern auch über einen festen Speicher, das ROM, das halb so groß wie das RAM ist, also 32 K umfaßt. Schauen wir uns zunächst einmal das RAM in seiner Gesamtheit an. Wir beginnen dabei an der Obergrenze des Speichers, bei Adresse 65535 oder &FFFF, die das hexadezimale Äquivalent darstellt.

Die **obersten 16 K** werden normalerweise vom Grafikspeicher eingenommen. Jeder Punkt auf dem Bildschirm wird hier durch seinen Farbcode abgelegt. Von **C000** nach unten läuft der Maschinenstapel. Dieser Speicherplatz ist dem Prozessor, der CPU, vorbehalten. Sie benutzt ihn, um Daten oder Sprungadressen abzulegen. Änderungen in diesem Bereich haben entweder keinen Effekt oder führen dazu, daß der Computer sich auf die eine oder andere Weise verabschiedet. Für den Anwender von größerem Interesse ist der nächste Bereich und hier insbesondere die Adressen von B900 bis BD37. Von **B900 bis B923** liegt der Kernel Jumpblock. Diese Sprungtabelle dient dazu, es dem Benutzer zu ermöglichen, auf die einzelnen Speicherbereiche zurückzugreifen, ROMs ein- und auszuschalten. Auf die einzelnen Aufrufe kommen wir am Ende dieses Kapitels noch zurück.

Noch interessanter für den Benutzer ist der Haupt-Firmware-Jumpblock zwischen **BB00** und **BDFF**. Hier finden sich mehrere hundert Einsprünge in das untere ROM, die der Anwender in seine eigenen Maschinenprogramme einbauen kann. Dabei wird die gesamte Funktionspalette abgedeckt. Es ist hiermit möglich, die Tastatur umzudefinieren, Zeichen auf dem Bildschirm auszugeben, Windows zu definieren, die Farben der einzelnen Windows zu setzen. Aber auch für Kassettenoperationen das Laden des Soundspeichers oder bei der Druckerausgabe stehen eine ganze Reihe von Maschinenhilfsroutinen zur Verfügung, die durch universelle Anwendbarkeit für viele Probleme in der Maschinensprache einsetzbar sind. Der Ansprung der einzelnen Betriebssystemroutinen erfolgt dabei über sogenannte LOWJUMP-RESTARTS. Jede Betriebssystemroutine ist in drei Byte abgespeichert. Das erste Byte wird dabei durch die Restartanweisung eingenommen. Danach folgt in zwei Byte die Adresse der Betriebssystemroutine, die im unteren ROM die gewünschte Funktion ausführt. Beim Aufruf dieser Routine wird automatisch die benötigte ROM-Konfiguration hergestellt, das heißt, der Benutzer muß sich nicht mehr darum kümmern, ob die richtigen ROM- und RAM-Bereiche eingeschaltet sind.

Der nächste Bereich auf unserem Weg nach unten (zwischen **B900** und **AC00**) gehört wiederum der Firmware. In diesem Teil geht es vor allem um die Tastatur und die Eingabe von Zeichen. Hier befinden sich der Zeicheneingabepuffer und auch die Definitionstabellen für die Tastaturbelegung sowie die Erweiterungszeichen. Daneben liegen hier eine Reihe von BASIC-Variablen, wie die Pointer für die Unterteilung des Benutzerspeichers, die wir im letzten Kapitel bereits des öfteren benutzt haben. Die Abspeicherung der relevanten Werte für die Kassettenausgabe (mehr dazu in Kapitel 8) erfolgt ebenfalls hier.



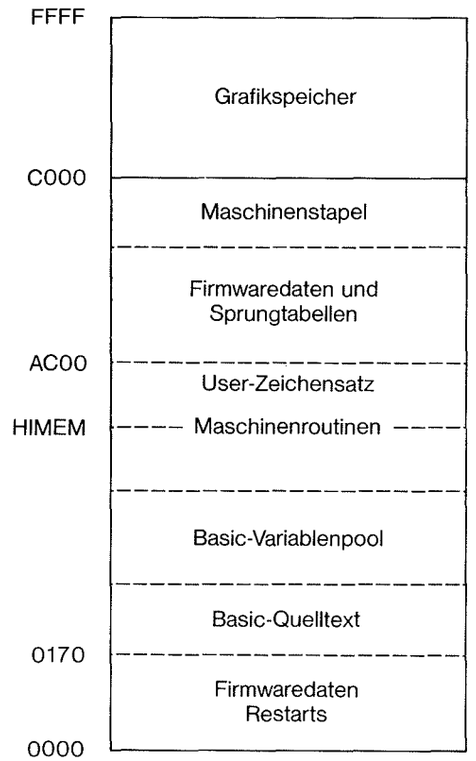



Bild 2.1: Unterteilung des RAM

Die Hex-Adresse **AC00** bezeichnet die Trennlinie zwischen Firmwarespeicher und Anwenderspeicher. Ab dieser Adresse nach unten ist das RAM vornehmlich dem Benutzer vorbehalten. Den obersten Teil des Benutzerspeichers, der variabel nach unten ausgedehnt werden kann, bildet der vom Benutzer frei definierbare Zeichensatz. Grundsätzlich ist es möglich, alle 256 Zeichen des Firmwarezeichensatzes umzudefinieren, jedoch sollte man dabei auf die unteren 32 Symbole verzichten, da diese Kontrollzeichen darstellen (vergleiche Kapitel 9, Seite 2f im Handbuch) und der Computer diese bei Änderung nicht mehr als Kontrollzeichen interpretieren kann.

Beim Einschalten sind die obersten 16 Zeichen, das heißt die Zeichen mit den Codes von 240 bis 255, durch den Benutzer frei definierbar. Die Variable **HIMEM**, die die Trennlinie zum weiter unten liegenden BASIC zieht, hat den Wert 43903 oder in hex AB7F, oder bei Floppyeinsatz entsprechend niedriger. Man kann sie jederzeit mit

PRINT HIMEM bzw. PRINT HEX$(HIMEM)

abfragen. Die Differenz zu den oben genannten AC00 stellt den Speicherbedarf für eben jene 16 Zeichen dar. Mit SYMBOL AFTER schafft man nun Platz für neue Zeichen. Dabei wird HIMEM tiefer nach unten im Speicher geschoben und somit Platz für die neuen Symbole (8 Byte für jedes Zeichen) reserviert. Gibt man zum Beispiel SYMBOL AFTER 32 ein, so liegt HIMEM jetzt bei 42239. Wird der Speicherplatz knapp und kann man auf frei definierbare Zeichen verzichten, so ist es auch möglich, mit SYMBOL AFTER 256 HIMEM nach oben auf 44031 oder hex ABFF zu schieben, wodurch man noch einmal 128 Bytes gegenüber dem Einschaltzustand an freiem Speicherraum gewinnt.

Unterhalb des Benutzerzeichensatzes ist es möglich, einen weiteren Bereich für Maschinenroutinen oder Datenablage zu reservieren. Dies geschieht mit dem Kommando

MEMORY

Mit MEMORY wird die Variable HIMEM noch weiter nach unten gesetzt. Der dazwischenliegende Speicher steht dann zur gesicherten Abspeicherung von Maschinenprogrammen etc. zur Verfügung. Hierbei ist jedoch darauf zu achten, daß das SYMBOL AFTER-Kommando recht allergisch gegen MEMORY reagiert. Will man also einen benutzerdefinierten Zeichensatz schaffen und hat davor schon HIMEM mittels MEMORY nach unten verändert, so gibt der CPC ein unfreundliches IMPROPER ARGUMENT aus. Bevor man also weiteren Speicherplatz für Maschinenprogramme reserviert, muß man erst das SYMBOL AFTER für die gewünschte Anzahl von Zeichen ausführen. Ein Beispiel: Für ein Spiel etc. sollen 50 neue Zeichen definiert werden, daneben benötigt man noch reservierten Speicherplatz für Maschinenprogramme von der Größe von 2K. Unter der Annahme, daß die Zeichen in den obersten 50 Charactercodes abgelegt werden sollen, definiert man also SYMBOL AFTER 206 (HIMEM liegt jetzt bei 43631) und verschiebt dann die Obergrenze des BASIC-Speichers mittels MEMORY noch



um 2 K auf zum Beispiel auf 41600, mit MEMORY 41600 herab. HIMEM nimmt dann diesen neuen Wert an. Eine Änderung der Anzahl frei definierter Zeichen in dem so umdefinierten Speicherbereich (zum Beispiel SYMBOL AFTER 200) führt nun zur Ausgabe IMPROPER ARGUMENT.

Unterhalb HIMEM beginnt das eigentliche Reich des Benutzers, der Benutzerspeicher. Wir haben uns mit seinem Aufbau schon im letzten Kapitel ausgiebig beschäftigt.

Von oben nach unten wachsen in diesem Teil des Speichers die STRINGS. In umgekehrter Richtung türmen sich die BASIC-Variablen (numerische Variable, Schleifenvariablen und Arrays) übereinander. Der Zwischenraum bleibt für besondere Aufgaben wie zum Beispiel das Laden von Programmen offen. Noch eine Etage weiter unten finden wir dann unseren BASIC-Quelltext. Bei jeder Änderung in diesem Bereich wird das gesamte dahinterliegende Variablenfeld mit verschoben. Deshalb bleiben die Variablen bei einer Veränderung im BASIC-Text oder bei Programmabbruch im Gegensatz zu anderen Rechnern (zum Beispiel VC 64) erhalten. Der Boden unseres Speichers gehört wiederum der Firmware, diesmal im wesentlichen einigen Systemvariablen und den RESTARTS. Daneben enthalten die **unteren 367 Byte** noch einigen Freiraum, der für die Zwischenspeicherung der BASIC-Zeile bei der Übersetzung dient.

### 2.1.1 Die Restarts

Eine Besonderheit des Z80-Prozessors sind die RESTART-Anweisungen. Dabei handelt es sich um eine abgekürzte Form des normalen Sprungbefehles. Während dieser das Format Befehlscode und zwei Byte für Adresse hat, genügt bei der RESTART-Anweisung nur der Befehlscode, und dennoch ist es dem Prozessor möglich, an die richtige Stelle zu springen. Das wird dadurch erreicht, daß die Einsprungpunkte für die RESTART-Anweisung am Anfang des Speichers fix definiert sind. Der erste RESTART (RST 0) liegt im Fußpunkt des Speichers bei Adresse 0000. Trifft der Prozessor also auf eine RST-0-Anweisung im Maschinenprogramm, so springt er zur Adresse 0 und führt die dort angegebene Befehlsfolge weiter aus. RST 1 beginnt ab Speicherstelle 8, RST 2 ab Speicherstelle 16 beziehungsweise 10 hex usw. bis RST 7, das bei der Speicherstelle 0038 hex beginnt. Da die RESTART-Anweisung vom Prozessor relativ schnell ausgeführt und außerdem aufgrund der Kürze des Befehls erheblich weniger Speicherplatz verbraucht wird, gehört die Belegung der 8 RESTART-Anweisungen mit möglichst häufig benutzten Routinen zu einer der wichtigsten Entscheidun-

gen bei der Entwicklung eines Z80-Systems. Schnelligkeit und Effizienz der Maschine hängen wesentlich davon ab, ob größere, häufig benutzte Routinen schnell durch RESTART angesprungen werden können.

Eines der Hauptprobleme beim CPC, wie auch bei vielen anderen Homecomputern, ist das Umschalten zwischen ROM und RAM, das durch das Huckepackverfahren (Doppelbelegung der Adreßleitung) notwendig wird. Da dies, wie wir schon gesehen haben, beim CPC nicht durch feste Verdrahtung, sondern durch Softswitching erfolgt, liegt im schnellen Umschalten zwischen den einzelnen Bausteinen eine der Hauptmöglichkeiten für Zeitersparnis bei der Maschine. Eine weitere Anwendungsmöglichkeit entsteht bei der Interpretation von Sprungtabellen, wie zum Beispiel des Hauptfirmware-Jumpblocks zwischen BD00 und BDFF, mit dem wir uns weiter oben schon etwas näher beschäftigt haben. Auch hier kommt es darauf an, den Aufruf der entsprechenden Unterroutinen möglichst schnell und effizient abzuhandeln.

Diese beiden Bereiche stellen dann auch das Haupteinsatzgebiet für die RESTART-Anweisung beim CPC dar. Bis auf den RST 6 ab hex 0030, der vom Anwender benutzt werden darf, sind alle anderen RESTARTS dem System vorbehalten. Dabei ist zu sagen, daß die RESTART-Anweisungen im RAM und im parallelliegenden ROM identisch abgelegt sind. Das heißt, ein Umschalten zwischen unterem ROM und RAM beeinflußt das Funktionieren der RESTARTS nicht. Dies geschieht dadurch, daß beim Einschalten, das heißt der Initialisierung, das ROM ganz einfach ins RAM kopiert wird. Da der CPC davon ausgeht, daß diese Kopie einwandfrei ist, greift er teilweise auf das ROM und auch teilweise auf die entsprechenden RESTART-Anweisungen im RAM in seinen Betriebssystemroutinen zurück. Es braucht daher wohl eigentlich nicht gesagt zu werden, daß der CPC auf Änderungen in diesem Bereich durch POKE etc. sehr unangenehm reagiert. Auf die Funktion der einzelnen Restarts kommen wir später noch einmal zurück. Hier soll eine Grobübersicht genügen.

Der mächtigste RESTART ist **RST 0**. Beim Einschalten ist das untere ROM aktiviert, der entsprechende RAM-Bereich ausgeschaltet, und die Z80 beginnt ab Speicherstelle 0, die Maschinenbefehle zu lesen. Sie trifft dabei auf eben diesen RESTART, versetzt das System in den Ausgangszustand, löscht den Speicher, definiert die Firmwarevariablen, kopiert Teilbereiche der ROMs ins RAM, initialisiert den Bildschirm und springt anschließend zum Editor. Außer beim Einschalten wird dieser RESTART noch benutzt, wenn man gleichzeitig CTRL, SHIFT und ESC drückt. Man kann ihn auch benutzen, wenn bei falscher Eingabe eines Codewortes etc. das komplette



Programm so gelöscht werden soll, daß für einen Unbefugten keinerlei Rückschlüsse mehr auf den Programmablauf oder nur die Speicherverteilung gezogen werden können. Die **RESTARTS 1 und 5** springen in das untere ROM, wobei die Adresse im ROM in den nachfolgenden 2 Byte nach der RESTART-Anweisung abgespeichert sein muß. Die **RESTARTS 2 und 3** sind für die Ansprache der parallelliegenden oberen ROMs bestimmt. **RESTART 6** ist dem Benutzer vorbehalten. Ist das ROM eingeschaltet, so wird es bei RST 6 ausgeschaltet und ins RAM gesprungen. Der Anwender kann die Byte von 002C bis 0037 einschließlich für Eigenentwicklungen, Ansprünge eigener Unterprogramme, Spracherweiterungen etc. benutzen.

Nach dem RAM kommen wir nun zu einer Betrachtung des **ROM**. Betrachtet man die Adressen, unter denen ROM angesprochen werden kann, so fällt auf, daß der feste Speicher unseres Rechners, obwohl in ein und demselben Baustein beheimatet, dennoch einer adreßmäßigen Teilung unterliegt. Wir können ein oberes ROM (Adreßbereich hex C000-FFFF) und ein unteres ROM (Adressen 0000 bis 3FFF) unterscheiden. Der obere ROM-Teil enthält dabei den BASIC-Interpreter, also jene Routinen, die den BASIC-Quelltext in Maschinenroutinen übersetzen und ausführen. Der untere Teil wird durch die Betriebssystemroutinen und die von jedem System und jeder Hochsprache benötigte Fließkommaarithmetik eingenommen.

Soviel zunächst zu einer Groborientierung im Speicher der Maschine.

## 2.2 Einfache Routinen für die Speicheranalyse

Nachdem wir uns relativ ausgiebig mit der Aufteilung des RAM und der Anordnung des parallelgelagerten ROM beschäftigt haben, stellt sich nun die Frage, wie wir einzelne Speicherbereiche auslesen können. In bezug auf das RAM haben wir wichtige Routinen bereits kennengelernt. Mit dem BASIC-Kommando PEEK können wir den Wert einer jeden Speicherstelle abfragen und diese dann mit HEX$ und BIN$ umformen und so in anderen Zahlensystemen darstellen. Wenn wir vermuten, daß es sich bei den dort abgelegten Daten um Texte handelt, bietet sich die Verwendung der Funktion CHR$ an, um die zugehörigen ASCII-Zeichen darzustellen. Hierbei sollten wir jedoch beachten, daß, wie wir schon gesehen haben, die ASCII-Codes zwischen 0 und 31 beim CPC Kontrollfunktion haben, also Cursorbewegungen ausführen, den Grafikmodus umschalten oder Farben setzen, so daß wir diese bei einer normalen Ausgabe nicht ausführen sollten, da ansonsten die gerade geschriebenen Änderungen im Bildschirmaufbau auf-

treten. Die nachfolgenden beiden Routinen können universell verwandt werden.

**Speicheranalyseroutine:**

```
10 INPUT"Bereichsanfang";A
20 INPUT"Bereichsende";B
30 FOR i=A TO B
40 PRINT i,PEEK(i),HEX$(PEEK(i))
50 NEXT
```

**Textanalyseroutine:**

```
10 INPUT"Bereichsanfang";A
20 INPUT"Bereichsende";B
30 FOR i=A TO B
40 PRINT i,PEEK(i),:IF PEEK(i)>31 THEN PRINT CHR$(PEEK(i))
50 NEXT
```

Die erste dieser beiden Routinen haben wir bereits bei der Analyse von Variablen und Arrays ausgiebig angewandt und auch die zweite Routine dürfte vom Verständnis her keine Probleme ergeben. Beim Gebrauch dieser Routinen sind wir jedoch auf das RAM beschränkt. Wir wollen uns nun damit beschäftigen, wie wir auch auf die beiden parallelgelagerten ROM-Speicher zurückgreifen können, um dort Informationen auszulesen und diese zu analysieren. Das Umschalten zwischen RAM und ROM kann nur via Maschinensprache geschehen. Wir müssen also auf eine kleine Maschinenspracheroutine zurückgreifen.

### 2.2.1 Die Ablage von Maschinenprogrammen

Es stellt sich zunächst die Frage, wo wir Maschinenprogramme sicher ablegen können. Da nämlich, wie wir gesehen haben, der CPC in seinem Benutzerspeicher bei der Anlage beziehungsweise Änderung von Daten ständig Datenmengen hin und her schiebt, ist dieser Bereich für die Ablage eines Maschinenprogramms grundsätzlich nicht besonders geeignet. Auf Ausnahmen kommen wir gleich noch zurück. Was wir brauchen, ist ein gesicherter Speicherbereich, auf den keine Routine des Betriebssystems, sei es die Bildschirmverwaltung, Ein-/Ausgabesteuerung oder der BASIC-Interpreter, zurückgreift. Eine solche Möglichkeit findet sich beim CPC an der oberen Speichergrenze. Wie wir schon gesehen haben, können wir mit dem



MEMORY-Kommando HIMEM, das heißt die Obergrenze des Benutzerspeichers, nach unten verschieben. Normalerweise liegt sie bei Speicherstelle 43903 beziehungsweise bei angeschalteter Floppy natürlich entsprechend niedriger. Geben wir nun

```
MEMORY 42999
```

ein, so steht uns der Speicherbereich von 43000-43903 zur Verfügung. Die Speicherstelle 42999 enthält dann das letzte Byte des ersten definierten Strings.

Im Zusammenhang mit Strings steht auch die zweite Möglichkeit, die sich allerdings nur für die Ablage von kurzen Maschinenprogrammen, die auf keine absoluten Adressen zurückgreifen, eignet. In Kapitel 1.3 haben wir den Variablenpointer "@", einen Zeiger auf das erste Byte eines abgespeicherten Strings, kennengelernt.

```
PRINT PEEK(@m$+1)+256*PEEK(@m$+2)
```

gibt uns diese Position an. Wir brauchen nun nur die einzelnen Maschinenbefehle und Daten mit CHR$ in Grafiksymbole umzuwandeln und zu addieren. Der so gebildete String enthält dann ein Maschinenprogramm, das wir unter Benutzung des Pointers anspringen können. Da aber Strings verlagert werden können, dürfen in einem solchen Programm keine absoluten Adressen benutzt werden.

**Der Aufruf eines Maschinenprogramms:**

Der Ansprung eines Maschinenprogramms geschieht mit Hilfe des BASIC-Kommandos CALL. Trifft der CPC auf diesen Befehl, so springt er zu dem mit der Anfangsadresse definierten Maschinenprogramm und legt gleichzeitig die Rücksprungadresse auf dem Maschinenstapelspeicher, dem STACK, ab. Nun wird das Maschinenprogramm abgearbeitet, so lange, bis der Computer auf den Code 201 beziehungsweise C9 hex trifft. Mit diesem Maschinensprache-Return-Kommando (RET) wird der Prozessor angewiesen, wieder in die nächst höhere Ebene zurückzukehren, und der Ablauf des BASIC-Programms setzt sich nahtlos fort.

Wie muß nun eine Routine aussehen, die es uns ermöglicht, Informationen aus dem unteren oder dem oberen parallelen ROM zu lesen?

### 2.2.2 Das Umschalten zwischen ROM und RAM

Aufgabe dieser Routine muß es sein, zunächst einmal das gewünschte ROM, oberes oder unteres ROM, einzuschalten, danach ein Byte einer vorher angegebenen Speicherstelle aus diesem zu lesen und dieses an einer Stelle im RAM zu speichern. Danach muß es die Kontrolle zurück an BASIC übergeben, jedoch sollte es vorab das ROM wieder in die Ruhelage zurückbringen. Das Hauptproblem stellt also das Ein- und Ausschalten der benötigten Speicher dar.

Glücklicherweise stellt uns hierfür allerdings das Betriebssystem des Rechners eine Reihe von Routinen zur Verfügung, die wir von unserem Maschinenprogramm als Unterprogramme aufrufen können. Der Aufruf dieser Systemroutinen geschieht mit Hilfe einer Sprungtabelle, die von hex B900 bis hex B923 im RAM des Rechners liegt. Die einzelnen Sprungzeiger führen dabei folgende Funktionen aus:

**B900 U ROM ENABLE**

Diese Routine schaltet das obere ROM ein. Der Aufruf ist an keine bestimmten Einsprungbedingungen geknüpft, beim Aussprung enthält das Register A des Prozessors den alten ROM-Status. Mit der Routine ROM RST (Adresse B90C) ist es möglich, den vorherigen Stand wiederherzustellen.

**B903 U ROM DISABLE**

Diese Routine ist das Gegenstück zu B900, sie schaltet das obere ROM aus. A enthält dabei wiederum den vorherigen ROM-Status, so daß es möglich ist, mit B90C den alten Zustand wiederherzustellen.

**B906 L ROM ENABLE**

Diese Routine schaltet das untere ROM ein. Dieses ist normalerweise gesperrt, wenn keine Firmwareroutine aufgerufen wird. A beinhaltet auch hier wieder den vorherigen ROM-Status. Einsprungbedingungen: keine.

**B900 L ROM DISABLE**

Diese Routine sperrt das untere ROM, ist also das Gegenstück zu B906. A beinhaltet wieder den alten Speicherzustand.



**B90C ROM RST**

Diese Routine stellt den vorherigen ROM-Status wieder her, das heißt, unabhängig davon, ob durch den Aufruf einer der Routinen von B900 bis B90B eine Änderung erfolgte oder nicht, wird immer der Ursprungszustand wiederhergestellt. Dieser muß dabei im Register A zwischengespeichert sein. Beim Aussprung aus der Routine ist das Register A zerstört. Es besteht natürlich auch die Möglichkeit, mit U ROM DISABLE die Auswirkungen von U ROM ENABLE wieder aufzuheben. Allerdings hat diese Methode den Nachteil, nicht immer den Ursprungszustand wiederherzustellen. Wurde zum Beispiel ein bereits eingeschaltetes unteres ROM mit U ROM ENABLE weiterhin eingeschaltet, so führt die Gegenoperation U ROM DISABLE nun natürlich nicht in den Ausgangszustand zurück. Daher ist für den Normalfall die Benutzung von ROM RST vorzuziehen. Sie benötigt jedoch den alten ROM-Status in Register A. Man muß also, falls man im Verlauf des eigenen Maschinenprogramms mit dem Register A weiterhin arbeiten will, diesen Wert zwischenspeichern.

Aus diesen Vorbemerkungen ergibt sich die Programmstruktur nun fast von selbst. Wir wollen uns das Ganze einmal am Beispiel der Untersuchung des oberen ROMs anschauen. Zunächst das Maschinensprachelisting:

```
CALL B900          CD 00 B9
LD A,(C000)        3A 00 C0
LD (A800),A        32 00 A8
CALL B903          CD 03 B9
RET                C9
```

Mit dem ersten Kommando rufen wir die Routine in B900 auf. Damit ist das obere ROM eingeschaltet, und wir können auf dieses beim Lesen zugreifen. Dazu benutzen wir den Maschinenbefehl CALL, der in etwa dem BASIC-Befehl GOSUB entspricht. Es wird auch hier eine Rücksprungadresse abgelegt, womit nach Durchlauf der Routine B900 der Prozessor beim nächsten Statement mit der Abarbeitung unseres Maschinenprogramms fortfährt. Als nächstes laden wir das Register A mit dem Inhalt der Speicherstelle C000, der ersten Speicherstelle im oberen ROM. Diese legen wir dann mit dem nachfolgenden Befehl im RAM an der Speicherstelle A800 ab. Mit einem weiteren CALL schalten wir das obere ROM wieder aus und kehren dann mit dem RETURN-Befehl wieder in unser BASIC-Programm zurück.

Die entsprechenden Codes für die vier verschiedenen Befehle lauten CD für den CALL-Befehl, das Laden von A mit einer Speicherstelle wird durch das Kommando 3A veranlaßt, die Umkehrung dazu erreichen wir mit dem Code 32, der RET-Befehl schließlich hat den Code C9 (Alle Angaben jeweils in HEX). Die oberen drei Befehle benötigen dabei jeweils noch eine 2-Byte-Adresse, die wie üblich im Format Lo-Hi abgespeichert ist. Schauen wir nun einmal, wie wir unser Programm ablegen und danach ausführen können.

### 2.2.2.1 Programmablage in einem String

Als erstes wollen wir uns mit der Ablage in einem String beschäftigen. Dazu benutzen wir das nachstehende kleine Programm.

```
10 ' **********************
20 ' ** oberes ROM mit $ **
30 ' **********************
40 DATA CD,00,B9,3A,00,C0,32,00,A8,CD,03,B9,C9,X
60 READ a$:IF a$="X" THEN 80
70 z$=z$+CHR$(VAL("&"+a$)):GOTO 60
80 CALL PEEK(@z$+1)+256*PEEK(@z$+2)
90 PRINT PEEK(&A800)
```

In Zeile 40 finden wir in Data abgelegt die HEX-Codes unseres Maschinenprogramms. Das X am Ende der Zeile stellt dabei eine Endemarkierung dar. Die Zeilen 60 und 70 werden so lange durchlaufen, bis der Computer bei dem READ-Befehl in Zeile 60 ein X einliest. Danach wird die nachfolgende IF-Abfrage wahr, und die weitere Ausführung des Programms läuft über Zeile 80. In Zeile 70 wird unser Maschinenprogrammstring zusammengefügt. Da diese Methode auf den ersten Blick etwas ungewöhnlich ist, wollen wir noch mehr auf sie eingehen. Mit dem erstmaligen Aufruf des READ-Kommandos in Zeile 60 wird in A$ ein C und ein D eingelesen. Diese stellen nun aber nicht direkt eine HEX-Zahl dar, sondern einfach nur die beiden Buchstaben des Alphabets, in 2 Byte werden sie hintereinander wie üblich abgespeichert. Die IF-Abfrage ist unwahr, also geht es nach Zeile 70. Hier geschieht nun die Umwandlung dieses Textes. Was wir nämlich benötigen ist ein ASCII-Zeichen, das genau als Zahlenwert dem Maschinenprogrammbefehl entspricht. Als erstes addieren wir dazu zu unserem A$ das "&"-Zeichen. Damit haben wir nun einen 3 Byte langen String erzeugt. Auf diesem können wir nun die Funktion VAL anwenden. VAL gibt uns den Dezimalwert eines Stringausdrucks wieder. Durch das



vorangestellte & wird nun unser hexadezimaler Textstring in einen Dezimalwert umgewandelt. Wir können uns dies einmal anschauen.

```
PRINT VAL("&"+"CD")
```

ergibt als Ergebnis den Wert 205: das Dezimaläquivalent für die HEX-Zahl CD. Die etwas kompliziert erscheinende Methode ist notwendig, da es außer in der Abspeicherung als String keine Möglichkeiten im BASIC gibt, um mit hexadezimalen Größen direkt rechnen zu können.

Von dem so herausgefundenen Dezimalwert können wir natürlich nun mit der CHR$-Funktion relativ einfach ein äquivalentes ASCII-Zeichen erzeugen. Da der Wert über 128 liegt, erhalten wir hier ein Zeichen des erweiterten ASCII-Codes, das heißt ein Grafikzeichen, einen von links oben nach rechts unten verlaufenden Schrägstrich. Welche Zeichen bei den einzelnen dezimalen, beziehungsweise HEX-Werten dabei auftauchen müssen, können Sie zum Beispiel in Anhang 3 des Betriebshandbuches nachsehen. Auf Seite 10 finden wir dort das gerade angesprochene Grafikzeichen. Nach dieser Methode fügt das Programm Zeichen für Zeichen an den Z$-String an, bis es auf das schon angesprochene X stößt. Danach folgt ein CALL-Befehl, das heißt der Aufruf eines Maschinenprogramms. Da uns die genaue Lage des Z$-Strings, speziell wenn wir mit mehreren Strings arbeiten, nicht bekannt ist, greifen wir hier wieder auf die Funktion "@" zurück. Der Variablenpointer "@" deutet auf das erste Byte nach dem Stringnamen, das heißt die Länge des Strings. Fragen wir dann die folgenden beiden Byte ab, so erhalten wir seine momentane Lage, die wir nach geeigneter Umrechnung, das heißt als Lo-Hi-Adresse interpretiert, zum Ansprung unseres Maschinenprogramms benutzen können. Unser Maschinenspracheprogramm legt das gesuchte Byte in der RAM- Speicherstelle A800 ab, wo wir es nach der Rückkehr aus der Maschinenroutine mit PEEK abfragen können. Als Ergebnis beim Durchlauf dieses Programms müssen Sie den Wert 128 erhalten.

### 2.2.2.2 Maschinenprogrammsicherung mit HIMEM

Schauen wir uns nun die zweite Variante an, die Abspeicherung im geschützten Speicherbereich. Das Prinzip ist identisch von der Maschinenspracheseite her, so daß wir hier gleich zum BASIC-Programm übergehen können.

```
10 ' ***************************
20 ' ** oberes ROM mit HIMEM **
30 ' ***************************
40 DATA 40000,CD,00,B9,3A,40,C0,32,00,A0,CD,03,B9,C9,X
50 READ j:i=j:MEMORY i-1
60 READ a$:IF a$="X" THEN 80
70 POKE i,VAL("&"+a$):i=i+1:GOTO 60
80 CALL j
90 PRINT PEEK(&A000)
```

Auch hier wird eine Variante verwendet, die leicht für andere Maschinenspracheprogramme benutzt werden kann und die wir deshalb im Verlauf des Buches noch einige Male anwenden werden. In der Datazeile finden wir als neuen Wert am Anfang die neue Speicherobergrenze. Auf diesen Wert beschränkten wir mit dem MEMORY-Kommando den Benutzerspeicher. Die Zeilen 60 und 70 haben dieselbe Funktion wie bei der Stringmethode. Auf die Erzeugung eines ASCII-Characters können wir hier verzichten, dafür muß der Dezimalwert in aufeinanderfolgende Speicherstellen hineingepoket werden. Wir benutzen dazu als Adresse die Variable I, die damit von den anfänglichen 40000 bis 40012 heraufläuft. Der Aufruf unseres Programms gestaltet sich nun einfacher. Die Ansprungadresse ist direkt bekannt, es handelt sich um die anfangs eingelesene 40000, die sich noch in J befindet. Unseren gesuchten Wert erhalten wir wiederum durch Abfrage einer Speicherstelle, diesmal A000 mit PEEK. Da wir diesmal die Speicherstelle C040 analysiert hatten, ergibt sich auch ein etwas anderer Wert, eine 66, der ASCII-Code für B. Was es mit diesem großen B und den nachfolgenden Zeichen auf sich hat, werden wir gleich noch sehen.

Zunächst wollen wir jedoch unsere theoretischen Betrachtungen über das Umschalten und Abfragen von ROMs in ein handliches, universell verwendbares Abfrageprogramm umsetzen. Dieses soll es uns ermöglichen, an einer beliebigen Stelle im oberen oder unteren ROM eine beliebige Anzahl von Stellen dezimal und hexadezimal anzuschauen. Ein solches kleines Programm stellt ROMREAD dar. Im Prinzip ist es mit unserem vorherigen identisch. Allerdings bietet es nun wahlweisen Zugriff auf eines der beiden ROMs, es werden 4 Byte gleichzeitig mit jedem Ansprung der Routine ausgelesen, und außerdem wurde die Programmierung sauberer. Während wir bei unserem letzten Programm noch mit den Routinen U ROM ENABLE und U ROM DISABLE gearbeitet haben, wollen wir nun lieber auf ROM RST zurückgreifen, da diese Routine unabhängig vom vorherigen Ausgangszustand die alte Konstellation von RAM und ROM wieder herstellt.



Beim Ansprung unserer letzten Routine konnten wir uns darauf verlassen, daß beide ROMs ausgeschaltet sind, da dies beim Ausführen des BASIC-Befehls CALL immer der Fall ist. In größeren und komplexen Programmsystemen aber, und hieran wollen wir uns schon möglichst schnell gewöhnen, kann derartiges Vertrauen zu höchst unangenehmen Konsequenzen führen und im Extremfall sogar dazu, daß sich der Rechner "aufhängt". Denken Sie zum Beispiel einmal daran, was passiert, wenn der Rechner im Vertrauen darauf, daß Sie ihm die alte Speicherkonfiguration wiederherstellen, eine Routine im BASIC-ROM anspringen will, er aber statt dessen durch Ihren Ausschaltbefehl im Grafikspeicher landet. Was beim ersten Durchdenken möglicherweise noch ganz lustig erscheint, kehrt sich in sein Gegenteil, wenn Sie sich vorstellen, daß Sie gerade zu diesem Zeitpunkt ein neues Programm entwickelt hatten, von dem unglücklicherweise noch keine Sicherungskopie gezogen wurde. Wir sollten uns also schon möglichst früh angewöhnen, den Speicherzustand wieder in die Ausgangslage zu versetzen. Trotzdem sollten Sie natürlich spätestens ab jetzt nicht mehr auf ein vorheriges Abspeichern des Programms vor dem eigentlichen Lauf verzichten.

Assemblerlisting ROMREAD

```
CALL U ROM ENABLE       CD 00 B9
LD B,A (ROMSTATE)       47
CALL L ROM ENABLE       CD 06 B9
LD HL,<ADRESSE>         21 00 00
LD A,(HL)               7E
LD(40050),A             32 72 9C
INC HL                  23
LD A,(HL)               7E
LD(40051),A             32 73 9C
INC HL                  23
LD A,(HL)               7E
LD(40052),A             32 74 9C
INC HL                  23
LD A,(HL)               7E
LD(40053),A             32 75 9C
LD A,B                  78
CALL ROM RESTORE        CD 0C B9
RET                     C9
```

Doch zurück zur Interpretation des Programms. Am Anfang steht der Aufruf der uns schon bekannten Routinen U ROM ENABLE und L ROM ENABLE. Nach dem Durchlauf von U ROM ENABLE enthält der Akkumulator die alte Speicheraufteilung von RAM und ROM. Diese müssen wir nun bis zum Anlauf der Routine ROM RST zwischenspeichern, falls wir den Akkumulator wie gewohnt bei der Abfrage von Speicherstellen benutzen wollen. Als Zwischenspeicher dient uns dabei ein anderes Register unseres Prozessors, das Register B. Mit dem Befehl

LD B,A

wird in Maschinensprache der Inhalt des Registers A in das Register B kopiert. Der HEX-Code dafür ist 47. Mit dem nächsten Befehl lernen wir eine andere Variante des Ladebefehls kennen. Hier werden zwei Register verändert, das Registerpaar HL. Die nachfolgenden beide Byte geben dabei an, auf welchen Wert das Registerpaar gesetzt werden soll. Die Abspeicherung erfolgt dabei wieder im Format Lo-Hi; das heißt: Das erste Byte wird in das Register L geschickt, das zweite Byte nach dem Befehlscode findet sich danach im Register H wieder. Diese Art des Ladens wird als immediate, also unmittelbar, bezeichnet, da hier die zu ladenden Werte direkt, explizit angegeben sind. Das Gegenstück dazu lernen wir beim nächsten Befehl kennen, das indirekte Laden. Hiermit wird das Register A mit dem Inhalt der Speicherstelle geladen, der durch das Registerpaar HL angegeben ist.

**Beispiel:**

HL wurde auf C000 gesetzt. A nimmt nun den Inhalt der Speicherstelle C000 an. Bei eingeschaltetem RAM wäre das das erste Byte des Bildschirmspeichers, ansonsten das erste Byte des oberen ROMs.

In dem nächsten Befehl wird der Inhalt von A nun an eine Stelle im RAM geschrieben, Speicherstelle 40050, was umgerechnet in HEX den etwas krummen Wert von 9C72 ergibt. Auch hier erfolgt die Angabe der Speicherstelle wieder im Format Lo-Hi. Damit erhält eine in RAM gelegene Speicherstelle, auf die wir jederzeit zugreifen können, den Wert eines Bytes aus dem ROM. Welches Byte dabei gelesen wird, wird, wie schon gesagt, durch das Laden von HL bestimmt. Poken wir also dann in die vier Nullen eine 16-Bit-Adresse, so können wir mit dieser Methode auf jedes beliebige Byte, gleich in welchem ROM, zurückgreifen. Nach der Rückkehr aus dem Maschinenprogramm finden wir den Wert dann in Speicherstelle 40050 wieder.



Als nächstes erhöhen wir den Inhalt des Registerpaares HL um 1. Dies leistet der INC-Befehl. Dadurch ist nun in HL die nächst höhere Zugriffsadresse, zum Beispiel C001, verfügbar. Wir laden den Inhalt dieser Speicherstelle wiederum in A und speichern es in die nächst höhere Zelle im RAM zurück, also nach 40051. Nachdem wir diese Prozedur noch zweimal wiederholt haben, sind vier aufeinanderfolgende Byte in das RAM zwischen Adresse 40050 und 40053 kopiert worden. Ein BASIC-Programm muß also nun nur noch den Anfang des auszulesenden Speicherbereichs in 2 Byte als Hi-Lo-Adresse zerlegt, an die Stelle, wo sich die vier Nullen im Maschinencode befinden, poken und kann dann nach Durchlauf des Maschinenprogramms den Inhalt der adressierten ROM-Zellen im RAM wiederfinden.

```
10 ' *************
20 ' ** romread **
30 ' *************
40 MEMORY 39999
50 DATA cd,00,b9,47,cd,06,b9,21,00,00
60 DATA 7e,32,72,9c,23,7e,32,73,9c,23
70 DATA 7e,32,74,9c,23,7e,32,75,9c
80 DATA 78,cd,0c,b9,c9,x
90 i=40000
100 READ a$:IF a$="x" THEN 120
110 POKE i,VAL("&"+a$):i=i+1:GOTO 100
120 INPUT"ab welcher Speicherstelle";s
130 IF s<0 THEN s= s+65536
140 s1=INT(s/256):s2=s-256*s1
150 POKE 40008,s2:POKE 40009,s1:CALL 40000
160 FOR i= 40050 TO 40053:PRINT i,PEEK(i),HEX$(PEEK(i)):NEXT
170 GOTO 120
```

Dies ist dann auch der genaue Funktionsablauf von ROMREAD. Zunächst wird in Zeile 40 mit dem MEMORY-Kommando die Speicherobergrenze auf 40000 gesetzt, so daß wir von 40000 nach oben über einen geschützten Speicher für Maschinenprogramme verfügen. In diesen laden wir dann das Maschinenprogramm ein, als Endemarkierung dient uns hier wiederum ein X.

Es folgt in Zeile 120 die Abfrage der gewünschten Speicherstelle. Diese kann wahlweise in dezimal oder in HEX eingegeben werden. Bei HEX ist jedoch ein vorangestelltes &-Zeichen notwendig. Um bei HEX-Zahlen von größer als 8000 nicht zu negativen Werten von S zu kommen, addieren wir, wie schon bei der Konvertierung von Zahlensystemen ineinander, wiederum $2^{16}$=65536.

Zeile 140 zerlegt diese Adresse dann in ein HIGH-Adreßbyte und ein LOW-Adreßbyte. Das HIGH-Byte ergibt sich dabei, indem man die Adresse durch 256 teilt. Die überbleibende Ganzzahl gibt den Block an, in dem die Speicherstelle liegt. Die Position innerhalb eines Blocks findet man dann, indem man von der Adresse die Anzahl der Blöcke, multipliziert mit 256 Byte je Block, abzieht. Zeile 150 poket dann die beiden Werte an die entsprechenden Stellen im Maschinenprogramm und ruft dieses danach auf.

Nach der Rückkehr können dann in einer FOR-TO-Schleife die vier RAM-Speicherstellen ausgelesen werden. Wollen wir statt der Ausgabeposition im RAM lieber unsere ROM-Speicherstellen in der ersten Spalte wiederfinden, so brauchen wir nur das PRINT-Kommando ein wenig zu ändern. Wir ersetzen dazu das erste I nach dem PRINT-Befehl einfach durch I-40050+S.

## 2.3 Ein Monitor für den CPC

Im Verlauf der letzten beiden Kapitel haben wir eine Reihe nützlicher Routinen kennengelernt, die es uns ermöglichten, Zahlenwerte ineinander zu konvertieren, ASCII-Werte im Text umzuformen oder uns Speicherstellen und Speicherbereiche auf die verschiedensten Arten und Weisen anzuschauen. Was aber noch fehlt, ist ein universell verwendbares Programm, welches all diese Möglichkeiten als Unterfunktionen enthält.

### 2.3.1 Anforderungen an den Monitor

Darüber hinaus sollte ein solches Programm natürlich auch noch über eine Reihe anderer Möglichkeiten verfügen. So sollten zum Beispiel Speicherbereiche löschbar sein, es müßte möglich sein, einen Teil des MEMORYs zu sichern und auch Daten vom Band in einen Bereich hineinzulesen. Daneben sollte ein solches Programm in der Lage sein, Maschinenprogramme aufzurufen und Maschinenprogramme zu disassemblieren, das heißt, die Kürzel oder Mnemonics für die einzelnen Befehle auszugeben, also zum Beispiel den Code 23 mit INC HL zu übersetzen und uns so ein Befehlslisting eines Maschinenspracheprogramms, ein sogenanntes Assemblerlisting, zu erstellen. Darüber hinaus sollte ein solches Programm natürlich jederzeit mit neuen Funktionen erweiterbar sein, um das Programm steigenden Bedürfnissen oder neuen Erkenntnissen anpassen zu können.



### 2.3.2 Das Programm

Das nachfolgend abgedruckte Programmlisting CPC MON stellt ein solches Programm, oder besser gesagt, den ersten Teil eines solchen Programms dar, denn einige Funktionen, die größere Kenntnisse im Bereich der Maschinensprache oder die Kenntnis des Z80-Befehlssatzes für die Programmierung des Programms voraussetzen, wurden hier noch nicht implementiert. Wir werden dieses im Verlauf des Buches nachholen und verfügen dann ab Kapitel 6 über das vollständige Programm, einen Monitor. Der Aufbau des Programms ist relativ einfach. Es besteht eigentlich nur aus einer Befehlsabfrageschleife, die die Zeilen 250-420 in Anspruch nimmt, und aus einem Inputbefehl sowie dessen Auswertung. Jede Funktion des Programms ist durch einen Buchstaben und gegebenenfalls noch einige Parameter gekennzeichnet. Werden die Parameter direkt mit eingegeben, so muß nach dem Buchstaben ein Leerzeichen stehen. Zur Bedeutung der einzelnen Buchstaben:

**A** Mit dieser Funktion wird das Ausgabegerät festgelegt. Die meisten Unterprogramme von CPC MON geben die Daten auf dem Schirm aus. Bei einigen größeren Ausgaben wie zum Beispiel beim Disassemblieren eines Programms oder beim Anzeigen von Speicherbereichen ist es jedoch wünschenswert, diese auch auf dem Drucker abbilden zu können. Wurde mit der Funktion A der Drucker eingeschaltet, so werden nun diese Ausgaben umdirigiert.

**B** Mit der Blocktransferroutine ist es möglich, Speicherbereiche zu verschieben. Diese Routine ist im Grundmodul noch nicht enthalten, wir werden sie später einfügen. Sie kann dazu verwendet werden, ein Maschinenprogramm, das zu groß geworden ist, an eine andere Stelle zu verlagern oder auch die Kopie bestimmter Variablenpointer zu ziehen, um diese dann in einem gesicherten Speicherbereich abzulegen und somit als Momentaufnahme einen Zustand eines BASIC-Programms einzufrieren.Zum Beispiel könnten die Variablenpointer, die den Benutzerspeicher unterteilen, also Anfang der Variablen, Anfang der Arrays, Ende der Arrays, Anfang der Strings und Ende der Strings zwischengespeichert werden.

**C** Zahlensysteme konvertieren. Hiermit können wir Werte von einem Zahlensystem in das andere umwandeln. Nach Eingabe einer Zahl (wahlweise dezimal, hex oder Dual) wird ihr Wert in den drei Zahlensystemen ausgegeben.

**D** Programm disassemblieren. Diese Routine übersetzt den Maschinencode eines Programms in den entsprechenden Assemblertext durch Ausgabe der Mnemomics. Sie stellt ebenso wie Blocktransfer eine Programmerweiterung dar, die wir im Kapitel 5 einfügen werden.

**G** Mit dieser Option ist es möglich, ein Maschinenprogramm durchlaufen zu lassen oder zu testen. G ohne Parameter dient dabei zum Fortsetzen eines durch einen Breakpoint abgebrochenen Programms, mit G <Adresse> sind wir in der Lage, ein Maschinenprogramm an einer bestimmten Stelle im Speicher anlaufen zu lassen. Die Ausführung dieser Routine bedarf einer Reihe interner Kenntnisse über den Prozessor in Z80 und kann deswegen erst im Kapitel 6 eingefügt werden. Eine erste Vorstufe wird in Kapitel 4 erarbeitet.

**H** Diese Unterfunktion gibt eine Hilfsliste des Befehlssatzes unseres Monitors aus.

**I** Diese Option erlaubt es uns, Speicherstellen zu ändern. Sie dient im wesentlichen dazu, Maschinenprogramme zu erstellen.Dabei kann in beliebiger Stelle der Code eingegeben werden. Die Beendigung der Eingabe geschieht, indem man ein X eingibt, dies wird als Endemarkierung aufgefaßt.

**K** Speicherbereiche löschen. Dies ist die Umkehrung zu I. Die zugehörige Unterroutine löscht, das heißt nullt einen vorher eingegebenen Speicherbereich.

**L** Mit dieser Routine können wir einen Speicherbereich von Kassette laden.

**M** Dieses Kommando steht für MEMORY, das Anzeigen eines Speicherbereichs. Die Darstellung erfolgt dabei sowohl durch HEX-Codes als auch im rechten Teil des Bildschirms, als Grafiksymbole ausgegeben. Es werden auf einer Zeile des Bildschirms jeweils die Inhalte von 8 Speicherstellen nebeneinander ausgegeben, so daß die in der ersten Spalte angegebene Speicheradresse im HEX-Format sich im 8er Rhythmus verändert. Einfaches M ohne weitere Parameterangaben stellt den Speicherbereich ab Speicherstelle 0 dar, jeweils für 15 Bildschirmzeilen, also 120 Byte Platz. Die erste Zahl nach dem M wird als Anfang des darzustellenden Bereichs angesehen, mit Semikolon ist es dann noch möglich, eine weitere Zahl, das Ende des abzubildenden Bereichs, anzugeben. Die Angabe der Parameter kann dabei dezimal,



hex oder binär erfolgen. Es sind bei anderer als dezimaler Eingabe allerdings die entsprechenden Kürzel, das heißt & beziehungsweise &X, zu verwenden. Bei der grafischen Ausgabe bedeutet ein Fragezeichen einen Wert >32, also ein Kontrollzeichen. Ob aus dem ROM oder RAM gelesen werden soll, wird mit der Funktion R festgelegt.

**P** Hiermit legen wir die Programmobergrenze fest. Die Variable HIMEM unterteilt den dem Benutzer verfügbaren Speicher in einen dem BASIC vorbehaltenen Speicher und einen gesicherten Bereich, den wir für die Ablage von Maschinenspracheprogrammen benutzen können. Beim Anlauf des Monitors setzt dieser die Variable HIMEM mittels MEMORY auf 39999, so daß die Speicherstellen von 40000 nach oben definiert sind. Je nachdem, ob man mit angeschalteter Floppy oder nur mit Kassette arbeitet, braucht der Rechner selber für die Abspeicherung seiner internen Variablen den Speicher von unterhalb des Bildschirmspeichers bis 43903 (Kassette) oder 42619 (Floppy). CPC MON benutzt den Bereich zwischen 40000 und 42000 für die Ablage seiner Variablen und gegebenenfalls der des Benutzers. Oberhalb von 42000 kann der Benutzer also Programme ablegen. Sollte dieser Platz nicht reichen, ist es dann möglich, mit P HIMEM noch weiter nach unten zu verschieben und den Bereich zum Beispiel von 38000-39999 zu benutzen.

**R** Diese Funktion dient der Umschaltung zwischen ROM und RAM als Basis für alle Leseoperationen. Es wird der Inhalt der BASIC-Variablen ROM in ihr Gegenteil verkehrt. Diese Variable dient den Funktionen D und M als Grundlage, um festzustellen, ob aus dem ROM oder dem RAM gelesen werden soll.

**S** Speicherbereich sichern. Diese Routine stellt ein Maschinen- SAVE dar, genauer gesagt, das Abspeichern eines binären Files. Nach Angabe von Ober- und Untergrenze des Bereiches, der abgespeichert werden soll, wird dieser auf Kassette gesichert. Diese Funktion eignet sich, um Texte in einem Maschinenspracheprogramm einfügen zu können. Oftmals ist es nötig, Kommandos, Erklärungen etc. auch durch ein Maschinenspracheprogramm ausgeben zu können. Die Unterroutine P legt nach Eingabe eines Strings diesen anhand seiner ASCII-Werte Zeichen für Zeichen anhand einer vorher eingegebenen Speicherstelle ab.

**X** Diese Funktion dient dazu, CPC MON zu verlassen und in BASIC zurückzukehren.

Schauen wir uns nun einmal CPC MON etwas näher an. Das Programm besteht aus drei Teilen. Den Anfang bildet der **Initialisierungsteil (bis Zeile 240)**. Er dient dazu, die Bildschirmaufteilung und die Speicherbereichsverteilung (HIMEM) festzulegen. In einem zweiten Schritt wird dann ein uns schon bekanntes Maschinenprogramm in den gesicherten Speicher ab 40000 gepoket, "ROMREAD". Als nächstes werden einige Variable gesetzt. ROMVAL gibt die Stelle im Speicher an, in die wir die Adresse einpoken müssen, ab der ROMREAD den Speicher ausliest. Die Anfangsadresse, das heißt, den Ansprungpunkt unserer Maschinenspracheroutine, finden wir in der Variablen ROMREAD wieder. Er liegt hier, wie auch schon im Programm ROMREAD selber, bei 40000. Danach werden zwei Variable gesetzt, die für die Ein-/Ausgabe notwendig sind. ROM legt fest, ob aus dem RAM oder ROM gelesen werden soll, STREAM definiert das Ausgabegerät, eine 0 bedeutet hier Schirm, eine -1 den Drucker.

Wir kommen nun zum **zweiten Teil** unseres Programms, der **Befehlsabfrageschleife**. Sie läuft von Zeile 250-420 und besteht im wesentlichen aus dem schon besprochenen INPUT-Befehl, der das Kommando einliest, der Ablösung des ersten Zeichens von links und Speicherung mit der Variablen C$ und darauf folgend einem Sprungverteiler, der in Abhängigkeit von C$ die einzelnen Unterprogramme aufruft. Hier können natürlich mit anderen Anfangsbuchstaben noch weitere Unterprogramme relativ problemlos eingefügt werden. Das Unterprogramm muß dabei dann jeweils nach Zeile 420 zurückspringen.

Der **dritte Teil** wird dann durch die verschiedenen **Interpretationsroutinen** gebildet. Den Anfang bildet dabei die Interpretationsroutine für das Kommando M, Speicher anzeigen.

In einem ersten Schritt werden hier auch anderweitig zur Zwischenspeicherung benutzte Variable Z$ und Y$ gelöscht. Danach wird der Kommandostring auf die Existenz eines Semikolons untersucht. War ein Semikolon vorhanden, so sind Anfangs- und Endadresse des anzuzeigenden Speicherbereichs angegeben worden. Ansonsten entweder nur ein M (hier müßte die Interpretation ab Speicherstelle 0 erfolgen), oder es wurde wenigstens die Anfangsadresse mitgegeben. Je nachdem, welchen Wert W annimmt, wird dann in Zeile 480 der Befehlsstring auf unterschiedliche Art und Weise behandelt. Falls kein Semikolon angegeben wurde, wird der gesamte rechte Bereich unseres Befehlsstrings ab der dritten Stelle als Angabe der Anfangsadresse aufgefaßt.



MID$(B,3)

löst diesen Teil aus dem Befehlsstring heraus. Mit der voranstehenden VAL-Funktion wird dieser String dann in einen Zahlenwert umgewandelt. Die Variable EN, die das Ende des auszulesenden Speicherbereichs enthält, wird in diesem Fall 112 höher angenommen, was damit die Ausgabe von insgesamt 15 Zeilen auf dem Bildschirm beziehungsweise dem Drucker zur Folge hat. Im anderen Fall wird der rechte Teil in unserem Befehlsstring in zwei Hälften zerlegt. Die erste geht vom dritten Zeichen bis zum Semikolon, die zweite Hälfte beginnt nach dem Semikolon. Beide Teilstücke werden in einen Zahlenwert umgewandelt und in den Variablen AN beziehungsweise EN zwischengespeichert. Diese werden dann noch in Zeile 490 und 500 daraufhin überprüft, ob sich durch die Eingabe von Anfangs- und Endpunkt im HEX-Code negative Werte ergeben hatten, und gegebenenfalls korrigiert.

Nachdem nun Anfangs- und Endpunkt des anzuzeigenden Speicherbereichs bekannt sind, beginnt die eigentliche Ausleseroutine. Diese besteht aus einer auf den ersten Blick relativ komplizierten Verschachtelung von Schleifen, aber wie gesagt nur auf den ersten Blick. Die wichtigste Schleife ist die I-Schleife. I läuft von AN bis EN, das heißt vom Anfang des auszugebenden Bereichs bis zu seinem Ende in Schritten von 8. Da jeweils die HEX-Codes von acht Zeichen nebeneinander in einer Bildschirmzeile dargestellt werden sollen, wird I also bei jeder Zeile weitergeschaltet. Innerhalb der Zeile sind nun beim Auslesen des ROMs zwei Abfragen nötig, da die Routine ROMREAD uns ja nur 4 Byte Speicherinhalt zurückgibt. J nimmt die Werte 0 und 1 an. Multipliziert man also J mit 4, so hat man genau einen Versatz um 4 Byte, der dann zur aktuellen Position (I) addiert, für J=0, die ersten 4 Byte der darzustellenden Adressen und für J=1 die nächsten 4 Byte angibt.

Es folgt eine Verzweigung in Abhängigkeit von der Variablen ROM. Falls das ROM ausgelesen werden soll, geht es bei Zeile 550 weiter. Nach dem uns schon bekannten Prinzip werden die Variablen S1 und S2 bestimmt und an die entsprechende Stelle im Maschinenprogramm gepoket. Danach folgt der Aufruf von ROMREAD, wonach die 4 auszugebenden Byte in den Speicherstellen 40050-40053 abgelegt wurden, und danach geht es weiter nach Zeile 590.

Zeile 570 und 580 behandeln den Fall, daß aus dem RAM gelesen werden soll. Hier kann der Zugriff in BASIC passieren, in die Speicherstellen 40050-40053 wird in Abhängigkeit von k eine Kopie der auszulesenden

RAM-Speicherstelle abgelegt. Dieses Verfahren hat den Vorteil, daß unabhängig davon, ob RAM oder ROM ausgelesen werden soll, immer der Inhalt in 40050-40053 abgelegt ist.

Bei der nun, das heißt ab Zeilennummer 590, folgenden Darstellung dieser Speicherstellen ist es daher nicht mehr notwendig, zwischen dem Auslesen von RAM und ROM zu unterscheiden. Die Zeilen 600 und 610 bauen zwei Strings auf. z$ enthält dabei die aneinandergereihten HEX-Codes der Speicherstellen und y$ ihren grafischen Inhalt. Nach dem zweiten Durchlauf von J sind 8 Adressen ausgelesen und aneinandergefügt worden. Die Zeile 630 addiert nun die beiden Strings für Grafik- und HEX-Werte und fügt am Anfang noch die aktuelle Speicheradresse im HEX-Code an. Dieser Gesamtstring wird dann auf dem gewählten Ausgabegerät ausgegeben. Die Multiplikation mit -8 leitet sich daraus ab, daß die Variable STREAM die Werte 0 und -1 annimmt. Bei -1 soll der Printer angesprochen werden, der ja bekanntlich auf Ausgangkanal 8 liegt, woraus sich die notwendige Multiplikation mit -8 ergibt. Zeile 640 löscht die beiden Stringvariablen wieder, und mit 650 beginnt der Aufbau der nächsten Zeile.

Als nächste Funktion im Programmverlauf stoßen wir auf ROMCHANGE. Diese Funktion hat keine andere Aufgabe, als die Variable ROM zu invertieren und dementsprechend dann den Text RAM beziehungsweise ROM im oberen Fenster, dem WINDOW 1, darzustellen. ROM fungiert als Flag für die Speicherabfrage, zum Beispiel beim Kommando 'M'.

Interessanter schon sind die Zeilen 740 und 750. Sie enthalten die **ERROR-HANDLING-Routine.** Damit CPC MON nicht bei jeder Fehleingabe >>aussteigt<<, wurde diese Fehlerbehandlungsroutine eingefügt. Am Anfang des Programms findet sich der Befehl

```
ON ERROR GOTO 740
```

Tritt nun im Verlauf des Programms ein Fehler auf, so wird diese Routine angesprungen. Sie gibt zwei Fehlermeldungen aus. Wird ein unzulässiges Verschieben der Speicherobergrenze, das heißt von HIMEM, versucht, so würde normalerweise ein

```
MEMORY FULL
```

auf dem Bildschirm erscheinen, und das Programm würde die weitere Bearbeitung verweigern. Durch das ON-ERROR-Kommando wird nun aber zu Zeile 740 verzweigt und dort als erstes geprüft, ob ein Error 7 aufgetreten



ist. Diese Fehlernummer entspricht genau dem Fehlertext MEMORY FULL. Dementsprechend gibt das Programm dann aus, daß der Speicherbereich belegt ist, eine Verschiebung also nicht erfolgen kann, und springt dann zurück in die Befehlsabfrageschleife nach 420. Jeder andere auftretende Fehler führt dagegen zur Ausgabe des Textes "falsche Befehlseingabe" und wiederum zum Rücksprung in die Abfrageschleife. Nach diesem Prinzip lassen sich natürlich auch noch andere Fehler herausfiltern. Dies sind jedoch die beiden wichtigsten Möglichkeiten.

Die nun folgende Routine erklärt sich eigentlich von selbst. Sie ist das Äquivalent zum Befehl **T** Texteingabe. Zunächst wird ein Text in Z$ eingelesen und dieser danach ab einer vorher gewählten Speicherstelle (W) mit den ASCII-Werten Zeichen für Zeichen abgelegt. Damit können Texte als Ausgabewerte oder Daten problemlos in ein Maschinenprogramm eingebaut werden.

Es folgt eine weitere Routine, die eigentlich auch keinerlei Schwierigkeiten bereitet, sofern man sich schon einmal mit der Technik des binären Ladens und Sicherns von Programmen beschäftigt hat. Der CPC stellt dafür ein eigenes Kommando zur Verfügung. Während ein normales

```
SAVE"<Name>"
```

ein BASIC-Programm speichert, wird mit

```
SAVE"<Name>",b,an,l
```

angezeigt, daß es sich um ein binäres FILE handelt, oder: im Klartext, um Maschinenspracheprogramme beziehungsweise -daten. Zur Ausführung dieses SAVE-Befehls braucht der Computer noch den Anfang des abzuspeichernden Datenbereichs und seine Länge. Diese beiden Größen werden über INPUT abgefragt, worauf einer Ausführung des Befehls nichts mehr im Wege steht.

Die nächste Funktion (Ausgabegerät umschalten) können wir überspringen. Hier verhält sich alles genauso wie bei der Funktion R, bloß, daß mit einer anderen Variablen gearbeitet wird und deswegen natürlich auch die Ausgaben im WINDOW 1 geändert werden müssen.

Etwas interessanter ist schon das nächste Unterprogramm. Es dient dazu, ein binäres FILE wieder in den Speicher zurückzuladen. Dazu gibt es zwei

Möglichkeiten. Möglichkeit eins: Man führt einen einfachen Ladebefehl aus, also

LOAD"<NAME>"

Der Computer erkennt dann die Existenz eines binäres FILEs auf der Kassette und lädt dieses ab der alten Anfangsadresse wieder in den Speicher. Es erfolgt also automatisch ein positionsrichtiges Laden. Dieses kann jedoch manchmal unerwünscht sein, zum Beispiel, wenn das Programm mit einem anderen Programm kombiniert werden soll, was auch gerade diesen Speicherbereich einnimmt. Daher ist es durch die Angabe einer Einleseanfangsadresse nach dem Programmnamen auch möglich, das Programm ab einer anderen Speicherstelle zu laden. In diesem Fall reagiert allerdings der CPC auf die Definition des Programmnamens mittels einer Stringvariablen relativ sonderbar. Er findet zwar das gesuchte Programm, lädt es jedoch nicht in den Speicher. Wie man den Computer dennoch dazu bringen kann, den Befehl auszuführen, sehen Sie in Zeile 11050. Man addiert einfach einen Nullstring zu dem zu suchenden Namen, und plötzlich funktioniert es.

Überspringen wir einmal die nächste Funktion, so kommen wir dann zum Äquivalent des C-Befehls, Zahlen konvertieren. Die hier auftauchenden Befehle und Befehlskombinationen haben wir im ersten Kapitel bereits relativ ausgiebig benutzt, so daß wir hierauf nicht näher einzugehen brauchen. Die Ausgabe des in einem beliebigen Format eingegebenen Wertes erfolgt in allen 3 Zahlensystemen.

Auch die nächsten beiden Unterfunktionen stellen uns vor keine größeren Probleme. Die Routine Speicherbereich löschen poket nach Eingabe von Anfang und Ende die dazwischenliegenden Adressen auf 0. HELP besteht im wesentlichen aus einer Aneinanderreihung von PRINT-Statements, die die Befehlsliste ausgeben. Durch ihren Aufruf im Hauptmenü können wir uns eine Übersicht über alle verfügbaren Kommandos ausgeben lassen.

Die letzte Routine, die wir noch betrachten müssen, ist damit Speicherbereiche eingeben, das Äquivalent zum I-Befehl. Zunächst wird der Anfang des einzugebenden Bereichs festgestellt und in W gespeichert. Es folgt eine Abfrage, in der überprüft wird, ob die Eingabe als HEX-Werte oder dezimal erfolgen soll. Je nachdem wird die Variable HEX auf 1 beziehungsweise 0 gesetzt. In Zeile 1800 wird dann Speicherstelle für Speicherstelle der neue Inhalt abgefragt und in den darauffolgenden Zeilen analysiert. Handelte es sich bei der Eingabe um ein X, so erfolgt die Rückkehr in die Befehlsabfrageschleife nach 420. Bei der Eingabe von HEX-Codes wird zu



dem eingelesenen Inhalt der Speicherstelle das & addiert und der Wert des dann entstehenden Ausdrucks in die Speicherstelle W gepoket. Ansonsten wird sofort das wertmäßige Äquivalent des Stringausdrucks mit VAL ermittelt und in der adressierten Speicherstelle abgelegt. In beiden Fällen wird W um 1 erhöht und die nächste Speicherstelle abgefragt.

Die nun folgenden drei Routinen haben im wesentlichen Platzhalterfunktionen für die hier später noch einzufügenden Programme.

```
10 ' *************
20 ' ** CPC-MON **
30 ' *************
40 '
50 ' Initialisierung
60 '
70 ON ERROR GOTO 740
80 MODE 1
90 INK 2,0:INK 3,21:WINDOW#0,1,40,4,25:WINDOW#1,1,40,1,3
100 PAPER#1,2:PEN#1,3:CLS:CLS#1
110 MEMORY 39999
120 DATA cd,00,b9,cd,06,b9,47,21,00,00
130 DATA 7e,32,72,9c,23,7e,32,73,9c,23
140 DATA 7e,32,74,9c,23,7e,32,75,9c
150 DATA 78,cd,0c,b9,c9,x
160 i=40000
170 READ a$:IF a$="x" THEN 190
180 POKE i,VAL("&"+a$):i=i+1:GOTO 170
190 romval=40008:romread=40000
200 rom=0:LOCATE#1,34,1:PRINT#1,"RAM"
210 stream=0:LOCATE#1,4,1:PRINT#1,"Schirm"
220 '
230 ' Befehlsabfrageschleife
240 '
250 INPUT b$
260 c$=LOWER$(LEFT$(b$,1))
270 IF c$="m" THEN 440
280 IF c$="r" THEN 710
290 IF c$="k" THEN 1410
300 IF c$="t" THEN 760
310 IF c$="c" THEN 1280
320 IF c$="l" THEN 1080
330 IF c$="s" THEN 890
340 IF c$="a" THEN 1020
350 IF c$="p" THEN 1200
360 IF c$="x" THEN END
370 IF c$="b" THEN 1870
380 IF c$="g" THEN 1920
390 IF c$="d" THEN 1970
400 IF c$="h" THEN 1530
410 IF c$="i" THEN 1740
420 GOTO 250
430 '
440 ' memory anzeigen
450 '
```

```
460 z$="":y$=""
470 w=INSTR(b$,";")
480 IF w=0 THEN an=VAL(MID$(b$,3)):en=an+112 ELSE an=VAL(MID$(b$,3
,w-3)):en=VAL(MID$(b$,w+1))
490 IF en<0 THEN en=en+65536
500 IF an<0 THEN an=an+65536
510 PRINT
520 FOR i=an TO en STEP 8
530 FOR j=0 TO 1
540 IF rom=0 THEN 570
550 s2=INT((i+4*j)/256):s1=i+4*j-256*s2
560 POKE romval,s1:POKE romval+1,s2:CALL romread:GOTO 590
570 FOR k=0 TO 3:POKE 40050+k,PEEK(i+4*j+k)
580 NEXT k
590 FOR k=0 TO 3
600 z$=z$+RIGHT$("0"+HEX$(PEEK(40050+k)),2)+" "
610 IF PEEK(40050+k)<32 THEN y$=y$+CHR$(24)+CHR$(63)+CHR$(24) ELSE
 y$=y$+CHR$(PEEK(40050+k))
620 NEXT k,j
630 PRINT#stream*(-8),RIGHT$("000"+HEX$(i),4)+"  "+z$+" "+y$
640 z$="":y$=""
650 NEXT i
660 PRINT
670 GOTO 420
680 '
690 ' romchange
700 '
710 rom=NOT(rom)
720 LOCATE#1,34,1:IF rom=0 THEN PRINT#1,"RAM" ELSE PRINT#1,"ROM"
730 GOTO 420
740 IF ERR=7 THEN PRINT"Speicherbereich belegt!!":RESUME 420
750 PRINT:PRINT"Falsche Befehlseingabe!!":RESUME 420
760 '
770 'Texteingabe
780 '
790 INPUT"Bitte geben Sie den Text ein (max.255 Zeichen)";z$
800 INPUT"Ab welcher Speicherstelle soll der Text liegen";w
810 IF w<0 THEN w=w+65536
820 FOR i=w TO w+LEN(z$)-1
830 POKE i,ASC(MID$(z$,i-w+1,1))
840 NEXT i
850 GOTO 420
860 '
870 ' Save
880 '
890 INPUT"Erstes Byte des Bereiches";an
900 IF an<0 THEN an=an+65536
910 INPUT"Letztes Byte des Bereiches";en
920 IF en<0 THEN en=en+65536
930 IF en<=an THEN PRINT"falscher Bereich!!"
940 INPUT"Name des Files";n$
950 PRINT"Speed Write 1 j/n"
960 z$=LOWER$(INKEY$):IF z$="j" THEN SPEED WRITE 1 ELSE IF z$="n"
THEN SPEED WRITE 0 ELSE 960
970 SAVE n$,b,an,en-an
980 GOTO 420
990 '
1000 ' Ausgabegeraet umschalten
1010 '
```



```
1020 stream=NOT(stream)
1030 LOCATE#1,4,1:IF stream=0 THEN PRINT#1,"Schirm " ELSE PRINT#1,
"Drucker"
1040 GOTO 420
1050 '
1060 ' File laden
1070 '
1080 PRINT
1090 INPUT"Name des Files";n$
1100 PRINT"Neue Adresse j/n"
1110 z$=LOWER$(INKEY$):IF z$="n" THEN LOAD ""+n$
1120 IF z$<>"j" THEN 1110
1130 INPUT"Ab welcher Adresse laden";w
1140 IF w<0 THEN w=w+65536
1150 LOAD ""+n$,w
1160 GOTO 420
1170 '
1180 ' Programmobergrenze festlegen
1190 '
1200 PRINT:PRINT"Alte Programmobergrenze";HIMEM
1210 INPUT"Neue Programmobergrenze (HIMEM)";w
1220 IF w<0 THEN w=w+65536
1230 MEMORY w
1240 PRINT:GOTO 420
1250 '
1260 ' Zahlen konvertieren
1270 '
1280 PRINT:INPUT"Zu konvertierende Zahl";w
1290 IF w<0 THEN w=w+65536
1300 IF w>255 THEN 1340
1310 PRINT"  dezimal   hexa        dual"
1320 PRINT"    "+RIGHT$("   "+MID$(STR$(w),2),3)+"       "+RIGHT$("
0"+HEX$(w),2)+"       "+RIGHT$("0000000"+BIN$(w),8)
1330 PRINT:GOTO 420
1340 PRINT"  dezimal     hexa          dual"
1350 PRINT"    "+RIGHT$("     "+MID$(STR$(w),2),5)+"       "+RIGHT$(
"000"+HEX$(w),4)+"    "+RIGHT$("0000000000000000"+BIN$(w),16)
1360 PRINT
1370 GOTO 420
1380 '
1390 ' Speicherbereich loeschen
1400 '
1410 INPUT"Erstes zu loeschendes Byte";an
1420 INPUT"Letztes zu loeschendes Byte";en
1430 IF an<0 THEN an=an+65536
1440 IF en<0 THEN en=en+65536
1450 PRINT"Loeschen von";an;"bis";en;"j/n"
1460 z$=LOWER$(INKEY$)
1470 IF z$="n" THEN 420
1480 IF z$<>"j" THEN 1460
1490 FOR i=an TO en:POKE i,0:NEXT i:GOTO 420
1500 '
1510 ' Help
1520 '
```

```
1530 CLS:PRINT"Befehlsvorrat:":PRINT
1540 PRINT"A Ausgabegeraet Drucker/Schirm"
1550 PRINT"B Blocktransfer"
1560 PRINT"C Zahlensysteme konvertieren"
1570 PRINT"D Programm disassemblieren"
1580 PRINT"G Programm anspringen"
1590 PRINT"H Hilfsliste ausgeben"
1600 PRINT"I Speicherstellen eingeben"
1610 PRINT"K Speicherbereiche loeschen"
1620 PRINT"L Speicherbereich laden"
1630 PRINT"M Speicherbereich darstellen"
1640 PRINT"P Programmobergrenze festlegen"
1650 PRINT"R ROM/RAM-Umschaltung"
1660 PRINT"S Speicherbereich sichern"
1670 PRINT"T Text eingeben"
1680 PRINT"X CPCMON verlassen"
1690 PRINT
1700 GOTO 420
1710 '
1720 ' Speicherbereiche eingeben
1730 '
1740 INPUT"Ab welcher Speicherstelle";w
1750 IF w<0 THEN w=w+65536
1760 PRINT CHR$(24)+"h"+CHR$(24)+"ex oder "+CHR$(24)+"d"+CHR$(24)+
"ezimal ?"
1770 z$=LOWER$(INKEY$):IF z$="h" THEN hex=1 ELSE IF z$="d" THEN he
x=0 ELSE 1770
1780 PRINT"Ende mit 'x'"
1790 PRINT
1800 PRINT"Inhalt Speicherstelle";w;:INPUT n$
1810 IF LOWER$(n$)="x" THEN PRINT:GOTO 420
1820 IF hex=1 THEN POKE w,VAL("&"+n$):w=w+1:GOTO 1800
1830 POKE w,VAL(n$):w=w+1:GOTO 1800
1840 '
1850 ' Blocktransfer
1860 '
1870 PRINT"Blocktransferroutine fehlt noch!!"
1880 GOTO 420
1890 '
1900 ' Maschinenprogramm anspringen
1910 '
1920 PRINT"Programmansprung fehlt noch!!"
1930 GOTO 420
1940 '
1950 ' Programm disassemblieren
1960 '
1970 PRINT"Disassembler fehlt noch!!"
1980 GOTO 420
```



# 3 Die Systemhardware

**Ein-Blick in den grauen Kasten**

Mit den Hilfsroutinen aus den letzten beiden Kapiteln haben wir uns ein umfangreiches Werkzeug erarbeitet, mit dem es uns möglich ist, unseren Computer näher zu erforschen. Schauen wir uns das Innenleben unserer Maschine ein wenig näher an. Als erstes wollen wir dazu die Systemhardware betrachten, das heißt die einzelnen Bausteine, aus denen unsere Maschine aufgebaut ist und ihre Funktionen im Zusammenwirken miteinander.

Bei der Analyse der Systemhardware gibt es im wesentlichen zwei Möglichkeiten:

**Möglichkeit 1:** Man beschäftigt sich mit den einzelnen integrierten Schaltkreisen, analysiert die Bedeutung der einzelnen Anschlüsse und gelangt schließlich über die Analyse der verschiedenen Datenwege zu einem Gesamtverständnis des Systems.

**Möglichkeit 2:** Man geht von einem Funktionsschaltbild der Datenwege aus und schaut sich darauffolgend die Bedeutung der Chips im Rahmen des Systems an.

Bei unserem Streifzug durch das System wollen wir eine Mischung aus diesen beiden Varianten wählen. Ausgangspunkt dazu wird ein Funktionsplan sein, der das Zusammenwirken der einzelnen Bausteine wiedergibt. Darauffolgend analysieren wir den Zusammenhang, das Zusammenwirken und die Verbindung der einzelnen Schaltkreise miteinander, und dort, wo es interessant ist, steigen wir auch einmal bis zur Funktion und Bedeutung der einzelnen PINS der ICs in das Systeminnere hinab.

## 3.1 Übersicht über das Gesamtsystem

Beginnen wir also mit einer Grobübersicht. Dazu haben Sie wiederum zwei Möglichkeiten:

**Möglichkeit 1:** Sie schrauben Ihren Computer einmal auf. Dies ist nach Herausdrehen der sechs Kreuzschlitzschrauben, die teilweise etwas versenkt im Gehäuseboden des Tastaturteils liegen, relativ leicht möglich, und schauen sich das Ganze in natura an. Die Steckverbindungen zwischen Tastaturteil und Monitor sollten Sie allerdings davor abgezogen haben. Beachten Sie bitte auch, daß damit gegebenenfalls die Garantie erloschen ist!

**Möglichkeit 2:** Sie beschäftigen sich nur mit dem Photo auf der nächsten Seite, das genau die Hauptplatine wiedergibt.

Wer schon einmal einen anderen Computer von innen gesehen hat, wird erstaunt sein, mit wie wenig Bausteinen hier doch relativ viel erreicht wurde. Das Schlagwort hierzu heißt Hoch- beziehungsweise sogar Höchstintegration, das heißt Komprimierung von sehr vielen Schaltfunktionen in einem einzelnen Baustein. Wir können uns daher auch bei unserer Analyse im wesentlichen auf die größeren Bausteine beschränken, die kleineren führen nur Hilfsfunktionen aus. Um Ihnen eine bessere Übersicht zu geben, haben wir die einzelnen Bausteine beschriftet. Im Blockschaltbild auf der nächsten Seite (Bild 3.2) finden Sie dann das Zusammenwirken der einzelnen Bausteine grafisch dargestellt. Dieses wird Basis für unsere weiteren Erörterungen sein.



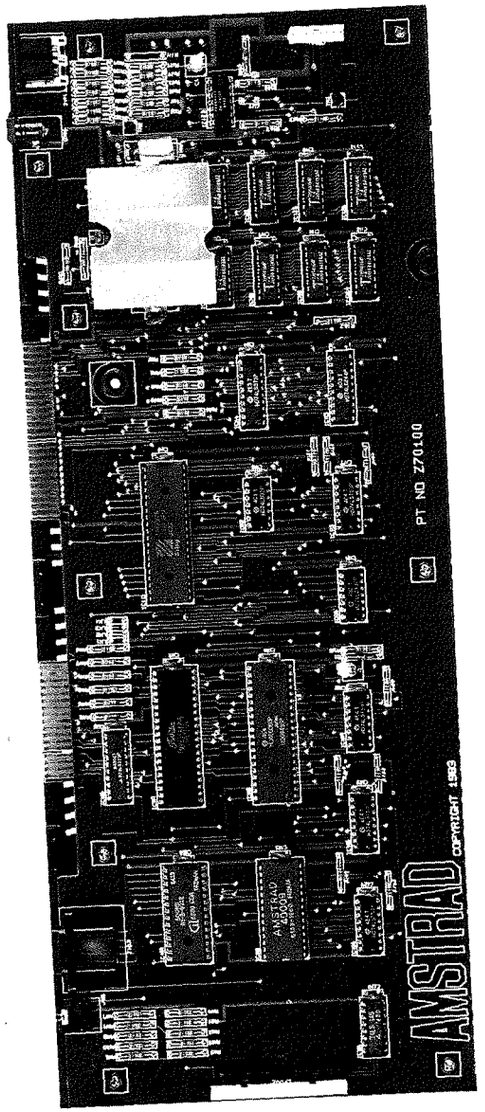



Bild 3.1: Die Hauptplatine

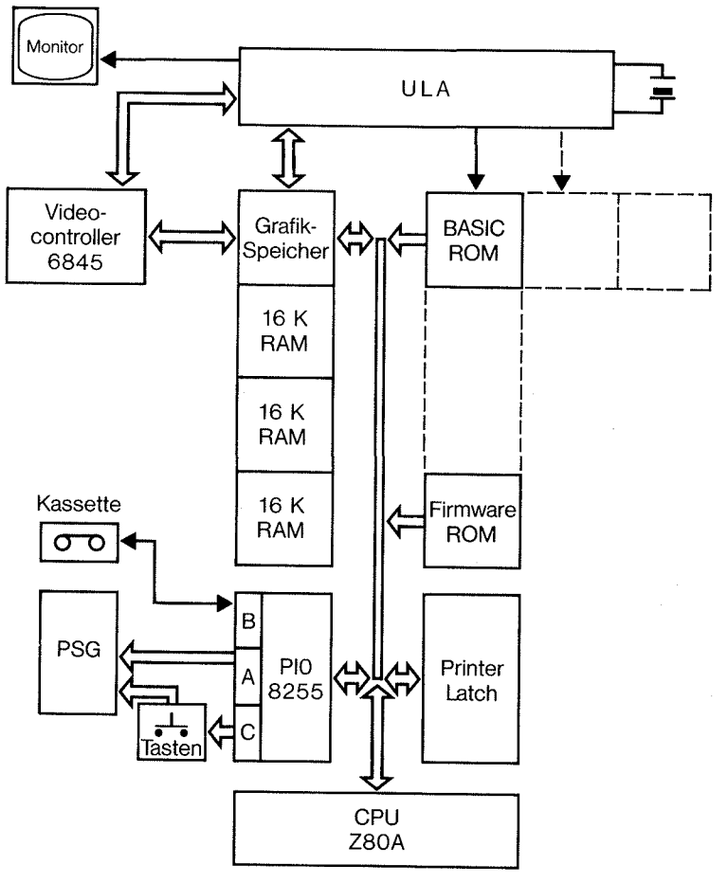



Bild 3.2: Blockschaltbild des Systems



### 3.1.1 Die Aufgaben des Prozessors im System

Herzstück unseres Systems ist ein Z80A-Prozessor, auch als CPU (Central Processing Unit = zentrale Prozesseinheit) bezeichnet. Diese stellt das Gehirn unseres Computers dar. Ihre Aufgabe besteht in der Kontrolle und Ausführung des internen und externen Datentransfers und in der Durchführung arithmetischer und logischer Funktionen. Dabei bietet der Prozessor so vielfältige und auch komplexe Möglichkeiten, daß ihm ein eigenes Kapitel, das nächstfolgende, gewidmet ist. Hier soll nur seine Stellung im Gesamtsystem und das Zusammenwirken mit anderen Bausteinen erklärt werden.

Zum Durchführen seiner Aufgaben benötigt ein Prozessor zunächst einmal Anweisungen, ein Maschinenprogramm. Dieses Maschinenprogramm muß von einem externen Speicher vorgegeben werden und wenigstens teilweise dort permanent verfügbar, das heißt resident, sein. Zumindest muß aber ein externer Speicher ein Ladeprogramm, einen sogenannten Urlader, enthalten, mit dem es möglich ist, ein gewünschtes Maschinenspracheprogramm einzuladen, das danach ausgeführt wird.

Neben dem Maschinenprogramm benötigt der Prozessor noch Bausteine, von denen er Daten empfangen kann, beziehungsweise an die er Daten senden kann. Die CPU unterscheidet dabei zwischen zwei Arten von Bausteinen, MEMORY-und I/O-Bausteine. Bei den MEMORY-Bausteinen handelt es sich, vereinfacht gesagt, um Speicher, also um RAM und ROM. Alle anderen Bausteine werden als I/O angesprochen. Zur Kommunikation mit seiner Umwelt verfügt der Prozessor über drei Gruppen von Leitungen:

**ADRESSBUS** - Er besteht aus 16 Leitungen (=16 Bit) und dient zur Adressierung, das heißt Auswahl eines bestimmten I/O-Bausteins beziehungsweise einer Adresse im Speicher. Hierbei handelt es sich um reine Ausgabeleitungen. Die 16 Adreßleitungen haben Nummern von 0-15. A0 stellt dabei das niedrigstwertige Bit dar, A15 das höchstwertige.

**DATENBUS**

- Er besteht aus 8 Leitungen (=8 Bit) und kann wahlweise auf Eingabe, das heißt den Empfang von Daten, oder Ausgabe programmiert werden. Dies geschieht jeweils prozessorintern bei der Ausführung eines Maschinenbefehls. Beim Z80A werden immer 8 Bit (=1 Byte) parallel eingegeben beziehungsweise ausgegeben. Der Datenbus trägt Nummern von 0-7. Die 0 stellt das niedrigstwertige, die 7 das höchstwertige Bit dar.

**STEUERBUS**

- Dieser enthält all die Leitungen, die für die Koordination mit den einzelnen externen Bausteinen notwendig sind. Im einzelnen sind dies:

**MREQ - MEMORY REQUEST (Speicheranfrage):**
Diese Leitung gibt an, daß eine Anfrage an einen Speicher, also an einen MEMORY-Baustein vorliegt.

**IORQ - IO REQUEST (E/A-Anfrage):**
Auf dieser Leitung zeigt der Prozessor an, daß ein I/O-Baustein angesprochen werden soll.

**RD - READ (Lesen von externem Baustein):**
Dies ist ein Lesesignal,das heißt, es sollen Daten durch den Prozessor empfangen werden.

**WR - WRITE (Schreiben an externen Baustein):**
Auf dieser Leitung gibt der Prozessor an, wenn er an einen I/O Baustein oder einen Speicher Daten aussenden will.

**IRQ - INTERRUPT REQUEST (Unterbrechungsanfrage):**
Auf dieser Leitung teilt ein externes Gerät dem Prozessor mit, daß er sein momentanes Programm abrechnen soll, um zum Beispiel Daten von einem jetzt sendebereiten Gerät zu übernehmen.



**WAIT (Bitte warten!):**
Dieser Eingang des Prozessors wird benutzt, um die Z80A mit langsameren ICs zu synchronisieren.Ein externer Baustein kann durch Setzen dieser Leitung den Prozessor zwingen, so lange mit dem weiteren Abarbeiten des Maschinenprogramms zu warten, bis er die bereitgestellten Daten empfangen kann. Beim CPC wird diese Leitung benutzt, um die Koordination zwischen Videocontroller und Z80A sicherzustellen. Beide ICs können nämlich auf den Grafikspeicher zurückgreifen, die Z80, um Bildinformationen zu schreiben, und der Videocontroller, um diese zu lesen und in das Signal für den Monitor umzuwandeln. Da ein gleichzeitiger Zugriff unvorhergesehene Konsequenzen mit sich bringen würde, wird der Prozessor in diesem Fall angehalten.

Neben diesen Leitungen verfügt der Prozessor noch über einige andere Steuerleitungen, die aber für das Verständnis des Systems nur untergeordnete Bedeutung haben, so daß wir hier auf sie verzichten können.

Durch geeignete Verknüpfung der Steuersignale MREQ, IORQ, RD und WR können wir kombinierte Signale erzeugen, die dann, zum Beispiel wenn der Prozessor einen Speicherbaustein lesen will (MREQ + RD), aktiv werden und so die gewünschte Information an das MEMORY weitergeben. Der prinzipielle Ablauf eines Speicherzugriffs würde dann wie folgt aussehen:

**Beispiel:**
Lesen der Speicherstelle C000: Die CPU setzt A14 und A15 auf 1, die anderen Adreßleitungen auf 0. Wenn Ihnen nicht klar ist warum, sollten Sie sich einmal mit der Funktion C des Monitors oder einem der Konvertierungsprogramme aus Kapitel 1 den Binärwert von hex C000 ausdrucken lassen und dann von rechts nach links beginnend mit 0 durchzählen.

Gleichzeitig wird der Datenbus auf Eingabe geschaltet, und es werden die beiden Steuerleitungen MREQ und RD auf Aktivpegel gesetzt. Beim nächsten Takt steht dann die Information aus dem Speicher auf dem Datenbus, und der Prozessor kann sie hereinlesen und gegebenenfalls weiterverarbeiten.

Analog zu diesen Operationen läuft auch die Ansprache eines I/O-Bausteins (Soundgenerator, Schnittstellenschaltkreise etc.) ab, nur daß hier die Steuersignale natürlich verschieden sind. In diesem Fall werden die Leitungen IORQ und RD aktiv. Und ein weiterer Unterschied: auch die Adressierung erfolgt auf andere Art und Weise.

Dazu schauen wir zunächst noch einmal auf den Speicherbereich zurück. Die **Ansprache eines Speicherbausteins** verläuft relativ einfach. Die benötigte 16-Bit-Adresse kommt entweder aus einem Registerpaar oder ist im Maschinenprogramm selber mit angegeben.

In der Routine ROMREAD aus Kapitel 2 hatten wir sie im Registerpaar HL gespeichert. Bei dem nachfolgenden LD A, (HL) war dann der Inhalt von HL auf den Adreßbus gesetzt worden, die Information der entsprechenden Speicherstelle wurde vom ROM auf den Datenbus gesetzt und danach in das Register A der CPU eingelesen.

Bei unserer Beschreibung haben wir allerdings noch eines unterschlagen, die Auswahl zwischen ROM und RAM. Dies geschieht durch zwei Freigabeleitungen, die durch einen Gatterbaustein, das ULA, je nach gewünschter Betriebsart gesetzt werden. Hat der Prozessor also bis jetzt auf das ROM zugegriffen, so muß er beim Lesen aus dem oberen RAM (hex C000 nach oben) zunächst einem anderen Baustein, dem ULA mitteilen, daß ab jetzt aus dem RAM gelesen werden soll und kann danach den bereits beschriebenen Befehlsablauf für das Einlesen von Informationen durchführen. Das ULA wird dabei als I/O-Baustein angesprochen.

Normalerweise beschränkt man sich bei Z80-Systemen auf die einfache **Adressierung von I/O-Bausteinen.** Da im Regelfall nicht mehr als 256 Eingabe- und Ausgabegeräte benötigt werden, ordnet man jedem Gerät eine Adresse, die sich aus den unteren 8 Adreßleitungen ergibt, zu.

Beim CPC läuft jedoch alles anders. Hier werden die oberen 8 Adressbits (A8 bis A15) für die Analyse des angesprochenen Gerätes benutzt. Die Decodierung geschieht dabei jedoch, wie auch bei den meisten anderen Homecomputern, nur unvollständig. Es werden also nicht alle Zahlen decodiert (was einen erheblichen Aufwand erfordern würde), sondern bestimmte Bausteine sind direkt an ausgewählte Adreßleitungen angeschlossen. Dadurch wird die Zahl der unterscheidbaren Geräte erheblich kleiner, und außerdem führt es dazu, daß einige Geräte auf mehreren I/O-Adressen zu finden sind. Auf die genaue Zuordnung der einzelnen Adressen zu den Bausteinen kommen wir noch bei der Behandlung der einzelnen ICs zurück.



Anhand der Unterscheidung des Prozessors zwischen Ein/Ausgabe- und Speicherbausteinen können wir eine Dreiteilung der Baugruppen unseres Computers vornehmen:

- Im oberen Teil des Blockschaltbildes finden wir den Speicherbereich, er besteht aus RAM, ROM, VIDEOCONTROLLER und als Systemkoordinator dem ULA.

- In der Mitte treffen wir auf den IO-Bereich. Er dient im wesentlichen der Kommunikation mit externen Geräten, also Kassette, Lautsprecher, Drucker, Floppy und natürlich der Tastatur.

- Übrig bleibt, am unteren Bildrand, als dritte Baugruppe das Gehirn des Rechners, die CPU. Wir wollen uns nun mit den einzelnen Abteilungen etwas näher beschäftigen.

## 3.2 Der MEMORY-Bereich

**Aufgaben:**

- Abspeicherung von Daten und Programmen
- Lieferung von Daten und Maschinenprogrammen an die CPU
- Ablage der Bildinformationen und Bilddarstellungen

**Bausteine:**

- **8 dynamische RAMs;** 64K x 1 Bit parallelgeschaltet als 64K x 1 Byte Speicher

- **1 ROM** mit 32K Bit Speicherumfang unterteilt in zwei Speicherbereiche. Die Adressen 0000-3FFF enthalten das Betriebssystem; also jene Routinen, die allgemein von jeder Hochsprache benötigt werden für die Tastaturabfrage, die Ausgabe eines Zeichens auf dem Bildschirm, das Ziehen von Linien, die Integerarithmetik und die Speicherverwaltung etc. Im oberen ROM (von C000-FFFF hex) findet sich der BASIC-Interpreter, also die Routinen, die zur Übersetzung des Hochsprachenbasics im Maschinencode notwendig sind.

- **1 Videocontroller 6845**. Dieser auch CRTC (Cathode Ray Tube Controller = Kathodenstrahlröhrencontroller) genannte Baustein liest aus einem frei wählbaren 16K-Bereich des RAM Byte für Byte die Bildinformation aus und übersetzt diese in das Bildsignal für den Monitor. Dabei ist der Videocontroller in weiten Bereichen durch Software programmierbar. So können zum Beispiel die Anzahl der Zeichen pro Zeile oder ähnliche, für den Bildaufbau notwendige, Daten softwaremäßig gesteuert werden. Dies geschieht über das Schreiben in interne Register des Videocontrollers. Der Benutzer braucht sich mit diesem Register nicht auseinanderzusetzen, weil sie vom Computer bei der Initialisierung auf die notwendigen Werte gesetzt werden und eine Änderung meistens nicht sinnvoll ist. Eine Ausnahme bilden die Register 12 und 13 (beziehungsweise 16 und 17), welche die Bildschirmsteueradresse respektive ein Light-Pen-Register enthalten. Auf ihre Benutzung kommen wir gleich noch zurück.

- **1 Gate Array** (Ferranti ULA). Dieser Baustein erfüllt so vielfältige Aufgaben, daß ihm im Rahmen des Systems fast die Rolle eines Hilfsprozessors neben der eigentlichen CPU zukommt. Technisch gesehen handelt es sich beim ULA um eine Matrix aus Widerständen, Kondensatoren und Transistoren (insgesamt mehr als 2200 Einzelkomponenten), die mit Hilfe des technischen Verfahrens der Maskenprogrammierung miteinander verbunden werden können. Das Kürzel ULA steht dabei für Uncomitted Logic Array, also unverbundene logische Gatter. Durch das Ziehen der Verbindungen ist es für einen Computerhersteller ohne größere Schwierigkeiten möglich, auf einem Baustein zum Teil sehr verschiedenartige Funktionen zu integieren.

  Dies ist auch hier der Fall. Hieraus resultiert nun, daß man beim CPC neben hochentwickelten Standardbausteinen und einigen wenigen Hilfschips keine nichtintegrierten Bauteile findet: alles ist im ULA zusammengefaßt.

**Das Array** nimmt beim CPC im wesentlichen drei verschiedene Aufgaben wahr. In Zusammenarbeit mit dem CRTC-Controller gibt es das Bildsignal aus. Der Gatterbaustein ist dabei für die Auswahl der abzubildenden Farbe eines Punktes zuständig und stellt überdies die unterste Adreßleitung zur Ansprache des Bildschirmspeichers zur Verfügung. Der CRTC-Controller



adressiert immer zwei Byte gleichzeitig. Welches Byte dann am Schluß bearbeitet wird, legt das ULA fest. Daneben enthält das GATE-ARRAY eine Reihe von Registern, die für die Zwischenspeicherung der aktuellen Kombination von INKs, also der verwendeten Farben, benutzt werden und mit deren Hilfe der Baustein die aktuelle auszugebende Farbkombination für einen Bildpunkt bestimmt.

Die **zweite Aufgabe des ULA** besteht in der Auswahl von Bildschirmmodus und Speicherbereichsverteilung. Dazu verfügt das ULA über ein weiteres Register, MODE-Register genannt, in dem einzelne Bits als Kennzeichen für die Speicherung des Bildschirmzustands benutzt werden. Der Bildschirmmodus ist für das ULA eine Kenngröße, anhand der es entscheidet, ob ein geliefertes Datenbyte als komplexe Farbinformation oder mehrere Bildschirmpunkte zu interpretieren ist. Mehr dazu im nächsten Abschnitt.

Der **dritte wichtige Aufgabenbereich des ULA** ist die zeitliche Koordination der einzelnen Bausteine. Das ULA erzeugt mit Hilfe einiger Gatter und eines Quarzes den Hauptsystemtakt von 16 Megaherz und liefert so fast alle anderen vom System benötigten Signale, so zum Beispiel den Prozessorsatz mit 4 Megaherz, die Takte für den Videocontroller und den Soundgenerator mit 1 Megaherz. Daneben liefert das ULA den Hauptinterrupttakt, eine Unterbrechung, die 300mal pro Sekunde auftritt und im Betriebssystem für den Aufruf ständig wiederkehrender schneller Ereignisse benutzt wird. Eine nochmalige Teilung durch 6 liefert dann den langsamen Unterbrechungstakt, ein Rhythmus, der für den Aufruf von langsamen Ereignissen, wie zum Beispiel der Tastaturabfrage, benutzt wird.

Beschäftigen wir uns nun etwas näher mit verschiedenen Aufgaben des Memorybereiches, speziell mit der Bildschirmdarstellung und der Speicherfreischaltung.

### 3.2.1 Die Bildschirmdarstellung

Der Bildschirmspeicher wird durch zwei Zeiger grob untergliedert, die Bildschirmbasis und den Bildschirmoffset.

Die **Bildschirmbasis** gibt an, in welchem 16K-Bereich des RAMs sich der Bildschirmspeicher befindet. Da im unteren Teil unseres RAMs die RESTART-Anweisungen liegen, und der Bereich zwischen 8000 und C000 den Firmwaresprungverteiler enthält, sind diese beiden Bereiche nicht für die Ablage von Bildinformationen geeignet. Was übrigbleibt, ist der Bereich zwischen hex 4000 und hex 7FFF beziehungsweise zwischen hex C000 und hex FFFF.

Bei der Initialisierung definiert der CPC den Bereich zwischen C000 und FFFF für die Ablage der Bildinformationen.

Um zwischen den einzelnen 16K-Bereichen umschalten zu können, stellt das Betriebssystem eine Routine zur Verfügung, SCR SET BASE mit der Adresse hex BC08. Bei dieser Routine muß der AKKUMULATOR das signifikante Byte, also das High-Byte der Anfangsadresse des Grafikspeichers, enthalten. Da die Umschaltung nur in Blöcken zu 16K erfolgen kann, sind dabei nur die oberen zwei Bits (Bit 6 und Bit 7) relevant. Sie definieren, in welchem der vier möglichen Speicherbereiche der Bildschirmspeicher nun liegt.

Mit einer weiteren Routine **SCR SET OFFSET** mit der HEX-Adresse BC05 ist es möglich zu bestimmen, ab welcher Position innerhalb dieses 16K-Bereiches der Bildschirm beginnt. Oder um es einfacher auszudrücken: Mit dieser Adresse wird der oberste linke Eckpunkt unseres Bildschirms festgelegt. Wir haben es also beim CPC nicht mit einem festen Speicher zu tun, bei dem Bildpunkte genau definierten Speicherstellen zugeordnet werden können. Sondern, je nach Bedeutung dieses **OFFSET**-Zeigers gibt eine Speicherstelle im Bildschirm-RAM eine andere Position auf dem Bildschirm wieder. Um uns die Wirkungsweise dieser beiden Routinen klarzumachen gibt es nur eines: ausprobieren. Dies wollen wir nun auch tun.

Zunächst einmal zur **Wirkungsweise der Routine SCR SETBASE.** Wir haben schon gesehen, daß der CRTC-Controller Über zwei Register verfügt, die ihm angeben, welche Positon im Speicher den oberen linken Bildpunkt darstellt. Dieser Zeiger erfordert 16 Bit und ist deswegen in zwei Registern abgelegt, den Registern 12 und 13 des Videochips. 12 beinhaltet dabei die höherwertigen Bits der Startadresse, 13 die niederwertigen.



Nun werden für die Auswahl eines Bytes aus einem 16K-Bereich nur 14 Bit benutzt, so daß 2 Bit für die Auswahl des entsprechenden 16K-Speicherbereichs im Register 12 freibleiben. Diese beiden oberen Bits werden von SCR SETBASE nach dem Willen des Benutzers geändert und schalten damit dann die gewünschten Speicherbereiche ein. Bei der Auswahl der Speicherbereiche ist jedoch Vorsicht oberstes Gebot, denn nur die Adressen im Bereich zwischen hex 4000 und 7FFF, beziehungsweise C000 und FFFF, sind ja disponibel.

Obwohl derart weitreichende Änderungen in der Softwarestruktur des Systems anfangs zu einigen Unsicherheiten führen werden, sollten wir dennoch auf gar keinen Fall darauf verzichten, denn sie eröffnen uns ungeahnte Möglichkeiten, zum Beispiel das Arbeiten mit zwei Grafikspeichern, eine beliebte Methode, die des öfteren bei Zeichenprogrammen verwendet wird.

Wir wollen uns das **Grundprinzip** einmal anhand eines Beispiels anschauen: Erster Grafikspeicher soll unser Normalspeicher sein, das heißt der Bereich von C000 nach oben. Als zweiten Nebengrafikspeicher verwenden wir den Bereich von hex 4000 bis 7FFF.

Zunächst einmal sollten wir überprüfen, wo wir überhaupt sind. Dies geht relativ einfach, indem wir den Bildschirm löschen und uns dann eine Speicherstelle in diesem Bereich, zum Beispiel mit dem Wert FF, poken. Nach

```
POKE &D700,&FF
```

erhalten wir ziemlich in der Mitte unseres Bildschirms einen ein Zeichen breiten gelben Strich, eine Bildschirmzeile hoch. Als nächstes sollten Sie den im folgenden abgedruckten kleinen Vierzeiler eintippen. Es handelt sich um ein Maschinenprogramm, das wir wieder ab 42000 nach oben ablegen.

```
10 MEMORY &3FFF
20 DATA 3e,40,cd,08,bc,c9
30 FOR i=42000 TO 42005:READ a$:POKE i,VAL("&"+a$):NEXT
40 CALL 42000
```

Der Inhalt des Programms ist relativ einfach. Zunächst einmal setzen wir die Speicherobergrenze auf 3FFF herab, um Kollisionen mit dem BASIC, speziell den Stringvariablen des BASIC, zu vermeiden. Danach wird das kleine Maschinenprogramm in den Speicherstellen 42000 bis 42005 gepoket und danach das Maschinenprogramm aufgerufen.

Das Maschinenprogramm selbst stellt uns vor keine größeren Probleme, da die Befehle schon weitgehend bekannt sind. 3E ist der Code für das Laden des Akkumulators mit der nachgestellten Zahl, also mit 40 hex. Danach folgt das CD für das Aufrufen eines Maschinenunterprogramms und schließlich mit C9 der Rücksprung ins BASIC. Die Zwei-Byte-Adresse BC08 nimmt dabei den Ansprungpunkt der aufzurufenden Routine an, hier ist dies SCR SET BASE.

Es stellt sich nun die Frage, warum wir den Akkumulator mit 40 geladen haben. Dies ergibt sich relativ schnell, indem wir uns die HEX-Zahl 40 einmal binär ausdrucken lassen.

```
PRINT BIN$(&40)
```

bringt uns als Ergebnis 1000000, also eine siebenstellige Binärzahl. Die linke Vornull wurde bei der Ausgabe unterdrückt. Schauen wir uns die obersten beiden Bits der ausgegebenen Zahl an, so erhalten wir die Werte 01. Dies ist genau die notwendige Adressierung für den zweiten Speicherbereich von unten, das heißt von 4000 bis 7FFF. Umgekehrt müßten wir zum Rückschalten auf den normalen Speicherbereich eine Zahl benutzen, die die obersten beiden Bits gesetzt hat, zum Beispiel F0 oder C0. Lassen wir unser Programm nun laufen, so passiert zunächst überhaupt nichts. Der Bildschirm löscht sich, und es erscheint die READY-Meldung, so als hätten wir ein CLS ausgeführt.

Allerdings, so einfach ist die Sache nicht. Poken wir nämlich jetzt wiederum D700, so passiert nichts. Der Grund: Wir haben ja die Basis unseres Bildschirmspeichers verändert. Die entsprechende Stelle liegt jetzt 32K tiefer bei hex 5700. Poken an diese Speicherposition bringt uns wieder den gewünschten Strich zurück.

Arbeiten wir nun ein wenig mit unserem COPY-Cursor in Zeile 20 und ändern die 40 in F0 ab. Lassen wir nun das Programm wiederum laufen, so erhalten wir unser altes Schirmergebnis vom ersten Programmdurchlauf wieder: Wir verfügen nun über zwei Grafikspeicher.

Durch diese Art der Umschaltung können wir nun vielfältige Effekte erzielen. Der Anwendungsbereich reicht vom Einblenden eines vorher aufgebauten hochauflösenden Grafikbildes bis zur schnelleren Bildschirmausgabe bei größeren Erklärungstexten; überhaupt, jede Art der Ausgabe von größeren Datenmengen auf dem Bildschirm wird nun einfacher.



Da aufgrund des hochauflösenden Grafikspeichers die Bildschirmausgabe beim CPC den größten Zeitaufwand aller Routinen beansprucht, läßt sich mit dem oben beschriebenen Verfahren eine komfortable und zugleich auch schnelle Bildschirmdarstellung erreichen. Allerdings hat diese auch ihren Preis, denn man verliert 16K freien Speicher. So dürfte eine Anwendung dieser Technik doch im wesentlichen bei Grafikentwicklungsprogrammen liegen. Nun ist es allerdings wenig sinnvoll, die Informationen auf dem einen Bildschirm auszugeben und dann auf den anderen umzuschalten. Dies eignet sich nur, wenn man von einer Grafikentwicklungsroutine, die auf den gesamten Bildschirm zurückgreift, in ein größeres Eingabemenü umschalten will.

Viel interessanter ist es dagegen, Daten bereits auf dem einen Bildschirm parat zu halten und dann mit einem kurzen Maschinenprogrammaufruf zwischen den beiden Bildschirmen zu switchen. Die Vorbereitung des Parallelbildschirms erfordert allerdings eine nähere Kenntnis der Abspeicherung der Bildinformationen, da das Setzen des Nebenspeichers bei eingeschaltetem Hauptspeicher nur durch direkten Speicherzugriff möglich ist.

Wir müssen uns daher mit der Speicherstruktur noch etwas näher beschäftigen. Zunächst einmal zur Routine **SCR SET OFFSET**. Wir hatten ihre grundsätzliche Bedeutung bereits kennengelernt. Sie legt den Anfangspunkt des Bildschirms (also die linke obere Ecke) im Speicher fest. Dies geschieht wiederum über die Änderung von Register 12 und Register 13 des CRTC-Controllers. Vor dem Ansprung der Routine müssen wir in das Registerpaar HL den neuen OFFSET einschreiben. Die oberen beiden Bits werden dabei vernachlässigt.

Es gibt jedoch noch eine andere Möglichkeit. Wir können nämlich den Videoprozessor direkt adressieren. Dies geschieht über den BASIC-Befehl OUT, ein Äquivalent des entsprechenden Maschinensprachekommandos. Damit ist es möglich, direkt Daten an ein Ausgabegerät zu senden. Der OUT-Befehl hat das Format

```
OUT<ADRESSE>,<INHALT>
```

Den CRTC können wir über drei Adressen ansprechen. So stellt sich zuerst einmal die Frage, wie man mit drei Adressen eine relativ große Anzahl von Registern (insgesamt verfügt der CRTC über 18 Stück) ansprechen und des weiteren noch eine Trennung von Ein- und Ausgabe bewerkstelligen kann. Dies wird über das sogenannte Adressregister geleistet. Vor der Abfrage

eines internen Registers des Videochips müssen wir zuerst die Nummer des Registers nebst Adressregister ausgeben und können danach dann auf das gewünschte Datenregister zurückgreifen. Das Adressregister hat die Ein-/Ausgabeadresse BCXX, wobei die XX bedeuten, daß diese beiden Stellen nicht dekodiert werden. Ihr Inhalt ist also ohne Belang. Wir werden sie immer durch FF ersetzen.

```
OUT&BCFF,12
```

teilt also dem Videogenerator mit, daß wir auf sein internes Register 12 zurückgreifen wollen, also den oberen Teil der OFFSET-Adresse kombiniert mit der Auswahl des Speicherbereichs (Bildschirmbasis). Unter der Ein-/Ausgabeadresse BDFF können wir Daten an ein Datenregister senden, mit BFFF wird ein Datenregister zur Eingabe adressiert. Fragen wir einmal den momentan gespeicherten OFFSET ab. Dies benötigt vier Kommandos. Zunächst müssen wir das Adressregister auf Register 12 setzen, dann aus diesem mit einem INPUT-Befehl der Umkehrung des OUT-Kommandos Daten einlesen, es folgt die Spezifizierung des nächsten Registers, in unserem Falle Register 13, und wiederum ein INPUT-Befehl. Definieren wir als H und L in Analogie zur Maschinensprache zwei Register, so liefert uns die folgende Zeile den aktuellen OFFSET.

```
50 OUT &BCFF,12:H=INP(&BFFF):OUT &BCFF,13:L=INP(&BFFF)
60 PRINT H,L
```

Hierbei ist zu beachten, daß wir über unsere Abfrage von H nicht feststellen können, welche Basisadresse der CRTC-Controller momentan benutzt. Die obersten beiden Bits sind nämlich bei der Ausgabe immer auf 0 gesetzt. Und noch einen weiteren Nachteil hat dieses Verfahren des Direktzugriffs. Wir teilen der Software, das heißt dem Betriebssystem nämlich nicht mit, daß wir den Speicher verändert haben. Da der OFFSET Rechner-intern für die Berechnung fast jeder Ausgabe auf dem Bildschirm benötigt wird, führen unvorsichtige Änderungen hier zu einer später falschen Bildschirmausgabe.

Für eigene Programmentwicklungen sollte man daher besser auf die Routine SCR SET OFFSET zurückgreifen. Mit dem OUT-Befehl haben wir allerdings eine gute Möglichkeit, um zu demonstrieren, wie der CPC bei nur



einem definierten WINDOW, das heißt WINDOW 0 = Gesamtbildschirm, auf einfache Art und Weise eine Zeile nach oben oder unten scrollt. Er versetzt dazu nämlich einfach den OFFSET-Pointer und löscht ein Teil des Bildschirms.

Zur näheren Erklärung müssen wir uns mit dem Aufbau des Bildschirmspeichers näher beschäftigen. Am besten macht man sich dies mit Hilfe einer kleinen Grafikroutine klar. Versetzen Sie doch einmal den Computer mit

```
<CTRL><SHIFT><ESC>
```

in den Ausgangszustand, und tippen Sie nach CLS die folgende Zeile ein:

```
FOR i=&C000 TO &FFFF:POKE I,&FF:LOCATE 1,1:PRINT
HEX$(i):
NEXT
```

Dieses Programm lädt den Bildschirmspeicher ab Speicherstelle C000 mit dem Wert FF. Wenn Sie sich den Ablauf dabei anschauen, werden Sie schnell die Speicherstruktur erkennen. Das Schreiben beginnt in der linken oberen Ecke, wie nicht anders zu erwarten war, da der OFFSET nach dem Einschalten immer 0 ist. Es werden jetzt rote Linien untereinander auf dem Bildschirm gezeichnet, und zwar für jede HEX-Zeile zuerst die oberste von acht Bildschirmzeilen.

Der Wert in der linken oberen Ecke gibt dabei jeweils die aktuell gepokete Speicherstelle an. Bei dem Wert C800, also 2K oberhalb des Anfangs der ersten Bildschirmzeile, beginnt der CPC mit der Ausgabe der zweiten Bildschirmzeile. Bei 8D00 folgt die nächste usw. (der Abstand beträgt immer 2K).

Jede Bildschirmzeile besteht dabei aus 640 Bildpunkten, so daß wir für ihre Abspeicherung 80 Byte benötigen (= 640 Bit oder aber 640 Bildpunkte). 25 Textzeilen mit jeweils 80 Byte für die Darstellung einer Bildschirmzeile ergeben dann auch einen Block von 2000 Byte, der alle obersten Bildschirmzeilen, beziehungsweise alle zweiten Bildschirmzeilen usw. enthält.

Nun haben wir es aber bei unserem Bildschirmspeicher nicht mit 16000 Byte zu tun, sondern mit 16K. Das wiederum führt dazu, daß nach dem Ende jedes Blocks (welcher ja 2K = 2048 Byte lang ist) 48 Byte frei blei-

ben. Sie können das verfolgen, indem Sie kurz vor dem Sprung von einer Bildschirmzeile zur nächsten auf den Zähler in der linken oberen Ecke schauen. Dieser läuft 48 Byte weiter, allerdings tut sich auf dem Bildschirm nichts. Erst nach Überschreiten der 2K-Grenze geht es links oben weiter.

**Wie sieht es nun mit der Abspeicherung der Punkte innerhalb einer Bildschirmzeile aus?**

Aufgrund des Ineinandergreifens von CPC-Controller und ULA kommt es bei der Farbauswahl zu einer sehr seltsamen Belegung von Speicherstellen, die schon fast an den Turmbau von Babel (jedenfalls der Sprachverwirrung nach) erinnert.

Im MODE 2 ist es relativ einfach. Hier entspricht jedes Bit im Bildschirmspeicher einem gesetzen Bildpunkt. Mit 0 ist dabei der am weitesten rechts liegende Bildpunkt, mit 7 das Gegenstück auf der linken Seite der 8 gespeicherten Bildpunkte je Byte adressiert. Probieren sie einmal nach

CTRL SHIFT ESC

und der Umschaltung auf MODE 2 die folgenden beiden POKEs aus (die linke obere Bildschirmecke sollten Sie davor durch Eingabe von Leerzeichen gelöscht haben):

```
POKE&C000,&01
```

und

```
POKE&C000,128
```

Mit diesen beiden Befehlen werden die beiden äußersten Bildpunkte gesetzt, die restlichen sieben jeweils gelöscht.

Etwas schwieriger gestaltet es sich dagegen, wenn wir in den beiden anderen MODEs operieren möchten. Die höheren farblichen Möglichkeiten werden bei diesen Bildschirmgrößen durch gleichzeitiges Setzen mehrerer Bildpunkte erreicht. So werden zum Beispiel im MODE 1 immer zwei Bildpunkte nebeneinander auf dieselbe Farbe gesetzt. Da man nun nicht mehr für jeden Bildpunkt ein einzelnes Bit zum Zweck der Abspeicherung braucht, können jetzt in zwei Bits die möglichen vier Farben codiert werden. Steht in den beiden Bits 00, so wird auf INK 0 zurückgegriffen, beim Wert 01 wird mit INK 1 geschrieben usw.



Nun wäre es schön und eigentlich auch logisch anzunehmen, daß die Bit 6 und 7 zusammengefaßt Farbinformationen für die beiden linken Bildpunkte liefern. Doch was schön und liebenswert wäre, ist beim CPC noch lange nicht natürlich. Die nachfolgende Tabelle gibt Ihnen die Zuordnung der verschiedenen Bits zu den Bildpunkten an. Die Abfolge der Bits gibt dabei an, in welcher Reihenfolge diese die zu setzende INK codieren.

Beispielsweise liegt im MODE 0 folgendes Byte vor (in binärer Schreibweise): 01010101. Als Farben ergeben sich nun für das linke PIXEL INK 15 und für das rechte PIXEL INK 0. Vollziehen Sie es einmal in Gedanken nach!

Nun zur Umkehrung. Es soll eine grüne Strichellinie auf dem Bildschirm dargestellt werden. Dazu sollen jeweils im MODE 1 vier Bildpunkte gesetzt, danach wieder vier Bildpunkte nicht gesetzt werden. Als Hintergrundfarbe diene INK 0 ; INK 3 sei auf Grün, das heißt 21, definiert worden. Wir gehen einmal davon aus, daß die linken und die rechten zwei Bildpunkte auf Hintergrundfarbe, die mittleren dagegen auf Grün gesetzt werden sollen. Somit ergibt sich als zu pokender Bytewert binär 01100110.

Nun fehlen uns noch die Ausgangsposition und die Länge unserer Linie. Wir nehmen hierbei einmal an, daß wir 20 Zeichen (beginnend mit dem 10. Zeichen) in der Mitte der dritten HEX-Zeile darstellen wollen. Als Mitte fassen wir hierbei die vierte Bildschirmzeile von oben auf.

Jetzt rechnen wir ein wenig. Die vierten Bildschirmzeilen sind im vierten Block abgespeichert, der mit C800 (das dezimal 55296 entspricht) beginnt. Dazu kommen jetzt 160 Byte für die ersten beiden HEX-Zeilen, die wir überspringen müssen, sowie weitere 30 Byte, damit unser 20 Byte langer Strich in der Mitte steht. Als Anfangswert für unsere Linie erhalten wir damit dezimal 55486 und kommen somit zu folgender Programmzeile:

```
INK0,0:INK3,21:FOR i=55486 TO 55506:POKE i,&X01100110
```

Sie sollten nun einmal probieren, die Linie selber auf andere Farbe zu setzen oder eine senkrechte Linie oder ein Rechteck auf dem Bildschirm mit Direktzugriff zu zeichnen. Eines müssen Sie dabei aber immer noch beachten. Das Verschieben von Linien, das heißt das Rollen des Bildschirms um eine Zeile von oben oder nach unten geschieht beim CPC, indem einfach der OFFSET-Pointer verschoben wird, vorausgesetzt, man arbeitet nur mit einem WINDOW, also auf dem Gesamtbildschirm.

Mit jedem SCROLLen verschieben sich also die Anfangskoordinaten des Bildschirms, und damit findet auch eine Ausgabe ihrerseits an einer ganz anderen Stelle statt. Einzige Möglichkeit, dies zu umgehen, ist es, den Bildschirm in zwei Unterbildschirme aufzuteilen. Zum Beispiel könnte man die Zeilen 1-12 und 13-25 als WINDOW#0 bzw. WINDOW#1 definieren. Dadurch wird der CPC gezwungen, beim SCROLLen wirklich Speicherbereiche gegeneinander auszutauschen, beziehungsweise einzelne Bereiche des Speichers mit INK 0 aufzufüllen.

Dies führt zwar bei Bildschirmbewegung zu einem erheblichen Geschwindigkeitsverlust (vergleichen Sie einmal die Ausführungsgeschwindigkeit eines LIST auf dem Gesamtbildschirm mit einem LIST in einem WINDOW - es ist deutlich langsamer, und jetzt wissen Sie, warum), hat dafür aber den Vorteil, daß der Stand unseres OFFSET-Pointers konstant bleibt. Definieren wir uns nach dem Einschalten auf diese Art zwei WINDOWs, so bleibt die linke obere Bildschirmecke immer an der Position C000.

### 3.2.2 Die Freischaltung der Speicherbausteine

Kommen wir nun zu einem anderen interessanten Gebiet im Speicherbereich, der Freischaltung von RAM beziehungsweise ROM. Die Problematik haben wir schon bei der Entwicklung unseres Monitors kennengelernt. RAM-Speicher und ROM belegen teilweise gleiche Adressen.

Es stellt sich die Frage, wer denn nun entscheidet, welcher dieser beiden Bausteingruppen Daten auf dem Datenbus, das heißt zur CPU, geben darf. Dieses ist wiederum das ULA. Es enthält ein Multifunktionsregister, das neben dem Bildschirmmodus auch die aktuelle Auswahl von ROM oder RAM enthält. Nun soll hier nicht so sehr der Zugriff auf das Register im GATE ARRAY interessieren, wichtig ist nur, daß für dieses Register eine genaue Kopie im Registersatz der Z80 vorhanden ist.

Dabei handelt es sich um das Register C des Parallelregistersatzes. Der Aufruf einer Routine, wie zum Beispiel KL U ROM ENABLE, die wir bei ROMREAD verwandt haben, führt nun die folgenden Operationen aus. Zuerst werden die erforderlichen Bits im Parallelregister C geändert und dann dieses an das GATE ARRAY ausgegeben, wo es dann die neue Ver-



teilung von Bildschirmmodi und Speicherabteilung herstellt. Die Bedeutung der einzelnen Bits im Parallelregister C ist dabei wie folgt: Bit 0 und Bit 1 definieren den Bildschirmmodus. Bit 2 und Bit 3 definieren die Speicherkonfiguration. Die Bedeutung der einzelnen Bits bzw. ihrer Kombinationen ist dabei wie folgt:

| Bit 1 | Bit 0 | Mode |
|---|---|---|
| 0 | 0 | Mode 0 (16 Farben) |
| 0 | 1 | Mode 1 (4 Farben) |
| 1 | 0 | Mode 2 (2 Farben) |
| 1 | 1 | Mode 0, die normale Farbdefinition durch Blinken ist abgestellt. |

| | |
|---|---|
| Bit 2 | unteres ROM freischalten<br>0=freigeschaltet 1=gesperrt |
| Bit 3 | oberes ROM freischalten<br>0=freigeschaltet 1=gesperrt |

**Bedeutung der Bits eines Farbbytes in den verschiedenen MODES**

| Pixel | MODE 0 | MODE 1 | MODE 2 |
|---|---|---|---|
| Linkes Pixel | Bit 1,5,3,7 | Bit 3,7 | Bit 7 |
| " | | | Bit 6 |
| " | | Bit 2,6 | Bit 5 |
| " | | | Bit 4 |
| " | Bit 0,4,2,6 | Bit 1,5 | Bit 3 |
| " | | | Bit 2 |
| " | | Bit 0,4 | Bit 1 |
| Rechtes Pixel | | | Bit 0 |

## 3.3 Der IO-Bereich

Ein kurzer Blick auf das Blockschaltbild unseres Computers liefert uns die zugehörigen Baugruppen:

**Ein PRINTER-Port (Centronixport)**

Dies ist ein 8-Bit-Ausgaberegister, das bei Anlegen eines Taktsignals Daten vom Datenbus übernimmt und bis zum Auftreten des nächsten Taktes zwischenspeichert. Des weiteren liefert es ein STROBE-Signal (als Kennzeichen dafür, daß neue Daten eingetroffen sind).

**Ein PIO 8255**

das Arbeitspferd im Eingabe-/Ausgabebereich. Hierbei handelt es sich um einen universellen Ein-/Ausgabebaustein mit drei wahlweise auf Eingabe oder Ausgabe programmierbaren 8-Bit-Datenregistern.

**Ein programmierbarer Soundgenerator (PSG)**

Dieses IC (AY 8912 von General Instruments) enthält drei Tongeneratoren, einen Rauschgenerator und ein programmierbares Ein- bzw. Ausgaberegister. Daneben gehören in diesen Bereich die Kassette und die Tastatur.

Schauen wir uns nun einmal die Ansprache und das Zusammenwirken der einzelnen Bausteine etwas näher an. Am einfachsten ist die Beschreibung des Centronixports. Seine Ausgänge befinden sich direkt auf der mit Printer beschrifteten Steckerleiste. Ihre Belegung finden Sie in Anhang V, Seite 2 des Bedienerhandbuches wieder. Der gesamte Ausgang besteht im wesentlichen aus neun Leitungen.

Im einzelnen sind das sieben Datenleitungen, der STROBE OUTPUT, welcher die Übergabebereitschaft von Daten signalisiert sowie der BUSY INPUT. Mit letzterem teilt das externe Gerät dem CPC mit, daß es noch nicht übernahmebereit ist (womit dann das Aussenden eines weiteren Datenbytes unterbleibt). Dieser Eingang führt in das Register B des PIO, und zwar an Bit 6. Mit der Abfrage von Register B und abschließendem Überprüfen dieses Bit ist es somit möglich festzustellen, ob ein externes Gerät Übernahmebereitschaft signalisiert. Zu diesem Zweck muß der BUSY-Eingang auf LOW, das heißt GROUND, gelegt werden.



Der Centronixport eignet sich relativ gut als universelle Schnittstelle für die Datenausgabe. Die Ansprache geschieht über die Adresse &EFFF. Wie bei den anderen IO-Bausteinen wird auch die Ausgabe an dieses Ein-/Ausgabegerät nur unvollständig decodiert (es wird nur der Zustand der Adreßleitung A12 berücksichtigt). Daher müssen alle anderen Leitungen auf 1 gesetzt sein, um Kollisionen mit anderen Bausteinen zu vermeiden.

Die Ausgabe eines Wertes an den Centronixport ist verhältnismäßig simpel. Sie erfolgt mit dem Kommando

OUT&EFFF,<WERT>

Im Zusammenhang mit der Druckerausgabe muß auf eine unangenehme Eigenschaft hingewiesen werden: der CPC sendet nur 7 Datenbits an den Drucker. Das oberste Bit 7 wird als STROBE-Signal benutzt. Dies bereitet besonders dann Ärger, wenn man mit grafikbefähigten Druckern arbeitet, die die Zeichen von 128-255 als Grafiksymbole interpretieren. Da beim CPC nur die unteren 7 Bit auf D0-D6 weitergegeben werden, ist es ohne Hardwareänderung nicht möglich, diese Grafikfähigkeiten auszunutzen.

Neben der Ansprache mit dem Hauptbefehl können wir jedoch auch auf eine Reihe von Maschinenroutinen zurückgreifen, um Werte an den Port zu senden beziehungsweise seine Empfangsbereitschaft abzutesten. Mehr dazu am Ende dieses Buches.

### 3.3.1 Der Schnittstellenbaustein 8255

Kommen wir nun zum Hauptbaustein im I/O-Bereich, dem 8255, einem sehr vielseitigen IC, das in vielen Z80-Systemen Verwendung gefunden hat. Er verfügt über 24 programmierbare Ein-/Ausgabeleitungen, die in drei Registern mit den Bezeichnungen A, B und C zusamamengefaßt sind. Dabei nimmt das Register C eine Sonderfunktion ein. Es kann nämlich den Ports A und B je zur Hälfte, das heißt mit den vier höherwertigen Leitungen zu Register A und mit den vier niederwertigen zu Kanal B zugeordnet werden; damit übernimmt es die Steuerfunktion. Man spricht dann von den Gruppen A und B. Es können drei wesentliche Betriebsarten durch die Systemsoftware festgelegt werden:

| | |
|---|---|
| **Betriebsart 0:** | einfache Ein-/Ausgabe |
| **Betriebsart 1:** | getaktete Ein-/Ausgabe |
| **Betriebsart 2:** | 2-Wegebus |

Die Betriebsarten der Kanäle A und B können unabhängig voneinander definiert werden, während Kanal C entsprechend den Erfordernissen der Kanäle A und B in zwei Teile aufgeteilt wird. Dabei können die Betriebsarten miteinander kombiniert werden. Es ist also möglich, PORT A in Betriebsart 0 und PORT B in Betriebsart 1 zu betreiben. Dabei ist nun noch zu beachten, daß es in jeder Betriebsart möglich ist, den Haupt-PORT und auch den entsprechenden Teil des Kanals C auf Ein- oder Ausgabe zu programmieren (und zwar unabhängig voneinander).

Eine Einschränkung ist hier allerdings gegeben. Die Gruppe B kann nur in den Betriebsarten 0 und 1 betrieben werden, die Gruppe A auch zusätzlich in der Betriebsart 2. Dieser Umstand wird allerdings sofort klar, wenn wir uns die einzelnen Betriebsarten anschauen.

**Betriebsart 0** stellt dabei den am häufigsten verwendeten Fall dar. Diese Funktionsanordnung ermöglicht eine einfache Ein- und Ausgabe für jeden der drei Kanäle. Ein Austausch von Steuersignalen (HANDSHAKING) ist hier nicht vorgesehen, die Daten werden einfach in den gewählten Kanal geschrieben oder aus ihm gelesen. Durch gleichgerichtete Programmierung von PORT C können somit drei PORTs parallel betrieben werden.

Das ist die beim CPC verwendete Betriebsart. Daneben existieren noch die anderen beiden Betriebsarten. Zuerst einige Bemerkungen zur **Betriebsart 1** (getaktete Ein-/Ausgabe). Hier werden die beiden Gruppen A und B gebildet. Sie bestehen aus jeweils einem 8Bit-Datenregister und den höherwertigen bzw. niederwertigen Leitungen des Registers C, die hier Quittierungssignale erzeugen respektive empfangen.

Die Übernahme von Daten erfolgt hier erst bei der Angabe eines Übernahmesignals auf Pin 4 von Kanal C für Kanal A bzw. Pin 2 für Kanal B. Auch werden an das eingebende Gerät mit Setzen der Leitung 5 (A) beziehungsweise 1 (B) Rückmeldesignale über die Übernahme der übermittelten Werte gegeben. Daneben erzeugen die Pins 0 für Kanal B und 3 für Kanal A noch jeweils ein Unterbrechungssignal, welches bei geeigneter Beschaltung der CPU dem Prozessor mitteilen kann, daß von einem externen Gerät Daten zur Verfügung gestellt wurden.

Im umgekehrten Fall liefert der Chip mittels anderer Pins ein Signal dafür, daß Ausgabedaten bereitstehen; zugleich empfängt er ein Übernahmesignal von dem externen Gerät und gibt danach wiederum ein Unterbrechungssignal an den Prozessor aus. Der letzte Vorgang in diesem Spiel ist also die Mitteilung an die CPU, daß die Daten übernommen wurden.



Noch etwas komplizierter gestaltet sich **Betriebsart 2**. PORT A ist jetzt in der Datenrichtung nicht mehr festgelegt, das heißt, er kann sowohl als Eingabe- wie auch als Ausgaberegister arbeiten. Für das HANDSHAKING (also die Kontrollmitteilung an ein externes Gerät) benötigt PORT A jetzt fünf Leitungen , die übrigen bleiben weiterhin wahlweise auf Ein- oder Ausgabe programmiert. Mit den verbleibenden drei Leitungen kann PORT B natürlich nicht auch noch in Betriebsart 2 betrieben werden (das erklärt unsere obige Restriktion).

Es stellt sich nun die Frage, wie all diese Betriebsarten und Ein-/Ausgabedefinitionen dem Chip mitgeteilt werden können. Dies geschieht über das Aussenden eines Datenwortes in das Steuerregister. In diesem Steuerwort hat jedes Bit eine eigene Funktion; Bild 3.3 schlüsselt die Bedeutung der einzelnen Bits auf.

Wie können wir Daten an den PIO senden, beziehungsweise von ihm empfangen. Wir müssen wissen. Der 8255 verfügt über zwei Adreßleitungen, die angeben, ob der Datenbus mit dem PORT A, B, C oder dem Steuerregister verbunden werden soll. Die Zuordnung ergibt sich dabei aus Bild 3.3.

**Ein-Ausgabe**

| Adresse | Ausgabe | Eingabe |
|---|---|---|
| F4xx | Port A Daten ausgeben | Port A Daten einlesen |
| F5xx | Port B Daten | Port B Daten |
| F6xx | Port C Daten | Port C Daten |
| F7xx | Steuerport schreiben | ---------- |

Wie Sie aus der Tabelle entnehmen können, genügen die unteren 2 Bit des Adreßbus, um die 4 verschiedenen Register zu decodieren. Nun muß nur noch sichergestellt werden, daß nicht zur gleichen Zeit auch noch andere Bausteine auf den Datenbus zugreifen. Dazu verfügt der 8255 noch über einen Chip-Select-Eingang. Legt man an diesen Low-Signal an, so wird der 8255 aktiv.

Der Chip-Select-Eingang ist mit dem Adreßbus Bit 11 verbunden, die Adresseingänge A1 und A0 liegen auf A9 und A8. Eine vollständige Decodierung findet auch hier wiederum nicht statt, so daß die anderen Bits unbedingt auf 1 gesetzt werden müssen, um nicht gleichzeitig eine Ansprache anderer I/O-Geräte, zum Beispiel des CRTC-Controllers oder des GATE ARRAYS, zu erreichen. Gleichzeitig setzt der Prozessor natürlich auch noch bei jeder Ein-/Ausgabeoperation die Signale IORQ, RD und WR, so daß die Ansprache eines I/O-Gerätes und die gewünschte Datenflußrichtung (Ein-/Ausgabe) gekennzeichnet sind.

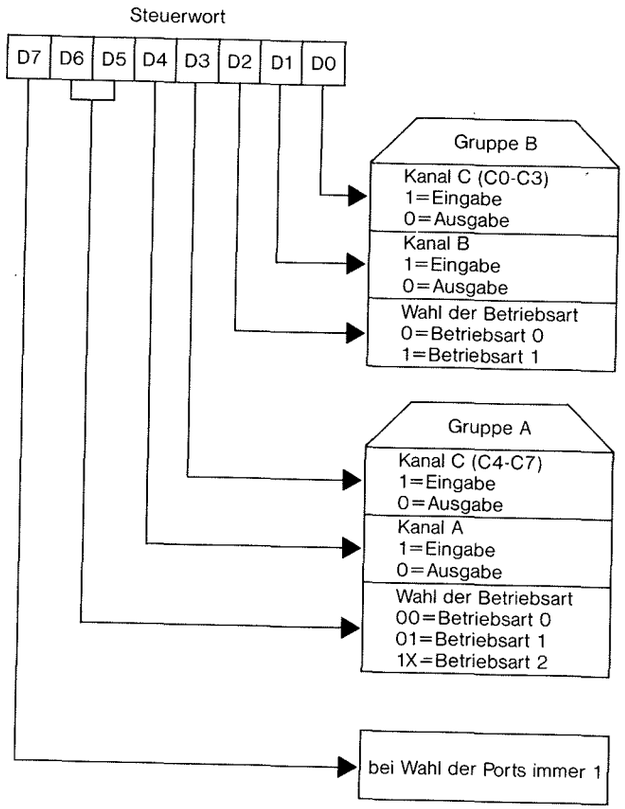



Bild 3.3: Steuerworte und Betriebsarten beim 8255



Wollen wir also nun den Ausgabe-PORT A ansprechen, so ist dies relativ einfach möglich. Wir setzen dazu einfach die Adressbit 11, 9 und 8 auf 0, was als Adresse zum Beispiel &F4FF ergibt (die unteren acht Bit des Adreßbus werden wieder nicht berücksichtigt (wichtig ist also nur die F4 im oberen Teil) und geben an diese Adresse dann unseren Wert aus. Zuvor muß natürlich im Steuerregister der PORT A als Ausgabe-PORT definiert worden sein. Hierzu wäre nur die Adressleitug 11, also der Chip-Select auf 0 zu setzen (A9 und A8 bleiben wegen des Codes des Steuerregisters jeweils auf 1), woraus sich als Adresse &F7FF ergibt.

Bevor wir jedoch weiterhin große geistige Aktivitäten daraufhin verwenden, wie wir die PORTs umdefinieren können, sollten wir uns erst einmal damit beschäftigen, wie der 8255 beim CPC eingesetzt wird.

Das Blockschaltbild zeigt uns, daß das IC mit einer Vielzahl von Baugruppen in Verbindung steht. Zunächst einmal hängt es natürlich am Datenbus, um eine Kommunikation mit der CPU zu ermöglichen. Schauen wir uns nun die einzelnen PORTs an. **PORT B** ist fest als Eingangs-PORT programmiert. Über diesen PORT werden alle Abfragen außer der Tastaturabfrage getätigt. Die einzelnen Bits haben dabei folgende Belegung: **Bit 0-4** sind für den Benutzer von nachrangigem Interesse. Hier ist der Zustand des SYNC-Impulses des CRTC abfragbar, und durch Drahtbrücken wird der gewünschte Firmenname in der Einschaltmeldung fixiert.

**Bit 5** ist mit der EXP-Leitung des Expansion Connectors verbunden. Setzt man P48 des Erweiterungsanschlusses (vergleiche Anhang V Seite 2 des Bedienerhandbuches) auf LOW PEGEL, so kann diese Veränderung durch Testen des Bit 5 in Kanal B festgestellt werden. Damit ist es zum Beispiel möglich, eine serielle Eingabe über den Erweiterungs-PORT laufen zu lassen.

**Bit 6** ist mit der BUSY-Leitung des Druckers verbunden. Ein angeschlossenes Gerät, zum Beispiel der Drucker, kann durch High-Legen dieses Anschlusses der Maschine mitteilen, daß er nicht empfangsbereit ist und sogar ein Zeichentransfer unterbinden. Wir haben uns mit diesen Möglichkeiten schon bei der Behandlung des PRINTER LATCH beschäftigt.

**Bit 7** dient dem Einlesen von Daten vom Recorder.

Kommen wir nun zum **PORT C**, der im CPC fix als Ausgangs-PORT definiert ist. Die unteren vier Bit steuern dabei die Tastaturmatrix. Die Tastatur ist in zehn Reihen mit je acht Spalten angeordnet. Durch Dekodierung der **unteren vier Bit** mit Hilfe eines BCD-Dezimaldekoders wird eine dieser Tastaturreihen auf LOW PEGEL gelegt.

Drückt man nun irgendeine Taste, so wird das LOW-Signal von einer Reihe auf eine Spalte weitergeschaltet; die restlichen Spalten bleiben HIGH. Im Prinzip taktet der CPC jede 50stel Sekunde alle zehn Reihen durch Anlegen verschiedener Werte an diese vier Bit durch und überprüft dann, welche Rückschlußinformationen er auf den Spalten wiederfindet. Dadurch kann er die gedrückten Tasten analysieren.

Mit **Bit 4** wird der Motor des Kassettenrecorders bedient. **Bit 5** stellt die Ausgabeleitung für die Datenübergabe an den Kassettenrecorder dar. Hier wird das Tonfrequenzensignal an den Datacorder weitergegeben. Die Bit 6 und 7 arbeiten als Chip-Select und STROBE-Signal für den Soundship.

**Port A** wird beim CPC sowohl für die Dateneingabe wie auch für die Ausgabe benutzt. Sollen Daten in den Soundprozessor geschrieben werden, so ist Kanal A auf Ausgabe programmiert. Die zu übermittelnden Daten werden in PORT A abgelegt, danach wird mittels Bit 6 und 7 von Kanal B das STROBE-Signal erzeugt, und der Soundchip übernimmt die Daten.

Nun zur Eingabeseite: Der Soundgenerator verfÜgt über einen 8-Bit-PORT, der wahlweise als Eingabe oder Ausgabe programmiert werden kann. Beim CPC ist er auf Eingabe programmiert. Auf diesem Kanal liegen die acht Spalten unserer Tastaturmatrix. Die mit dem Register C angewählte Reihe (vergleiche Kanal C) führt bei Tastendruck zu einem Setzen irgendeines Bits im 8-Bit-PORT des programmierbaren Soundgenerator (PSG). Von dort werden sie dann mit einer Eingabeoperation in das Register A des 8255 eingelesen und gelangen schließlich zur Z80-CPU.

### 3.3.2 Der Soundchip

Bei der Beschäftigung mit der Tastaturabfrage haben wir bereits einen kleinen Teil des Soundgenerators kennengelernt. Aber die Hauptaufgabe eines Soundprozessors liegt natürlich nicht darin, die Tastatur abzufragen. Hier wurde eher ein "Schmutzeffekt" benutzt.



Der Soundgenerator AY-8912 verfügt über drei getrennt programmierbare Tongeneratoren und einen Rauschgenerator, der wahlweise einem oder mehreren Registern zugeschaltet werden kann. Dabei sind die Frequenzen der Tongeneratoren und des Rauschgenerators via Software programmierbar. Die Lautstärke des gemischten Signals, das heißt, Tonsignal plus Rauschen, kann ebenfalls softwaremäßig eingestellt werden. Neben diesen Möglichkeiten verfügt der Soundchip noch über eine programmierbare Hardware-Hüllkurve, das heißt, der Lautstärkeverlauf im Zeitablauf kann also bereits im Soundchip durch die Auswahl einer Höhekurve vorgenommen werden.

Beim CPC-BASIC wird diese Möglichkeit nicht genutzt, das heißt, Lautstärkeänderungen und Tonänderungen, die technischen Äquivalente zu den Befehlen ENV und ENT, werden durch ständige Änderungen von Registern des Soundgenerators vorgenommen.

Der PSG verfügt über insgesamt 16 Register, von denen wir 15 nutzen können, die Bedeutung der einzelnen Register ist dabei wie folgt:

| | |
|---|---|
| Register 0-5 | bestimmen die Periodendauer und damit die Frequenz des Tonsignals an den Ausgängen A, B und C. Diese sind als 12-Bit-Wert gespeichert. Dabei enthalten die Register 0, 2 und 4 die niedrigwertigen 8 Bit für die Generatoren A, B und C. Die unteren vier Bit der Register 1, 3 und 5 stellen die Frequenz grob ein. Je kleiner der 12-Bit-Wert eines der Register wird, desto höher ist der Ton. Um einen Ton über den Kanal auszugeben, muß jedoch das entsprechende Bit im Freischaltregister, im Register 7, gelöscht sein. |
| Register 6 | gibt die mittlere Frequenz des Rauschgenerators an. Sie ist in den unteren 5 Bit enthalten. Die oberen 3 werden vernachlässigt. Die Zuschaltung des Rauschens zu den einzelnen Tongeneratoren ergibt sich wiederum aus Register 7. |
| Register 7 | Das Freischaltregister 7 sagt aus, ob der Ton- oder der Rauschgenerator auf die Ausgänge weitergegeben werden, und es bestimmt ebenso, ob der Ein-Ausgabe-PORT im Eingabe- oder Ausgabe-MODUS arbeitet. |

Die Bits sind dabei wie folgt belegt:

Bit 0: Freischaltung Kanal A (0=freigeschaltet, 1=gesperrt)
Bit 1: Freischaltung Kanal B (0=freigeschaltet, 1=gesperrt)
Bit 2: Freischaltung Kanal C (0=freigeschaltet, 1=gesperrt)
Bit 3: Rauschaddition Kanal A (zugeschaltet=0, abgeschaltet=1)
Bit 4: Rauschaddition Kanal B (zugeschaltet=0, abgeschaltet=1)
Bit 5: Rauschaddition Kanal C (zugeschaltet=0, abgeschaltet=1)
Bit 6: PORT-MODUS (0=Eingabe, 1=Ausgabe)
Bit 7: nicht verwendet (Ausgabe-MODUS des nicht herausgeführten 2. Ports)

Register 8-10 — bestimmen die Lautstärke. Dabei werden die Bit 0-3 für die Definition der Lautstärke benutzt, so daß sich Werte von 0-15 ergeben. Das vierte Bit hat eine Sonderfunktion. Wird es gesetzt, so wird nicht auf die unteren vier Bit zurückgegriffen, sondern es wird der Lautstärkeverlauf durch die Hardwarehüllkurve bestimmt (Register 11-13). Register 8 legt die Lautstärke für den Kanal A fest, 9 für B und 10 für C.

Register 11-13 — bilden den Hüllkurvengenerator. Eine Hüllkurve legt den Ablauf der Lautstärke über den Zeitablauf fest. Mit den Registern 11 und 12, Lo-Hi abgespeichert, wird die Periodendauer, das heißt die Länge der Kurve festgelegt. Register 13, genauer gesagt, die Bit 0-3 von Register 13 bestimmen die Kurvenform des Höhekurvengenerators. Bild 3.4 gibt die Höhekurven für die einzelnen Bitkombinationen wider. Dabei wurden nur die Hüllkurven für Werte der vier Bit zwischen 8 und 15 angegeben. Die Lautstärkenverläufe mit Werten zwischen 0 und 7 stellen eine Kopie derselben dar.



Register 14 — ist das letzte uns interessierende Register, es enthält den Ein-/Ausgabekanal. Der Betriebs-MODUS des Kanals wird durch den Text des Freischaltregisters bestimmt. Sie sollten dabei jedoch beachten, daß der CPC bei der Tastaturabfrage den PORT im Eingabe-MODUS erwartet. Änderungen an diesem Bit führen also dazu, daß er die Tastatur nicht mehr abfragen kann und somit auch jegliche Unterbrechungsmöglichkeiten via Tastatur, wie zum Beispiel das Drücken der ESC-Taste, verhindert sind. Das System hängt sich auf.

Trotz dieser Schreckensvision sollten Sie sich nicht davon abhalten lassen, einmal mit den verschiedenen Registern des Soundgenerators zu experimentieren. Da der Umweg über PORT B etwas lästig ist und Sie gegebenfalls Probleme mit der Tastaturabfrageroutine beim Setzen im BASIC bekommen können, existiert im Betriebssystem eine sehr vielseitige Maschinenroutine MC SOUND REGISTER, die wir gut für unsere Zwecke benutzen können. Sie ermöglicht einen problemlosen Zugriff auf die verschiedenen Register des Chips.

Dabei muß der Akkumulator, also das CPU-Register A, die Nummer des Tonregisters enthalten, CPU-Register C die zu sendenden Daten. Diese Routine eignet sich allerdings nur für die Ausgabe, das heißt das Setzen eines Tones. Die Abfrage des Eingabe-PORTs ist damit nicht möglich.

Bei den vielfältigen Möglichkeiten des 8912 hilft nur eines: Spielen, experimentieren und ausprobieren. Und dies sollten Sie nun auch tun. Als Hilfe dazu dient das nachfolgend abgedruckte Programm Soundregister. Sein Prinzip ist schnell erklärt.

Zur Ablage eines Maschinenprogramms benutzen wir wiederum den Speicherbereich von 42000 nach oben. Das Maschinenprogramm selber stellt uns vor keine großen Probleme. Den Code 3E kennen wir schon. Er bedeutet das Laden des Akkumulators, Register A mit der nachfolgenden HEX-Zahl, 0E leistet dasselbe für C. Damit sind die Register A und C auf die gewünschten Werte gesetzt, danach wird mit dem CALL-Befehl MC SOUND REGISTER aufgerufen (die HEX-Adresse ist BD34), und es folgt mit C9 ein Rücksprung in unser BASIC-Programm.

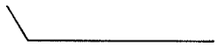
| Nummer | Ablauf | Text |
|---|---|---|
| 8 |  | umgekehrter Sägezahn |
| 9 |  | Abwärts-Rampe nach Sprung und HOLD auf 0 |
| 10 |  | umgekehrtes Dreieck |
| 11 |  | Abwärts-Rampe, Aufwärts-Sprung und HOLD auf 15 |
| 12 |  | normaler Sägezahn |
| 13 |  | Aufwärts-Rampe und HOLD auf 15 |
| 14 |  | normales Dreieck |
| 15 |  | Aufwärts-Rampe, Abwärts-Sprung und HOLD auf 0 |

Bild 3.4 Die Hardwarehüllkurven des Soundgenerators

Anstelle der beiden Nullen müssen wir nun nur noch das gewünschte Register beziehungsweise den zu sendenden Wert POKEN. Dies geschieht nach der INPUT-Abfrage in Zeile 90. Es folgt der Aufruf der Maschinenroutine und ein Rücksprung nach Zeile 70 für weitere Experimente. Abschließend noch zwei kleine Tips:

Wenn Sie beim Setzen der Periode eines Tongenerators trotz aufgedrehtem Lautstärkeregler nicht gleich einen Ton aus dem Lautsprecher vernehmen, so haben Sie wahrscheinlich das entsprechende Bit im Freischaltregister



noch nicht gelöscht. Besonders bei der Eingabe dieses Registers sollten Sie sich an eine angenehme Eigenschaft des CPC erinnern, die Eingabe von Binärzahlen mittels des vorangestellten Kürzels &X. Damit können Werte Bit für Bit bestimmt werden, was natürlich besonders beim Beschreiben des Freischaltregisters von großem Nutzen ist. Um Ihnen gleich ein anfängliches Erfolgserlebnis zu ermöglichen, hier einmal eine sehr interessante Kombination:

| Register | Wert |
|---|---|
| 0 | 65 |
| 7 | &X00110110 |
| 8 | &00010000 |
| 13 | 14 |
| 11 | 255 |

Es entsteht ein Science-fiction-Klang, den man gut in einem Weltraumspiel oder ähnlichem verwerten kann. Variieren Sie nun Register 11, schalten Sie den Rauschgenerator durch Setzen von Register 7 wieder ab, und experimentieren Sie mit den anderen Registern. Schalten Sie Klanggeneratoren zu und wieder ab und erleben Sie völlig ungeahnte Soundkombinationen. Spätestens wenn Sie dies durchgespielt haben, sollte Ihnen klar sein, warum es sich auch beim CPC lohnt, in die Maschinenspracheebene hinabzutauchen, denn diese Sounds sind mit BASIC-Befehlen zum großem Teil nicht erzielbar. Wir benötigen den Direktzugriff.

```
10 '
20 ' Sound-Register
30 '
40 MEMORY 41999
50 DATA 3e,00,0e,00,cd,34,bd,c9
60 FOR i=0 TO 7:READ a$:POKE 42000+i,VAL("&"+a$):NEXT
70 INPUT"Register";reg
80 INPUT "Wert";wert
90 POKE 42001,reg:POKE 42003,wert
100 CALL 42000
110 GOTO 70
```

# 4 Das Herzstück des Systems: der Z80 A

Nachdem wir uns eine ganze Reihe von Seiten im letzten Kapitel mit peripheren Bausteinen, der Eingabe-/Ausgabesteuerung sowie der Grafik beschäftigt haben, kommen wir nun zu den zentralen Regionen unseres Computers zurück und befassen uns mit dem Innenleben seines Gehirns, dem Z80A-Prozessor. Ausgangspunkt soll dabei das Blockschaltbild (Bild 4.1) des Prozessors sein. Es gibt eine Übersicht über die internen Register des Z80A, soweit sie für den Benutzer von Interesse sind.

Der Z80 verfügt noch über einige weitere Register, welche er jedoch für die Befehlsinterpretation und das Ausführen von Befehlen (hier speziell die Zwischenspeicherung von Daten) benötigt. Da auf diese Register vom Benutzer nicht zugegriffen werden kann, sind sie für uns hier nicht weiter von Interesse.

Schon beim ersten Blick auf den Funktionsplan fällt die Unterteilung der Register in bestimmte Gruppen auf. Jedes Register ist mit einem Buchstaben bezeichnet. Einige Register existieren doppelt. Daneben gibt es noch Größenunterschiede zwischen den einzelnen Registern. Es gibt 8-Bit-Register und 16-Bit-Register. Des weiteren existieren 8-Bit-Register, die zusammen mit einem anderen 8-Bit-Register als Registerpaar ein 16-Bit-Register bilden. Damit haben wir auch schon alle vier Typen von Speicherstellen im Prozessorinnern kennengelernt, die wir nun einzeln untersuchen wollen.

## 4.1 Die 16-Bit-Register

### 4.1.1 Die Adressregister

Diese Register sind allesamt 16-Bit-Register. Ihre Aufgabe ist in erster Linie die Adressierung von externen Speicherstellen bzw. geräten. Dabei kommen den einzelnen Registern unterschiedliche Aufgaben zu. Je nachdem stellt der Befehlssatz des Z80-Prozessors auch für die einzelnen Register höchst unterschiedliche Befehle zur Verfügung.

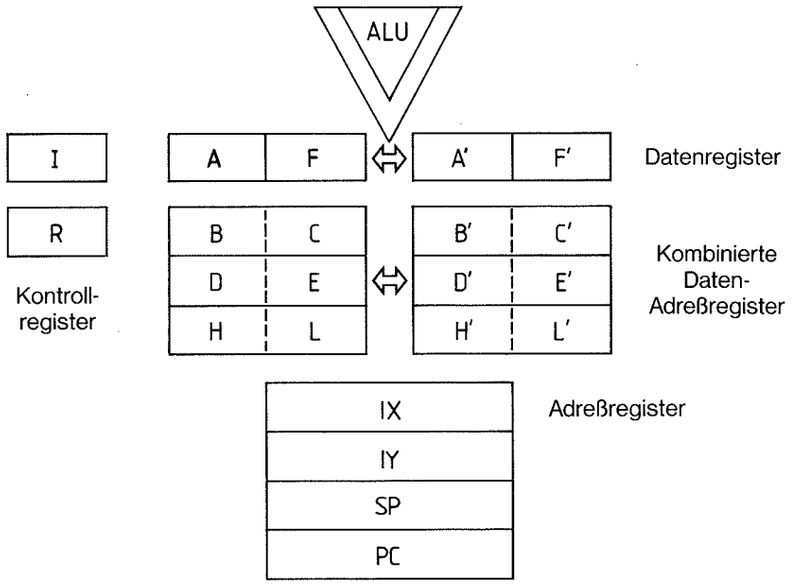



Bild 4.1: Registeraufbau des Z-80-Prozessors

**PC (Programm Counter = Programmzähler).**

Dieses Register enthält die Adresse des momentan bearbeiteten Befehlscodes beziehungsweise eines zugeordneten Datenwortes. Wie wir schon gesehen haben, sind Maschinenspracheprogramme hintereinander in aufsteigender Richtung im Speicher abgelegt. Zum Beispiel wurde ein Ladebefehl mit dem Akkumulator (Register A) in den Speicher geschrieben, indem wir zuerst den Befehlscode für das Laden des Registers A im RAM abgelegt haben. Die nachfolgende Speicherstelle hatten wir dann mit dem einzulesenden Datenwert belegt. Der Ablauf eines Maschinenprogramms sieht nun wie folgt aus:



1. Hole das durch den Programmzähler adressierte Byte aus dem Speicher, interpretiere es als Befehlswort und erhöhe den Programmzähler um 1.
2. Lies das Daten-Byte in das Register A ein und erhöhe PC um 1.

Der Programmzähler zeigt nun auf den nächsten Befehl, und das Spiel beginnt von neuem.

Beim Einschalten wird der Programmzähler auf den Wert 0 gesetzt, und der Prozessor beginnt ab dieser Speicherstelle mit der Programmausführung. An dieser Adresse muß sich also notwendigerweise die Initialisierungsroutine eines jeden Z80-Systems befinden. Alternativ kann hier auch ein Verzweigungsbefehl zu einer Routine des Betriebssystems, die die Anfangswerte setzt (wie z.B. Aufteilung der Speicherbereiche, Belegung der Tastatur, Löschen und Initialisieren des Bildschirms etc.), stehen.

**SP (Stack Pointer = Stapelzeiger).**

Ebenso wie der Programmzähler stellt auch das Register SP im Endeffekt einen Zeiger auf einen externen Speicherbereich dar. Während aber PC die Adresse des aktuellen Maschinenbefehls enthält, ist SP ein Zeiger, der mit der Datenablage zu tun hat.

Obwohl der Z80-Prozessor über eine ganze Reihe von Registern verfügt (wie das Schaubild schon auf den ersten Blick zeigt), stößt man doch schon bei den ersten etwas komplizierteren Operationen auf ein Phänomen, welches man leider bei fast jedem Prozessor vorfindet: nämlich den Mangel an Speicherplatz. Abhilfe schafft hier nur die Verlagerung von Daten in einen externen Speicherbereich, also ins RAM. Dabei werden den einzelnen Registern oder Registerpaaren nun nicht etwa spezielle Speicherplätze für die Abspeicherung zugewiesen, sondern das Register SP enthält die aktuelle Position der zuletzt gespeicherten Daten. Das Ganze hört sich etwas kompliziert an, wird aber sofort einsichtig, wenn wir es an einem Beispiel durchspielen.

Mit dem Anschalten des Computers wird der STACK POINTER auf den Wert C000 gesetzt, wobei nun von diesem Wert an abwärts gespeichert

wird. Nehmen wir nun einmal an, das Registerpaar HL soll gesichert werden, da es kurzfristig für eine Speicheradressierung benötigt wird (siehe das Auslesen in der Routine ROMREAD im letzten Kapitel). Dieser Effekt wird durch das Maschinenkommando

PUSH

erzielt. Dieser Befehl legt eines der Registerpaare AF, BC, DE, HL, IX UND IY auf dem Stapel ab.

Der Vorgang spielt sich im einzelnen wie folgt ab: Zunächst wird der STACK POINTER um 1 erniedrigt. Die Abspeicherung erfolgt also von oben nach unten, umgekehrt zum Lauf des Programmzählers. Damit hat der STACK POINTER jetzt den Wert BFFF (C000-1). An dieser Stelle wird nun das HIGHER BYTE abgelegt. Danach erfolgt eine weitere Verminderung um 1; in Position BFFE, also genau ein Byte tiefer, wird das Register L geschrieben.

Die Umkehroperation zu PUSH ist der Befehl POP. Er sorgt dafür, daß eines der Registerpaare wieder mit Daten vom Stapel geladen wird. Dazu wird zunächst das niederwertige Byte gelesen und dann in das niederwertige Register (d.h. Register C, E, L, oder F bzw. die unteren 8 Bit von IX oder von IY) geschrieben. Danach erfolgt eine Erhöhung des STACK POINTERs um 1, das Auslesen des HIGHER BYTE und daraufhin nochmal eine Erhöhung des STACK POINTERs um 1.

Der Stapelzeiger wird allerdings noch für eine zweite Anwendung benutzt, nämlich die Ablage der Rücksprungadressen von Maschinenprogrammen. Bei einem Sprungbefehl wird der PC auf einen anderen Wert gesetzt. Dies führt dazu, daß nach einem Sprungbefehl nicht der nächstfolgende Befehl bearbeitet wird; statt dessen läuft das Maschinenprogramm aufgrund der Veränderung vom PC an einer anderen Stelle weiter.

So gibt es zwei Arten von Sprungbefehlen. JUMP-Befehle (das Kürzel dafür ist JMP) ändern nur den PC; die Unterprogrammaufrufe schieben den alten PC auf den Stapel, bevor der Programmzähler mit der neuen Adresse geladen wird. Zu dieser Befehlsgruppe gehören CALL und RST. Die Unterprogrammansprünge werden dabei mit dem Befehl

RETURN (RET)



beendet. Hiermit wird dafür Sorge getragen, daß vom Stapelzeiger zwei Byte wieder ins PC zurückgelesen werden. Damit setzt dann der Z80-Prozessor nach dem Ausführen des Maschinenprogramms das alte Programm an der Stelle PC+1 wieder fort. Die Ablage der Rücksprungadressen geht dabei ebenso vor sich wie die Ablage eines Registers beziehungsweise eines Registerpaares: Zuerst werden die acht niedrigwertigen Bit abgelegt, danach die acht höherwertigsten.

Hierbei ist noch anzumerken, daß bei dieser Ablage keinerlei Kennzeichen mitgegeben wurde, ob es sich bei den abgelegten Daten nun um einen ehemaligen Inhalt eines Registerpaares oder um eine Rücksprungadresse handelt. Auch wird nicht gesagt, von welchem Registerpaar die Daten kamen. Es werden also nur zwei Byte hintereinander in den Speicher geschrieben.

Die oben beschriebene Methode ist sehr gefährlich, bietet gleichzeitig aber auch ungeahnte Möglichkeiten. So ist es unter anderem möglich, in einem Unterprogramm beispielsweise das Registerpaar HL abzuspeichern; der nachfolgende RETURN-Befehl springt dann nicht wieder in das alte Programm zurück, sondern an die durch HL angegebene Stelle. Man kann selbstverständlich auch das Registerpaar HL mit

PUSH HL

abspeichern, um es dann mittels des Befehls

POP BC

in das Registerpaar BC zu kopieren. Geschehen derartige Operationen bewußt, so kann man viel erreichen. Wird dagegen aus Unachtsamkeit ein zwischengespeichertes Registerpaar vor dem Unterprogrammrücksprung nicht wieder abgerufen, so kann das zum Absturz des Computers führen, weil er an eine unsinnige Programmstelle zurückspringt. Auf die Anwendung des STACK POINTERs und seine vielfältigen Möglichkeiten kommen wir am Ende dieses Kapitels noch einmal mit einigen Beispielen zurück.

### IX und IY (Indexregister)

Neben SP und PC verfügt der Z80 Prozessor noch über zwei reine 16-Bit-Register. Rein bedeutet hierbei, daß die oberen und die unteren 8 Bit nicht als getrennte Register angesprochen werden können. Die Indexregister ermöglichen Operationen, die sich mit einer Speicheradresse beschäftigen;

mittels dieser Register kann man also einen Datentransfer vom externen Speicher zum Prozessor und zurück bewirken und einfache Operationen mit einer Speicherstelle, wie das Inkrementieren (Erhöhen um 1) oder Dekrementieren (Vermindern um 1) ausführen.

Die Adresse der angesprochenen Speicherstelle ergibt sich dabei aus einer Kombination, genauer einer Addition, von zwei Größen. Typischerweise handelt es sich dabei um eine Konstante und natürlich den Inhalt des Indexregisters. Ein indizierter Befehl lautet zum Beispiel: Lade den Akkumulator (ein weiteres Register der CPU) mit der Speicherstelle, welche sich aus der Summe einer Konstanten zuzüglich dem Wert des Indexregisters ergibt.

Derartige Befehle lassen sich relativ gut einsetzen, um auf Daten innerhalb eines Blocks, der an verschiedenen Stellen liegen kann, auf ein bestimmtes Byte ohne Rücksicht auf die Blockanfangsadresse zurückgreifen zu können. Haben wir zum Beispiel einen Datenblock ab der Adresse A800 gepoket und wollen nun auf das 64. Byte in diesem Datenblock zurückgreifen, so bieten sich zwei Möglichkeiten.

**Möglichkeit 1:** Wir laden ein Registerpaar, zum Beispiel HL mit A840, und laden dann den Inhalt der so addressierten Speicherstelle.

**Möglichkeit 2:** Wir laden ein Indexregister, zum Beispiel IX, mit dem Wert A800 und führen dann folgenden Befehl aus:

```
LD A,(IX+40)
```

Der Vorteil dieser Art von Addressierung ist, daß wir uns nicht mehr um die aktuelle Blockanfangsadresse kümmern brauchen. So lange die Struktur innerhalb des Blocks konstant ist, genügt ein Laden der Anfangsadresse in ein Indexregister, um das gewünschte Byte innerhalb des Blocks zu adressieren. Als Anwendungsmöglichkeit sei hier einmal die Abspeicherung von benutzerdefinierten Zeichen beim CPC betrachtet. Wenn Sie sich einmal den Anhang III Ihres Bedienerhandbuches angeschaut haben, werden Sie feststellen, daß jedes Zeichen in einer Matrix von 8 x 8 Punkten dargestellt ist. Jede Reihe eines Characters nimmt also 1 Byte in Anspruch.

Um ein solches Zeichen neu zu definieren, können wir nun zwei Wege beschreiten: Entweder wir berechnen nun die Position einer jeden Zeile, oder wir laden einmalig, z.B. IY mit der gesuchten Anfangsposition und ändern dann (IY+0), (IY+1) usw.



Das zusätzliche Datenbyte fungiert hier also als Index innerhalb des Blocks, weshalb diese Art der Ansprache von Speicherstellen auch als indirekt indizierte Adressierung bezeichnet wird. Neben dieser Hauptanwendung kann man die Indexregister noch benutzen, wenn man 16-Bit-Werte zwischenspeichern will. Beim CPC dient IX z.B der Zwischenspeicherung des Stackpointers.

## 4.2 Die Universalregisterpaare BC, DE und HL

Wie der Name schon sagt, handelt es sich bei diesen Registern um Ablageplätze innerhalb der CPU, die nicht an bestimmte Verwendungszwecke gebunden sind. Dennoch weist jedes der Register bestimmte Stärken und Schwächen in den verschiedenen Anwendungsbereichen auf. Dies resultiert daraus, daß bestimmte Befehle ganz gewisse Register verlangen.

Die Universalregister können sowohl einzeln wie auch als Registerpaar angesprochen werden. Als Registerpaar dienen sie zwei Zwecken, der Adressierung von Speicherstellen und dem Rechnen mit 16-Bit-Werten. Als Einzelregister können sie sehr vielfältige Aufgaben wahrnehmen. Wir werden diese bei der Analyse des Befehlssatzes im nächsten Kapitel noch näher unter die Lupe nehmen. Hier sollen nur einige grundsätzliche Schwerpunkte beschrieben werden.

Beim Vorgänger des Z80-Prozessors, der INTEL 8080, war das Registerpaar HL in erster Linie für die Adressierung von Speicherstellen verantwortlich, woraus sich auch die etwas seltsame Buchstabenbezeichnung HL (für HIGH LOW) ableitet. Dies ist auch beim Z80A so gelieben. BC wird dagegen vorwiegend für die Adressierung peripherer Geräte, das heißt für die I/O-Befehle, benutzt. DE wird als Universalregisterpaar aufgefaßt und kann daneben in einigen Bereichen HL ersetzen. Als Einzelregister sind die sechs Register gleichwertig.

Von jedem Register existiert innerhalb der CPU noch einmal ein Double, im zweiten Registersatz. Diese Register werden mit "'" bezeichnet. Ein direkter Zugriff auf die Register des sogenannten Parallelregistersatzes ist nicht möglich. Jedoch können alle Universalregister gleichzeitig gegen ihre 6 Parallelregister ausgetauscht werden. Der Parallelregistersatz hat insofern hauptsächlich die Aufgabe einer Zwischenspeicherung. Hier werden Werte,

die permanent vom System benötigt werden und schnell verfügbar sein müssen, wie z.B. die aktuelle Speicheraufteilung in ROM und RAM, abgelegt. Man kann dann relativ schnell auf diese Werte zurückgreifen, indem man die Registersätze vertauscht, etwaige Änderungen durchführt und dann wieder rückwechselt.

Die einzelnen Universalregister haben darüber hinaus noch einen anderen Anwendungsbereich. Sie können bei den arithmetischen und logischen Operationen, die in der ALU, der Arithmetik-Logik-Einheit, erfolgen, als Datenlieferant (Quelle) oder -empfänger (Ziel) benutzt werden.

Bei diesem Teil des Prozessors handelt es sich um eine Serie von Schaltkreisen und Gattern, die es ermöglichen, Daten wahlweise im 8-Bit- oder 16-Bit-Format zu verknüpfen. Das Ergebnis wird dabei immer wieder in einem Universalregister oder -registerpaar abgelegt (**Beispiel**: Addiere den Inhalt des Registers B mit dem Akkumulator (Register A)).

### 4.3 Der Akkumulator und die Flags

Der Akkumulator ist das Register, welches der ALU am nächsten steht. Seine Aufgabe besteht darin, die ALU mit Daten zu versorgen. Gleichzeitig kann er jedoch auch als Zielregister benutzt werden. Eine typische Operation besteht darin, den Akkumulator mit dem Inhalt eines der anderen Universalregister zu vergleichen, zum Beispiel: Vergleiche den Inhalt des Akkumulators mit Register B (CP B). Addiere den Inhalt des Registers C zum Akkumulator (ADD A,C). Außer bei einfachen Operationen befindet sich das Ergebnis dann normalerweise immer im Akkumulator.

Neben dem Akkumulator ist der ALU noch ein weiteres Register eng beigeordnet, das Flagregister F. Außer dem eigentlichen Verknüpfen der Daten oder dem Abtesten der gelieferten Daten auf bestimmte Bits, dem Setzen oder Rücksetzen einzelner Bits eines gegebenen Datenbytes, nimmt die ALU nämlich noch eine weitere Funktion wahr: sie setzt nach dem Ergebnis der ausgeführten Operation bestimmte Merker im Register F, die sogenannten FLAGs.

6 Bit des Registers F werden für die Abspeicherung dieser Eigenschaften benutzt, die anderen zwei sind ohne Bedeutung. Wir wollen uns nun einmal die Aussagen der einzelnen FLAGs anschauen und beginnen dabei mit dem obersten Bit, das heißt Bit 7.



**Bit 7 Vorzeichenflag (S-Flag):**

Dieses Bit gibt an, ob das Ergebnis einer Operation negativ ist. Wir haben schon bei der Beschäftigung mit der Konvertierung von Zahlensystemen gesehen, daß der CPC Zahlen zwischen 32767 und 32768 in zwei Byte abspeichert, wobei er das höchste Bit, das heißt Bit 15, für die Definition des Vorzeichens benutzt. Eine 0 zeigt hier eine positive Zahl an, eine 1 eine negative.

Dadurch kam es zu dem etwas seltsamen Ergebnis, daß wir beim Hochzählen von hex 7FFF auf hex 8000 einen Sprung vom Positiven ins Negative erlebten. Jetzt wurde plötzlich das 15. Bit unseres 2-Byte-Wertes gesetzt. Dadurch interpretierte der Computer die Zahl als negativ. Wenn wir nur mit 8-Bit-Werten arbeiten, so wird analog das Bit 7 für die Vorzeichenangabe benutzt und dieses wird ins S-FLAG kopiert. Wir können also durch die Abfrage des S-FLAGs feststellen, ob ein gerade bearbeiteter Wert nun als Ergebnis positiv oder negativ geworden ist und davon zum Beispiel die weitere Programmausführung abhängig machen. Das geht zum Beispiel durch einen bedingten Sprungbefehl (z.B. "Springe, wenn S=0=positiver Wert" oder "Springe, wenn S=1=negativer Wert").

**Bit 6 ZERO FLAG (Z-Flag):**

Dieses Bit gibt an, ob der Wert eines Bytes, das berechnet oder übertragen wurde, 0 ist. Es wird auch bei Vergleichsoperationen angewandt, um anzuzeigen, daß zwei Datenbytes identisch sind. Das Kommando CP (Compare=Vergleiche) setzt das Z-Flag und ermöglicht dann eine bedingte Verzweigung in Abhängigkeit vom Ergebnis dieser Operation.

**Bit 5: Unbenutzt**

**Bit 4: Halbübertrags-FLAG (H-Flag):**

Dieses Bit funktioniert wie Bit 0, allerdings gibt es an, ob ein Übertrag vom Bit 3 zum Bit 4 erfolgt ist. Diese Angabe war bei früheren Computern von großem Vorteil, da man dort Zahlenwerte im sogenannten BCD-Code (Binär codiert dezimal) abspeicherte. Dieser Code benutzt 4 Bit, um eine Zahl von 0-9 zu dekodieren, das heißt, es werden anstelle der Dezimalzahlen die binären Äquivalente von 0000-1001 verwendet. Die höheren Binärzahlen von 10-15 bleiben dabei unbenutzt. Da die Abspeicherung einer Dezimalzahl genau 4 Bit benötigte, konnte man in 8 Bit genau 2 Dezimalzahlen abspeichern.

Der Übertrag von einer Stelle zur nächsten fand also bei dieser Art der Codierung zwischen dem Bit 3 und dem Bit 4 eines Bytes statt. Derartige Überträge wurden durch das Halbübertrags-FLAG angezeigt, und mittels geeigneter Operationen konnte dann eine Verrechnung mit der nächst höheren Dezimalstelle vorgenommen werden. Bei den heutigen Computersystemen wird diese Art der Abspeicherung nicht mehr benutzt, so daß das H-FLAG für den CPC-Benutzer von geringem Interesse ist.

**Bit 3: Unbenutzt**

**Bit 2: Parität/Überlauf (P/V-Flag)**

Den Begriff der Parität haben wir bereits bei der Beschreibung des ASCII-Codes eingeführt. Der CPC verfügt über eine Reihe von Befehlen, die je nach der Parität des Ergebnisses das P/V-Flag setzen beziehungsweise rücksetzen. Eine gerade Parität ergibt sich dabei aus einem Paritätsbit mit dem Wert 1, bei ungerader Parität wird P/V auf 0 rückgesetzt. Basis für diese Betrachtung bildet wiederum die Anzahl der Einsen im Ergebnis.

Die zweite wesentliche Funktion des P/V-FLAGs ist die Angabe des Überlaufs. Wenn wir zum Beispiel zu dem Wert 7F 1 addieren, so müßte sich aus einer positiven Zahl wiederum eine positive Zahl ergeben. Da aber nun ein Überlauf (vergleiche S-Flag) stattgefunden hat, wurde S auf 1 gesetzt. Das heißt, der Zahl wurde ein negatives Vorzeichen zugeordnet. In solchen Fällen wird das P/V FLAG gesetzt und dadurch signalisiert, daß hier eine Korrektur erfolgen muß. In Abhängigkeit von P/V kann man dann einen bedingten Sprung zur Korrekturroutine durchführen.

**Bit 1: Substraktions-FLAG (N-Flag):**

Dieses FLAG wird vom Z80 für den internen Abgleich nach der Ausführung von BCD-Kommandos benutzt (siehe Halbübertrags-FLAG). Für den Benutzer ist es von geringem oder gar keinem Interesse.

**Bit 0: CARRY FLAG (C-Flag):**

Dieses FLAG nimmt wie bei den meisten Mikroprozessoren, so auch beim Z80, eine Doppelrolle ein. Zum einen gibt es an, ob ein Übertrag erfolgt ist, das heißt, ob bei einer Addition die Anzahl von 8 Bit für die Darstellung des Ergebnisses überschritten wurde. Es hätte also ein 9. Bit gesetzt



werden müssen, welches aber nicht vorhanden ist. Dies ist immer dann der Fall, wenn man zwei Binärzahlen addiert, bei denen Bit 7 gesetzt ist. Zum Beispiel: 10000000 und 11000000. Das Ergebnis führt zu einem neuen Bit-Wert, nämlich 101000000.

Das CARRY FLAG nimmt nun die Funktion dieses 9. Bit wahr. Es enthält also in diesem Fall eine 1. Der umgekehrte Fall ergibt sich, wenn man bei einer Subtraktion eine 1 aus dem nicht vorhandenen 9. Bit "borgen" müßte. In beiden Fällen wird das CARRY FLAG auf 1 gesetzt. Eine Abprüfung kann hier wiederum zu einer Korrekturroutine überleiten. Eine andere Möglichkeit besteht darin, daß man bei der Behandlung der höherwertigen Bytes bei Mehr-Byte-Operationen arithmetische Befehle benutzt, die das CARRY bereits mit verwenden (ADC,SBC). Diese beiden Kommandos führen eine Addition, beziehungsweise eine Subtraktion, unter Berücksichtigung des Wertes des CARRY FLAGs durch. Man hat damit die Möglichkeit, eine Korrektur des Übertrags im nächsthöheren Byte bei in mehreren Bytes gespeicherten Zahlenwerten vorzunehmen.

Bei gesetztem CARRY wird bei ADC in der letzten Stelle zusätzlich noch 1 addiert, bei SBC und gesetztem CARRY wird Bit 0 um 1 vermindert.

Eine andere Funktion hat das Carry bei den Routier- und Verschiebebefehlen. Mit diesen ist es möglich, Register Bit für Bit in das Carry zu schieben oder durch dieses zu rotieren. Dadurch kann eine schnelle Abprüfung von Registerinhalten erfolgen, und auch eine Multiplikation mit 2 ist so binär durchführbar. Mehr dazu bei der Beschreibung der Befehle im nächsten Kapitel.

Die Hauptanwendung der FLAGs besteht darin, nachfolgende Operationen von ihrem Inhalt abhängig zu machen, das heißt zum Beispiel, einen Sprung durchzuführen, falls ein Zahlenwert negativ wurde oder ein Übertrag erfolgte etc. Normalerweise können die FLAGs nur gelesen werden, das heißt ausgewertet werden. Die einzige Ausnahme bildet das CARRY FLAG. Dieses kann mit den Befehlen

SET CARRY FLAG (SCF)

und

COMPLEMENT CARRY FLAG (CCF)

gesetzt, beziehungsweise sein Inhalt umgekehrt werden. Die Auswertung der FLAGs geschieht entweder durch Befehle, die direkt auf diese reagieren, wie zum Beispiel ADC und SBC, oder durch bedingte Verzweigungen und Sprünge. Die meisten Sprung- und Rückkehrbefehle kann man sowohl ohne Bedingung wie auch in Abhängigkeit von verschiedenen FLAGs ausführen. Je nachdem wird dem Befehlskürzel noch ein weiteres Kürzel angehängt, welches den Zustand eines FLAGs angibt. Die Bedeutung der Kürzel zeigt Tabelle 4.1.

**Tabelle 4.1**: Die Bedeutung der Flagabkürzungen

| Kürzel | Bedeutung |
|---|---|
| C | CARRY FLAG gesetzt |
| NC (not carry) | CARRY FLAG gelöscht |
| Z (zero) | ZERO FLAG gesetzt |
| NZ (not zero) | ZERO FLAG gelöscht |
| PO (Parity Odd) | ungerade Parität, also P/V=0 |
| PE (Parity Even) | gerade Parität, also P/V=1 |
| P= (Plus) | S=0 |
| M= (Minus) | S=1 |

## 4.4 Spezialregister (I,R)

Bei diesem Registerpaar handelt es sich um zwei Steuerregister, die unterschiedliche Funktionen wahrnehmen, das INTERRUPT- Register I und das REFRESH-Register R.

Das **I-Register** wird in einer speziellen Unterbrechungsbetriebsart des Prozessors, dem INTERRUPT-MODUS 2, benutzt. Durch das Anlegen eines INTERRUPT-Signals an einen PIN des Prozessors kann ein externer Baustein die CPU dazu bringen, das momentan bearbeitete Programm abzubrechen und an einer anderen Stelle, normalerweise der INTERRUPT-Behandlungsroutine, fortzufahren. Beim INTERRUPT-MODUS 2 wird beim Auftreten eines INTERRUPTs ein Byte, das vom externen Baustein auf den Datenbus gelegt werden muß, eingelesen, dazu werden als obere acht Bit die acht Bit des Registers I genommen, und diese 16-Bit-Adresse ergibt den Punkt, an dem das Programm weiterlaufen soll.

Das **REFRESH-Register** wird beim Arbeiten in speziellen RAM-Bausteinen in dynamischen RAMs benutzt. Diese verlieren ihren Speicherinhalt nach einer gewissen Zeit, falls sie nicht fortlaufend aufgefrischt werden. Der



Z80 leistet dieses Auffrischen selbständig. Dazu wird das Register R als Zähler für die einzelnen Speicheradressen benutzt. Zusammengefaßt kann gesagt werden, daß diese beiden Register für den Benutzer nur in Grenzfällen von Interesse sein dürften, so daß wir hier nicht mehr auf sie eingehen werden.

## 4.5 Eine Möglichkeit zur Registerkontrolle: 'GOMACH'

Auf den letzten Seiten haben wir in einer Grobübersicht die einzelnen Register des Z80 Prozessors und ihre Funktionen kennengelernt. Wichtig für unsere weiteren Operationen und Untersuchungen wäre es nun, uns bei der Beschäftigung mit den einzelnen Befehlen jederzeit den Inhalt der einzelnen Register und natürlich auch der FLAGs anschauen zu können. Mit unserem Monitor ist dies bis jetzt noch nicht möglich, so daß wir uns einige neue Routinen überlegen müssen.

Schauen wir uns zunächst das Problem etwas näher an. Beim Aufruf eines Maschinenspracheprogramms, und dies ist ja der Hauptanwendungsbereich für eine solche Routine, verläßt der Prozessor den BASIC-Interpreter, setzt die Register auf bestimmte Werte, auf deren Bedeutung wir am Ende des Kapitels noch zurückkommen werden, und springt dann in unsere Maschinenroutine. Aus der Maschinenroutine kehren wir dann mit dem Befehl

RETURN (RET)

wieder in das BASIC-Programm zurück. Es werden neue Daten in dem Register geladen, und die Interpretation des Programms geht weiter. Wir haben keine Möglichkeit, von BASIC aus eines oder gar alle Register des Prozessors abzufragen. Unsere einzige Möglichkeit besteht also darin, dem Prozessor vor dem Rücksprung aus dem Maschinenprogramm die Anweisung zu erteilen, die Register zwischenzuspeichern. Die Abspeicherung sollte dabei an einer Stelle erfolgen, wo sie möglichst unangreifbar sind, das heißt, nicht von anderen Daten überschrieben werden.

Nun ist es aber lästig, wenn wir an jede Maschinenroutine, die wir entwickeln wollen, am Ende vor dem Rücksprung einen Teil anhängen müssen, der dies ermöglicht. Besonders ärgerlich ist dies, wenn wir aus verschiedenen Punkten oder verschiedenen Teilprogrammen wieder in das BASIC-Programm zurückspringen, weil in Abhängigkeit von verschiedenen Bedingungen einzelne Teile unseres Maschinenspracheprogramms benutzt werden (zum Beispiel in Abhängigkeit von verschiedenen FLAGs).

Hier gibt es nur eine Möglichkeit: Wir schalten zwischen BASIC und unser Anwendermaschinenprogramm eine weitere Maschinenroutine, die diese Funktionen wahrnimmt. Der Ablauf ist dann wie folgt:

BASIC springt in unser Hilfsprogramm. Dieses springt in die Maschinenroutine. Die Maschinenroutine wird am Schluß mit dem Befehl RETurn beendet und kehrt in das Hilfsprogramm zurück. Das Hilfsprogramm sichert die Register und gibt dann wiederum zurück an den BASIC-Interpreter. Dieses Prinzip ist relativ einfach. Wir haben jedoch noch ein Problem. Wie schaffen wir es, mit möglichst wenigen Befehlen die Ablage der Register vorzunehmen?

Die Lösung dieses Problems gelingt leicht mit einem kleinen Trick: Wir können mit Hilfe der Befehle PUSH und POP jederzeit Daten auf dem Stapel ablegen oder von diesem lesen. Die Sache hat jedoch einen Nachteil. Der Stapelzeiger wird nämlich auch noch vom BASIC-Interpreter benutzt. Da dieser nach der Rückkehr aus unserem Maschinenprogramm leider auf die von uns gespeicherten Daten keinerlei Rücksicht nimmt, kann es sein, daß er sie einfach überschreibt.

Wir haben daher nur zwei Möglichkeiten. Möglichkeit 1: Wir speichern Register für Register an festgelegten Speicherstellen im Speicher ab. Möglichkeit 2: Wir verändern Register SP, schaffen uns also einen neuen Maschinensprachestapel, der dann nur noch unsere gesicherten Register enthält. Nach Beendigung der Speicheroperation setzen wir dann SP wieder auf den alten Wert zurück, worauf der Computer mit dem alten Stapel weiterarbeitet. Bei der Entwicklung unseres Hilfsprogramms wollen wir auf die zweite Variante zurückgreifen. Dabei werden wir auch die Abspeicherung von Daten beim Laufen unseres Maschinenprogramms über den neuen Maschinenstapel laufen lassen. Wir setzen also bereits beim ersten Ansprung unsere Hilfsroutine SP auf den neuen Stapelwert, fahren dann unser Programm durch und setzen nach der Rückkehr und der Abspeicherung der Register SP wieder auf den alten Wert zurück.

Die Ablage dieser kleinen Maschinenhilfsroutine soll dabei ab Speicherstelle 40000 im gesicherten Teil unseres Speichers erfolgen. Bei der nun folgenden Erklärung des Programms sollten Sie immer die Abbildung im Bild 4.2 beziehungsweise 4.3 vor Augen haben. Diese beiden Bilder geben die Abfolge der Sprünge und die Speicherbereichsaufteilung des Programms wieder. Sie sollten diese Bilder im Hinterkopf behalten, wenn wir uns nun die Funktion des Programms etwas näher anschauen.



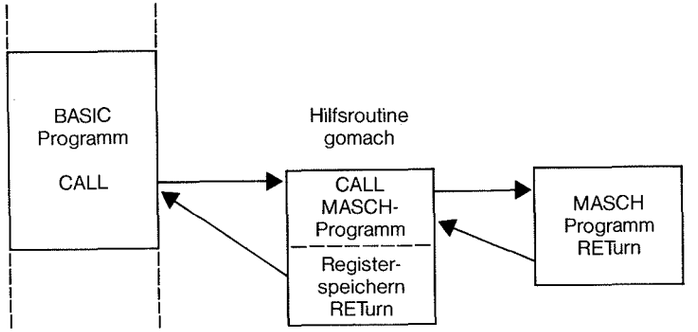



Bild 4.2: Die Sprungfolge bei zwischengeschaltetem Hilfsprogramm

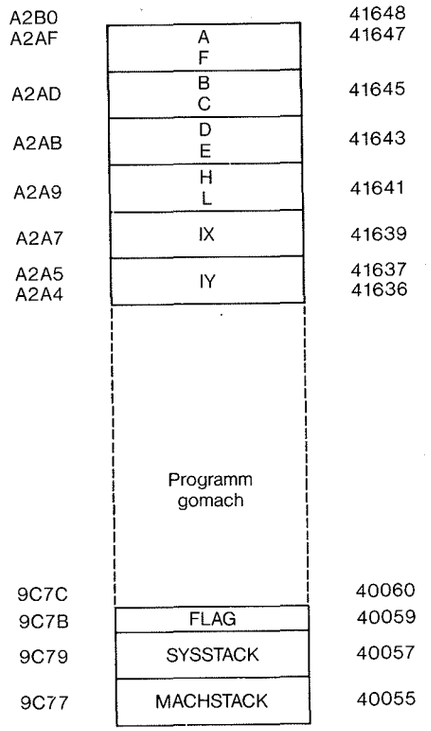



Bild 4.3: Stackbelegung der Hilfsroutine

Die Maschinenroutine benutzt 5 Speicherstellen für die Abspeicherung von altem Stack (2 Byte zu Sicherungszwecken), neuem Stack (2 Byte; unser gewählter Speicherbereich) und einem FLAG, das wir bei dieser Routine noch nicht brauchen, aber welches uns bei einer Weiterentwicklung schnell nützlich sein kann. Das FLAG benutzt die Speicherstelle 40059, darunter liegen die Variablen oder besser gesagt die Zeiger SYSSTACK (40057, 40058), dieses enthält den Wert des System-Stacks und MACHSTACK (40055, 40056). An dieser Speicherstelle werden wir unseren eigenen Maschinensprache-Stack zwischenspeichern. Nachfolgend das Assemblerlisting:

**Assemblerlisting 'GOMACH'**

| | |
|---|---|
| LD (SYSSTACK),SP | ED 73 79 9C |
| LD SP, A2B0 | 31 B0 A2 |
| CALL <Adresse> | CD 00 00 |
| PUSH AF | F5 |
| PUSH BC | C5 |
| PUSH DE | D5 |
| PUSH HL | E5 |
| PUSH IX | DD E5 |
| PUSH IY | FD E5 |
| LD A,0 | 3E 00 |
| LD (FLAG),A | 32 7B 9C |
| LD (MACHSTACK),SP | ED 73 77 9C |
| LD SP,(SYSSTACK) | ED 7B 79 9C |
| RET | C9 |

Der Ablauf unserer Maschinenhilfsroutine ist relativ einfach zu verstehen. Als erstes wird der System-Stack in den Speicherstellen 40057, 40058 bzw. 9C79 und 9C7A in Sicherheit gebracht. Danach laden wir SP mit dem Beginn unseres neuen Maschinensprachestapels. Hier wurde der Wert A2B0 gewählt, was Kollisionen mit der Floppy ausschließt. Dieser Wert stellt eine willkürlich vorgegebene Größe dar.

Allerdings ist hierbei zu sagen, daß man sich bei dieser Obergrenze natürlich an der Benutzung des Restsystems im Speicher orientieren muß. So ist zu berücksichtigen, daß die Floppy und auch eine Umdefinition des Zeichensatzes mittels SYMBOL, den vom System benutzten Speicher, dessen



Untergrenze die Variable HIMEM angibt, nach unten ausdehnen. Er liegt also dann nicht mehr auf 43903, sondern zum Beispiel auf 42619 bei eingeschalteter Floppy oder noch niedriger, wenn mehr als die oberen 15 Zeichen umdefiniert werden sollen.

Mit den hier gewählten A2B0 hat man oberhalb unseres neuen Stapels noch ungefähr 600 Byte bei normalem Betrieb zur Verfügung, in denen man Maschinenspracheprogramme ablegen und austesten kann. Bei voller Zeichensatzumdefinition schrumpft dieser Wert allerdings auf 0. Man muß dann die Speichergrenze von 40000 nach unten verschieben und kann dann unterhalb des Maschinenprogramms bis 40000 seine Programme austesten.

Nach diesen beiden Operationen haben wir die neue Stapelzeigersituation hergestellt. Als nächstes folgt mit einem Unterprogrammaufruf der Ansprung unseres Maschinenprogramms. Von Hause aus wird hier die Adresse 0000 angesprungen. Es würde also in diesem Fall ein System-Restart erfolgen, der den kompletten CPC wieder in den Einschaltzustand zurückversetzt. Diese beiden Werte sind also unbedingt mit der Anfangsadresse unseres anzuspringenden Maschinenspracheprogramms zu poken.

Wir verlassen unser Maschinenspracheprogramm durch den Befehl RET (siehe Bild 4.2) und kehren damit aus der Unterprogrammebene wieder in unser Hilfsprogramm zurück. Es folgen eine Reihe von Speicherbefehlen, die nacheinander sämtliche Register, bis auf SP und PC, auf den Maschinenstapel ablegen. Das Kommando hierfür heißt PUSH.

Als nächstes laden wir das FLAG (Adresse 9C7B) mit 0 und stellen den alten Stapelzustand wieder her. Der Wert unseres Maschinenstapelzeigers wird in MACHSTACK gesichert, anschließend wird SP wieder auf den alten Wert des Systemstapels geladen. Es folgt der Rücksprungbefehl, der die CPU anweist, wieder an den BASIC-Interpreter zurückzugeben.

Nach dieser detaillierten Erklärung des Maschinenprogramms dürfte es nun kein Problem mehr für Sie sein, auch den Restüberbau, das heißt das BASIC-Programm, welches die gelieferten Daten auswertet und anzeigt, zu verstehen. Sie finden es nachfolgend abgedruckt.

```
10 ' ************
20 ' ** gomach **
30 ' ************
40 ' machstack=40055
50 ' sysstack =40057
60 ' flag=40059
70 WINDOW#0,1,40,4,25:WINDOW#1,1,40,1,3
80 INK 2,0:INK 3,21:PAPER#1,2:PEN#1,3
90 CLS:CLS#1
100 MEMORY 39999
110 DATA ed,73,79,9c,31,b0,a2,cd,00,00
120 DATA f5,c5,d5,e5,dd,e5,fd,e5,3e,00
130 DATA 32,7b,9c,ed,73,77,9c,ed,7b
140 DATA 79,9c,c9
150 DATA x
160 i=40060
170 READ a$:IF a$="x" THEN 190
180 POKE i,VAL("&"+a$):i=i+1:GOTO 170
190 '
200 ' *****************
210 ' ** gomach demo **
220 ' *****************
230 INPUT"Wieviele Bytes braucht das Maschinenpro-gramm";b
240 INPUT"Ab welcher Speicherstelle soll das Maschprogramm liegen"
;st
250 IF st<0 THEN st=st+65536
260 FOR i=st TO st+b:PRINT"Speicherstelle";i;:INPUT"Welcher Wert";
w:POKE i,w
270 NEXT
280 ' ***************
290 ' ** demoende **
300 ' ***************
310 ' Ansprung von gomach
320 '
330 INPUT"Ansprunstelle";s
340 IF s<0 THEN s= s+65536
350 s1=INT(s/256):s2=s-256*s1
360   POKE 40068,s2:POKE 40069,s1:CALL 40060
370 ' Stack auslesen
380 machstack=PEEK(40055)+256*PEEK(40056)
390 FOR i=machstack TO machstack+10 STEP 2
400 z$=RIGHT$("0"+HEX$(PEEK(i+1)),2)+RIGHT$("0"+HEX$(PEEK(i)),2)+"
  "+z$
410 NEXT
420 z$=z$+RIGHT$("0"+HEX$(PEEK(40058)),2)+RIGHT$("0"+HEX$(PEEK(400
57)),2)
430 n$=" AF    BC    DE    HL    IX    IY    SP"
440 PRINT#1,n$,z$
```



GOMACH besteht aus drei Teilen. Im ersten Teil wird das Maschinenspracheprogramm an Speicherstelle 40060 in den Speicher gepoket. Es folgt eine kleine Demonstrationsroutine, die dazu dient, die Wirkung des Programms und auch seine Funktionen zu überprüfen. Sie ermöglicht es, ein Maschinenprogramm beliebiger Länge im HEX-CODE einzugeben.

Ab Zeile 330 erfolgt dann der Ansprung der Hilfsroutine und indirekt damit auch der Aufruf des eingegebenen Maschinenspracheprogramms. Dessen Ansprungstelle wurde in S eingelesen. Nach der uns schon bekannten Methode zerlegen wir S in ein LOWER und ein HIGHER BYTE und poken diese Werte in die Speicherstellen 40068 und 40069. Dies sind genau die Adressen hinter dem CALL-Befehl, wo normalerweise die 4 Nullen stehen. Unser Hilfsprogramm ist somit aktiviert und auf das aktuelle Maschinenspracheprogramm angepaßt, worauf dann der Ansprung erfolgt.

Ab Zeile 380 werden dann mit Hilfe des Zeigers MACHSTACK die mit PUSH gespeicherten Registerinhalte von unserem Zweit-Stack wieder ausgelesen. Zunächst wird die Endadresse unseres Maschinenstapels, die in der Variablen MACHSTAG in 40055 und 40056 am Ende unserer Hilfsroutine abgelegt wurde, wieder zu einer BASIC-Variablen zusammengebaut. Die Schleife in Zeile 390-410 liest dann Speicherstelle für Speicherstelle die einzelnen Registerinhalte aus und fügt sie zu einem geeigneten STRING zusammen. In Zeile 420 wird an diesen noch der Wert des STACK POINTERs angehängt; danach erfolgt der Ausdruck.

Hierbei ist darauf hinzuweisen, daß wir die Speicherstellen 40058 beziehungsweise 40057 für die Abfrage des Stapelzeigers benutzt haben. Es wird also der Wert des System-STACKs ausgegeben. Wollen wir uns unseren eigenen Maschinenstapel anschauen, so müssen wir die Werte nur um 2 vermindern (also auf 40056 beziehungsweise 40055).

In unserer ersten Untersuchung wollen wir uns zunächst jedoch noch mit dem Systemstapel beschäftigen. Als Testprogramm nehmen wir dabei das kürzest mögliche, nämlich einfach den Befehl

C9

also RETURN. Damit gibt das angesprungene Maschinenprogramm sofort wieder an unsere Hilfsroutine zurück. Die Register werden dabei so ge-

sichert, wie sie beim Ansprung des Maschinenspracheprogramms vorhanden sind. Beim Lauf des Programms müßten Sie also die folgenden Eingaben machen: Programmlänge 1; ab welcher Speicherstelle: z.B. 42000; Wert dieser Speicherstelle: &C9; Ansprungstelle 42000.

Sie erhalten nun (korrekte Eingabe des Programms vorausgesetzt) in den obersten drei Bildschirmzeilen eine zweizeilige Ausgabe. Die oberste Zeile enthält dabei die Registernamen, beginnend mit AF auf der linken Seite, bis hin zu SP ganz rechts. Darunter finden sich als 4-Byte-HEX-Werte die Inhalte der einzelnen Register. Auf ihre Bedeutung wollen wir nun im Folgenden etwas näher eingehen.

| AF | BC | DE | HL | IX | IY | SP |
|---|---|---|---|---|---|---|
| 0068 | 20FF | 9C7C | 0560 | BFFE | B0C2 | BFF8 |

Fangen wir bei den leichtesten Angaben an. Es sind dies die Inhalte der Registerpaare DE beziehungsweise HL. Dieses enthält die Position im BASIC-Quelltext, ab welcher der Ansprung der Maschinenspracheroutine erfolgt ist (hier wäre dies der CALL-Befehl in Zeile 360). HL verweist auf das direkt folgende Statement. Es stellt also die Rücksprungangabe in den BASIC-Interpreter dar.

Als Benutzer kann man durch die Überprüfung von HL feststellen, von welchem Punkt eines BASIC-Programms aus eine mehrfach benutzte Maschinenroutine angesprungen wurde; davon kann man dann gegebenenfalls deren Handlungsweise abhängig machen. Bei normaler Eingabe des Programms sollten Sie hier den Wert 0560 erhalten. Es kann jedoch durch Einfügen bzw. Auslassen von Spaces im BASIC-Quelltext zu Abweichungen um einige Spaces nach oben oder unten kommen.

DE enthält die Anfangsadresse der angesprungenen Routine. In unserem Fall war dies 42000 oder 9C7C. Diese Zahl findet sich dann auch in DE wieder. Eine ähnliche Bedeutung hat das Register IY. Der BASIC-Interpreter hat eine 2-Byte-Adresse, in der er Integernvariable (also z.B. auch Ansprungadressen) zwischenspeichert. Dies sind die Speicherstellen B0C2 und B0C3. Hier findet sich wiederum, im Format LOW HIGH abgespeichert, unsere Ansprungadresse 9C7C. Und zwar enthält B0C2 den HEX-Wert 7C, B0C3 den HEX-Wert 9C.

Sollten Sie bei der Abfrage dieser Speicherstellen am Programmende ganz andere Werte vorfinden, wundern Sie sich nicht. Denn wie der Name schon sagt, handelt es sich bei diesen Adressen um einen relativ häufig vom Computer benutzten Zwischenspeicher, der folglich sehr oft seinen Wert ändert. Dies gilt jedoch nur für die Analyse von BASIC aus. Beim Einsprung in eine Maschinenspracheroutine können wir uns darauf verlassen, daß diese beiden Variablen immer den Ansprungpunkt enthalten. Dies ist u.a. dann von Vorteil, wenn man die Rückkehr aus dem Maschinenspracheprogramm von unterschiedlichen Ansprungpunkten abhängig machen will.



SP enthält den Wert des Systemstapelzeigers nach dem Rücksprung aus der Maschinenroutine. IX und A bilden ein sehr interessantes Duo bei der Übergabe von Parametern. Der CALL-Befehl ist nämlich nicht auf eine Adresse beschränkt. Es können insgesamt bis zu 32 16-Bit-Werte übergeben werden. Das ermöglicht es, eine ganze Reihe von Variablen mit ihren Adressen zu bezeichnen. Die Adresse kann man relativ einfach durch den Variablenpointer @ übergeben. Bei unserem Durchlauf des Programms haben wir in IX den Wert BFFE und in A 0 gefunden. Es wurden somit 0 Parameter übergeben; damit blieben nur 2 Byte auf dem Maschinensprache-STACK in den Positionen BFFF und BFFE: nämlich die Rücksprungadresse zum BASIC-Interpreter.

Wenn wir Daten mit im Programm übergeben, sieht der Ablauf wie folgt aus: beim Laden unseres Maschinenprogramms haben wir in den ersten Zeilen den String A$ benutzt. A$ ist auch noch beim Ansprung unserer Maschinenroutine als String angelegt, da er ja nicht gelöscht wird. Wir könnten ihn also theoretisch zur weiteren Bearbeitung mit an das Maschinenprogramm durch seine Anfangsadresse übergeben.

Hierzu fügen wir zunächst in Zeile 360 hinter dem CALL-Befehl das Kürzel @A$ mit Komma getrennt an. Um einer falschen Ausgabe vorzubeugen, sollten wir nun noch mit

```
385 Z$=""
```

Z$ löschen und können danach einen zweiten Durchlauf mit

```
GOTO 330
```

wagen. Ansprungstelle ist dabei wieder 42000.

Unsere Ausgabe hat sich jetzt in vielen Punkten verändert. Zunächst einmal hat sich der Wert in HL erhöht. Das liegt daran, daß wir unseren CALL-Befehl durch die Anfügung verlängert haben. Damit ist auch der absolute Rücksprungpunkt weiter im Speicher nach oben gerutscht. IY enthält wei-

terhin den Verweis auf die Integernvariable, also auf B0C2. IX und SP sind um je 2 nach unten verschoben worden. An dieser Stelle befindet sich jetzt ein Zeiger auf A$, nämlich der Wert unseres Variablenpointers (siehe Kapitel 1).

Der Akkumulator Register A gibt uns die Gesamtzahl der abgespeicherten Variablen an (also 1), Register B die Differenz zur möglichen Maximalzahl von 32 Variablen (= 20 HEX Variablen). Hier findet sich jetzt der Wert 1F im Vergleich zu der 20 vom ersten Durchlauf. Bei zwei übergebenen Parametern würden sich IX und SP nochmals um 2 vermindern, in A fände sich dann eine 2, in B eine 1E.

Als letztes zu erklärendes Register bliebe noch DE übrig. Während wir im ersten Fall den Ansprungpunkt unserer Routine zwischengespeichert hatten, findet sich jetzt hier der Wert des übergebenen Variablenpointers. DE enthält also immer die zuletzt ausgerechnete und danach auf den Stapel abgelegte Integergröße.

Und noch eines dürfte Ihnen aufgefallen sein. Das FLAG-Register hat sich nämlich geändert. Hatten wir beim ersten Durchlauf noch den Wert hex 68, so findet sich nun hier eine 28. Schauen wir uns das Ganze einmal binär an, so erhalten wir als Ausgabe die Größen 00101000 beziehungsweise 01101000. Der Unterschied zwischen den beiden Zahlen besteht also darin, daß bei der 68 Bit 6 gesetzt ist, bei der 28 dagegen nicht.

Wie wir schon wissen, stellt Bit 6 das ZERO FLAG dar. Werden keine Daten mit im CALL-Befehl übergeben, so ist das ZERO FLAG gesetzt (wegen A=0), ansonsten ist es rückgesetzt. In unserem angesprungenen Maschinenspracheprogramm können wir den Zustand des ZERO FLAGs zur Erkennung benutzen, ob Daten Übergeben wurden, und damit gegebenenfalls einen Dateneinleseteil unseres Programms überspringen.

Sie sehen: Die Maschine stellt Ihnen eine ganze Serie von nützlichen Informationen für die Programmierung eigener Maschinenspracheprogramme zur Verfügung. Um diese vielfältigen Chancen etwas näher auszuprobieren, sollten Sie sich unbedingt mit diesem Mechanismus vertraut machen. Definieren Sie doch einmal mit Zeilennummer zwischen 330 und 340 eine Reihe von Variablen, und übergeben Sie diese dann via Variablenpointer im CALL-Befehl, durch weiteres Anhängen mit Komma. Wenn Sie dann das Programm wieder mit

```
GOTO 330
```



laufen lassen, werden Ihnen die Funktionen der einzelnen Register relativ schnell klarwerden.

Mit diesem Programm haben wir nun eine gute Arbeitshilfe geschaffen, die es uns ermöglicht, den Befehlssatz des Z80 Prozessors in all seinen Auswirkungen auf die verschiedenen Register zu untersuchen. GOMACH ist alleine lauffähig.

Wir können allerdings das Programm, und dies ist natürlich auch das Endziel, mit in unseren Monitor unter dem Kommando G implementieren. Dies geht relativ problemlos. Sie setzen dazu einfach den Einleseteil mit den DATA-Statements und der POKE-Schleife hinter die POKE-Schleife von ROMREAD. Der DEMO-Teil zwischen Zeile 200 und 300 entfällt. Den Rest des Programms numerieren Sie in den für die GO-Routine reservierten Bereich des Monitors hoch. Nach ein paar kleinen Anpassungen am Anfang und am Ende des Programms läuft es dann ohne Probleme.

Wenn Sie Schwierigkeiten damit haben sollten oder nicht ganz so neugierig sind, können Sie sich auch bis zum Ende des 6. Kapitels gedulden. Dort findet sich dann das Gesamt-LISTING des Monitors mit allen Sonder- und Spezialfunktionen und natürlich auch mit einer leistungsstarken GO-Routine.

# 5 Der Befehlssatz des Z80-Prozessors

Zum Zeitpunkt der Entwicklung unseres Prozessors stand man vor zwei Problemen. Zum einen war für das neue Gerät weder Software vorhanden, noch konnte sie von dem kleinen Unternehmen schnell genug bereitgestellt werden. Zum anderen war ja auch der Z80 von Anfang an als Verbesserungsmodell für den 8080-Prozessor gedacht. Dies führte dazu, daß man den Chip aufwärtskompatibel gestalten wollte. Der große Vorteil dieser Konzeption ist, daß alle 8080-Programme auch auf dem Z80 laufen.

Dafür stellte sich dann allerdings das große Problem der Befehlsdecodierung. Der 8080 hatte ja bereits einen relativ umfangreichen Befehlssatz, welcher die zur Verfügung stehenden 8 Bits mit ihrer Möglichkeit, 256 Befehle zu decodieren, weitgehend ausnutzte. Nun waren aber für den neuen Prozessor fast 700 Befehle geplant. Es stellte sich daher die Frage, wie man mit 256 verschiedenen Codes 700 Möglichkeiten decodieren kann. Das Stichwort hierzu heißt Erweiterungscodes.

Der Z80 verfügt über 4 Erweiterungsbytes mit den Bezeichnungen CB, BD, ED und FD jeweils im HEX-Code. Diese Bytes bzw. Kombinationen davon führen in insgesamt sechs Erweiterungstabellen, die dann die restlichen Befehle enthalten. Das Prinzip ist dabei wie folgt: Trifft der Prozessor bei der Interpretation auf einen solchen Erweiterungscode, so geht er in eine andere Befehlstabelle und interpretiert in dieser dann den Inhalt der nächsten Speicherstelle als Erweiterungsbefehl.

Für diese Erweiterungsbefehle bestehen grundsätzlich zwei Möglichkeiten. Ein Teil stellt vollkommen neue Befehle dar. Der Rest erweitert den Befehlssatz derart, daß solche Befehle, die in der Grundtabelle nur mit dem Akkumulator durchführbar sind, in den Erweiterungstabellen dann auch zusammen mit den anderen Universalregistern funktionieren.

Ein Z80-Maschinensprachebefehl besteht immer aus zwei Teilen: dem Befehlscode (auch OP CODE = Operation Code genannt) und einem Datenteil, der gegebenenfalls auch wegfallen kann. Der OP CODE ist zwischen einem und drei Bytes lang, der Datenteil zwischen null und zwei Bytes. Besteht der OP CODE aus drei Bytes, wird nur ein Datenbyte benötigt. Somit ergeben sich als Maximum 4 und als Minimum 1 Byte Länge für einen Z80-Maschinensprachebefehl. Die Aufteilung zwischen Befehls- und Datenteil sowie die Gesamtlänge eines Maschinensprachestatements ist also in Grenzen variabel und wird durch den am Anfang befindlichen OP CODE angegeben.

## 5.1 Befehlsarten und Adressierungstechniken

Bei unserer weiteren Analyse des Befehlssatzes werden wir vier Arten von Befehlen unterscheiden, die wir an dieser Stelle nun definieren wollen.

Datentransferbefehle: Diese Gruppe beinhaltet all jene Z80-Kommandos, die sich damit beschäftigen, Daten von bestimmten Punkten des Systems (Quelle) zu einem Bestimmungsort (Ziel) zu verlagern. Quelle und Ziel können identisch sein. Sowohl Register und/oder CPU als auch externe Speicher (RAM, ROM, I/O-Geräte, STACK) können als Quelle bzw. Ziel fungieren. Eine Veränderung der Quelldaten erfolgt nicht. Der Datentransfer existiert in zwei Varianten:

a) als Austausch (die Inhalte von Ziel und Quelle werden vertauscht)

b) als Kopie (das Ziel erhält den Wert der Quelle)

Verzweigungsbefehle: Diese Gruppe beinhaltet all jene Z80-Befehle, mit denen arithmetische oder logische Operationen sowie Bitmanipulationen durchgeführt werden können. Es tritt eine Veränderung der Daten und/oder der FLAGS ein.



Sprungbefehle: Mit diesen Steuerbefehlen wird der Prozessor angewiesen, die Programmausführung an einer anderen Stelle fortzusetzen.

Steuerbefehle: Diese Befehlsgruppe verändert die Befehlsausführung der CPU und schafft damit die Basis für einen veränderten Ablauf bestimmter Befehle.

Befehle, die sich mit dem Datentransfer oder dem Verändern von Daten beschäftigen, können sowohl auf ein Byte wie auch auf zwei Byte, also ein 16-Bit-Wort, oder auch auf einen Block von Daten wirken. Die Sprungbefehle können unbedingt oder unter einer Bedingung erfolgen. Die Abhängigkeit von einer Bedingung wird dabei durch nachgestellte Kürzel angedeutet (Näheres siehe Kapitel 4 FLAGs).

Viele Befehle existieren in mehreren Adressierungsarten. Bevor wir uns daher den einzelnen Befehlsgruppen zuwenden, wollen wir noch kurz auf die verschiedenen Möglichkeiten der Adressierung eingehen. Zunächst jedoch noch eine formelle **Definition**:

Adressierungsart ist die Methode, auf Daten hinzuweisen, die im Programm benötigt werden.

Wir werden im folgenden zwischen acht Adressierungstechniken unterscheiden, die nun im einzelnen vorgestellt werden sollen:

### Die unmittelbare Adressierung

Bei der unmittelbaren Adressierung folgen die benötigten Daten dem OP CODE beziehungsweise den OP CODEs direkt nach. Es können ein Datenbyte oder zwei Datenbytes (erweiterte unmittelbare Adressierung) übergeben werden. **Beispiele:**

```
LD A,30 H
LD BC,3040 H
```

Werden zwei Byte mitgeliefert, so wird der Wert als im Format LOW-HIGH gespeichert betrachtet, das heißt, das erste Datenbyte im zweiten Befehl würde in C geladen werden, das zweite Datenbyte fände sich am Schluß in Register B.

Schreibweise/Kürzel:

Die unmittelbaren Daten werden durch Komma getrennt, an den OP CODE und gegebenenfalls die Registeradresse angehängt.

**Die absolute Adressierung:**

Bei der absoluten Adressierung, die auch als erweiterte Adressierung bezeichnet wird, folgt dem OP CODE die Adresse, an der die gewünschten Daten stehen. **Beispiel:**

LD A,(C000)

A enthält nach dem Ausführen des Befehls den Inhalt der Speicherstelle C000. Die indirekte Angabe der Daten wird durch Klammer ausgedrückt.

**Die indirekte Adressierung:**

Das Prinzip ist hier ähnlich wie bei der absoluten Adressierung, nur daß die Adresse nicht als Daten nach dem OP CODE mitgeliefert wird, sondern durch den Inhalt eines Registerpaares bestimmt ist. **Beispiel:**

LD A,(HL)

Hier wird A mit dem Wert der Speicherstelle geladen, die durch HL bestimmt wird.

Schreibweise/Kürzel:
Die indirekte Angabe der Daten wird durch die Klammer ausgedrückt.



**Die implizite Adressierung:**

Bei dieser Adressierungsart werden Ziel und Quelle durch den OP CODE bereits ausreichend bestimmt. **Beispiel:**

LD A,B

Schreibweise/Kürzel:
Keine (der OP CODE gibt bereits Ziel und Quelle an).

**Die modifizierte Nullseitenadressierung:**

Diese spezielle Adressierungsart kommt nur bei einer besonderen Variante der Sprungbefehle, den RESTART-Anweisungen (RST), vor. Es handelt sich um eine Art implizite Adressierung für die Sprungbefehle. Bei dieser verkürzten Ansprungangabe wird das HIGH BYTE der Adresse immer als 0 betrachtet, ebenso die Bit 0 bis 2, 6 und 7 des LOW BYTEs. Die restlichen drei Bit der Adresse sind im RST-Kommando bereits enthalten und geben die acht RST-Ansprungpunkte an.

Schreibweise/Kürzel:
Die OP CODEs der RST-Anweisungen enthalten bereits die benötigte Adressenangabe.

**Die relative Adressierung:**

Hierbei handelt es sich um eine weitere Adressierungsart, die nur für Sprünge benutzt wird. Sprungbefehle verändern im wesentlichen den Inhalt von PC. Bei der relativen Adressierung wird PC nun nicht auf einen extern vorgegebenen Wert gesetzt, sondern es erfolgt ein Sprung vor oder zurück, um eine bestimmte Anzahl von Speicherstellen, das sogenannte DISPLACEMENT, was symbolisch auch mit "dis" abgekürzt wird.

Die Verschiebung ergibt sich dabei aus einem Byte, von dem das oberste Bit als Vorzeichen-Bit benutzt wird. Ist es gesetzt, so ist das DISPLACEMENT negativ, es folgt also ein Sprung rückwärts. Bei nicht gesetztem Bit wird PC um die in den nächsten sieben Bit angegebene Zahl von Speicherstellen weitergesetzt.

Eine kurze Behandlung dieser Technik haben wir schon bei der Beschäftigung mit dem S-FLAG im letzten Kapitel kennengelernt. Wir wollen das Ganze nun einmal anhand von einigen Beispielen durchspielen. Bei der Berechnung der Rücksprünge sind mehrere Punkte zu beachten. Zum einen gelten die Zahlen von hex 80 bis hex FF als negativ, wobei ein Wert von FF ein Rücksprung um 1, ein Wert von FE ein Rücksprung um 2 usw. ist.

Zum anderen ist aber auch die Berechnungsbasis zu beachten. Nach jedem Befehl befindet sich PC auf dem Anfang, das heißt, dem ersten Byte des nächsten Befehls. Steht ein relativer Sprungbefehl wie JR also auf der Anfangsadresse A000, so befindet sich das DISPLACEMENT-BYTE in A001, und nach der Befehlsausführung steht der PC auf A002. Dieser Wert ist die Anfangsadresse, zu der wir nun unsere Verschiebung addieren, beziehungsweise von der wir sie subtrahieren müssen.

Ist die Verschiebung 0, so tritt keine Veränderung von PC ein. Haben wir dagegen die hex-Folge 18 01 gegeben (18 hex ist der Code für den unbedingten relativen Sprung JR), so wird PC um 1 erhöht, der Befehl A002 also übersprungen. Maximal können wir nach dieser Methode bis hex 7F, also dezimal 127, nach vorne springen, dagegen um 128 zurück. Beziehen wir dies jedoch auf den Beginn unseres Sprungbefehls, das heißt auf die Adresse A000, so sieht das Ganze nun völlig anders aus. Wir springen nämlich um bis zu 129 nach vorne und maximal 126 zurück.

Schreibweise/Kürzel:
Der Wert der Verschiebung wird nach dem Befehl durch "Abstand getrennt" angegeben, also zum Beispiel JR 30 H.

Manchmal wird er auch als dezimal angegebener Plus-/Minuswert geschrieben, beispielsweise JR +10 oder JR -20.

Gute DISASSEMBLER, Programme, die ein in hex-Code geschriebenes Maschinenprogramm wieder in die MNEMONICs zurückverwandeln, geben darüber hinaus häufig den absoluten Wert der anzuspringenden Speicherstelle an.



**Indizierte Adressierung:**

Die indizierte Adressierung stellt die wohl schwierigste Adressierungsart dar. Hierbei ergibt sich die Position des zu benutzenden Datenbytes aus einer Kombination des Indexregisters (wahlweise IX oder IY) zuzüglich eines DISPLACEMENT-Byte. Es handelt sich also um eine indirekte Adressierung, bei der zusätzlich noch eine Verschiebung eingerechnet wird. Die Funktion der einzelnen Adreßteile haben wir schon bei der Behandlung der Indexregister kennengelernt.

**Beispiele:**

```
LD A,(IX+30 H)
IX:=A000
```

Der Akkumulator nimmt nun den Wert an, der in der Speicherstelle A030 zu finden ist.

```
LD A,(IY+F0 H)
IY:=C000
```

A enthält jetzt den Inhalt von BFF0. Sie sollten unbedingt jetzt einmal durchrechnen, warum dies so ist. Ein kleiner Denkanstoß: FF hat den Wert -1, führt also von der Speicherstelle C000 zu BFFF. Vermindern Sie nun die Speicherstelle und das DISPLACEMENT jeweils gleichzeitig in Einserschritten, so erhalten Sie das Ergebnis.

Schreibweise/Kürzel:
In der Indirektklammer steht das Symbol für das Indexregister sowie mit "+" verbunden das DISPLACEMENT. Beides zusammen wird durch Komma vom OP-CODE getrennt.

## 5.2 Die Analyse der Befehle beim Z80

Z80 ist ein bitorientierter Prozessor. Mehr als andere Chips herrscht bei ihm eine bitorientierte Arithmetik vor. Es gibt eine ganze Reihe von Operationen und Befehlen, die der Veränderung, dem Testen, Abfragen, Rücksetzen und Setzen für Bits dienen, und auch bei der Befehlsanalyse greift der Prozessor auf die einzelnen Bits in sehr starkem Maße zurück.

Dies führt dazu, daß man anhand des Binärwertes eines OP CODES die Bedeutung der einzelnen Befehle relativ gut herauslesen kann. Wir werden dies bei der Entwicklung unses Disassemblers im nächsten Kapitel noch näher vertiefen. Hier wollen wir nur einmal die Grobstruktur des Z80-Befehlssatzes aufzeigen, um einen Ordnungsrahmen für die nachfolgende Beschäftigung mit den einzelnen Befehlen zu schaffen. Als erstes zur **Grundtabelle**: Diese enthält alle 1-Byte-OP-CODES. Nach den höchsten beiden Bits, das heißt den Bit 6 und 7 des OP CODES, lassen sich hier 4 Befehlsgruppen analysieren:

**00** Diese 64 OP CODES enthalten die Befehle für unmittelbares Laden, für das INKrementieren und DEKrementieren, für Register und Registerpaare, einen Teil der 16-Bit-Lade- und Arithmetikbefehle sowie einige bitorientierte Befehle, die relativen Sprungkommandos und die Befehle NOP und EX AF,AF'.

**01** Diese Befehlsgruppe enthält alle Befehle zum Datentransfer zwischen einzelnen Universalregistern sowie das Interrupt- Kommando HALT.

**10** Diese Befehlsgruppe enthält die Arithmetik- und Logikbefehle für den Akkumulator.

**11** Mit diesen ersten Bits beginnen alle Sprungbefehle, außer denen in Gruppe 00. Sie enthalten die Kommandos PUSH, POP, IN, OUT, Befehle für den Austausch von Registerpaaren, zur Interruptfreigabe und -sperrung. Auch finden sich hier die Codes für die vier Erweiterungstabellen.

**Erweiterungstabelle CB (2 Byte OP CODE)**

Unter dem Befehl CB erreicht man eine weitere Sprungtabelle mit 248 neuen Befehlen. Diese dienen im wesentlichen zum Verschieben und zum Routieren von Universalregistern sowie zum Setzen, Rücksetzen und Testen von Bits.



**Erweiterungstabelle DD (2 Byte OP CODE)**

Diese Erweiterungstabelle enthält alle Befehle, die das Indexregister IX benötigen, beziehunsweise auf es zurückgreifen.

**Erweiterungstabelle FD (2 Byte OP CODE)**

Diese Erweiterungstabelle leistet dasselbe wie DD, allerdings für das Indexregister IY. Die Funktion der nachstehenden Codes ist jedoch mit denen der Tabelle DD identisch!

**Erweiterungstabelle ED (2 Byte OP CODE)**

Mit dem Erweiterungsbefehl ED erreichen wir eine weitere Hilfstabelle, die uns weitere Eingabe-/Ausgabebefehle, Befehle für 16 Bit, Transfer und Arithmetik, Block- und Interrupt-Kommandos sowie die Spezialbefehle RRD, RLD und NEG zur Verfügung stellt.

**Erweiterungstabelle DDCB (3 Byte Operation Code)**

Diese Tabelle ist nur mit einem Achtel aller möglichen Kodierungen belegt. Sie führt dieselben Kommandos wie CB aus, allerdings wird hier nicht ein Register verändert, sondern eine Speicherstelle, die indiziert adressiert wird, und zwar mit dem Indexregister IX.

**Erweiterungstabelle FECB (3 Byte Operation Code)**

Diese Befehlsmatrix bietet dieselben Funktionen wie DDCB, allerdings jetzt für das Indexregister IY.

Nach soviel Systematik und Unterteilung wenden wir uns nun der Praxis zu und schauen uns die Wirkung der einzelnen Befehle an. Dazu sollten Sie unbedingt den Monitor geladen haben und Befehl für Befehl, Beispiel für Beispiel durchspielen, um sich mit den Funktionen und den manchmal auch etwas ungewohnten Nebenwirkungen schnell auseinanderzusetzen und vertraut zu machen.

Wir wollen nacheinander die Befehle zum Datentransfer, den Bereich Logik - Arithmetik, die Sprungbefehle und am Ende auch die Steuerkommandos analysieren. Jede Befehlsgruppe läßt eine Unterteilung in weitere Untergruppen zu, die nacheinander beschrieben werden. Am Ende jeder Untergruppe stehen einige Beispielprogramme und Befehle, die Sie mit dem Monitor austesten sollten, um sich mit den Befehlen näher vertraut zu machen.

Am Ende jedes Unterkapitels, das heißt am Abschluß jeder Befehlsgruppe, finden sich dann noch ein oder mehrere Anwendungen, typischerweise aus dem ROM-LISTING des CPC, die die Funktionsweise der einzelnen Befehle illustrieren und den Ablauf bestimmter Grundarbeitsweisen im Computerinnern demonstrieren. Wir beginnen mit den Befehlen zum Datentransfer.

## 5.3 Datentransfer

Bei der Beschreibung der Datentransferbefehle kann man zwei grundsätzliche Unterteilungen vornehmen: Zum einen eine Unterteilung nach der Nähe der Befehle zur CPU. Danach unterscheidet man:

### Interner Datentransfer

Dieser findet zwischen Registern beziehungsweise zwischen Registerpaaren der CPU statt. Externe Bausteine sind nicht beteiligt.

### Erweiterter interner Datentransfer

Dieser beinhaltet drei weitere Datenquellen beziehungsweise Zieladressen, die von der CPU mittels verkürzter und damit relativ schneller Befehle bearbeitet werden können. Es sind dies die durch das Registerpaar HL adressierte Speicherstelle, die aktuelle Position auf dem Stack, das heißt die durch das Registerpaar SP adressierte Speicherstelle und unmittelbar dem Befehlscode nachfolgende Daten. Alle drei Varianten greifen dabei zwar auf den Speicher, typischerweise das RAM, zurück; ihr Vorteil liegt jedoch in der Kürze des Befehlscodes und in der schnelleren Ausführungszeit. Sie stellen somit ein Mittelding zwischen dem internen und dem externen Datentransfer dar.



### Externer Datentransfer

Dieser existiert in zwei Varianten. Der externe Datentransfer zwischen Registern und Speicher oder zwischen Registern und einem Ein-/Ausgabebaustein.

Eine weitere Unterteilung ist nach der Datenmenge möglich. Es können 8 Bit, 16 Bit oder ein Datenblock von spezifizierter Länge mit einem Datentransferbefehl behandelt werden. Daneben kommen für die Daten zwei grundsätzliche Operationen in Frage:

**Ladevorgänge:** Bei diesen nimmt der Inhalt des Zielregisters den Wert der Quelldaten an. Ziel und Quelle können dabei sein: interne Register beziehungsweise Registerpaare, Speicherbausteine, Eingabe-/Ausgabebausteine beziehungsweise deren Register.

**Austauschvorgänge:** Bei diesen werden zwei Registerpaare beziehungsweise eine Gruppe von Registerpaaren ausgetauscht. Während bei den Ladevorgängen die Inhalte des Zielspeichers mit dem Quellspeicher übereinstimmen, finden sich nach einem Austauschvorgang die Werte eines Registerpaares im jeweils anderen wieder.

Für unsere weitere Darstellung werden wir den internen und den erweiterten internen Datentransfer zum internen Datentransfer zusammenfassen und nur noch von einer Zweiteilung ausgehen.

### 5.3.1 Interner Datentransfer

Beim internen Datentransfer können wir zwischen zwei Datenmengen unterscheiden. 8-Bit-Datentransfer und 16-Bit-Datentransfer. Die zugehörigen Befehle lauten:

**KOMMANDOS:**

- LD (LOAD) für Ladevorgänge
- EX (Exchange) für Austauschvorgänge
- PUSH und POP für die Zusammenarbeit von Registern mit dem Stack

Der Ladebefehl hat das Format LD Ziel, Quelle, wobei Ziel und Quelle ein Universalregister oder einen 8-Bit-Datenwert darstellen.

**FLAGS:**

Grundsätzlich beeinflussen die Befehle des internen Datentransfers die Flags nicht. Ausnahmen: LD A,I
LD A,R
und die Blockladebefehle.

### 5.3.1.1 8-Bit-Ladebefehle: Laden zwischen Universalregistern/(HL)

**Befehl**: LD Ziel, Quelle

Es wird also ein Universalregister oder (HL) mit dem Inhalt eines anderen Universalregisters (HL) geladen. Der Code ist dabei bitorientiert. Er Lautet

01aaabbb

Dabei stellen aaa und bbb jeweils einen 3-Bit-Registercode dar. Die Zuordnung ist dabei wie folgt:

000 Register B
001 Register C
010 Register D
011 Register E
100 Register H
101 Register L
110 Die durch HL adressierte Speicherstelle <(HL)>
111 Register A (AKKUMULATOR)

Einen Ladebefehl kann man nun erzeugen, indem man die Bit 7 und 6 des Befehlscodes auf 01 setzt und anstelle der Symbole aaa und bbb die entsprechenden Registercodes einfügt.

**Beispiel**: Laden des Registers B mit dem Inhalt von C.
Kürzel : LD B,C
Binärer Code: 01 000 001



Für diese Art der Darstellung gibt es allerdings eine Einschränkung. Der Befehl LD (HL),(HL) existiert nicht. Dieser Code wird durch den Steuerbefehl HALT (hex 76= binär 01110110) belegt. Mehr dazu bei den Steuerkommandos in Kapitel 5.6.

Wir können diese Befehle auch in einer Matrix aus 8 x 8 Stellen darstellen. Diese Matrix enthält auf der einen Achse das Zielregister, auf der anderen die Quelldaten. Sie finden sie in Tabelle 5.1. Neben den 63 Ladebefehlen für die Universalregister existieren noch vier weitere 8-Bit-Ladebefehle mit der Adressierungsart Implizit. Alle liegen in der Erweiterungstabelle ED. Es sind dies mit ihren hex-Codes:

| | |
|---|---|
| LD R,A | ED4F |
| LD A,R | ED5F |
| LD I,A | ED47 |
| LD A,I | ED57 |

Diese Kommandos ermöglichen einen Datenaustausch zwischen dem Akkumulator und den Spezialregistern. Auf die Bedeutung dieser Register haben wir schon an anderer Stelle (Kapitel 4.3) hingewiesen. Mehr zum Register I und seiner Funktion findet sich darüber hinaus bei den Steuerbefehlen Kapitel 5.6, Befehl IM2. Da diese Befehle für den Normalbenutzer kaum nutzbar sind, wollen wir hier nicht näher darauf eingehen.

**Tabelle 5.1**: Die 8-Bit-Registerladebefehle

| | | Quelle | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | A | B | C | D | E | H | L | (HL) |
| | A | 7F | 78 | 79 | 7A | 7B | 7C | 7D | 7E |
| | B | 47 | 40 | 41 | 42 | 43 | 44 | 45 | 46 |
| | C | 4F | 48 | 49 | 4A | 4B | 4C | 4D | 4E |
| Ziel | D | 57 | 50 | 51 | 52 | 53 | 54 | 55 | 56 |
| | E | 5F | 58 | 59 | 5A | 5B | 5C | 5D | 5E |
| | H | 67 | 60 | 61 | 62 | 63 | 64 | 65 | 66 |
| | L | 6F | 68 | 69 | 6A | 6B | 6C | 6D | 6E |
| | (HL) | 77 | 70 | 71 | 72 | 73 | 74 | 75 | -- |

### 5.3.1.2 Laden eines Registers/(HL) mit einem Absolutwert

**Befehl**: LD Register,<Wert>; 2-Byte-Befehl; OP-Code: 00aaa110

**Funktion**: Der im zweiten Byte spezifizierte Wert wird in das durch aaa angegebene Universalregister Übertragen. Für aaa gelten dabei dieselben Registercodes wie schon beim impliziten Laden.
Also:

| LD B | LD C | LD D | LD E | LD H | LD L | LD (HL) | LD A |
|---|---|---|---|---|---|---|---|
| 06 | 0E | 16 | 1E | 26 | 2E | 36 | 3E |

**Beispiel**: Sie sollten nun einmal den Monitor zur Hand nehmen beziehungsweise die Routine GOMACH, falls Sie diese noch nicht eingebaut haben, und einige Befehle in ihrer Funktionsweise erproben. Beginnen wir einmal mit C9. Dieser Befehl führt uns zu der schon bekannten Registerausgabe. Wir wollen nun unser Programm etwas erweitern, und zwar zunächst auf zwei Byte, die Kommandos 47 und C9. C9 ist dabei wiederum der Rücksprung. 47 ist, wie Sie Tabelle 5.1 entnehmen können, der Befehl

```
LD B,A
```

Wenn Sie sich nach dem Programmdurchlauf nun den Registersatz ansehen, werden Sie die Verschiebung relativ leicht feststellen. Statt der gewohnten 20 findet sich eine 00 in Register B. Der Registerinhalt von A wurde nach B übertragen.

**2.Beispiel**: Wir wollen nun einmal Register C mit einem Absolutwert laden. Als Beispiel soll dabei der Wert 80 hex dienen. Dies läuft wie folgt. Zunächst benötigen wir den Code für das Laden des Registers, dieser ist 0E. Als zweites Byte müssen die Daten folgen, in unserem Fall steht hier also eine 80. Das Programmende bildet wieder der RETURN-Befehl (C9). Nach dem Durchlauf dieses Programms (Eingabe ab 42000 und Programmstart bei 42000) erhalten Sie in Register C die erwartete Änderung. Es hat den Wert 80 angenommen. Um sich mit den Ladebefehlen und auch mit der Funktionsweise von GOMACH noch etwas näher vertraut zu machen, sollten Sie das Ganze jetzt mit einigen anderen Kombinationen durchspielen.



### 5.3.1.3 Interner Datentransfer mit 16 Bit

Der Datentransfer zwischen Registerpaaren erfolgt in zwei Varianten, als Lade- und als Austauschvorgang. Es existieren dabei Befehle für das Laden des Stack-Pointers aus einem Registerpaar sowie für das unmittelbare Laden eines Registerpaares mit einem nachgestellten Datenwert. Darüber hinaus sind Austauschvorgänge zwischen Registerpaaren möglich.

#### 5.3.1.3.1 Laden eines Registerpaares mit einem absoluten Wert

Dies ist für BC, DE, HL, SP, IX und IY möglich. Dabei nimmt das Registerpaar den nachfolgenden 2-Byte-Wert an. Das erste Datenbyte wird dabei als LOWER BYTE betrachtet, geht also in das niederwertige Register, das zweite geht in das höherwertige Register. Die Codes sind wie folgt:

| | |
|---|---|
| LD BC | 01 aa bb |
| LD DE | 11 aa bb |
| LD HL | 21 aa bb |
| LD SP | 31 aa bb |
| LD IX | DD 21 aa bb |
| LD IY | FD 21 aa bb |

#### 5.3.1.3.2 Datenaustausch zwischen Stack und einem Registerpaar

Hier gibt es zwei verschiedene Gruppen. Zunächst einmal das Laden des Stack-Pointers aus einem Universalregisterpaar. Dies ist mit den Registern IX, IY und HL möglich. Die Kommandos lauten:

| | |
|---|---|
| LD SP,IX | DD F9 |
| LD SP,IY | FD F9 |
| LD SP,HL | F9 |

Die zweite Gruppe bilden die Befehle, die ein Registerpaar auf dem Stapel ablegen oder von diesem wieder laden. Das Kommando für das Ablegen heißt

PUSH

für das Wiederholen, das heißt das erneute Laden eines Registerpaares aus dem Stapel

POP

Die Operationen sind mit den folgenden Registerpaaren möglich: AF, BC, DE, HL, IX, IY. Die einzelnen Kommandos lauten:

| Registerpaar: | AF | BC | DE | HL | IX | IY |
|---|---|---|---|---|---|---|
| POP | F1 | C1 | D1 | E1 | DD E1 | FD E1 |
| PUSH | AF | BC | DE | HL | IX | IY |
| | F5 | C5 | D5 | E5 | DD E5 | FD E5 |

#### 5.3.1.3.3 Datenaustausch zwischen Registerpaaren/(SP)

Neben diesen Kommandos existieren eine Reihe von Befehlen, die den Datenaustausch zwischen dem Inhalt der durch SP adressierten Adresse und einem Universalregisterpaar beziehungsweise zwischen Universalregisterpaaren ermöglichen. Das Kommando hierfür lautet: EX (=Exchange, Vertauschen). Es sind dies mit ihren Codes:

| | |
|---|---|
| EX DE,HL | EB |
| EX (SP),HL | E3 |
| EX (SP),IX | DD E3 |
| EX (SP),IY | FD E3 |
| EX AF,AF' | 08 |
| EXX | D9 |

Die letzten beiden Kommandos beziehen sich dabei auf den Austausch zwischen Registern und Parallelregistern. Wir haben schon bei der Beschreibung des Registeraufbaus der CPU in Kapitel 4 gesehen, daß das Akkumulator- und auch die Universalregister in doppelter Ausfertigung als Zwillingsregister vorhanden sind. Mit dem Befehl

EX AF,AF'



ist es möglich, den Akkumulator und das Statusregister gegen das Zwillingspärchen auszutauschen. Es wird also der alte Akkumulator in den Parallelakkumulator geladen und umgekehrt, dasselbe wiederholt sich für die FLAGs. Noch stärker ist der Befehl **EXX**. Er tauscht die Registerpaare BC, DE und HL gegen ihre jeweiligen Parallelregister aus. Ein einzelner Austausch zwischen Register und Parallelregister ist nicht möglich. Der Parallelregistersatz wird von der Routine GOMACH nicht angezeigt, was seinen Grund darin hat, daß der CPC ja eine Reihe von hochwichtigen Systemvariablen, wie zum Beispiel die Speicherbereichsverteilung, den Bildschirm-MODE etc. zwischengespeichert hat, die er permanent für die Speicheradressierung benötigt. Änderungen im Parallelregistersatz sind daher nur für den Profi geeignet und bedürfen einer ausgezeichneten Kenntnis der internen Abläufe in der Maschine. Daher haben wir in diesem Buch diese Möglichkeiten weggelassen.

**Beispiele:**

Um sich mit den Funktionen der 16-Bit-Ladebefehle vertraut zu machen, können Sie wie bei den 8-Bit-Ladebefehlen vorgehen. Wir wollen hier nur zwei Befehle besonders besprechen. Als erstes kommen wir zu EX DE, HL. Wenn Sie die Codes EB und das C9 für RETURN nacheinander eingeben, werden Sie den Austausch der Universalregisterpaare im Ausgabefeld von GOMACH sehen.

Auch das Laden eines Registerpaares dürfte keine Schwierigkeit darstellen. Wir wollen einmal das Registerpaar BC mit 77 88 laden. Die Codes dafür sind: 01 88 77 und C9 für die RÜckkehr (warum wurden die Werte 77 und 88 vertauscht?) Das Ausgabefenster von GOMACH zeigt uns nun die gewünschten Werte. Die Funktion des Stack-Pointers und das Arbeiten mit dem Parallelregistersatz wird an einem Beispiel am Ende von Kapitel 5.3 intensiv besprochen.

### 5.3.2 Externer Datentransfer

**Funktion:** Mit den Befehlen des externen Datentransfers ist es möglich, Register und Registerpaare mit Daten aus dem Speicher (RAM oder ROM) zu laden oder diese wegzuschreiben. Daneben ist ein Datentransfer zu oder von externen Geräten möglich. Die vorhandenen Befehle lauten:

LD (LOAD) - für die Kommunikation zwischen Speicher und Prozessor
IN - für das Einlesen von Daten von einem Ein-Ausgabegerät
OUT - für die Übergabe von Daten an ein IO-Gerät

Beim Speichern spricht man von Adressen, die die Daten enthalten. Die empfangende oder sendende Stelle eines Ein-/Ausgabegerätes wird als PORT bezeichnet.

**DATENFORMATE:**

8-Bit- und 16-Bit-Werte für den Verkehr zwischen Speicher und CPU, 8-Bit-Werte für den Datenaustausch zwischen IO und CPU. Daneben ist mit speziellen Blockkommandos die Übertragung eines Datenbereichs möglich.

**FLAGS:**

Das Statusregister wird generell von den Befehlen des externen Datentransfers nicht beeinflußt. Ausnahmen: der Befehl IN reg,(C)
und die Blockbefehle.

### 5.3.2.1 Externer Datentransfer zwischen Register und Speicher

**Datenaustausch zwischen einer indirekt adressierten Speicherstelle und dem Akkumulator.**

Hierzu stehen die Registerpaare BC, DE und das schon vom internen Datenaustausch bekannte Registerpaar HL zur Verfügung. Daneben ist es auch möglich, A aus einer fix vorgegebenen Stelle NN zu laden beziehungsweise in diese abzulegen. Damit ergeben sich die folgenden Kommandos:

| Akkumulator an Speicher | Speicher an Akkumulator |
|---|---|
| LD (BC),A 02 | LD A,(BC) 0A |
| LD (DE),A 12 | LD A,(DE) 1A |
| LD (HL),A 77 | LD A,(HL) 7E |
| LD (nn),A 32 aa bb | LD A,(nn) 3A aa bb |



**Laden einer indirekt adressierten Speicherstelle mit einem Datenbyte**

Hier sind drei Möglichkeiten gegeben. Die zu benutzende Speicherstelle kann durch den Inhalt des Registers HL oder indiziert durch IX beziehungsweise IY definiert werden. Dabei stellt der Parameter dd eine Distanzadresse dar. Enthält also IX beispielsweise den Wert A000 und ist dd = 20 hex, so wird die Speicherstelle A020, das heißt die Summe aus Register X plus der Distanzadresse benutzt. dd und nn stellen Symbole für die hex-Codes der Distanzadresse und des zu ladenden Datenbytes dar. Die Befehle lauten:

| Befehl | Code |
|---|---|
| LD (HL),n | 36 nn |
| LD (IX+d),n DD | 36 dd nn |
| LD (IY+d),n FD | 36 dd nn |

**Datenaustausch zwischen einem Universalregister und einer indirekt indiziert adressierten Speicherstelle**

Als Universalregister sind hier alle uns schon bekannten Register erlaubt. Eine Ansprache der Speicherstelle (HL) ist allerdings nicht möglich. Die Ladebefehle benötigen drei Byte, das erste Byte DD oder FD gibt an, ob das Register IX oder IY benutzt wird. Das zweite Byte decodiert das anzusprechende Register. Als drittes Byte folgt die Distanzadresse, der OFFSET,

| | |
|---|---|
| LD r,(IX+d) | DD 01aaa110 dd |
| LD r,(IY+d) | FD 01aaa110 dd |
| LD(IX+d),r | DD 01110aaa dd |
| LD(IY+d),r | FD 01110aaa dd |

mit den folgenden Registercodes:

| | |
|---|---|
| A 111 | E 011 |
| B 000 | H 100 |
| C 001 | L 101 |
| D 010 | |

| | A | B | C | D | E | H | L | |
|---|---|---|---|---|---|---|---|---|
| DD/FD | 77 | 70 | 71 | 72 | 73 | 74 | 75 | dd |

**Datenaustausch zwischen zwei aufeinanderfolgenden Speicherstellen und einem Registerpaar**

Als Registerpaare kommen alle internen Register in Frage: BC, DE, HL, SP, IX, IY. Die Arbeitsweise ist dabei wie folgt: Jeder Befehl besteht aus vier Byte. Das Arbeiten mit HL ist mit einem verkürzten Befehl, der nur einen Befehlscode benötigt, auch mit drei Byte möglich. Hier existieren also zwei in ihrer Funktion identische Kommandos. Die Ablage und das Auslesen erfolgen im Format LOW-HIGH. Der Befehl

LD (nn),bc

lädt also zuerst den Inhalt von Register C in die durch nn adressierte Speicherstelle. Danach wird B in die nächsthöhere Adresse geladen. Umgekehrt lädt der Befehl

LD DE,(nn)

Register E mit der durch nn angegebenen Speicherstelle, D enthält danach den Inhalt der nächsthöheren Adresse. Es existieren die folgenden Kommandos:

Registerpaar an Speicherstelle Speicherstelle an Registerpaar

| Befehl | Code | Befehl | Code |
|---|---|---|---|
| LD(nn),BC | ED 43 aa bb | LD BC,(nn) | ED 4B aa bb |
| LD(nn),DE | ED 53 aa bb | LD DE,(nn) | ED 5B aa bb |
| LD(nn),HL | ED 63 aa bb<br>22 aa bb | LD HL,(nn) | ED 6B aa bb<br>2A aa bb |
| LD(nn),SP | ED 73 aa bb | LD SP,(nn) | ED 7B aa bb |
| LD(nn),IX | DD 22 aa bb | LD IX,(nn) | DD 2A aa bb |
| LD(nn),IY | FD 22 aa bb | LD IY,(nn) | DD 2A aa bb |

**Beispiel**: Sie sollten jetzt wieder GOMACH oder den Monitor zur Hand nehmen und einmal einige Befehle ausprobieren. Erste Anwendung: Wir wollen den Inhalt von VAR START, jener 2 Byte Adresse, die den Beginn der Variablen, also das Ende des BASIC-Programms kennzeichnet, in ein Registerpaar laden, um mit einer Maschinenroutine den Zugriff auf bestimmte Variable zu ermöglichen. Dieser POINTER ist im Bereich der BASIC-Firmware in den Adressen AE85 und AE86 abgelegt (vergleichen Sie dazu einmal Kapitel 1.3). Zielregisterpaar soll DE sein, womit wir auch den zugehörigen Befehl definiert hätten: **LD DE,(AE85)**



Unser Programm besteht aus fünf Byte: ED 5B 85 AE C9. Nach dem Durchlauf des Programms erhalten Sie nun im Ausgabe-WINDOW die folgende Anzeige:

| AF | BC | DE | HL | IX | IY | SP |
|---|---|---|---|---|---|---|
| 0068 | 20FF | XXXX | 0560 | BFFE | B0C2 | BFF8 |

Die X stehen dabei stellvertretend für den Inhalt des Variablenpointers. Sie können die richtige Funktion des Programms überprüfen, indem Sie sich einmal mit

PRINT HEX$(PEEK(AE85)) und PRINT HEX$(PEAK(AE86))

den Inhalt der beiden Speicherstellen kontrollhalber ausgeben lassen.

### 5.3.2.2 Blockladebefehle

Der Z80 Prozessor verfügt über vier sehr starke Kommandos, die es ermöglichen, Speicherbereiche im Variablenspeicher zu verschieben. Der Ablauf ist dabei wie folgt: Zwei Registerpaare (DE und HL) enthalten die Adressen. HL enthält dabei die zu lesende Adresse, DE den Speicherplatz, an den die Daten geschrieben werden sollen. Die Anzahl der zu verschiebenden Bytes ergibt sich aus dem Wert des Registerpaares BC.

Der Computer beginnt also, indem er den Inhalt der durch HL adressierten Speicherstelle liest, und diesen an die durch DE gegebene Adresse speichert. Danach wird das Registerpaar BC um 1 vermindert. BC fungiert also bei dieser Operation als Zähler. Der weitere Ablauf hängt nun von der Art des Befehls ab. Es existieren die Kommandos

| | | |
|---|---|---|
| -LD I | (lade- und inkrementiere) | ED A0 |
| -LD D | (lade- und dekrementiere) | ED A8 |
| -LD IR | (lade- und inkrementiere wiederholt) | ED B0 |
| -LD DR | (lade- und dekrementiere wiederholt) | ED B8 |

Bei LD I werden die Registerpaare DE und HL jeweils um 1 erhöht, BC um 1 vermindert. Mit dieser Änderung ist der Befehl abgeschlossen. Es erfolgt also ein einmaliges Umspeichern und eine Veränderung der Zeiger und des Zählregisterpaares. Bei LD DR werden alle drei Registerpaare um 1 vermindert. Der Unterschied zwischen Dekrementieren und Inkrementieren besteht also in der Richtung, in der abgespeichert wird. Bei LD D und

LD DR beginnt man mit der obersten Speicherstelle und tastet sich dann nach unten weiter vor. Bei LD I und LD IR läuft das Ganze in der umgekehrten Richtung.

Der Unterschied zwischen den einfachen Befehlen und denen mit angehängtem R besteht nun darin, daß die REPEAT-Befehle den einfachen Vorgang so lange wiederholen, bis BC den Wert 0 angenommen hat. LD I und LD D leisten also eine Verlagerung von einem Byte mit nachfolgendem Umsetzen der POINTER und des Zählers, LD IR und LD DR verschieben einen ganzen durch seine Länge (BC und Start- bzw. Endadresse HL) spezifizierten Datenblock auf die neue Anfangs- beziehungsweise Endadresse DE.

Dabei ist natürlich zu beachten, daß das Schreiben von Daten nur in dem RAM-Speicher vonstatten gehen kann. Ein Lesen dagegen sowohl aus ROM wie auch aus RAM.

Da der CPC in wesentlichen interessanten Bereichen, nämlich den oberen und den unteren 16 K des RAM ständig zwischen ROM und RAM umherschaltet, existieren im Betriebssystem 2 nützliche Routinen, die die Beschäftigung mit dieser Problematik überflüssig machen. Es handelt sich um die KERNEL-Routinen KL LDIR und KL LDDR mit den Adressen B91B beziehungsweise B91E. Diese führen den LDIR beziehungsweise LDDR-Befehl bei gesperrten ROMs aus. Es ist also RAM im ganzen Speicher adressiert.

**Beispiel:** Als Anwendungsbeispiel wollen wir uns einmal eine Verschiebung des Grafikspeichers vornehmen. Wir hatten schon bei der Beschäftigung mit dem Videocontroller in Kapitel 3 gesehen, daß es möglich ist, mit SCR SET BASE die Basisadresse des Bildschirmspeichers zu verschieben und so zwischen zwei Bildschirmen zu switchen. Der zweite Bildschirm muß dabei ab Adresse 4000hexj nach oben liegen, um Kollisionen mit den weiter unten liegenden RESTARTs beziehungsweise der Firmwaresprungtabelle im Bereich B000 bis C000 zu vermeiden.

Wir wollen nun einmal eine vollständige Kopie des Grafikspeichers in diesem Speicherbereich anlegen. Dazu verschieben wir als erstes mit dem Befehl MEMORY die höchste mögliche Speicheradresse auf 3FFF.

Um nun eine Verschiebung mit dem Befehl LD IR durchzuführen, müssen wir wie folgt vorgehen. Die Anfangsadresse des alten Bereichs gehört in HL. Dazu benutzen wir am einfachsten den Befehl



```
LD HL,C000
```

Analog dazu nimmt DE den Anfang des neuen Bereichs (hex 4000) auf. Die Anzahl der zu übertragenden Bytes beträgt genau ein Viertel des verfügbaren Speichers, 16 K oder in hex ausgedrückt 4000. Diese Anzahl speichern wir als Zählwert in Register BC. Das Kommando

```
LD IR
```

und der RETURN-Befehl schließen unser kleines Maschinenprogramm ab.

```
LD HL,C000      21 00 C0
LD DE,4000      11 00 40
LD BC,4000      01 00 40
LDIR            ED B0
RET             C9
```

Wir werden das Programm in zwei Stufen durchführen. In der ersten fügen wir nach dem Laden von BC ein RETURN ein. Dies führt dazu, daß uns zunächst einmal die Registerinhalte angezeigt werden. Wenn Sie diesen Teil des Programms durchlaufen lassen, gibt GOMACH die folgende Ausgabe:

| AF | BC | DE | HL | IX | IY | SP |
|---|---|---|---|---|---|---|
| 0068 | 4000 | 4000 | C000 | BFFE | B0C2 | BFF8 |

Wir geben nun ab Speicherstelle 42000 das Gesamtprogramm ein. Ganz Clevere können auch erst ab Speicherstelle 42009 beginnen, da ja die ersten neun Byte identisch sind. Bei diesem Beginn wird das C9 als erstes Byte überschrieben. Es brauchen dann nur noch die drei übrigen Codes eingegeben zu werden. Ansprungstelle bleibt weiterhin 42000. Nach dem Durchlauf des Programms erhalten Sie folgende Ausgabe von GOMACH:

| AF | BC | DE | HL | IX | IY | SP |
|---|---|---|---|---|---|---|
| 0040 | 0000 | 8000 | 0000 | BFFE | B0C2 | BFF8 |

Um die korrekte Funktion zu überprüfen, sollten Sie sich jetzt schnell einmal die Inhalte von C000 und 4000 mit dem PEEK-Kommando und gegebenenfalls hex$ ausgeben lassen. Wenn Sie GOMACH direkt nach dem Einschalten beziehungsweise nach Systemreset geladen hatten und somit noch keine Verschiebung des Bildschirm-OFFSETs (vergleiche Kapitel 3) stattgefunden hat, müßten Sie jetzt in beiden Fällen die Ausgabe 0 erhalten. Sie können sich natürlich auch, wie in Kapitel 3 beschrieben, mit SCR SET

BASE den neuen Bildschirmspeicher adressieren. Hier noch einmal das dazu notwendige Programm:

```
LD A,40          3E40
CALL SET BASE    CD 08 BC
RET              C9
```

Wenn Sie dieses Programm mit GOMACH durchfahren, erhalten Sie im oberen Bildschirm-Window die übliche Ausgabezeile. Diese wurde bereits nach der Umschaltung in den neuen Bildschirmspeicher geschrieben. Etwas verwundern werden Sie dabei möglicherweise die Werte für A und HL. Dies ist einer der unangenehmen Nebeneffekte von SCR SET BASE. Sie zerstört den Inhalt dieser Register. In den anderen Registern müßten Sie dagegen die alten Werteingaben des zuerst eingegebenen Maschinenprogramms mit dem LD-IR-Befehl vorfinden.

Natürlich ist es auch kein Problem, wieder zurück in den normalen Grafikspeicher zu switchen. Dazu ersetzen Sie einfach in dem Programm die 40 durch eine C0, lassen das Ganze noch einmal laufen. Sie bekommen nun wieder die eben eingegebene Ausgabe zurück auf den Schirm. Wenn sie nun einmal mit der nachfolgenden FOR-TO-Schleife den neuen Bildschirmspeicher wieder löschen, bekommen Sie auch einen guten Eindruck davon, was Maschinenspracheprogrammierung leisten kann. Während das Umschieben mit dem LD IR-Befehl wie der Blitz abgelaufen ist, die Verzögerungen ergaben sich durch die nachfolgenden BASIC-Kommandos, so werden Sie jetzt einige Zeit auf das Setzen Ihrer 16384 Speicherstellen warten dürfen.

```
FOR I=16384 TO 32767:POKE I,0:NEXT I
```

Wir waren bei unserer BASIC-Schleife aber unfair,denn wir haben immer einen konstanten Wert (0) gepoket. Bei einer genauen Verschiebung müßten wir mit PEEK an die Stelle I den Inhalt von PEEK(I+32768) ablegen.

### 5.3.2.3 Datentransfer zwischen CPU-Registern und Ein-/Ausgabegeräten

Der Z80-Befehlsvorrat enthält eine Reihe von Ein-/Ausgabekommandos, die die Kommunikation zwischen Universalregistern und externen Bausteinen ermöglichen. Es existieren Befehle sowohl für die Ausgabe einzelner Bytes als auch für den Datentransfer eines ganzen Speicherbereichs zu einem externen Gerät. Alle Kommandos bauen auf den Befehlen OUT und IN auf. Der OUT-Befehl hat das Format



OUT PORT,Register

Der IN-Befehl umgekehrt

IN Register, PORT

Im ersteren Fall wird also an die Adresse des externen Gerätes, den PORT, der Inhalt des nachstehenden Registers übergeben beziehungsweise umgekehrt ein CPU-Register mit dem Inhalt eines angesprochenen PORTs geladen. Die PORT-Adresse kann dabei sowohl als Absolutwert wie auch durch den Inhalt des Registers C mitgegeben werden. Nach der Art der übergebenen Datenmenge können wir zwei Gruppen von Befehlen unterscheiden:

1. einfache Ausgabe eines 8-Bit-Wertes an ein externes Gerät beziehungsweise das Einlesen von diesem,

2. die Ein-/und Ausgabe von Speicherbereichen an ein Ein-/Ausgabegerät. Dabei fungiert die CPU im wesentlichen als Durchlaufregister, das heißt, sie liest den Inhalt einer Speicherstelle und gibt sie an die gewünschte PORT-Adresse des IO-Geräts weiter.

Grundsätzlich ist zu beachten, daß der Z80-Prozessor bei der Decodierung davon ausgeht, daß die Unterscheidung zwischen den verschiedenen Ein-/Ausgabegeräten anhand der Adreßleitung A0-A7 erfolgt. Die meisten Befehle sehen nur diese Decodierung vor, weshalb sie beim CPC nicht verwendet werden können, da dieser die Adreßleitung **A8-A15** zur Ansprache der einzelnen Bausteine benutzt. Von den möglichen 12 verschiedenen Kommandos des CPC, von denen eines allerdings noch für mehrere Universalregister gilt, können somit insgesamt nur zwei benutzt werden.

Es sind dies die Kommandos IN reg,(C) und OUT(C),reg. Der Ablauf Befehle ist wie folgt:

**IN reg,(C)**

Mit diesem Befehl wird der Inhalt eines PORTs des durch Register C adressierten Ein-/Ausgabegerätes gelesen und in das angegebene Register übertragen. Das Register C enthält dabei die Bit A0-A7 des Adreßbus, B die Bit A8-A15. Als anzusprechende Laderegister kommen alle Universalregister in Frage, die Codes sind dabei wie gewohnt. Der IN-Befehl besteht aus zwei Byte, dem Tabellencode ED und einem weiteren Byte der Form 01aaa000, welches das anzusprechende Register decodiert. "aaa" bedeuten

dabei wie üblich: A-111, B-000, C-001, D-010, E-011, H-100, L-101, so daß sich die folgenden Befehlscodes ergeben:

| | A | B | C | D | E | H | L |
|---|---|---|---|---|---|---|---|
| ED | 78 | 40 | 48 | 50 | 58 | 60 | 68 |

**FLAGs**: Neben den Blockbefehlen ist das Kommando IN reg,(C) der einzige Datentransferbefehl, der die FLAGs ändert. Je nach den Eigenschaften des eingelesenen Datenbytes (positiv, negativ, 0 etc.) werden die einzelnen FLAGs gesetzt.

**IN A,(n)**

Dieser Befehl liest ein Datenbyte aus dem durch den Absolutwert N spezifizierten PORT eines Ein-/Ausgabegerätes. N wird dabei als A0-A7 auf den Adreßbus gelegt. Der Akkumulator selber liefert A8-A15. Er ist also gleichzeitig oberes Adressenbyte und Eingaberegister. Beim CPC, bei dem die Bit 8 - 15 für die Ausgabegerätadressierung verantwortlich sind und die Bit 0 - 7 beliebig sind, darf also N einen beliebigen Wert enthalten, dafür aber muß das Register A mit der Adresse des Eingabegerätes geladen werden.

**OUT(C),reg**

Dieser Befehl gibt den Inhalt eines Universalregisters an die durch Register C spezifizierte PORT-Adresse aus. C liefert dabei die Bit A0-A7 des Adreßbus, Register B A8-A15. Der Befehl hat das Format ED 01aaa001.

"aaa" steht dabei stellvertretend für die Registercodes. Es können alle Universalregister verwandt werden (siehe IN reg,(C)). Somit ergeben sich die folgenden Byte-Codes:

| | A | B | C | D | E | H | L |
|---|---|---|---|---|---|---|---|
| ED | 79 | 41 | 49 | 51 | 59 | 61 | 69 |

**OUT(n),A**

Dieser Befehl gibt den Inhalt des Akkumulators an das periphere Gerät aus, welches durch das mitgelieferte Datenbyte adressiert wird. Dieses Kommando darf aber nicht beim CPC verwandt werden, da hier die Decodierung mit den höheren 8 Adreßleitungen erfolgt.



### 5.3.2.4 Blockausgabebefehle

Im Z80-Maschinensprachesatz existieren 8 Blockausgabebefehle, die es ermöglichen, Daten von einem externen PORT in den Speicher als Block einzulesen beziehungsweise in der umgekehrten Richtung an den PORT auszugeben. Der grundsätzliche Funktionsablauf ist dabei wie folgt:

Der Datenaustausch findet zwischen der durch HL adressierten Speicherstelle und der durch C bestimmten PORT-Adresse statt. Register B fungiert dabei als Zählregister (vgl. Blockladebefehle LD IR, LD DR). Nach dem einmaligen Ablauf des Befehls wird HL um 1 erhöht oder vermindert, B um 1 vermindert, und gegebenenfalls erfolgt eine Wiederholung bis B = 0. Die verschiedenen Befehlsvarianten entsprechen dabei den einzelnen Möglichkeiten, die wir schon beim Blockladen (Befehle LD DR, LD D, LD I, LD IR) kennengelernt haben. Im einzelnen gibt es die folgenden Kommandos:

| IN-Befehle | IN-Codes | OUT-Befehle | OUT-Codes |
|---|---|---|---|
| INI | EDA2 | OUTI | EDA3 |
| IND | EDAA | OUTD | EDAB |
| INIR | EDB2 | OTIR | EDB3 |
| INDR | EDBA | OTDR | EDBB |

Da B bei all diesen Befehlen für die Zählfunktion verwendet wird und sich deshalb während der Befehlsausführung die PORT-Adresse ändern würde, sind alle Blockbefehle beim CPC leider nicht benutzbar. Das Betriebssystem arbeitet daher im wesentlichen mit den Kommandos OUT(C),reg und IN reg,(C). Den typischen Ablauf einer Betriebssystemroutine, die ein externes Gerät anspricht, finden Sie dabei in den nachfolgenden Anwendungen.

**ANWENDUNG**: Das Umschalten zwischen verschiedenen Speicherbereichen im ROM und im RAM (Softswitching)

Wir haben uns schon des öfteren mit dem Umschalten zwischen festen und variablen Speichern (ROM und RAM) bei der Speicherbereichsverteilung beschäftigt. Wir wollen uns nun einmal anschauen, wie das Ganze vom Betriebssystem her abläuft. Schon in Kapitel 3 hatten wir gesehen, daß die Auswahl zwischen ROM und RAM beim Lesen durch das ULA vorgenommen wird. Dieser Baustein verfügt über mehrere Register, wovon eins, das Multifunktionsregister, neben dem Bildschirmmodus in 2 Bit abgespeichert, auch die aktuelle Speicherkonfiguration enthält. Es sind dies Bit 2 und 3 des Multifunktionsregisters. Ein gesetztes Bit heißt, daß RAM adressiert ist,

ansonsten wird das ROM angesprochen. Die zur Umschaltung zwischen den verschiedenen Speicherbereichen benutzten Routinen B900, B903, B906, B909 haben wir schon kennengelernt. Mit der Routine B900, die den oberen ROM-Bereich einschaltet, wollen wir uns nun den konkreten Ablauf anschauen. Das Maschinensprache-LISTING der Routine ist relativ kurz. Es umfaßt nur 8 Befehle.

```
BA5E    F3      DI
BA5F    D9      EXX
BA60    79      LD A,C
BA61    CB 99   RES 3,C
BA63    ED 49   OUT (C),C
BA65    D9      EXX
BA66    FB      EI
BA67    C9      RET
```

Zunächst ein paar Vorbemerkungen zur Adressierung. Möglicherweise werden Sie sich wundern, daß die Routine ab BA5E hier aufgelistet ist. Dies ist der Platz, an dem sich die Routine wirklich befindet. In B900 liegt nur ein Sprungzeiger auf eben dieser Routine. Beim Ansprung von B900 wird also nach BA5E weitergesprungen und der RETURN-Befehl in BA67 bedeutet dann den Rücksprung zum aufrufenden Programm.

Zunächst der Programmanfang. In Position BA5E findet sich der Befehl DI (Dissable Interrupt). Dieses Steuerkommando weist die CPU an, keine Unterbrechungen von außen mehr zuzulassen (vergleiche Steuerbefehle Kapitel 5.5). Damit kann das nachfolgende Programm bis zum Befehl EI (Enable Interrupt) in BA66 ungestört von fremden Einflüssen ablaufen. Es folgt das Kommando EXX. Damit werden die Registerpaare BC, DE und HL mit ihren Parallelregistern ausgetauscht. Die Vordergrundregister werden vom CPC für vielfältige Zwecke benutzt. In den Hintergrundregistern hat er einige dauerhaft benötigte Systemdaten gespeichert. Beispiele dafür sind Bildschirmmodus und ROM- Verteilung, die als Kopie des Multifunktionsregisters des ULA in Register C permanent existieren. Die Adresse des Multifunktionsregisters im ULA befindet sich ebenso permanent in Register B'. Damit ist es möglich, mit dem Befehl

```
OUT(C),C
```

diese wichtigen Steuerdaten relativ schnell an das ULA zu übermitteln. Da der CPC aufgrund der Parallelschaltung von BASIC-Interpreter und Grafikspeicher, speziell bei der Ausführung von Grafikroutinen, häufig



zwischen oberem RAM und oberem ROM umschalten muß, wäre es viel zeitaufwendiger, wenn der Wert des Multifunktionsregisters erst eingelesen oder gar aus einer Speicherstelle in ein Universalregister übertragen werden müßte.

Vor dem OUT-Befehl befinden sich noch zwei Kommandos. Zur Datensicherung der alten Speicherbereichsverteilung dient das Kommando

```
LD A,C
```

Damit wird der Akkumulator mit der vorherigen Speicherbereichsverteilung geladen. Dabei ist zu beachten, daß es sich beim Register A um den normalen Akkumulator handelt, da beim Kommando EXX der Akkumulator und das Statusregister ja nicht mit vertauscht werden. Wir haben also nun im aktuellen Akkumulator die alte Speicherbereichsverteilung gespeichert. Die komplementäre Routine

```
ROM RESTORE
```

kann deshalb später auf diese Verteilung zurückgreifen. Die neue Aufteilung, die sich nach Rücksetzen des 3. Bits in Register C mit dem Kommando

```
RES 3,C
```

ergibt, kann dann (Speicherstelle BA61) an das ULA ausgegeben werden. Nochmaliges EXX bringt die Hintergrundregister wieder in Sicherheit, wonach der normale Programmablauf fortgesetzt werden kann. Die Speicherumschaltung ist erfolgt.

## 5.4 Arithmetik- und Logikbefehle

Der Z80-Befehlssatz stellt eine Reihe hocheffektiver Kommandos zur Verfügung, die es ermöglichen, Daten logisch und arithmetisch zu verknüpfen sowie bitorientierte Operationen auszuführen. Kommandos: Diese sind generell alle für die Behandlung von 8-Bit-Werten, teilweise aber auch im Bereich der 16-Bit-Arithmetik verfügbar. Es können vier Befehlsgruppen unterschieden werden:

**Arithmetikbefehle:** Diese Kommandos verknüpfen Register A beziehungsweise bei 16 Bit Operationen eines der Registerpaare HL, IX oder IY mit einem ein- oder zwei Byte Datenwert, der direkt als Inhalt eines Registers oder als indirekt indizierte Speicherstelle mitgegeben werden kann. Die Befehle Increment und Decrement sind darüber hinaus für alle Universalregister und Registerpaare verfügbar. Es können Addition und Subtraktion sowie davon getrennt das Erhöhen und Vermindern um 1 ausgeführt werden.

**Logikbefehle:** Diese Befehle stellen eine logische Verknüpfung zwischen dem Wert des Akkumulators und einem durch ein weiteres Register oder absolut vorgegebenen 8-Bit-Wert her. Das Ergebnis befindet sich dabei im Akkumulator. Verfügbar sind die logischen Verknüpfungen AND, OR, XO sowie die Vergleichsoperation CP (für Compare = Vergleiche).

**Verschiebebefehle:** Mit diesen ist es möglich, ein Universalregister oder eine indirekt indiziert adressierte Speicherstelle mit oder ohne Berücksichtigung des CARRY-Bits zu rotieren beziehungsweise zu verschieben. Diese Operationen sind dabei in beiden Richtungen (nach rechts und nach links) durchführbar.



**Bitbefehle :** Mit diesen Kommandos ist es möglich, einzelne Bits eines Universalregisters oder einer indirekt indiziert adressierten Speicherstelle zu setzen, rückzusetzen oder auf ihren momentanen Zustand zu überprüfen. Dabei umfaßt der Z80-Befehlssatz sowohl Kommandos für die Behandlung von 16-Bit-Werten als auch das Arbeiten von 8-Bit-Daten.

Neben diesen Befehlen existieren im Bereich der Arithmetik- und Logikkommandos noch einige Spezialbefehle für den Akkumulator und das Statusregister. Beim Register A handelt es sich dabei im wesentlichen um Kommandos, die mit der BCD-Darstellung zu tun haben, sowie um den Befehl NEG. Darüberhinaus ist das Setzen beziehungsweise Komplementieren des CARRY-FLAGs möglich.

**FLAGs:**

Das Statusregister F wird grundsätzlich durch alle Befehle aus der Arithmetik- Logikgruppe nach dem Endergebnis gesetzt. Aber eine Ausnahme bilden die Bitmanipulationskommandos SET und RESET sowie das Inkrementieren und Dekrementieren von Registerpaaren, wobei hier auch IX und IY als Registerpaar zu betrachten sind. In diesen Fällen wird F nicht verändert.

### 5.4.1 Die Arithmetikbefehle

Mit sechs verschiedenen Arithmetikbefehlen können vier grundsätzlich verschiedene Operationen ausgeführt werden:

| | |
|---|---|
| Addieren | ADD (Addiere ohne Berücksichtigung des C-Flags) |
| | ADC (Addiere mit Carry) |
| Subtrahieren | SUB (Subtrahiere ohne Berücksichtigung des Carry-Flags) |
| | SBC (Subtrahiere mit Carry) |
| Inkrementieren | INC (Erhöhen um 1) |
| Dekrementieren | DEC (Vermindern um 1) |

Die Kommandos ADC und SBC führen eine Addition bzw. Subtraktion mit Berücksichtigung des CARRY-Bits durch. Bei ADC wird bei gesetztem CARRY eine 1 in der niedrigsten Stelle, das heißt zu Bit 0 addiert, bei SBC

analog dazu eine 1 in der niedrigsten Stelle subtrahiert. Unabhängig davon setzen die vier Kommandos ADD, ADC, SUB und SBC das CARRY-FLAG, falls ein Übertrag (positiv oder negativ) erfolgen muß.

INC und DEC greifen auf ein Register oder Registerpaar oder eine der indirekt beziehungsweise indirekt indiziert adressierten Speicherstellen (HL), (IX+D), (IY+D) zurück.

Die **8-Bit-Arithmetikbefehle** können als Daten den Inhalt eines Registers, ein mitgeliefertes Datenbyte oder den Inhalt einer indirekt oder indiziert angesprochenen Adresse im MEMORY benutzen. Das Endergebnis steht dabei immer im Akkumulator. Eine Übersicht über die gesamten Befehle zur 8-Bit-Arithmetik findet sich am Ende der Logikbefehle.

Bei der **16-Bit-Arithmetik** müssen wir zwischen HL auf der einen Seite und IX, IY auf der anderen Seite unterscheiden. Alle drei fungieren als 16-Bit-Zielregister. Der Befehl

ADD

ist für alle drei Registerpaare verfügbar. Dieser addiert zwei Registerpaare, ohne Berücksichtigung des CARRYs. Nur mit dem Registerpaar HL ist es darüber hinaus möglich, auch eine Addition oder Subtraktion unter Berücksichtigung des Übertrags-FLAGs vorzunehmen. Der Befehl SUB (Subtraktion ohne Carry) ist nur im 8-Bit-Bereich vorhanden. Quellregisterpaare können BC, DE, HL sowie der Stackpointer (SP) sein. Diese Registerpaare können darüber hinaus noch mit den Befehlen INC und DEC bearbeitet werden.

Allen 16-Bit-Kommandos ist dabei gemeinsam, daß der Übertrag zwischen den niedrigeren 8 Bit und dem HIGHER-BYTE automatisch erfolgt. Das Carry wird gesetzt, wenn ein Übertrag in Bit 16 erfolgen müßte; das H-Flag zeigt einen Übertrag zwischen den Bits 11 und 12, also zwischen den beiden Halbbytes (Nibbles) der oberen 8 Bit an. Es ergeben sich somit die folgenden Kommandos:



**Tabelle 5.2**: 16-Bit-Arithmetik-Befehle

| | Quellregister | | | | | |
|---|---|---|---|---|---|---|
| Befehl | BC | DE | HL | SP | IX | IY |
| ADD HL, | 09 | 19 | 29 | 39 | | |
| ADD IX, | DD 09 | DD 19 | | DD 39 | DD 29 | |
| ADD IY, | FD 09 | FD 19 | | FD 39 | | FD 29 |
| ADC HL, | ED 4A | ED 5A | ED 6A | ED 7A | | |
| SBC HL, | 42 | 52 | 62 | 72 | | |
| INC | 03 | 13 | 23 | 33 | DD 23 | FD 23 |
| DEC | 0B | 1B | 2B | 3B | DD 2B | FD 2B |

### 5.4.2 Die Logikgruppe

Der Z80-Prozessor ste llt vier Befehle aus dem Logikbereich zur Verfügung, mit denen es möglich ist, die Verknüpfungen AND, OR und XOR zwischen 8-Bit-Werten herzustellen sowie zwei 8-Bit-Werte zu vergleichen. All diese Operationen laufen zwischen dem Akkumulator und einem 8-Bit-Wert ab, der durch ein Universalregister, als externes Byte oder durch HL indirekt, beziehungsweise durch IX und IY indiziert, adressiert werden kann. Alle Verknüpfungen laufen Bit für Bit ab. Es gelten dabei die folgenden Verknüpfungsregeln:

AND: Logische UND-Verknüpfung

Bei der logischen UND-Verknüpfung wird ein Bit des Ergebnisses dann gesetzt, wenn dasselbe Bit bei beiden INPUT-Größen gesetzt war. Ansonsten bleibt es 0. Es ergibt sich somit die folgende Wahrheitstabelle für jeweils 1 Bit:

Verknüpfung logisch UND

| | | ' Operand 1 ' | |
|---|---|---|---|
| | | 0 | 1 |
| Operand 2 | 0 | 0 | 0 |
| | 1 | 0 | 1 |

Das Durchführen dieser Operation für alle 8 Bit der beiden Operanden liefert das Endergebnis. Beispiel:

```
A:Inhalt:   33 00100001
B:Inhalt:   44 00101100
--------------------
Ergebnis:   32 00100000
```

Sie können sich das Wirken dieser Operation auch durch den BASIC-Befehl AND anschauen.

```
PRINT 33 AND 44
```

liefert uns das gesuchte Ergebnis 32.

**XOR**: Logische EXKLUSIV-ODER-Verknüpfung

Bei dieser Verknüpfungsart wird ein Bit im Ergebnis-Byte dann gesetzt, wenn die entsprechenden beiden Bits der Operanden unterschiedlich sind, ansonsten wird es rückgesetzt. Somit ergibt sich die folgende Wahrheitstabelle:

Verknüpfung logisch XOR

| | | ' Operand 1 ' | |
|---|---|---|---|
| | | 0 | 1 |
| Operand 2 | 0 | 0 | 1 |
| | 1 | 1 | 0 |

Eine Simulation des XOR-Befehls durch BASIC-Kommandos ist direkt nicht möglich, hier würde bereits ein kleines Programm benötigt werden. Wir können uns das Ganze allerdings in Maschinensprache mit GOMACH anschauen.



**Beispiel**: Wir laden den Akkumulator mit hex 55 und verknüpfen das Ergebnis dann mit Register B, das wie üblich beim Ansprung eines Maschinenprogramms den Wert 20 enthält. Ein Programm, welches dieses leistet, ist 4 Byte lang:

```
LD A, 55        3E 55
XOR B           A8
RET             C9
```

Sie erhalten dann als Ausgabe in Register A den Wert 75. Um dessen Zustandekommen zu erläutern, sollten Sie sich einmal mit

```
PRINT BIN$(&55,8),BIN$(&20,8),BIN$(&75,8)
```

die einzelnen Zahlen als Binärwerte ausgeben lassen. Wenn Sie nun an jede Stelle, an der die ersten beiden Zahlen nicht übereinstimmen, eine 1 setzen und im umgekehrten Fall eine 0, erhalten Sie die dritte Zahl, die XOR-Verknüpfung.

**OR**: Logische ODER-Verknüpfung

Bei dieser Verknüpfungsart werden die Bits des Ergebnisses dann gesetzt, wenn in einem der beiden Operanden das entsprechende Bit gesetzt ist. Es ergibt sich somit die folgende Wahrheitstabelle für jedes Bit:

Verknüpfung logisch OR

| | | ' Operand 1 ' | |
|---|---|---|---|
| | | 0 | 1 |
| Operand 2 | 0 | 0 | 1 |
| | 1 | 1 | 1 |

**Beispiel**: Um uns die Wirkung des OR-Kommandos anzuschauen, können wir wieder auf BASIC-Kommandos zurückgreifen. Der BASIC-Befehl OR stellt nichts anderes als ein erweitertes Maschinenspracheäquivalent dar. Lassen Sie sich einmal mit

```
PRINT &55 OR &20 und

PRINT BIN$(&55,8),BIN$(&20,8)BIN$(&75,8)
```

die zu untersuchenden Werte ausgeben. Das Endergebnis (&75) stimmt hier mit der logischen XOR-Verknüpfung überein. Überlegen Sie einmal, warum dies so ist.

Neben diesen drei logischen Verknüpfungen gehört in die Logikgruppe noch das Kommando

CP (für Compare = Vergleichen)

Bei der Ausführung dieses Befehls wird der Wert des Akkumulators mit einem extern vorgegebenen Datenbyte verglichen. Es wird eine Subtraktion durchgeführt, die jedoch nicht in den Akkumulator zurückgespeichert wird, sondern nur die FLAGs beeinflußt. Das Kommando eignet sich daher sehr gut, um Sprünge aufgrund einer Eigenschaft eines Datenbytes (0, <0, >0 etc.) durchzuführen.

Außer dem normalen ComPare existieren beim Z80-Prozessor noch vier Blockbefehle. Mit diesen ist es möglich einen Datenbereich nach einem bestimmten Wert zu durchsuchen. Die einzelnen Varianten entsprechen dabei den Blockladebefehlen. Das Prinzip ist dabei wie folgt: A wird mit dem Inhalt der durch HL angegebenen Speicherstelle verglichen, und die FLAGs werden entsprechend gesetzt. Das Registerpaar BC fungiert dabei als Zähler, es wird in jedem Fall um 1 vermindert. Je nach Suchrichtung wird danach HL um 1 erhöht (CPI, CPIR) oder um 1 vermindert (CPD, CPDR). Die einfachen Blocksuchbefehle sind damit beendet.

Bei einem REPEAT-Kommando wiederholt sich dieser Vorgang so lange, bis A und der Inhalt von (HL) übereinstimmen oder BC den Wert 0 erreicht. Die Blocksuchbefehle haben die nachfolgenden Codes:

| | |
|---|---|
| CPD (Vergleiche und dekrementiere) | ED A9 |
| CPDR (Blockvergleich und Dekrementieren) | ED B9 |
| CPI (vergleiche und inkrementiere) | ED A1 |
| CPIR (Blockvergleich und Inkrementieren) | ED B1 |

Das Ergebnis der Vergleichsbefehle ist dabei mit den Flags Z beziehungsweise PV kontrollierbar. Z wird gesetzt, falls eine Übereinstimmung stattgefunden hat, das heißt A=(HL). PV wird zurückgesetzt, wenn BC=0, das heißt alle Bytes durchsucht worden sind, ansonsten ist es gesetzt: Durch Abfrage dieser beiden FLAGs kann also überprüft werden, ob innerhalb des durchsuchten Blocks eine Übereinstimmung stattgefunden hat und wenn ja, ob innerhalb eines Blocks oder erst am Blockende.



Die nachfolgende Befehlsübersicht bietet einen Überblick über die einzelnen Varianten der Arithmetik- und Logikbefehle, soweit es sich um einfache 8-Bit-Werte handelt. Die Abkürzungen B2 und B3 stehen dabei für das 2. und 3. Byte eines Befehles und geben das mitgelieferte Datenbyte beziehungsweise bei den indirekt indizierten Adressierungsarten, die Distanzadresse an.

**Beispiele:** Wir wollen uns mit Hilfe von GOMACH einmal die Wirkung einiger Befehle aus dem A/L-Bereich anschauen. Besondere Beachtung verdienen dabei in jedem Fall die Flags. Sie sollten sich also immer mit der Funktion C oder

PRINT BIN$(<Flagwert>,8)

das Verhalten der einzelnen Flags überprüfen. Zunächst einmal zu den Arithmetik-Kommandos. Wir wollen einmal das unterschiedliche Funktionieren von ADD und ADC untersuchen. Dazu benutzen wir den Akkumulator und das Register B. Vergleichen Sie nun einmal die Ausgabe bei den folgenden Befehlen:

| ADD | | ADC | |
|---|---|---|---|
| LD A, 90 | 3E 90 | LD A,90 | 3E 90 |
| LD B,A | 47 | LD B,A | 47 |
| ADD A,B | 80 | ADC A,B | 88 |
| ADD A,B | 80 | ADC A,B | 88 |
| RET | C9 | RET | C9 |

In beiden Varianten werden zunächst A (LD A,90) und dann auch durch den Kopierbefehl (LD B,A) die Register A und B mit hex 90 geladen. Danach erfolgt zweimal hintereinander eine Addition.

Register B wird zu A addiert. Sie sollten zunächst einmal nach dem ersten Additionsbefehl ein RET einfügen und sich die Ausgabe anschauen. Sie erhalten dann für AF bei beiden Befehlsketten die nachfolgenden Werte.

AF
2025

Das Carry ist jetzt also gesetzt (BIN$ liefert uns die Ausgabe: 00100101). Da das C-Flag beim Ansprung einer Maschinenspracheroutine von BASIC aus immer 0 ist, reagierten beide Kommandos identisch. Führen wir jetzt

aber unseren zweiten Additionsbefehl aus, so erhalten wir eine unterschiedliche Reaktion:

| ADD A,B | ADC A,B |
|---|---|
| AF | AF |
| B0A0 | B1A0 |

In beiden Fällen wurde also das Carry gelöscht. Nur bei ADC fand dagegen der Übertrag in die unterste Stelle des Akkumulators statt. Die Werte für den Akkumulator ergeben sich übrigens relativ leicht, wenn man die folgenden Additionen nachvollzieht:

Akkumulator zu Beginn: 90 B zu Beginn: 90

1. ADD A,B oder ADC A,B: 90 hex + 90 hex = 120 hex
   ergibt: 20 hex in A und Übertrag (Carry) = 1

2. ADD A,B : 20 hex + 90 hex = B0 hex
   kein Übertrag

oder ADC A,B : 20 hex + 90 hex
+ Carry-Flag = B1 hex
kein Übertrag

Sie sollten als Übung dieselben Operationen einmal mit SUB und SBC durchspielen. Auch hier werden Sie dann verschiedene Ausgaben erhalten. Wenn sie dann mit den 8-Bit-Operationen einigermaßen vertraut sind, können Sie zur 16-Bit-Arithmetik übergehen. Die notwendigen Kommandos finden Sie am Ende der 16-Bit-Kommandos und in der nachfolgenden Tabelle. Eine Übersicht über die Funktion der Logik-Befehle gibt der nun folgende Anwendungsteil.



**Tabelle 5.3**: Die 8-Bit-Arithmetik-Logik-Befehle

| | Register | | | | | | | indirekt | indiziert | | unmittelbar |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | A | B | C | D | E | H | L | (HL) | (IX+d) | (IY+d) | n |
| Addiere<br>ADD | 87 | 80 | 81 | 82 | 83 | 84 | 85 | 86 | DD<br>86<br>d | FD<br>86<br>d | C6<br>n |
| Addiere<br>mit carry<br>ADC | 8F | 88 | 89 | 8A | 8B | 8C | 8D | 8E | DD<br>8E<br>d | FD<br>8E<br>d | CE<br>n |
| Subtrahiere<br>SUB | 97 | 90 | 91 | 92 | 93 | 94 | 95 | 96 | DD<br>96<br>d | FD<br>96<br>d | D6<br>n |
| Subtrahiere<br>mit carry<br>SBC | 9F | 98 | 99 | 9A | 9B | 9C | 9D | 9E | DD<br>9E<br>d | FD<br>9E<br>d | DE<br>n |
| AND<br>Verknüpfung | A7 | A0 | A1 | A2 | A3 | A4 | A5 | A6 | DD<br>A6<br>d | FD<br>A6<br>d | E6<br>n |
| XOR<br>Verknüpfung | AF | A8 | A9 | AA | AB | AC | AD | AE | DD<br>AE<br>d | FD<br>AE<br>d | EE<br>n |
| OR<br>Verknüpfung | B7 | B0 | B1 | B2 | B3 | B4 | B5 | B6 | DD<br>B6<br>d | FD<br>B6<br>d | F6<br>n |
| Vergleiche<br>CP | BF | B8 | B9 | BA | BB | BC | BD | BE | DD<br>BE<br>d | FD<br>BE<br>d | FE<br>n |
| Erhöhe<br>um 1 INC | 3C | 04 | 0C | 14 | 1C | 24 | 2C | 34 | DD<br>34<br>d | FD<br>34<br>d | |
| Vermindere<br>um 1 DEC | 3D | 05 | 0D | 15 | 1D | 25 | 2D | 35 | DD<br>35<br>d | FD<br>35<br>d | |

RicherzComputerGrafik

**Anwendungen:**

Eine der bekanntesten und nützlichsten Anwendungen der Logikbefehle besteht in der Maskierung von Daten. Als Maskierung bezeichnet man die Verknüpfung von zwei Datenbytes, bei denen das eine quasi als Durchgangsschleuse für die einzelnen Bits des anderen Datenbytes fungiert. Das Maskenbyte gibt also an, welche Bits des Datenbytes erhalten bleiben, die anderen werden normalerweise auf 0 gesetzt.

Bei der gerade beschriebenen Maskierung hätten wir es mit einer AND-Maskierung zu tun. Denkbar ist auch eine OR-Maskierung. Bei gesetztem Maskenbit würde das Datenbyte auf jeden Fall gesetzt, bei nichtgesetztem Maskenbit bliebe der alte Zustand erhalten. Den Fall einer XOR-Maskierung können Sie selber einmal durchdenken.

Wir wollen uns das Ganze nun einmal in einem konkreten Beispiel anschauen. Der häufigste Anwendungsbereich für Maskierung ist das Ausblenden unzulässiger Werte. Ein Beispiel aus dem Betriebssystem, wo der CPC die Maskierungstechnik anwendet, ist die Routine SCR SET BASE, die wir ja schon des öfteren benutzt haben.

Der Grafikspeicher kann in vier Bereichen zu je 16 K gelagert werden. Die Definition, welcher der vier Bereiche (0000-3FFFF, 40000-7FFFF, 8000-BFFF und C0000-FFFF) benutzt wird, wurde mit dem Inhalt des Registers A spezifiziert. A stellt dabei gleichsam das HIGH BYTE einer Adresse dar, mit der der Bildschirmspeicher adressiert wird. Das LOW BYTE dieser Adresse wäre dabei immer 00.

Nun sind nur die oberen beiden Bit, also B6 und B7, für die Entscheidung relevant, da nur sie für die Auswahl der vier 16-K-Blöcke zuständig sind. Die unteren sechs Bit geben eine Position eines Blocks von 256 Byte innerhalb der 16K. Das auf 0 gesetzte LOWER BYTE würde dann eine Position innerhalb eines Blocks von 256 Byte definieren.

Der Videocontroller kann aber nur zwischen diesen vier Bereichen unterscheiden und erwartet dabei eine eindeutige Adressenvorgabe. Es ist zum Beispiel nicht möglich, den Anfang auf A000 oder auf 5173 zu setzen. Es müßten schon ganze 16K sein. Um bei einer Fehleingabe von A dennoch zu einem wenigstens möglichen Speicherbereich zu kommen, wird Register A vor der Ausgabe an den Videocontroller mit hex C0 maskiert.



Wenn Sie sich diesen Wert einmal binär ausgeben lassen, so erscheint auf dem Bildschirm die folgende Zahl:

```
11000000
```

Betrachten wir nun einmal, was passiert, wenn wir ein Datenbyte mit dieser Maske AND-verknüpfen. Wenn Sie sich noch einmal die Wahrheitstafel vor Augen halten, so ist das Ergebnis schnell erkennbar: Die oberen beiden Bits unseres Datenbytes bleiben erhalten, denn entweder waren sie 0, so tritt der Fall 0-1 ein, und das Ergebnis bleibt 0. Bei 1-1 bleibt die 1 im Ergebnis erhalten.

Die unteren sechs Datenbit werden aber in jedem Fall gelöscht. Damit bleiben Fehleingaben ohne negative Auswirkung. Sie sollten nun einmal zum Training das Ganze mit der OR-Funktion und mit XOR durchspielen, um sich klarzumachen, welche Änderungen bei einer Maskierung mit diesen Funktionen auftreten würden.

Neben dem Ausblenden unliebsamer Bits gibt es natürlich speziell bei der Verknüpfung mit OR eine andere Möglichkeit, das Einblenden erwünschter Bits. Wenn man mit einer OR-Maske ein Datenbyte maskiert, so bleiben jene Bits, die in der Maske auf 0 sind, wertmäßig erhalten. Die in der Maske auf 1 gesetzten Bits werden auch im Ergebnis immer auf 1 gesetzt. Man kann also mit einer OR-Verknüpfung eine ganze Reihe von Bits auf einmal setzen. Mit AND ist die Gegenoperation (das Löschen einer Reihe von Bits auf einmal) möglich.

Der CPC benutzt diese Technik zum Beispiel bei der Ansprache des Bildschirmspeichers. Wie Sie sicher wissen, verfügt der Computer über eine ganze Reihe von Bildschirmverknüpfungen. Eine ist zum Beispiel der Transparent-Modus. Man schaltet diesen Mode mit

```
PRINT CHR$(22)+CHR$(1)
```

ein. Angehängtes CHR$(0) schaltet ihn wieder aus. Im Transparent- Modus findet eine OR-Verknüpfung zwischen dem Hintergrund und dem Schreibstift statt. Dadurch bleibt der alte Hintergrund erhalten. Wer noch weiter experimentieren möchte, sei hier auf den Charactercode 23 verwiesen. Dieser setzt den Grafikfarbstiftmodus. Hier ist man neben der normalen Verknüpfung auch in der Lage, auf die einzelnen Pixel, das heißt Bildpunkte, die Verknüpfungen XOR und AND anzuwenden. Man bekommt hierdurch einen guten (grafischen) Eindruck von der Wirkung der einzelnen

Kombinationen. Die notwendigen Parameter finden Sie im Bedienerhandbuch in Kapitel 9, Seite 3.

### 5.4.3 Verschiebebefehle

Der Z80-Befehlssatz enthält eine Reihe von Kommandos, mit denen es möglich ist, ein Register oder eine indirekt oder indirekt indiziert angegebene Speicherstelle als Ganzes um 1 Bit nach rechts oder links zu verschieben. Dabei kann man zwei grundsätzliche Varianten unterscheiden: Rotieren (ROTATE) und das eigentliche Verschieben (SHIFT).

#### Rotieren

Beim **Rotieren** wird ein Register um ein Bit verschoben. Man kann dabei zwischen dem Rotieren durch das Carry-Bit (9-Bit-Rotation) und einem Rotieren in das CARRY-Bit (8-Bit-Verschiebung) unterscheiden. Bei der letztgenannten Variante wird dabei das angesprochene Register in sich selbst um eine Stelle rotiert. Gleichzeitig wird das höchste beziehungsweise niedrigwertigste Bit, je nach Schieberichtung, ins CARRY kopiert.

Es ergeben sich somit die folgenden vier Befehle:

**RLC (Rotate left into CARRY - Rotieren nach links ins CARRY)**

Dieser Befehl führt eine Rotation um ein Bit des angegebenen Universalregisters nach links durch. Bit 7 wird in Bit 0 geschoben und gleichzeitig als Kopie ins CARRY übertragen. Die anderen Bits verschieben sich um jeweils eine Stelle nach links.

**RRC (Rotate right into CARRY - Rotieren nach rechts ins CARRY)**

Dieser Befehl läuft analog zu RLC ab, nur daß sich hier die Bewegungsrichtung geändert hat. Bit 0 wird in Bit 7 und ins CARRY geschoben, die anderen Bits rücken jeweils um eine Stelle nach rechts.

**RL (Rotate left - Rotiere nach links durch das CARRY)**

Dieser Befehl führt eine 9-Bit-Rotation durch das CARRY durch. Bit 7 wird ins CARRY kopiert, das C-Flag gelangt in Bit 0, die anderen Bits werden um jeweils eine Stelle nach links verschoben.



**RR (Rotate right - Rotiere nach rechts durch das CARRY)**

führt ebenfalls eine 9-Bit-Rotation durch das CARRY durch, in umgekehrter Richtung zu RL. Das CARRY liegt jetzt unterhalb von Bit 0 des angesprochenen Registers. Der Inhalt von Bit 0 gelangt ins CARRY, das CARRY wird in Bit 7 geschoben, die Bit 7 bis 1 um jeweils eine Stelle nach rechts verrückt.

#### Verschieben

Diese Befehle führen ebenso wie die Rotationskommandos eine Verschiebung des Registers um eine Stelle in der gewünschten Richtung aus. Allerdings unterbleibt die Rückführung des herausgeschobenen Bits in die untere oder obere Position des Registers. Der herausgeschobene Wert gelangt ins CARRY, die nun freie Position wird durch 0 (SRL und SLA) beziehungsweise durch eine Kopie von Bit 7 (SRA) belegt. Im einzelnen existieren die folgenden drei Befehle:

**SRL (Shift right logical = logisches Schieben nach rechts)**

Hier werden die Bits des angegebenen Registers beziehungsweise der indirekt oder indiziert bestimmten Adresse um eine Stelle nach rechts verschoben. Bit 0 gelangt ins CARRY, Bit 7 wird auf 0 gesetzt.

**SLA (Shift left arithmetical=arithmetisches Schieben nach links)**

Dieser Befehl schiebt den Inhalt des Registers oder der adressierten Speicherstelle um eine Stelle nach links. Bit 7 gelangt ins CARRY, Bit 0 wird auf 0 gesetzt.

**SRA (Shift right arithmetical=arithmetisches Schieben rechts)**

Dieser Befehl verschiebt den Registerinhalt oder eine indirekt oder indiziert angegebene Speicherstelle um ein Bit nach rechts. Bit 0 wandert ins CARRY. Bit 7 wird dupliziert, das heißt, Bit 6 und Bit 7 haben nach der Ausführung des Kommandos den Wert des alten Bits 7.

Von der Funktion her kann grundsätzlich gesagt werden, daß achtmaliges Ausführen eines Rotationskommandos den Ursprungszustand - gegebenenfalls unter Nichtberücksichtigung der FLAGs - wiederherstellt, während Verschiebebefehle eine Änderung des Registerinhaltes zur Folge haben. Die SHIFT- und ROTATE-Kommandos sind für alle Universalregister und die

durch HL oder indiziert durch IX und IY adressierten Speicherstellen verfügbar. Alle diese Befehle erreicht man mit dem Erweiterungscode CB. Zusätzlich dazu existieren für den Akkumulator noch die Befehle

RLCA, RRCA, RLA und RRA

Sie sind neben den CB-Erweiterungstabellen auch noch einmal in der Grundtabelle verfügbar, um eine volle Kompatibilität des Z80-Prozessors mit seinem Vorgänger, dem 8080, zu gewährleisten. Diese Befehle sind funktionsmäßig identisch mit den Kommandos der Erweiterungstabelle, allerdings setzen sie nur das CARRY-FLAG. Ihre Codes lauten:

| | |
|---|---|
| RLCA | 07 |
| RRCA | 0F |
| RLA | 17 |
| RRA | 1F |

**FLAGs**: Sowohl die Rotations- wie auch die Shiftbefehle setzen das Statusregister nach dem Ergebnis des verschobenen Wertes. Bei den Kommandos der Grundtabelle (RLCA, RLCA, RLA und RLA) wird nur das CARRY-FLAG gesetzt, ansonsten werden alle FLAGs beeinflußt.

Das Bild auf der nächsten Seite gibt einen Überblick über die Funktionsweise der verschiedenen Verschiebebefehle. Die nachstehende Tabelle gibt dann die hex-Codes der Kommandos für die verschiedenen Operatoren an.

**Beispiele**: Um uns zunächst einmal mit den Befehlen etwas näher vertraut zu machen, wollen wir die Verschiebe- und Rotationskommandos einmal mit dem Inhalt des Registers B durchspielen. Sie sollten die einzelnen Kommandos, jeweils mit RETURN (C9) abgeschlossen, mittels GOMACH eingeben und sich dann die Änderungen von Register B und den Wert des CARRY-FLAGs anschauen.

Dazu sollten Sie sich den Inhalt des Statusregisters in binärer Schreibweise ausgeben lassen, was Sie entweder mit der Funktion C im Monitor, falls Sie GOMACH bereits eingebaut haben, oder ansonsten durch das Kommando

PRINT BIN$(<Statuswert>,8)

erreichen können. Für den Statuswert müssen Sie dabei dann den jeweiligen Wert in der Anzeige unter Register F einsetzen.



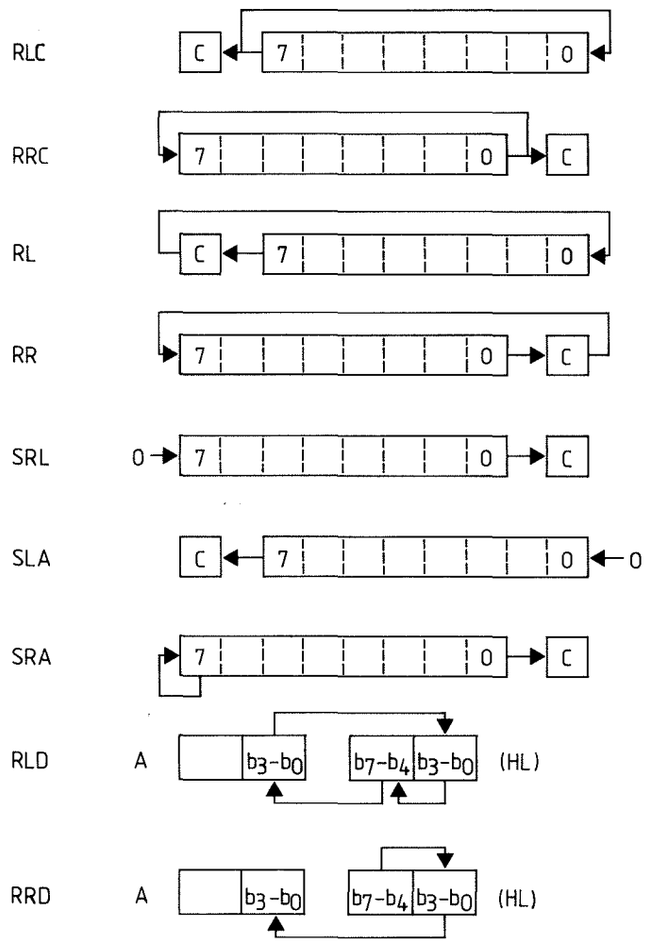



**Bild 5.1**: Die Schiebe- und Rotationsbefehle

**Tabelle 5.4**: Die Verschiebe- und Rotationskommandos

| Befehl | A | B | C | D | E | H | L | (HL) | (IX+d) | (IY+d) |
|---|---|---|---|---|---|---|---|---|---|---|
| RLC | CB | CB | CB | CB | CB | CB | CB | CB | DD | FD |
| | 07 | 00 | 01 | 02 | 03 | 04 | 05 | 06 | CB | CB |
| | | | | | | | | | d | d |
| | | | | | | | | | 06 | 06 |
| RRC | CB | CB | CB | CB | CB | CB | CB | CB | DD | FD |
| | 0F | 08 | 09 | 0A | 0B | 0C | 0D | 0E | CB | CB |
| | | | | | | | | | d | d |
| | | | | | | | | | 0E | 0E |
| RL | CB | CB | CB | CB | CB | CB | CB | CB | DD | FD |
| | 17 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | CB | CB |
| | | | | | | | | | d | d |
| | | | | | | | | | 16 | 16 |
| RR | CB | CB | CB | CB | CB | CB | CB | CB | DD | FD |
| | 1F | 18 | 19 | 1A | 1B | 1C | 1D | 1E | CB | CB |
| | | | | | | | | | d | d |
| | | | | | | | | | 1E | 1E |
| SLA | CB | CB | CB | CB | CB | CB | CB | CB | DD | FD |
| | 27 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | CB | CB |
| | | | | | | | | | d | d |
| | | | | | | | | | 26 | 26 |
| SRA | CB | CB | CB | CB | CB | CB | CB | CB | DD | FD |
| | 2F | 28 | 29 | 2A | 2B | 2C | 2D | 2E | CB | CB |
| | | | | | | | | | d | d |
| | | | | | | | | | 2E | 2E |
| SRL | CB | CB | CB | CB | CB | CB | CB | CB | DD | FD |
| | 3F | 38 | 39 | 3A | 3B | 3C | 3D | 3E | CB | CB |
| | | | | | | | | | d | d |
| | | | | | | | | | 3E | 3E |

RicharzComputerGrafik



Sie können auch einmal die verschiedenen Befehle miteinander kombinieren und sich dann das Ergebnis binär, das heißt mit Hilfe von BIN$ oder der Funktion C anschauen. Die benötigten Befehle entnehmen Sie dabei bitte der Tabelle.

**Anwendung:**

Die Verschiebe- und Rotationsbefehle haben ein breites Anwendungsfeld. Zunächst einmal zur SHIFT-Gruppe. Lesen Sie einmal in ein beliebiges Universalregister den Wert 40 hex ein, und führen Sie dann einmal das SLA-Kommando aus. Bitmäßig betrachtet ist das ganze relativ einfach: Sie haben eine Verschiebung um eine Stelle nach links erreicht, und die letzte Stelle wurde auf 0 gesetzt.

Wie sieht das Ganze aber nun wertmäßig aus? Wenn Sie sich den neuen Binärwert dezimal ausgeben lassen, werden Sie feststellen, daß hier eine genaue Verdopplung des Zahlenwertes erfolgt ist, und dies ist, wie man leicht durchdenken kann, bei jeder Zahl der Fall.

Eine arithmetische Verschiebung nach links bedeutet also eine Verdopplung des Zahlenwertes, nach rechts eine Halbierung. Wenn man diese Gedankengänge im Hinterkopf behält, wird auch die Funkionsweise des etwas seltsam - jedenfalls auf den ersten Blick - anmutenden SRA-Kommandos schnell verständlich.

Dazu benötigen wir ein kleines Experiment. Wir laden A mit hex AA, was dezimal 170 entspricht. Nach Durchlauf einer kleinen Routine (3E, AA, CB, 2F, C9) erhalten wir als Ausgabe für Akkumulator und Statusregister den Wert D5 80. Betrachten wir nun die Werte einmal, was wir durch die folgende Eingabe relativ einfach erreichen können:

```
PRINT BIN$(&AA,8),BIN$(&D5,8),BIN$(&80,8)
```

Sie haben damit eine Binärausgabe des Eingabewertes sowie des Zustandes von A und F nach Durchlauf des Programms auf dem Schirm. Der Effekt ist relativ einfach. Schauen wir uns zunächst einmal das Register F an. Nach Durchlauf des Programms ist das CARRY gelöscht, das ZERO ebenso, da ja kein Wert von 0 erreicht wurde. Das S-FLAG zeigt uns an, daß die Ergebniszahl negativ ist (Bit 7 = 1). Die Anzahl der Einsen im Ergebnis ist ungerade, weshalb PV auf 0 gesetzt ist.

Nun zu den eigentlichen Daten: Das Eingabebyte war relativ einfach zu beschreiben (ständiger Wechsel von 0 und 1). Das Ausgabebyte zeigt nun die erfolgte Veränderung. Alle Bits wurden um 1 nach rechts verschoben: Bit 7 allerdings bleibt dabei wertmäßig erhalten. Durch diese Art der Verschiebung bleibt also bei 7-Bit-Zahlen, bei denen das 8. Bit das Vorzeichen enthält, der Vorzeichenwert der Zahl trotz durchgeführter Verschiebung erhalten.

Neben dem Verdoppeln und Halbieren von Zahlenwerten, das heißt dem Multiplizieren mit oder Teilen durch 2, lassen sich speziell mit den Rotationsbefehlen allerdings noch eine Reihe anderer Effekte erzielen. Zum ersten ist es natürlich möglich, ein oder mehrere Bit, die auf einer "falschen Position" in einem Byte liegen, in eine "richtige Position" zu verschieben. Des weiteren kann man ein Datenbyte mit einem anderen, welches verändert werden soll, verknüpfen, um zum Beispiel eine neue Speichersituation anzuzeigen (siehe Anwendung zu den Verknüpfungskommandos AND und OR).

Ein weiterer wichtiger Anwendungsbereich ist aber auch das schnelle Abtesten von Zuständen mittels der Rotationsbefehle. Dies geschieht beispielsweise im Rahmen der Kassettenoperationen. Der Datacorder speichert mit einer relativ hohen Geschwindigkeit ab und muß mit derselben natürlich auch lesen können.

Den technischen Ablauf der Funktion haben wir bereits in Kapitel 3 kennengelernt, als wir uns mit den Ein-/Ausgabebausteinen beschäftigt haben. Unser PIO verfügt mit dem Register B über einen Eingabe-PORT, der in der Lage ist, relativ schnell Daten einzulesen und dann an die CPU weiterzugeben. Die höchste Datenleitung dieses Eingabe-PORTs, also Bit 7, ist direkt mit der Kassettenelektronik verbunden. Liest man den Wert dieses Eingabekanals in ein Universalregister ein, so kann mit den Befehlen

RL oder RLC

eine Verschiebung des höchsten Bits ins CARRY erfolgen. Da sich mit dem CARRY unter Verwendung der Befehle

JR C und JR NC

eine unkomplizierte und damit direkte Verzweigung erzeugen läßt, haben wir somit eine schnelle Abfragemöglichkeit geschaffen.



Lange Programmschleifen und komplizierte Operationen, die auch in Maschinensprache Zeit kosten würden, sind durch solche Tricks umgehbar. Zwar ist der Z80-Prozessor relativ schnell; doch muß man sich dabei immer noch vor Augen halten, daß das Abspeichern der einzelnen Bits ja mit einigen tausend Schwingungen pro Sekunde vor sich geht. Das wiederum bedeutet, daß man am besten einige 10000mal pro Sekunde den Eingabe-PORT abfragt, um die Länge der verschiedenen Signale mit möglichst geringem Fehlergehalt feststellen zu können.

Denn schließlich kommt es darauf an, den Wechsel von 0 auf 1 beziehungsweise umgekehrt möglichst exakt festzuhalten. Bei solch schneller Abfrage kommt es dann auch auf jeden einzelnen Maschinenzyklus mehr oder weniger an. Die Abfrage mit Hilfe der Rotationsbefehle stellt hier die schnellste Möglichkeit dar.

### 5.4.4 Kommandos für die Bitmanipulation

Haben wir im letzten Kapitel noch Datenbytes als Ganzes behandelt, so kommen wir nun auf eine Gruppe von Befehlen zurück, die den direkten Zugriff auf einzelne Bits eines Universalregisters bzw. einer durch HL und/oder die Indexregister adressierten Speicherstelle ermöglichen. Es sind dies die Befehle

**SET, RES und BIT.**

Die Kommandos haben das Format

Befehl <Stelle>, <Register>.

Befehl bedeutet dabei eines der obigen drei Kommandos SET, RES oder aber BIT. Die Stelle gibt die Position im Universalregister an, die bearbeitet werden soll, also das zu ändernde oder zu überprüfende Bit. Register enthält als Angabe eines der Universalregister respektive die durch HL indirekt oder durch IY und IX indiziert angesprochene Speicherstelle.

**Tabelle 5.5:** Die Bitmanipulationsbefehle

| | | REGISTER ADRESSIERUNG | | | | | | | REG. INDIR | INDEXIERT | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | BIT | A | B | C | D | E | H | L | (HL) | (IX +d) | (IY +d) |
| TEST "BIT" | 0 | CB 47 | CB 40 | CB 41 | CB 42 | CB 43 | CB 44 | CB 45 | CB 46 | DD CB d 46 | FD CB d 46 |
| | 1 | CB 4F | CB 48 | CB 49 | CB 4A | CB 4B | CB 4C | CB 4D | CB 4E | DD CB d 4E | FD CB d 4E |
| | 2 | CB 57 | CB 50 | CB 52 | CB 52 | CB 53 | CB 54 | CB 55 | CB 56 | DD CB d 56 | FD CB d 56 |
| | 3 | CB 5F | CB 58 | CB 59 | CB 5A | CB 5B | CB 5C | CB 5D | CB 5E | DD CB d 5E | FD CB d 5E |
| | 4 | CB 67 | CB 60 | CB 61 | CB 62 | CB 63 | CB 64 | CB 65 | CB 66 | DD CB d 66 | FD CB d 66 |
| | 5 | CB 6F | CB 68 | CB 69 | CB 6A | CB 6B | CB 6C | CB 6D | CB 6E | DD CB d 6E | FD CB d 6E |
| | 6 | CB 77 | CB 70 | CB 71 | CB 72 | CB 73 | CB 74 | CB 75 | CB 76 | DD CB d 76 | FD CB d 76 |
| | 7 | CB 7F | CB 78 | CB 79 | CB 7A | CB 7B | CB 7C | CB 7D | CB 7E | DD CB d 7E | FD CB d 7E |
| RESET BIT "RES" | 0 | CB B7 | CB B0 | CB B1 | CB B2 | CB B3 | CB B4 | CB B5 | CB B6 | DD CB d B6 | FD CB d B6 |
| | 1 | CB BF | CB B8 | CB B9 | CB BA | CB BB | CB BC | CB BD | CB BE | DD CB d BE | FD CB d BE |
| | 2 | CB 97 | CB 90 | CB 91 | CB 92 | CB 93 | CB 94 | CB 95 | CB 96 | DD CB d 96 | FD CB d 96 |
| | 3 | CB 9F | CB 98 | CB 99 | CB 9A | CB 9B | CB 9C | CB 9D | CB 9E | DD CB d 9E | FD CB d 9E |
| | 4 | CB A7 | CB A0 | CB A1 | CB A2 | CB A3 | CB A4 | CB A5 | CB A6 | DD CB d A6 | FD CB d A6 |
| | 5 | CB AF | CB A8 | CB A9 | CB AA | CB AB | CB AC | CB AD | CB AE | DD CB d AE | FD CB d AE |
| | 6 | CB B7 | CB B0 | CB B1 | CB B2 | CB B3 | CB B4 | CB B5 | CB B6 | DD CB d B6 | FD CB d B6 |
| | 7 | CB BF | CB B8 | CB B9 | CB BA | CB BB | CB BC | CB BD | CB BE | DD CB d BE | FD CB d BE |
| SET BIT "SET" | 0 | CB C7 | CB C0 | CB C1 | CB C2 | CB C3 | CB C4 | CB C5 | CB C6 | DD CB d C6 | FD CB d C6 |
| | 1 | CB CF | CB C8 | CB C9 | CB CA | CB CB | CB CC | CB CD | CB CE | DD CB d CE | FD CB d CE |
| | 2 | CB D7 | CB D0 | CB D1 | CB D2 | CB D3 | CB D4 | CB D4 | CB D6 | DD CB d D6 | FD CB d D6 |
| | 3 | CB DF | CB D8 | CB D9 | CB DA | CB DB | CB DC | CB DD | CB DE | DD CB d DE | FD CB d DE |
| | 4 | CB E7 | CB E0 | CB E1 | CB E2 | CB E3 | CB E4 | CB E5 | CB E6 | DD CB d E6 | FD CB d E6 |
| | 5 | CB EF | CB E8 | CB E9 | CB EA | CB EB | CB EC | CB ED | CB EE | DD CB d EE | FD CB d EE |
| | 6 | CB F7 | CB F0 | CB F1 | CB F2 | CB F3 | CB F4 | CB F5 | CB F6 | DD CB d F6 | FD CB d F6 |
| | 7 | CB FF | CB F8 | CB F9 | CB FA | CB FB | CB FC | CB FD | CB FE | DD CB d FE | FD CB d FE |

RicharzComputerGrafik



Hier nun die Erläuterungen zu den einzelnen Kommandos:

**SET:** dient zum Setzen eines Bits in einem Universalregister oder einer durch HL, IX oder IY adressierten Speicherstelle.

**RES:** setzt ein Bit in einem Universalregister oder einer durch HL, IX oder IY indirekt spezifizierten Adresse zurück.

**BIT:** testet ein Bit eines Universalregisters oder einer durch HL, IX oder IY vorgegebenen Speicherstelle. Das Ergebnis findet sich dabei im ZERO-FLAG wieder. Das heißt: War das entsprechende Bit gesetzt, wird das ZERO-BIT auf 0 gesetzt; ansonsten wird das ZERO-BIT auf 1 gesetzt. Eine Abfrage des FLAGs mit den Kommandos

JR Z und JR NZ

läßt dann Sprungverzweigungen in Abhängigkeit vom Zustand eines Bits zu (vergleiche Kapitel 5.4).

**FLAGs:** Die Befehle SET und RESet haben keinerlei Einfluß auf das Statusregister. Das Testkommando BIT setzt das ZERO FLAG und auch die anderen FLAGs werden entsprechend dem Zustand des betrachteten Datenbytes gesetzt.

### 5.4.5 Sonderbefehle für den Akkumulator und die FLAGs

Neben den bisher behandelten Befehlsgruppen existieren im Bereich der Arithmetik und Logikbefehle noch einige Sonderfunktionen, die sich mit der BCD-Darstellung, dem Komplementieren des Akkumulators und dem Setzen beziehungsweise Komplementieren des CARRY-FLAGs beschäftigen.

#### BCD-Operationen

Die grundsätzliche Problematik einer BCD-Darstellung haben wir bereits kennengelernt. Zwei Ziffern von 0-9 werden in je vier Bit codiert, wobei der Inhalt der einzelnen Bits dem Wert der Dezimalzahl entspricht. Die daraus resultierenden beiden Halbbyte oder auch Nibbles werden dann zu einem 8-Bit-Wert in einem Register oder einer Speicherstelle zusammenge-

fügt. Eine Speicherstelle enthält somit zwei aufeinanderfolgende Dezimalzahlen.

Probleme tauchen bei dieser Art der Darstellung immer dann auf, wenn man mit den normalen Additions- und Subtraktionskommandos zwei BCD-Zahlen miteinander verknüpft. Da Werte zwischen 10 und 15 beziehungsweise hex A bis hex F für BCD-Zahlen naturgemäß gesperrt sind, muß hier eine Korrektur, das heißt ein Übertrag, in die nächst höhere Stelle erfolgen.

Beispiel: Der Akkumulator enthält den Wert 09, was auch dezimal 9 bedeuten würde. Wir addieren nun 2:

```
ADC A,2
```

Wenn wir dies mit GOMACH durchführen, ist das Ergebnis klar. Der Akkumulator enthält jetzt den Wert 0B. In BCD-Schreibweise müßte hier aber 11 erscheinen. Daher ist eine Adjustierung notwendig. In unserem Beispiel müßte 6 addiert werden, um die richtige Darstellung zu erhalten:

```
0B + 06 = 11
```

und damit genau der entsprechende BCD-Wert. Im umgekehrten Fall, das heißt bei einer Subtraktion, müßte analog dazu eine Subtraktion mit 6 durchgeführt werden, um das richtige Ergebnis zu erhalten.

Diese Umformungen leistet das Kommando DAA. Falls zwischen den beiden Nibbels ein Übertrag vorgenommen werden müßte, addiert DAA in der kleineren Stelle 6, dies führt dann zu dem gewünschten Übertrag. Im umgekehrten Fall, das heißt bei einer Subtraktion, werden 6 von der niederwertigen Stelle subtrahiert, beziehungsweise es wird das CARRY gesetzt oder auch rückgesetzt.

**CODE:** 27
**FLAGs:** Das gesamte Statusregister wird durch DAA beeinflußt.

Ebenfalls im Rahmen der BCD-Darstellung werden die Kommandos

RRD (rotiere rechts dezimal) und
RLD (rotiere links dezimal)



benutzt. Diese dienen dazu, vier Bit aus dem Akkumulator mit vier Bit aus einer durch HL adressierten Speicherstelle auszutauschen. Damit ist es möglich, eine BCD-Zahl gegen eine andere BCD-Zahl auszutauschen. Dabei findet sowohl innerhalb des Akkumulators als auch innerhalb der adressierten Speicherstelle eine Verschiebung der anderen Zahlen statt. Den genauen Ablauf des Verschiebevorgangs zeigt Abbildung 5.4.

**CODE:** RRD ED 67; RLD ED 6F
**FLAGs:** Alle FLAGs, mit Ausnahme des CARRYs werden durch die Verschiebeoperation beeinflußt.

### Komplementieren des Akkumulators

Beim Komplementieren wird jedes Bit in sein Gegenteil verkehrt. War der Wert 1, so erhält das Bit jetzt den Wert 0 und umgekehrt. Man kann zwischen dem Einer-Komplement, durch den Befehl CPL erzeugt, und dem Zweier-Komplement, durch NEG erzeugt, unterscheiden.

Das Einer-Komplement stellt genau diesen Vorgang dar. Beim Zweier-Komplement handelt es sich im Endeffekt um eine Subtraktion von 0. Der Akkumulator wird von 0 abgezogen, bekommt also ein negatives Vorzeichen. Es entsteht eine 7-Bit-Zahl, wobei das 8. Bit das Vorzeichen repräsentiert.

Die unterschiedliche Funktion der beiden Komplemente kann man sich am einfachsten verdeutlichen, wenn man annimmt, daß das Register A 0 enthält. Beim Einer-Komplement werden alle Bits "umgedreht", und das Ergebnis ist damit FF oder 256. Im anderen Fall bleibt es 0.

Im Bereich der Arithmetik- und Logik-Befehle existieren außerdem noch zwei Befehle, die im Zusammenhang mit dem CARRY wirken. Der erste (SCF=SET CARRY FLAG) setzt das CARRY auf 1. Der zweite (CCF= COMPLEMENT CARY FLAG) invertiert den Inhalt des CARRYs. War das CARRY 0, wird es nun gesetzt, ansonsten rückgesetzt.

| | | |
|---|---|---|
| **CODES:** | NEG | ED 44 |
| | CPL | 2F |
| | SCF | 37 |
| | CCF | 3F |

**FLAGs:** Alle vier Befehle beeinflussen das Statusregister. SCF und CCF beeinflussen nur das CARRY. CPL beeinflußt das N- und das H-FLAG. Die vier Hauptmerker (S, Z, P/V, C) bleiben unverändert.

**Beispiele:**

| | | |
|---|---|---|
| NEG: | Vorher: | 00110010 |
| | Nachher: | 11001110 |
| CPL: | Vorher: | 00110010 |
| | Nachher: | 11001101 |

Die Wirkung von CPL können wir auch mit dem BASIC-Kommando NOT darstellen.

```
PRINT HEX$(NOT(&AA))
```

liefert uns als Ergebnis das Komplement FF55. Die FF können wir dabei vernachlässigen, sie stellen das Komplement des hier nicht angegebenen HIGHER BYTES 00 dar. Die 55 dagegen liefert uns das Komplement von AA, wohlgemerkt das Einser-Komplement. Das Zweier-Komplement können wir direkt mit einem BASIC-Befehl nicht erzeugen.

### 5.5 Sprungbefehle

Der Z80-Prozessor verfügt über eine Reihe von Kommandos, mit denen es möglich ist, ein Programm an einer Stelle abzubrechen und an einer anderen Stelle fortzusetzen (Sprung). Man kann verschiedene Unterteilungen nach der Wirkung und nach der Abhängigkeit von Bedingungen unter den einzelnen Sprungbefehlen treffen. Grundsätzlich sind drei Arten von Sprüngen nach der Adressierung möglich:

Relativer Sprung: Bei dieser Adressierungsart wird der neue Ansprungpunkt relativ zum Sprungbefehl festgelegt. Berechnungsgrundlage ist dabei das erste Byte nach dem Sprungkommando. Ab dieser Position können maximal 127 Byte nach vorne oder 128 Byte zurück adressiert werden. Der Programmablauf setzt sich dann an der so errechneten Stelle fort.



Bei den relativen Sprüngen kann man noch zwischen drei Unterarten unterscheiden. Dies sind:

- der unbedingte Sprung. Hier wird in jedem Fall verzweigt.

- der bedingte Sprung. Hier findet die Verzweigung nur dann statt, wenn eine bestimmte Bedingung erfüllt ist, ein FLAG im Statusregister gesetzt beziehungsweise nicht gesetzt ist.

- Darüber hinaus existiert ein weiterer besonderer Befehl, das Kommando DJNZ (decrement and jump not zero = decrementiere und springe wenn nicht 0). Dieses Kommando führt einen n-maligen Rücksprung durch. Damit ist die Programmierung von Schleifen möglich.

Absoluter Sprung: Diese Kommandos stellen die größte Befehlsgruppe bei den Sprungbefehlen dar. Der neuen Ansprungpunkt, an dem die Programmausführung fortgesetzt werden soll, wird hier als absolute Adresse (oder gegebenenfalls verkürzt absolute Adresse) mitgegeben. Er folgt entweder in zwei Datenbyte auf den Befehl oder ist bereits (bei den RESTARTs) im Kommando implizit enthalten.

Neben den RESTARTs existieren zwei größere Gruppen von Sprungbefehlen; normale Sprungbefehle und Unterprogrammaufrufe. Der normale Programmaufruf lädt PC mit einem neuen Wert und setzt die Bearbeitung an dieser Stelle fort. Der Unterprogrammaufruf (CALL) adressiert ein Unterprogramm. Er läuft wie der direkte Sprungbefehl (JP) ab, allerdings wird hier eine Rücksprungadresse auf dem STACK abgelegt. Ebenso wie bei den RESTARTs ist mit Hilfe dieser Rückkehradresse die Möglichkeit vorhanden, mit dem Kommando RETURN (RET) wieder aus dem Unterprogramm in das Hauptprogramm zurückzukehren.

JP auf der einen beziehungsweise RESTART (RST) und CALL auf der anderen Seite verhalten sich also von der Funktion wie die BASIC-Kommandos GOTO beziehungsweise GOSUB.

**Bedingung:** Die RESTART-Kommandos erfolgen alle unbedingt. Die Ausführung der normalen sowie der Unterprogrammsprünge und auch der Rückkehrbefehle aus einem Unterprogramm kann dagegen jeweils unbedingt oder in Abhängigkeit eines FLAGs (S-FLAG, CARRY, ZERO- FLAG, PV-FLAG) erfolgen. Dabei sind auch Kombinationen möglich. Man kann also ein Programm bei Eintritt einer Bedingung (z.B. ZERO FLAG gesetzt) anspringen und aus verschiedenen Punkten des Unterprogramms mit anderen Bedingungen (+ - CARRY o.ä.) zurückkehren. Wichtig bei der Rückkehr aus einem Unterprogramm ist dabei jedoch, daß zu diesem Zeitpunkt der Stapel wieder abgearbeitet sein muß, da die aktuellen zwei Byte des STACK in jedem Fall und ohne Prüfung als Rücksprungadresse betrachtet werden. Liegen hier falsche Werte, kommt es zum Ansprung unsinniger Routinen oder gegebenenfalls zum "Aufhängen" des Rechners.

Indirekte Sprünge: Es existieren drei indirekte Sprungkommandos, mit denen es möglich ist, eine Programmfortsetzung an dem Punkt zu erreichen, der durch den Inhalt eines der Registerpaare HL, IX oder IY vorgegeben wird. Diese Kommandos sind nur als direkter Ansprung, das heißt ohne Ablage einer Rücksprungadresse und auch nur unbedingt ausführbar.

Eine grafische Übersicht über die Aufteilung der verschiedenen Sprungbefehle gibt das nachfolgende Schaubild.



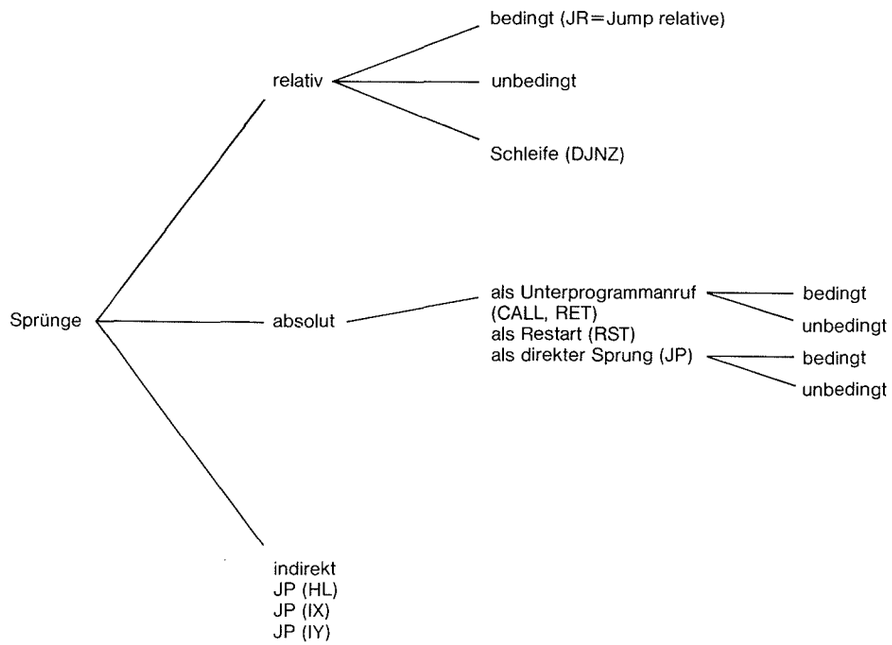



**Bild 5.2:** Die Sprungbefehle

**Relative Sprünge:**

Ein relativer Sprung kann unbedingt oder in Abhängigkeit vom CARRY (C,NC) und vom ZERO-FLAG (Z,NZ) erfolgen. Daneben existiert das Kommando DJNZ. Dieses dekrementiert das Register B und springt, falls B ungleich 0, um die gegebene Anzahl von Bytes. Berechnungsgrundlage ist dabei das erste Byte nach dem Befehl, womit sich ein Sprungumfang, berechnet auf den Befehlscode von -126 bis +129 Byte ergibt.

Die relativen Sprungbefehle bestehen aus zwei Byte. Das erst Byte wird durch den Befehlscode eingenommen, das zweite Byte gibt als Zweierkomplement die zu überspringende Anzahl von Bytes, den OFF SET, an; in der Tabelle mit diff bezeichnet.

**Absolute Sprünge:**

Bei den absoluten Sprungbefehlen muß man zwischen Ansprung und Unterprogrammansprung (JP und CALL) unterscheiden. Die absoluten Sprünge bestehen aus drei Byte. Das erste wird vom Befehlscode eingenommen. Die darauffolgenden beinhalten den Ansprungpunkt als Zwei-Byte-Adresse.

Der Ansprung kann dabei ohne Bedingung oder in Abhängigkeit eines der vier FLAGs (CARRY, ZERO, PV-FLAG, S-FLAG) erfolgen. Ein Unterprogrammaufruf muß dabei mit RETURN (RET) abgeschlossen werden. Das RETURN-Kommando ist ebenso unbedingt oder in Abhängigkeit eines der vier FLAGs ausführbar. Die RETURN-Befehle sind dabei Ein-Byte-Kommandos, sie enthalten bereits die FLAG-Bedingung.

Als Sonderform (verkürzte Unterprogrammaufrufe) existieren acht sogenannte RESTART-Anweisungen. Diese führen einen Unterprogrammaufruf bestimmter, fix vorgegebener Ansprungstellen im unteren Bereich des Speichers in der Seite 0 aus. Die RESTART-Anweisungen sind Ein-Byte-Befehle, wobei die Bit 3 bis 5 die anzuspringende Speicherstelle decodieren. Der Befehl hat das Format: 11aaa111. Ersetzen der drei durch Platzhalter belegten Stellen und Setzen auf Null aller anderen Bits gibt dabei die Ansprungadresse an.

Man kann deren Wert auch dadurch feststellen, daß man alle anderen Bits des Befehlswortes auf 0 setzt und das so veränderte Befehlsbyte als LOWER BYTE einer Zwei-Byte-Adresse mit ebenfalls auf 0 gesetzten HIGHER BYTE betrachtet. Die Werte für aaa entsprechen dann den folgenden hexadezimalen Ansprungadressen:

**Tabelle 5.4:** Die Restart-Kommandos

| Befehl | Code | aaa | hex-Adresse |
|---|---|---|---|
| RST0 | C7 | 000 | 00 |
| RST08 | CF | 001 | 08 |
| RST10 | D7 | 010 | 10 |
| RST18 | DF | 011 | 18 |
| RST20 | E7 | 100 | 20 |
| RST28 | EF | 101 | 28 |
| RST30 | F7 | 110 | 30 |
| RST38 | FF | 111 | 38 |



**Indirekte Sprünge:**

Der Z80-Prozessor verfügt über drei indirekte Sprungkommandos. Es ist möglich, ein Programm an der Stelle, die durch HL, IX oder IY spezifiziert wird, fortzusetzen. Eine Übersicht über die Gesamtheit aller Sprungbefehle bietet die nachfolgende Tabelle.

**Tabelle 5.6:** Sprungbefehle

| | Bedingung | | | | | |
|---|---|---|---|---|---|---|
| | keine | Carry C NC | Zero Z NZ | Parität PE PO | Vorzeichen M P | Reg.B = 0 |
| absoluter Sprung<br>JUMP: JP nn | C3<br>nn<br>nn | DA D2<br>nn nn<br>nn nn | CA C2<br>nn nn<br>nn nn | EA E2<br>nn nn<br>nn nn | FA F2<br>nn nn<br>nn nn | |
| Unterprogramm-<br>aufruf: CALL nn | CD<br>nn<br>nn | DC D4<br>nn nn<br>nn nn | CC C4<br>nn nn<br>nn nn | EC E4<br>nn nn<br>nn nn | FC F4<br>nn nn<br>nn nn | |
| Unterprogramm-<br>rückkehr: RET | C9 | D8 D0 | C8 C0 | E8 E0 | F8 F0 | |
| relativer Sprung<br>JR diff | 18<br>dd | 38 30<br>dd dd | 28 20<br>dd dd | | | |
| Schleife<br>DJNZ diff | | | | | | 10<br>dd |
| indirekter Sprung<br>JP (HL) | E9 | | | | | |
| indizierter Sprung<br>JP(IX) | DD<br>E9 | | | | | |
| indizierter Sprung<br>JP (IY) | FD<br>E9 | | | | | |

RicharzComputerGrafik

**FLAGs:**

Das Statusregister wird durch sämtliche Sprungkommandos nicht beeinflußt. Allerdings sind viele Sprungkommandos ihrerseits vom Inhalt einzelner FLAGs abhängig.

**Beispiele:** Die Sprungkommandos sind relativ einfach zu verstehen, so daß wir uns nicht besonders ausführlich damit beschäftigen müssen. Hier seien nur ein paar kleinere Probleme angesprochen.

Einen einfachen Sprungbefehl z.B. zur Adresse A000 erzeugen wir mit JP A000 beziehungsweise in hex-Code C3 00 A0. Sie sollten dabei immer daran denken, daß die Adresse LOW HIGH zu spezifizieren ist.

Ein anderes häufig auftretendes Problem ist ein relativer Sprung, der über mehr als die möglichen 126 beziehungsweise 129 Byte in eine Richtung verlaufen soll. Hier bieten sich mehrere Möglichkeiten zur Lösung des Problems:

Zum einen könnte man am Punkt der maximalen Entfernung eine Weiterverzeigerung vornehmen. Soll zum Beispiel bei nichtgesetztem CARRY der Programmablauf normal weiterlaufen, ansonsten aber ein Sprung um 200 Byte nach vorne erfolgen, so läßt sich dies relativ einfach realisieren.

Man führt zunächst das Kommando JRC+120 (38 78) aus; an der damit erreichten Stelle postiert man dann einen weiteren unbedingten relativen Sprung (JR+78 bzw. 18 4E hex), womit dann die Gesamtverschiebung um 200 Byte erreicht wird. Die 78 ergibt sich dabei aus der normal zu erwartenden 80, indem wir berücksichtigen, daß unser zweiter relativer Sprungbefehl ja wieder zwei Byte kostet.

Eine andere Möglichkeit besteht darin, die aktuelle Adresse im Register HL zu speichern. Man addiert dann zu dieser Adresse 200 hinzu und führt daraufhin einen indirekten Sprung in Abhängigkeit von HL durch. Dieser Vorgang läßt sich natürlich ebenso mit IX oder IY vornehmen.

Bei den obigen Beispielen handelte es sich um noch relativ einfache Probleme und deren Lösungen im Bereich der Sprungbefehle. Wie aber können wir unsere neuen Kenntnisse wirklich gewinnbringend anwenden?



Der STACK oder besser gesagt die Ablage der Sprunganweisungen auf dem STACK, die von der Ablage von Variablen nicht zu unterscheiden ist, bietet noch eine ganze Reihe anderer Anwendungsmöglichkeiten. Wir kommen damit dann zu einem Bereich, den man als die hohe Schule der Sprungbefehle beschreiben könnte.

**Anwendung: Die Routine LOW-JUMP-RESTART**

Was man bei konsequenter Ausnutzung sämtlicher Kniffe und Tricks in diesem Bereich erreichen kann, zeigt uns die Ansprungroutine für die LOW-JUMP-RESTARTs beim CPC.

Worum handelt es sich dabei? Wir haben schon des öfteren Firmwareroutinen benutzt. Diese hatten Nummern zwischen B900 und BDFF. An diesen Stellen befinden sich aber gar keine eigentlichen Maschinenroutinen, sondern nur Zeiger auf Maschinenprogramme im unteren oder teilweise im oberen ROM.

Wird eine solche Routine angesprungen, so muß also zunächst die benötigte ROM-Konfiguration hergestellt werden. Dann muß ein Sprung in das jeweils gewünschte Unterprogramm, welches sich z.B. im unteren ROM befindet, erfolgen, und nach der Rückkehr sollte der alte Speicherzustand wiederhergestellt werden.

Nun ist es selbstverständlich, daß man diese Operationen nicht für jede einzelne Routine durchführen kann. Bei den insgesamt etwa 300 Firmwareansprungpunkten, die im Sprungblock enthalten sind, würde das den Rahmen jedes 16-Bit-Adreßraumes sprengen. Daher ist eine allgemeine Interpretationsroutine für die Sprungtabellen vonnöten, die die oben angesprochenen Funktionen wahrnimmt.

Dabei sollte jeder Ansprungpunkt mit einer möglichst minimalen Zahl von Bytes auskommen. Beim CPC sind dies drei Byte. In der ersten Stelle findet sich ein RESTART-Kommando, also ein verkürzter Unterprogrammaufruf. Bei einem Ansprung einer Routine im unteren ROM ist dies RESTART 1, ansonsten RESTART 5.

Danach folgt in zwei Byte die Ansprungadresse im ROM. Davon werden jedoch nur 14 Bit benötigt, da jedes ROM ja nur über 16K Speicher verfügt. Die restlichen beiden Bits, also Bit 14 und Bit 15, decodieren den ROM-Zustand. Bit 15 ist dabei für das obere ROM zuständig; ist es auf 1

gesetzt, so ist die von ihm zu kontrollierende Hälfte des ROM gesperrt. Bit 14 nimmt die analoge Funktion für das untere ROM wahr.

Um den konkreten Ablauf des Ansprungs einer Routine im unteren ROM darzustellen, wollen wir uns im folgenden die Druckerhilfsroutine MC PRINT CHAR anschauen.

Sie hat die Ansprungadresse BD2B und führt einen Sprung ins untere ROM aus. Ihre Aufgabe besteht darin, ein Zeichen an den Drucker auszusenden. Dabei muß der Akkumulator das zu sendende Zeichen enthalten. Wenn die Routine erfolgreich ausgeführt wurde, ist das CARRY-FLAG an; ansonsten ist es rückgesetzt. Wir nehmen nun einmal an, daß wir diese Routine mit

CALL BD2B

aufrufen können. Den Punkt, von dem ab dieser Aufruf erfolgt, wollen wir mit aabb bezeichnen. Beim Ansprung von BD2B werden diese Koordinaten, oder besser gesagt die Rücksprungadresse (d.h. das erste auf den Befehl folgende Byte) auf dem STACK abgelegt. Daraus resultiert auch die Erhöhung von BD um 3 (das CALL-Kommando besteht ja aus drei Byte).

Was passiert nun ab BD2B? Schauen wir uns die einzelnen Speicherstellen einmal mit

PRINT BIN$(PEEK(&BD2B)) bis PRINT BIN$(PEEK(&BD2D))

an. Wir erhalten die folgenden Ausgaben:

11001111
11110010
10000111

Der erste Wert stellt das uns schon bekannte RESTART-Kommando dar. An zweiter Stelle finden wir das LOW-BYTE der anzuspringenden ROM-Adresse. Das dritte Byte gibt nun die schon besprochene Kombination von High-Adress und ROM-Auswahl wieder. Bit 15 ist gesetzt, mithin das obere ROM ausgeschaltet. Da das untere ROM adressiert ist bzw. noch adressiert werden soll (denn zu diesem Zeitpunkt ist ja noch irgendein anderer Speicherzustand vorhanden), ist Bit 14 nicht gesetzt.

Der RESTART-Aufruf stellt ebenso wie das CALL-Kommando einen Unterprogrammaufruf dar. Bei seiner Ausführung wird also auch die Rücksprungadresse auf dem STACK abgelegt. In Position 008 steht ein einfacher Sprung, der wieder in den Bereich der Firmwaresprünge (nämlich in die Routine B982) zurückführt. Dies ist die Ansprungadresse der eigentlichen LOW-JUMP-RESTART-Routine, d.h. jenes Programms, welches die ROM-Umschaltung bewirkt und auch den Ansprung der entsprechenden ROM-Routine vornimmt.



Wir wollen uns diese Routine etwas näher anschauen, weil sie den STACK exzessiv nutzt und darüber hinaus eine Vielzahl der bis jetzt verwendeten Befehle anwendet.

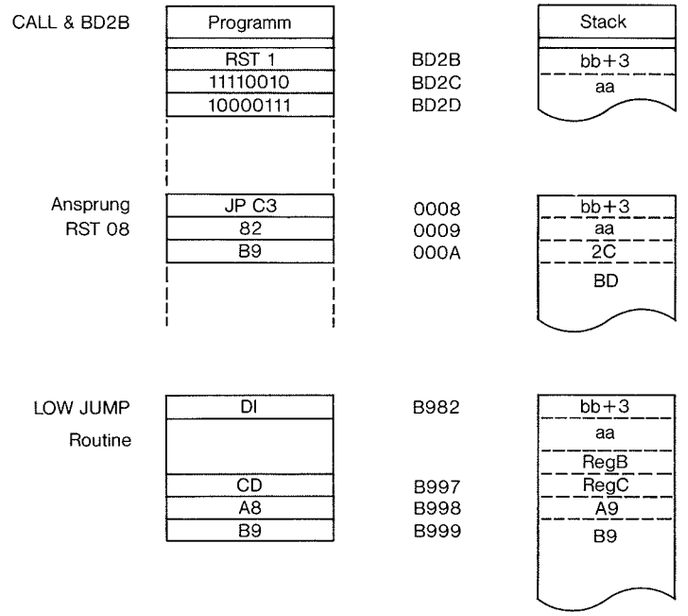



Bild 5.3: STACK-Ablauf bei LOW-JUMP-RESTART

Assemblerlisting LOW JUMP RESTART

```
B982    F3       DI          Unterbrechungen verhindern
B983    D9       EXX
B984    E1       POP HL
B985    5E       LD E,(HL)   Ansprung LOW
B986    23       INC HL
B987    56       LD D,(HL)   Ansprung HIGH
B988    08       EX AF,AF'
B989    7A       LD A,D
B98A    CBBA     RES 7,D
B98C    CBB2     RES 6,D
B98E    07       RLCA        ROM-Bit in Bit 2 und 3 rotieren
B98F    07       RCLA
B980    07       RCLA
B991    07       RCLA
B992    A9       XOR C       und mit Register C verknüpfen
B993    E60C     AND 0C
B995    A9       XOR C
B996    C5       PUSH BC     Alte Speicheraufteilung sichern
B997    CDA8B9   CALL B9A8   und Routineanspringen
B99A    F3       DI          Rückkehr aus ROM-Routine
B99B    D9       EXX
B99C    08       EX AF,AF'
B99D    79       LD A,C
B99E    C1       POP BC
B99F    E603     AND 03
B9A1    CB89     RES 1C
B9A3    CB81     RES 0C
B9A5    B1       OR C
B9A6    1801     JR 1 = B9A9
B9A8    D5       PUSH DE     ROM-Routine
B9A9    4F       LD C,A
B9AA    ED49     OUT (C)C
B9AC    B7       OR A
B9AD    08       EX AF,AF'
B9AE    D9       EXX
B9AF    FB       EI          IP
B9B0    C9       RET         ins UP oder zurück
```



Den Anfang des Programms bilden einige schon bekannte Kommandos. Zunächst werden Unterbrechungen abgestoppt, danach wird auf den Parallelregistersatz umgeschaltet.

Es folgt ein etwas seltsamer Befehl: POP HL. Mit diesem Kommando wird die Rücksprungadresse (in diesem Fall also BD2C) in HL übernommen und der STACK-Pointer um 2 erhöht. Er zeigt also jetzt wieder auf AA.

Anschließend kommen zwei indirekte Ladebefehle, mit denen nun die hinter dem RESTART liegenden beiden Bytes, d.h. der Inhalt von BD2C und BD2D, in Register E und D geladen wird. D Enthält dabei das höherwertige Byte, was in unserem Fall dem Inhalt von BD2D entspricht.

Als nächstes (Speicherstelle B989) laden wir den Akkumulator mit D und setzen danach die beiden höchsten Bit von D, welche ja bekanntlich die ROM-Konfiguration decodieren, auf Null zurück. Dies ist eine vorsorgliche Maßnahme. D und E geben jetzt wirklich als 16-Bit-Adresse den Ansprungpunkt im unteren ROM. A wird dagegen nun viermal rotiert. Damit befinden sich die beiden relevanten Bit jetzt in Bit 2 und Bit 3.

Wenn Ihnen dies nicht klar sein sollte, können Sie den Verschiebevorgang ja einmal simulieren. Zu diesem Zweck brauchen Sie nur auf ein Blatt Papier acht Felder aufzumalen; dann können Sie mit Reißzwecken, Büroklammern oder ähnlichen Utensilien das Verschieben der Bits durchspielen.

Zurück zu unserem Programm. Bit 2 und Bit 3 sind genau jene Bits, in denen auch im Register C die ROM-Konfiguration gespeichert ist. Wir erinnern uns: Register C im Parallelregistersatz enthält eine Kopie des Multifunktionsregisters im ULA (vgl. Kapitel 3). Zunächst werden die Register A und C OR-verknüpft, danach mit dem Wert 0C maskiert. Hier bleiben nur noch die Inhalte von B2 und B3 erhalten, das Ganze wird dann wiederum XOR-verknüpft.

Damit enthält der Akkumulator die geänderte Version des Registers C; also diejenige, in welcher die neue ROM-Konfiguration bereits gespeichert ist. Die alte Speicherverteilung, die sich noch in BC befindet, wird nun zunächt auf dem STACK gesichert. Danach erfolgt ein Unterprogrammsprung nach B9A8. Dabei wird wiederum die Rücksprungadresse B99A auf dem Stapel abgelegt.

Was jetzt folgt, ist ein wirklich schöner Trick. Es wird DE, d.h. die Ansprungadresse im unteren ROM, auf den Stapel gepusht. Danach schalten

wir (wie schon in den Anwendungen zu Kapitel 5.2 gesehen) die Speicherverteilung mit dem OUT-Befehl um. Interrupts werden wieder zugelassen (B9AF); somit erfolgt nun ein Rücksprung aus einem Unterprogramm.

Da aber als letztes DE auf dem Stapel abgelegt wurde, findet nun ein Ansprung der Routine im unteren ROM statt. Wir haben also durch diesen Umweg einen indirekten Unterprogrammaufruf erzeugt. Die ROM-Routine 07F2 (MC PRINT CHAR) gibt ihrerseits wiederum mit RETURN an das Oberprogramm zurück.

Da der STACK zu diesem Zeitpunkt sauber abgearbeitet ist, trifft der Prozessor nun wieder einmal auf eine wirkliche Rücksprungadresse, nämlich die durch RECALL-Befehl abgelegte B99A. An dieser Stelle kehrt das Unterprogramm wieder in LOW-JUMP-RESTART zurück.

Zunächst wird wiederum auf Parallelregistersatz umgeschaltet, danach erfolgt durch POP-BC und die nachfolgenden Operationen die Rückschaltung auf die alte Speicherbereichsverteilung. Diese wird unter nochmaliger Benutzung der Teile ab B9A8 (diesmal allerdings ohne das PUSH-Kommando, welches durch den relativen Sprung in B9A6 übersprungen wird) ausgegeben.

Abschließend kehrt LOW-JUMP-RESTART mit dem RETURN-Befehl wieder in unser Hauptprogramm (genauer an die Stelle AABB+3) zurück, womit die Benutzung der Systemroutine abgeschlossen ist.

Sie sehen: Bei entsprechender Kombination der verschiedenen Sprungbefehle lassen sich mit minimalem Aufwand auch sehr komplexe Programme realisieren; allerdings sollten Sie bei Ihren ersten Experimenten nicht gleich auf derartige Tricks verfallen. Hier ist bereits einiges Training und vor allem genaueste Kalkulation notwendig. Ein solches Programm muß gedanklich Schritt für Schritt mit allen Varianten und Möglichkeiten durchgespielt werden.



## 5.6 Systemsteuerbefehle

Die bis jetzt behandelten Kommandos des Z80A verfügten alle über eine Gemeinsamkeit: die Kontrolle ging nämlich fast vollständig vom Prozessor aus. Er forderte Daten an, verknüpfte sie und sandte sie an Speicher- oder externe Bausteine weiter.

Die im folgenden beschriebenen Befehle ermöglichen es, den Prozessor einem anderen Gerät unterzuordnen oder ihn mit diesem zu koordinieren. Dazu verfügt der Prozessorchip über zwei Eingänge:

INT (INTERRUPT = Unterbrechungsanfrage)

und

NMI (NON MASKABLE INTERRUPT = nicht maskierbarer Interrupt).

Treten an diesen Pints Signale auf, so arbeitet der Prozessor erst den gerade anhängigen Befehl ab; dann führt er einen Sprung zur Unterbrechungsbehandlungsroutine durch, die dann auf den Interrupt reagieren muß. Der nicht maskierbare Interrupt ruft dabei in jeden Fall die Unterbrechungsbehandlungsroutine auf; ein Signal am IRQ-Eingang dagegen nur, wenn Unterbrechungen erlaubt sind. Wir können mit dem Kommando

DI (DISABLE INTERRUPT: F3)

einen solchen Interrupt sperren, mit

EI (ENABLE INTERRUPT: FB)

solche Interrupts wieder zulassen. Die Rückkehr aus einer Interruptbehandlungsroutine erfolgt mit den Kommandos

RET I (RETURN FROM INTERRUPT: ED 4D)

und

RET N (RETURN FROM NOT MASKABLE INTERRUPT:ED 45).

Dabei ist zu beachten, daß vor RET I ein EI ausgeführt werden muß, um Interrupts wieder freizugeben. Damit sind dann auch verschachtelte Interrupts möglich.

Es stellt sich nun die Frage, wie die Ansprungadresse der Interruptbehandlungsroutine dem Prozessor mitgeteilt wird. Der Z80 verfügt über drei Möglichkeiten (die sogenannten Interruptmodi):

**IM 0** (setze Interruptmodus 0) **CODE:** ED 46

In dieser Interruptbetriebsart kann der unterbrechende Baustein einen Befehl zur Ausführung auf den Datenbus legen. Der Prozessor nimmt ihn dann herein und führt ihn aus.

**IM 1** (setze Interruptmodus 1) **CODE:** ED 56

Tritt hier ein Interrupt auf, so führt der Prozessor ein RST 38 aus (siehe Sprungbefehle, Kapitel 5.4). Dies ist auch die Betriebsart, in der beim CPC Interrupts behandelt werden. An dieser Stelle befindet sich ein Sprung zur Interruptbehandlungsroutine, die dann via RETURN wieder in das unterbrochene Programm zurückkehrt.

Interrupts treten beim CPC ziemlich häufig auf; genaugesagt 300 mal jede Sekunde. In diesem Rhythmus arbeitet der CPC nämlich schnelle Ereignisse im Bereich der Bildschirmsteuerung etc. ab. Daneben existiert noch ein 6mal langsamerer Takt (gleich 50 Hertz), der z. B. für die Tastaturabfrage benutzt wird. Das Unterbrechungssignal wird dabei im Endeffekt vom ULA geliefert.

**IM 2** (setze Interruptmodus 2) **CODE:** ED 5E

In diesem Modus muß der unterbrechende Baustein ein Datenbyte liefern, welches dann als untere Hälfte, d.h. LOWER-ADRESS-BYTE, benutzt wird. Das HIGHER-ADRESS-BYTE wird dem Register I entnommen. Dieser kombinierte Adressvektor adressiert nun eine Speicherzelle. Der Inhalt dieser Speicherzelle wird in den Befehlszähler geladen. Dort beginnt die Ausführung.

Neben den Interruptkommandos existieren noch zwei Befehle, welche die CPU dazu bringen, nichts zu tun:

**NOP** (keine Operation) **CODE:** 00

Trifft die CPU auf dieses Kommando, tut sie einen Maschinenzyklus lang nichts. Das Kommando kann für sehr schnelle Verzögerungen oder auch



dazu benutzt werden, in einem Maschinenprogramm Speicherplatz zu reservieren.

Da der NOP-Befehl keinerlei Veränderungen durchführt, ist es später möglich, anstatt einiger NOPs im Programm noch andere Befehle einzufügen.

**HALT** (CPU anhalten) **CODE:** 76

Hierbei handelt es sich um ein fortgesetztes NOP. Die CPU unterbricht so lange die Operation und führt NOPs aus, bis ein Interrupt oder ein RESET eintritt.

Dabei sind sämtliche Ein- und Ausgänge hochohmig geschaltet. Die CPU trennt sich damit quasi vom Restsystem ab, was es einem anderen Baustein ermöglicht, die volle Kontrolle über das System zu übernehmen.

Dies ist z.B. dann vorteilhaft, wenn größere Datenmengen von der Floppy durch Direktadressierung in den Speicher übertragen werden sollen oder umgekehrt. Will man dazu diese nicht erst in die CPU laden und dann wieder in den Speicher wegschreiben, so kann man auf sogenannte DMA (DIRECT MEMORY ACCESS = direkter Speicherzugriff) zurückgreifen, die dann den Speicher direkt adressieren und so einen viel schnelleren Datentransfer ermöglichen. Beim CPC wird dieses Verfahren allerdings nicht angewandt.

# 6 Ein Disassembler für den CPC

Im letzten Kapitel haben wir den kompletten Befehlssatz des Z80-Prozessors kennengelernt. Der Z80 verfügt über eine Vielzahl verschiedener Befehle, mit denen die verschiedensten und teilweise auch komplexesten Funktionen nachgebildet werden können. Allerdings hat diese Medaille auch eine Kehrseite. Der Umfang des Befehlssatzes macht es nämlich unmöglich, alle Befehle im Kopf zu behalten, und auch das Durchsuchen längerer Tabellen ist relativ aufwendig. Schon früh hat man deshalb für die Maschinenspracheprogrammierung Hilfsprogramme entwickelt:

### Assembler und Disassembler

Basis beider Programmtypen sind die sogenannten Mnemonics, Kürzel für die Maschinencodes der Prozessorsprache. Ein Assembler übersetzt Mnemonics in Maschinencodes, ordnet also den symbolischen Befehlsnamen (wie LD, XOR, JP und so weiter) die entsprechenden hex-Befehle zu. Ein Disassembler arbeitet in der umgekehrten Richtung. Er übersetzt ein im hex-Code abgespeichertes Maschinenspracheprogramm, Speicherstelle für Speicherstelle, in die Befehle, daß heißt Mnemonics, zurück.

Im folgenden Kapitel wollen wir nun einmal einen solchen Disassembler für unseren CPC entwickeln. Machen wir uns zunächst ein paar Gedanken darüber, wie wir beim Aufbau eines solchen Programms vorgehen müssen.

Schon bei der Behandlung der einzelnen Befehle und der zugehörigen Codes wird Ihnen aufgefallen sein, daß innerhalb der einzelnen Befehlsgruppen bestimmte Gesetzmäßigkeiten herrschen. Der Befehlssatz des Prozessors ist also nicht willkürlich, sondern nach bestimmten Regeln aufgebaut. So konnten wir z.B. im Bereich der 8-Bit-Ladebefehle feststellen, daß sich das Laden zwischen 2 Registern binär betrachtet wie folgt decodieren läßt:

Die ersten beiden Bit (01) gaben die Befehlsgruppe an. Danach spezifizierten je 3 Bit das Ziel- und das Quellregister. Vergleichen Sie dazu einmal die Befehlstabelle zu den 8-Bit-Ladebefehlen in Kapitel 5.3. Jedem Register und auch jedem Registerpaar sind also Codes zugeordnet, nach denen der Prozessor sie identifiziert und die entsprechenden Operationen ausführt.

Schlagen wir noch einmal zurück auf den Beginn von Kapitel 5. Bei der Einleitung zu den verschiedenen Befehlen hatten wir auch eine andere Art von Unterteilung kennengelernt: die Grundbefehle und die Erweiterungsbefehle. Wir müssen zwischen Ein-Byte-, Zwei-Byte- und Drei-Byte-OP-Codes, also Befehlscodes mit unterschiedlicher Länge unterscheiden.

Maximal kann ein Befehl 4 Byte lang sein. Befehle, die mehr als Ein-Byte für die Befehlsdecodierung benötigen, haben dabei als erstes Byte und Verweis auf die entsprechenden Erweiterungstabellen einen der Codes CB, DD, ED, oder FD. Neben der Aufteilung der Befehlscodes in verschiedene Bits und Gruppen von Bits können wir also auch noch eine Grobteilung in Grundbefehle und Erweiterungsbefehle vornehmen. Damit haben wir schon den Ordnungsrahmen für unseren Disassembler bestimmt.

Ein Problem bei der **Befehlsdecodierung** besteht nun noch darin, daß die Befehlslänge nicht konstant ist. Sie kann zwischen einem und vier Byte je Befehl variieren. Damit ist es notwendig, die Interpretation des nächsten Befehls vom Vorgänger abhängig zu machen. Der korrekte Anfang eines neuen Befehls innerhalb eines Maschinenprogramms läßt sich nur dadurch feststellen, daß man den Beginn seines Vorgängers kennt und anhand der Interpretation des vorangegangenen Kommandos dann feststellt, wieviel Byte dieser benötigt hat.

Ein Suchen nach bestimmten **TOKENs** oder ähnlichen Gliederungsmerkmalen in einem Maschinenprogramm ist nicht möglich. Diese Gedankengänge sollten Sie im Hinterkopf behalten, wenn wir uns nun mit dem Aufbau unseres Disassemblers beschäftigen.

Um eine möglichst gute **Integrationsfähigkeit** des Programms für die spätere Einpassung in den Monitor zu erhalten, wurde der eigentliche Disassembler als Prozedur ausgeführt. Diese Prozedur stellt anhand von vier aufeinanderfolgenden Byte den gespeicherten Befehl fest und gibt diesen zurück.



Der Disassembler benötigt dabei als Eingangswerte sechs Strings: die formatierten Binärwerte der ersten beiden Byte sowie das hex-Ergebnis aller vier Adressen.

Als Ausgabe liefert die Prozedur in dem String b$ den übersetzten Befehl und in der Variablen u die Länge des Befehls. Diese kann dann zum Weiterschalten eines Adresszählers benutzt werden.

Die Formatierung der Eingabewerte besagt dabei, daß die Binärwerte als acht Zeichen langer String, also gegebenenfalls mit Vornullen, und die hex-Werte mit zwei Ziffern angegeben werden müssen. Dies kann man relativ einfach erreichen, indem man in den Umwandlungsbefehlen bin$ und hex$ den Längenparameter spezifiziert. (Vergleiche Zeile 90 und Zeile 100 des nachfolgenden Programms.)

Um die Prozedur zu Testzwecken und auch für kleine Anwendungen schon vorab, das heißt vor Einpassung in den Monitor, benutzen zu können, sind im Listing zwei Programmteile vorgeschaltet. Mit dem REM-Vermerk "testeingaben" stoßen wir auf einen Programmteil, der es uns ermöglicht, vier Byte hintereinander einzugeben. Danach springt dieser Teil die Prozedur an und gibt den Befehl und die Befehlslänge in Bytes aus. Dieser Teil dient im wesentlichen dazu, die einzelnen Teile der Prozedur zu testen.

Am Anfang des Programms findet sich dann noch eine kleine Demonstrationsroutine, mit der wir alle Programme, die im RAM gespeichert sind, übersetzen können. Nach den üblichen Initialisierungsbefehlen, wie der Festlegung von Windows, Farben etc., wird die erste Speicherstelle des zu interpretierenden Maschinenprogramms abgefragt.

Diese wird in IC zwischengespeichert, und ab dieser Position wird mit der Interpretation der Befehle begonnen. Aus Zeile 110 erfolgt jedesmal der Sprung in die Prozedur. Danach wird die Ausgabe vorbereitet, eine Umformung des Befehlsstrings (Zeile 120) vorgenommen, worauf dann die eigentliche Ausgabe erfolgt.

## 6.1 Die Disassemblerprozedur

Schauen wir uns nun die eigentliche Prozedur an. Anhand der REM-Vermerke können wir die einzelnen Teile des Programms gut unterscheiden.

Zu Beginn des Programms treffen wir auf den Hauptsprungverteiler. Hier wird zunächst überprüft, ob der erste OP-Code den Verweis auf eine Erweiterungstabelle enthält, G also mit CB, DD, ED oder FD Übereinstimmt. Die Interpretation für CB beginnt ab Zeile 1090, für ED ab 1360. Die Tabellen für DD und FD unterscheiden sich nur in der Wahl des Indexregisters.

Beide Tabellen enthalten nur indizierte Lade-, Additions- und Sprunganweisungen. Der einzige Unterschied besteht darin, daß bei erstem Befehlscode DD das Register ix, bei FD dagegen iy benutzt wird. Deswegen wird dieselbe Routine für die Interpretation der Befehle benutzt, und schon im Vorfeld werden diese beiden Tabellen gleichgeschaltet, indem der String r$ auf die indizierte Variable gesetzt wird.

Ab Zeile 360 beginnt die Interpretation der Grundtabelle, das heißt der Ein-Byte-OP-Codes. In 360 erfolgt zunächst ein Sprung nach 420. In dem dort liegenden Unterprogramm wird der binäre Befehlsstring , also das erste Befehlsbyte, zerteilt. Dabei werden die drei Integervariablen b1%, b2% und b3% mit dem Wert der höchsten beiden Bit, der Bit drei bis fünf bzw. der niederwertigsten drei Bit des ersten Befehlscodes geladen.

Anhand dieser drei Variablen erfolgt nun die weitere Befehlsanalyse. Zunächst wird in Abhängigkeit von den höchsten beiden Bit, also von b1%, eines der vier Unterprogramme DISI 0 bis DISI 3 über den **Sprungverteiler** in Zeile 360 angesprungen. Innerhalb dieser Befehlsuntergruppen erfolgt die Befehlsanalyse mit Hilfe von DATA-Tabellen. Diese enthalten Register bzw. Registerpaare, wie sie den einzelnen Befehlsbits zugeordnet sind, und darüber hinaus natürlich die Mnemonics für die einzelnen Befehle.

Das Disassemblieren besteht nun darin, daß man aus den verschiedenen DATA-Tabellen die zugehörigen Codes und Register zusammensucht und auf geeignete Art und Weise miteinander verbindet. Das Prinzip ist relativ einfach, die Ausführung im wesentlichen Fleißarbeit. Hier soll nur eine Spezialität bei der Abspeicherung der verschiedenen DATAs betrachtet werden. Wenn Sie das Programm anschauen, werden Sie feststellen, daß des öfteren ein kleines "ö" im Programmlisting auftaucht, was Sie von den Befehlscodes nicht gewöhnt sind. Dies ist das Druckeräquivalent für den Erweiterungsstrich <Shift>+"@". Ein Problem bei der Abspeicherung von DATA-Statements besteht nämlich darin, daß es nicht möglich ist, in einer DATA-Zeile Kommas direkt abzuspeichern, da Kommas für die Trennung zwischen den einzelnen DATAS benützt werden.



Anstelle eines Kommas wird deswegen der Erweiterungsstrich benutzt. Bei der Ausgabe wird dieser dann mit einer Instringroutine wieder durch ein Komma ersetzt. (Vergleiche Zeile 120, Zeile 190). Wir wollen uns die Funktion der einzelnen Programmteile einmal an einigen Beispielen klarmachen.

**Beispiel:** Beginnen wollen wir dabei mit einem der am leichtesten zu interpretierenden Kommandos, dem direkten Laden eines acht Bit Wertes aus einem Universalregister in ein anderes

```
LD B,C.
```

Ein kurzes Rückschlagen in Kapitel 5.2 liefert uns den zugehörigen hex-Code: 41
oder binär : 01000001

Teilen wir diesen Binärstring nach dem zweiten und fünften Zeichen von links, so erhalten wir die Werte für unsere Integervariablen b1%, b2% und b3% mit 1, 0 und wiederum 1. Dies ist der Ausgangspunkt unserer Erörterung ab Zeile 360.

Da b1%=1 ist, wird in Zeile 370 das Unterprogramm ab 740 angesprungen, also DISI 1. Hier wird zunächst die Befehlslänge (u) auf 1 festgelegt und abgeprüft, ob es sich um den Befehl HALT handelt. Dies ist in unserem Fall nicht gegeben. Es folgt der Befehl RESTORE 560. Als aktuelle DATA-Zeile werden also die darin enthaltenen einzelnen Universalregister spezifiziert. Das nun angesprungene Unterprogramm in 1010 liest so lange Daten ein, bis der Zähler i der Variablen b2% gleicht. In unserem Fall (b2%=0) würde hier nur das erste DATA-Statement eingelesen, also "B".

Mit dem in z$ zwischengespeicherten Register kehren wir nach 750 zurück und fügen dies mit dem vorangestellten LOAD-Kommando (LD) in den Befehlsstring b$ ein. Es erfolgt eine Verschiebung: B2% wird gleich b3% gesetzt, und die Prozedur wiederholt sich. B3% ist 1, so daß nun in 1010 Zeile 560 bis zum "C" weitergelesen wird. Nach der Rückkehr nach 750 enthält z$ nun "C", und dieses wird nach dem als Komma fungierenden ö

an den Befehlsstring angehängt. Die Interpretation ist abgeschlossen, und es erfolgt die Rückkehr ins aufrufende Hauptprogramm.

Schauen wir uns also nun einmal an, wie sich dieser Vorgang für einen etwas komplexeren Befehl darstellt.

**Beispiel 2:** BIT 3, (IY+diff)

Dieses indizierte Testkommando stellt wohl einen der am schwierigsten zu interpretierenden Befehle dar. Zunächst einmal die hex-Codes:

FD CB dd 5E

Schauen wir uns nun den Ablauf vom Disassembler her an. Das erste Befehlsbyte ist FD, was dazu führt, daß in Zeile 350 r$ auf IY gesetzt wird, worauf dann der Sprung nach 1180 erfolgt. Auch die nächste Bedingung (Byte 2 =CB) ist erfüllt, so daß keine Verzweigung nach 1260 stattfindet, sondern das Programm normal durchläuft.

Da bei 3-Byte-OP-Codes das 4. Byte ( das 3. gibt ja die Differenz an) für die Befehlsdecodierung benötigt wird, müssen wir zunächst dieses in die drei üblichen Variablen b1%, b2% und b3% zerlegen. Dies erfolgt in 1190 mit dem Ansprung von 420. 5E läßt sich binär als 01 011 110 schreiben, womit wir die Werte für die einzelnen Register ablesen können: B1%=1, b2%=3, b3%=6

Da für alle Befehle dieser Erweiterungstabelle gilt, daß b3%=6 sein muß, können wir schon eine große Anzahl aller Befehlskombinationen als unsinnig (was durch ??? repräsentiert wird) ausscheiden. Es folgt die eigentliche Befehlsinterpretation (b1=1). Der nach der IF-Bedingung stehende Einzeiler leistet den Rest. Das RETURN führt wiederum ins Hauptprogramm zurück.

Auch die Interpretation der anderen Befehle läuft nach diesem Prinzip ab. Sie sollten nun einmal, nachdem Sie den Disassembler eingetippt haben, verschiedene Befehlscodes ausprobieren, um die korrekte Funktion zu überprüfen. Wenn Sie glauben, daß das Programm richtig läuft, können sie zu Größerem übergehen. Eine erste Möglichkeit besteht darin, daß Sie sich die Interpretationsroutine LOW JUMP, die wir in Kapitel 5.4 behandelt haben, einmal selbst mit dem Disassembler auslesen.



```
10 REM ******************
20 REM ** Disassembler **
30 REM ******************
40 ZONE 18
50 WINDOW#1,1,40,1,3:WINDOW#0,1,40,4,22:WINDOW#3,1,40,23,25
60 INK 0,0:INK 1,2:INK 2,6:INK 3,21:PAPER#1,1:PEN#1,2:PAPER 0:PEN
3:PAPER#3,1:PEN#3,2:CLS#1:CLS#2:CLS#3
70 LOCATE#1,15,2:PRINT#1,"DISASSEMBLER"
80 INPUT#3,"Beginn in Hex";ic$:ic=VAL("&"+ic$)
90 h1$=HEX$(PEEK(ic),2):h2$=HEX$(PEEK(ic+1),2):h3$=HEX$(PEEK(ic+2)
,2):h4$=HEX$(PEEK(ic+3),2)
100 b1$=BIN$(PEEK(ic),8):b2$=BIN$(PEEK(ic+1),8)
110 GOSUB 310:PRINT HEX$(ic,4)+"  ";:z$=h1$:IF u>1 THEN z$=z$+h2$:
IF u>2 THEN z$=z$+h3$:IF u>3 THEN z$=z$+h4$
120 IF INSTR(b$,"ö")<>0 THEN b$=LEFT$(b$,INSTR(b$,"ö")-1)+","+MID$
(b$,INSTR(b$,"ö")+1)
130 PRINT z$,b$:ic=ic+u
140 IF INKEY$=" " THEN 140 ELSE 90
150 '
160 'testeingaben
170 '
180 INPUT"h1$";h1$:INPUT"h2$";h2$:INPUT"h3$";h3$:INPUT"h4$";h4$:b1
$=BIN$(VAL("&"+h1$),8):b2$=BIN$(VAL("&"+h2$),8)
190 GOSUB 310:IF INSTR(b$,"ö")<>0 THEN b$=LEFT$(b$,INSTR(b$,"ö")-1
)+","+MID$(b$,INSTR(b$,"ö")+1):PRINT b$,u:GOTO 180
200 '
210 '
220 REM ***************
230 REM ** proc:DISI **
240 REM ***************
250 REM
260 REM ----------------------------
270 REM IN: b1$,b2$,h1$,h2$,h3$,h4$
280 REM OUT:b$,u
290 REM ----------------------------
300 REM
310 IF h1$="CB" THEN GOSUB 1090:RETURN
320 IF h1$="CB" THEN GOSUB 1090:RETURN
330 IF h1$="DD" THEN R$="IX":GOSUB 1180:RETURN
340 IF h1$="ED" THEN GOSUB 1360:RETURN
350 IF h1$="FD" THEN R$="IY":GOSUB 1180:RETURN
360 b0$=b1$:GOSUB 420:REM Binteil
370 ON b1%+1 GOSUB 500,740,790,840
380 RETURN
390 REM ******************
400 REM ** sub:Binteil **
410 REM ******************
420 b1%=VAL("&x"+LEFT$(b0$,2)):b2%=VAL("&x"+MID$(b0$,3,3)):b3%=VAL
("&x"+MID$(b0$,6))
430 RETURN
440 REM ************************
450 REM ** Ein-Byte-OP-Codes **
460 REM ************************
470 REM ****************
480 REM ** sub:DISI 0 **
490 REM ****************
```

```
500 IF b3%<4 OR b3%>6 THEN GOSUB 570 ELSE GOSUB 520
510 RETURN
520 RESTORE 560:GOSUB 1010
530 IF b3%=6 THEN b$="LD "+z$+","+h2$:u=2:RETURN
540 IF B3%=4 THEN B$="INC " ELSE B$="DEC "
550 B$=B$+Z$:U=1:RETURN
560 DATA B,C,D,E,H,L,(HL),A
570 DATA NOP,"EX AFöAF'",DJNZ,JR,"JR NZö","JR Zö","JR NCö","JR Cö"
580 DATA "LD BCö","ADD HLöBC","LD DEö","ADD HLöDE","LD HLö","ADD H
LöHL","LD SPö","ADD HLöSP"
590 DATA INC BC,DEC BC,INC DE,DEC DE,INC HL,DEC HL,INC SP,DEC SP
600 DATA RLCA,RRCA,RLA,RRA,DAA,CPL,SCF,CCF
610 DATA "LD(BC)öA","LD Aö(BC)","LD(DE)öA","LD Aö(DE)"
620 IF B3%=0 THEN RESTORE 570:GOSUB 1010:B$=Z$:IF B2%<2 THEN U=1:R
ETURN ELSE B$=B$+STR$(VAL("&"+H2$)+256*(VAL("&"+H2$)>127))+" = "+R
IGHT$("0000"+HEX$(ic+2+VAL("&"+H2$)+256*(VAL("&"+H2$)>127)),4):U=2
:RETURN
630 IF b3%=1 THEN RESTORE 580:GOSUB 1010:b$=z$:IF INT(b2%/2)<>b2%/
2 THEN u=1:RETURN ELSE b$=b$+h3$+h2$:u=3:RETURN
640 IF B3%=3 THEN RESTORE 590:GOSUB 1010:B$=Z$:u=1:RETURN
650 IF b3%=7 THEN RESTORE 600:GOSUB 1010:b$=z$:u=1:RETURN
660 IF B2%<4 THEN RESTORE 610:GOSUB 1010:b$=z$:u=1:RETURN
670 u=3:IF b2%=4 THEN b$="LD("+h3$+h2$+")öHL":RETURN
680 IF b2%=5 THEN b$="LD HLö("+h3$+h2$+")":RETURN
690 IF b2%=6 THEN b$="LD("+h3$+h2$+")öA":RETURN
700 b$="LD Aö("+h3$+h2$+")":RETURN
710 REM ************
720 REM ** DISI 1 **
730 REM ************
740 u=1:IF h1$="76" THEN b$="HALT":RETURN
750 RESTORE 560:GOSUB 1010:b$="LD "+z$:b2%=b3%:RESTORE 560:GOSUB 1
010:b$=b$+"ö"+z$:RETURN
760 REM ************
770 REM ** DISI 2 **
780 REM ************
790 u=1:RESTORE 800:GOSUB 1010:b$=z$:b2%=b3%:RESTORE 560:GOSUB 101
0:b$=b$+z$:RETURN
800 DATA "ADD Aö","ADC Aö","SUB ","SBC Aö","AND ","XOR ","OR ","CP
 "
810 REM ************
820 REM ** DISI 3 **
830 REM ************
840 IF b3%=7 THEN u=1:b$="RST "+HEX$(8*b2%):RETURN
850 IF b3%=0 THEN u=1:RESTORE 950:GOSUB 1010:b$="RET "+z$:RETURN
860 IF B3%=1 THEN RESTORE 960:GOSUB 1010:b$=z$:u=1:RETURN
870 IF b3%=6 THEN RESTORE 800:GOSUB 1010:b$=z$+h2$:u=2:RETURN
880 IF b3%=2 THEN RESTORE 950:GOSUB 1010:b$="JP "+z$+"ö"+h3$+h2$:u
=3:RETURN
890 IF b3%=4 THEN RESTORE 950:GOSUB 1010:b$="CALL "+z$+"ö"+h3$+h2$
:u=3:RETURN
900 IF b3%=3 AND b2%=0 THEN b$="JP "+h3$+h2$:u=3:RETURN
910 IF b3%=5 AND b2%=1 THEN b$="CALL "+h3$+h2$:u=3:RETURN
920 IF b3%=5 THEN RESTORE 970:b2%=b2%/2:GOSUB 1010:b$="PUSH "+z$:u
=1:RETURN
930 IF b2%>3 THEN RESTORE 970:GOSUB 1010:b$=z$:u=1:RETURN
940 u=2:IF b2%=2 THEN b$="OUT("+h2$+")öA":RETURN ELSE b$="IN Aö("+
h2$+")":RETURN
```



```
950 DATA NZ,Z,NC,C,PO,PE,P,M
960 DATA "POP BC","RET","POP DE","EXX","POP HL","JP(HL)","POP AF",
"LD SPöHL"
970 DATA BC,DE,HL,AF,"EX(SP)öHL","EX DEöHL","DI","EI"
980 '
990 ' Befehl aus DATA lesen
1000 '
1010 FOR I=0 TO B2%:READ Z$:NEXT I:RETURN
1020 REM ****************************
1030 REM ** Erweiterungskommandos **
1040 REM ****************************
1050 '
1060 REM *************
1070 REM ** DISI CB **
1080 REM *************
1090 DATA RLC,RRC,RL,RR,SLA,SRA,???,SRL
1100 b0$=b2$:GOSUB 420:u=2:IF b1%=0 THEN RESTORE 1090:GOSUB 1010:b
$=z$:RESTORE 560:b2%=b3%:GOSUB 1010:IF B$<>"???" THEN b$=b$+" "+z$
:RETURN ELSE RETURN
1110 V=B2%:B2%=B3%:RESTORE 560:GOSUB 1010
1120 IF b1%=1 THEN b$="BIT"+STR$(V)+"ö"+z$:RETURN
1130 IF b1%=2 THEN b$="RES"+STR$(V)+"ö"+z$:RETURN
1140 IF b1%=3 THEN b$="SET"+STR$(V)+"ö"+z$:RETURN
1150 REM *****************
1160 REM ** DISI DD+FD **
1170 REM *****************
1180 IF h2$<>"CB" THEN 1260
1190 U=1:b0$=RIGHT$("00000000"+BIN$(VAL("&"+h4$)),8):GOSUB 420:IF
b3%<>6 THEN b$="???":RETURN
1210 DATA RLC,RRC,RL,RR,SLA,SRA,???,SRL
1220 U=4:IF B1%=0 THEN RESTORE 1210:GOSUB 950:IF h4$="36" THEN b$=
z$:u=1:RETURN ELSE B$=Z$+"("+R$+"+"+H3$+")":RETURN
1230 IF B1%=1 THEN B$="BIT"+STR$(B2%)+"ö("+R$+"+"+H3$+")":RETURN
1240 IF B1%=2 THEN B$="RES"+STR$(B2%)+"ö("+R$+"+"+H3$+")":RETURN
1250 B$="SET"+STR$(B2%)+"ö("+R$+"+"+H3$+")":RETURN
1260 b0$=B2$:GOSUB 420:IF B1%=3 THEN U=2:IF H2$="E5" THEN B$="PUSH
 "+R$:RETURN ELSE IF H2$="E1" THEN B$="POP "+R$:RETURN ELSE IF H2$
="E9" THEN B$="JP("+R$+")":RETURN ELSE IF H2$="E3" THEN B$="EX(SP)
ö"+R$:RETURN ELSE IF H2$="F9" THEN B$="LD SPö"+R$:RETURN
1270 IF B1%=2 THEN IF B3%=6 THEN RESTORE 800:GOSUB 950:B$=Z$+"("+R
$+"+"+H3$+")":U=3:RETURN
1280 IF B1%=1 THEN U=3:IF B2%=6 AND B3%=6 THEN U=1:B$="???":RETURN
 ELSE IF B2%=6 THEN RESTORE 560:B2%=B3%:GOSUB 950:B$="LD("+R$+"+"+
H3$+")ö"+Z$:RETURN ELSE IF B3%=6 THEN RESTORE 560:GOSUB 950:B$="LD
 "+Z$+"ö("+R$+"+"+H3$+")":RETURN
1290 IF B1%<>0 THEN 1320 ELSE U=2:IF H2$="09" THEN B$="ADD "+R$+"ö
BC":RETURN ELSE IF H2$="19" THEN B$="ADD "+R$+"öDE":RETURN ELSE IF
 H2$="29" THEN B$="ADD "+R$+"ö"+R$:RETURN ELSE IF H2$="39" THEN B$
="ADD "+R$+"öSP":RETURN
1300 IF H2$="23" THEN B$="INC "+R$:RETURN ELSE IF H2$="2B" THEN B$
```

```
="DEC "+R$:RETURN ELSE IF H2$="34" THEN B$="INC("+R$+"+"+H3$+")":U
=3:RETURN ELSE IF H2$="35" THEN B$="DEC("+R$+"+"+h3$+")":u=3:RETUR
N ELSE U=4
1310 IF H2$="36" THEN B$="LD("+R$+"+"+H3$+")ö"+H4$:RETURN ELSE IF
H2$="22" THEN B$="LD("+H4$+H3$+")ö"+R$:RETURN ELSE IF H2$="2A" THE
N B$="LD "+R$+"ö("+H4$+H3$+")":RETURN ELSE IF H2$="21" THEN B$="LD
 "+R$+"ö"+H4$+H3$:RETURN
1320 U=1:B$="???":RETURN
1330 REM **************
1340 REM ** DISI ED **
1350 REM **************
1360 BO$=B2$:GOSUB 420:IF B1%>2 OR B1%=0 THEN 1470
1370 IF B1%=2 AND(B2%<4 OR B3%>3) THEN 1470
1380 IF B1%=2 THEN B2%=B2%-4+4*B3%:RESTORE 1390:GOSUB 950:U=2:B$=Z
$:RETURN
1390 DATA LDI,LDD,LDIR,LDDR,CPI,CPD,CPIR,CPDR,INI,IND,INIR,INDR,OU
TI,OUTD,OTIR,OTDR
1400 IF B3%>3 THEN U=2:IF H2$="44" THEN B$="NEG":RETURN ELSE IF H2
$="4D" THEN B$="RETI":RETURN ELSE IF H2$="45" THEN B$="RETN":RETUR
N ELSE IF H2$="46" THEN B$="IM 0":RETURN ELSE IF H2$="56" THEN B$=
"IM 1":RETURN ELSE IF H2$="5E" THEN B$="IM 2":RETURN
1410 IF B3%>3 THEN IF H2$="57" THEN B$="LD AöI":RETURN ELSE IF H2$
="47" THEN B$="LD IöA":RETURN ELSE IF H2$="6F" THEN B$="RLD":RETUR
N ELSE IF H2$="67" THEN B$="RRD":RETURN ELSE IF H2$="4F" THEN B$="
LD R,A":RETURN ELSE IF H2$="5F" THEN B$="LD A,R":RETURN
1415 IF B3%>3 THEN 1470
1420 DATA "SBC HLöBC","ADC HLöBC","SBC HLöDE","ADC HLöDE","SBC HLö
HL","ADC HLöHL","SBC HLöSP","ADC HLöSP"
1430 IF B3%=2 THEN RESTORE 1420:GOSUB 1010:B$=Z$:U=2:RETURN
1440 IF B3%=3 THEN U=4:IF B2%=0 THEN B$="LD("+H4$+H3$+")öBC":RETUR
N ELSE IF B2%=1 THEN B$="LD BCö("+H4$+H3$+")":RETURN ELSE IF B2%=2
 THEN B$="LD("+H4$+H3$+")öDE":RETURN ELSE IF B2%=3 THEN B$="LD DEö
("+H4$+H3$+")":RETURN
1450 IF B3%=3 THEN IF B2%=4 THEN B$="LD("+H4$+H3$+")öHL":RETURN EL
SE IF B2%=5 THEN B$="LD HLö("+H4$+H3$+")":RETURN ELSE IF B2%=6 THE
N B$="LD("+H4$+H3$+")öSP":RETURN ELSE IF B2%=7 THEN B$="LD SPö("+H
4$+H3$+")":RETURN ELSE 1470
1460 IF B2%=6 THEN 1470 ELSE U=2:RESTORE 560:GOSUB 950:IF B3%=0 TH
EN B$="IN "+Z$+"ö(C)":RETURN ELSE B$="OUT (C)ö"+Z$:RETURN
1470 U=1:B$="???":RETURN
1480 DATA ööööööö"
```

Mit dem Disassembler verfügen wir nun über ein ausgezeichnetes Werkzeug, um Maschinenprogramme zu analysieren. Unangenehmerweise stellt aber nun nicht jedes Byte im Adressraum des CPC ein Maschinenprogramm dar, so daß es angenehm wäre, wenn wir zusätzlich noch die Möglichkeiten unseres Monitors zur Verfügung hätten. Wir müssen also den Disassembler mit CPC-MON verbinden. Den Platz dafür hatten wir ja sowieso schon reserviert.



Die am Disassembler vorzunehmenden Änderungen sind dabei denkbar gering. Der Teil Testeingaben kann natürlich entfallen. Auch ist die Angabe der WINDOWS und der Farben nicht mehr notwendig. Dies wird nun vom Hauptprogramm übernommen. Daneben sind einige Anpassungen in bezug auf die Adressierung notwendig, um es zu ermöglichen, daß der Disassembler sowohl aus dem ROM als auch aus dem RAM liest.

Zur Analyse des ROM benutzen wir wiederum ROMREAD, das ja als Unterprogramm im Monitor zur Verfügung steht. Unsere hex-Strings greifen in diesem Fall nicht direkt auf die Speicherstellen zu, sondern werden aus 40050 bis 40053 geladen. Auch ist es in der geänderten Version möglich, wahlweise auf dem Bildschirm oder auf dem Drucker auszugeben. Die Umschaltung geschieht dabei, wie schon vom Monitor gewohnt, mit der Funktion A.

Der erweiterte Monitor verfügt auch noch über 2 andere neu integrierte Routinen. Zum ersten haben wir 'gomach' in das Programm eingebunden (Zeile 2150 bis 2270), und auch eine Blocktransferroutine findet sich einige Zeilen darüber. Der Maschinenladeteil von GOMACH wurde hinter die entsprechende Laderoutine von ROMREAD, wie schon in Kapitel 4 beschrieben, abgelegt; das Programm wurde also gesplittet.

Wenn Sie ein bißchen blättern und die Programme vergleichen, werden Ihnen die Unterschiede schnell klarwerden, und es dürfte dann ein leichtes sein, die Programme mit MERGE und RENUM zusammenzubauen. Als Vergleichsgröße hier noch einmal das jetzt endgültig fertige Listing unseres Monitors.

```
10 ' *************
20 ' ** CPC-MON **
30 ' *************
40 '
50 ' Initialisierung
60 '
70 ON ERROR GOTO 910
80 MODE 1
90 INK 0,0:INK 1,21:INK 2,2:INK 3,24:WINDOW#0,1,40,4,25:WINDOW#1,1
,40,1,3
100 PAPER 0:PEN 1:PAPER#1,2:PEN#1,3:CLS:CLS#1
110 MEMORY 39999
120 '
130 ' data romread
140 '
150 DATA cd,00,b9,cd,06,b9,47,21,00,00
160 DATA 7e,32,72,9c,23,7e,32,73,9c,23
170 DATA 7e,32,74,9c,23,7e,32,75,9c
180 DATA 78,cd,0c,b9,c9,x
190 i=40000
200 READ a$:IF a$="x" THEN 300
210 POKE i,VAL("&"+a$):i=i+1:GOTO 200
220 '
230 ' data gomach
240 '
250 DATA ed,73,79,9c,31,b0,a2,cd,00,00
260 DATA f5,c5,d5,e5,dd,e5,fd,e5,3e,00
270 DATA 32,7b,9c,ed,73,77,9c,ed,7b
280 DATA 79,9c,c9
290 DATA x
300 i=40060
310 READ a$:IF a$="x" THEN 360
320 POKE i,VAL("&"+a$):i=i+1:GOTO 310
330 '
340 ' variablen initialisieren
350 '
360 romval=40008:romread=40000
370 rom=0:LOCATE#1,34,1:PRINT#1,"RAM"
380 stream=0:LOCATE#1,4,1:PRINT#1,"Schirm"
390 '
400 ' Befehlsabfrageschleife
410 '
420 INPUT b$
430 c$=LOWER$(LEFT$(b$,1))
440 IF c$="m" THEN 610
450 IF c$="r" THEN 880
460 IF c$="k" THEN 1580
470 IF c$="t" THEN 930
480 IF c$="c" THEN 1450
490 IF c$="l" THEN 1250
500 IF c$="s" THEN 1060
510 IF c$="a" THEN 1190
520 IF c$="p" THEN 1370
530 IF c$="x" THEN END
540 IF c$="b" THEN 2040
550 IF c$="g" THEN 2150
560 IF c$="d" THEN 2310
```



```
570 IF c$="h" THEN 1700
580 IF c$="i" THEN 1910
590 GOTO 420
600 '
610 ' memory anzeigen
620 '
630 z$="":y$=""
640 w=INSTR(b$,";")
650 IF w=0 THEN an=VAL(MID$(b$,3)):en=an+112 ELSE an=VAL(MID$(b$,3
,w-3)):en=VAL(MID$(b$,w+1))
660 IF en<0 THEN en=en+65536
670 IF an<0 THEN an=an+65536
680 PRINT
690 FOR i=an TO en STEP 8
700 FOR j=0 TO 1
710 IF rom=0 THEN 740
720 s2=INT((i+4*j)/256):s1=i+4*j-256*s2
730 POKE romval,s1:POKE romval+1,s2:CALL romread:GOTO 760
740 FOR k=0 TO 3:POKE 40050+k,PEEK(i+4*j+k)
750 NEXT k
760 FOR k=0 TO 3
770 z$=z$+RIGHT$("0"+HEX$(PEEK(40050+k)),2)+" "
780 IF PEEK(40050+k)<32 THEN y$=y$+CHR$(24)+CHR$(63)+CHR$(24) ELSE
 y$=y$+CHR$(PEEK(40050+k))
790 NEXT k,j
800 PRINT#stream*(-8),RIGHT$("000"+HEX$(i),4)+"  "+z$+" "+y$
810 z$="":y$=""
820 NEXT i
830 PRINT
840 GOTO 590
850 '
860 ' romchange
870 '
880 rom=NOT(rom)
890 LOCATE#1,34,1:IF rom=0 THEN PRINT#1,"RAM" ELSE PRINT#1,"ROM"
900 GOTO 590
910 IF ERR=7 THEN PRINT"Speicherbereich belegt!!":RESUME 590
920 PRINT:PRINT"Falsche Befehlseingabe!!":RESUME 590
930 '
940 'Texteingabe
950 '
960 LINE INPUT"Bitte geben Sie den Text ein (max.255 Zeichen)";z$
970 INPUT"Ab welcher Speicherstelle soll der Text liegen";w
980 IF w<0 THEN w=w+65536
990 FOR i=w TO w+LEN(z$)-1
1000 POKE i,ASC(MID$(z$,i-w+1,1))
1010 NEXT i
1020 GOTO 590
1030 '
1040 ' Save
1050 '
1060 INPUT"Erstes Byte des Bereiches";an
1070 IF an<0 THEN an=an+65536
1080 INPUT"Letztes Byte des Bereiches";en
1090 IF en<0 THEN en=en+65536
1100 IF en<=an THEN PRINT"falscher Bereich!!"
```

```
1110 INPUT"Name des Files";n$
1120 PRINT"Speed Write 1 j/n"
1130 z$=LOWER$(INKEY$):IF z$="j" THEN SPEED WRITE 1 ELSE IF z$="n"
 THEN SPEED WRITE 0 ELSE 1130
1140 SAVE n$,b,an,en-an
1150 GOTO 590
1160 '
1170 ' Ausgabegeraet umschalten
1180 '
1190 stream=NOT(stream)
1200 LOCATE#1,4,1:IF stream=0 THEN PRINT#1,"Schirm " ELSE PRINT#1,
"Drucker"
1210 GOTO 590
1220 '
1230 ' File laden
1240 '
1250 PRINT
1260 INPUT"Name des Files";n$
1270 PRINT"Neue Adresse j/n"
1280 z$=LOWER$(INKEY$):IF z$="n" THEN LOAD ""+n$
1290 IF z$<>"j" THEN 1280
1300 INPUT"Ab welcher Adresse laden";w
1310 IF w<0 THEN w=w+65536
1320 LOAD ""+n$,w
1330 GOTO 590
1340 '
1350 ' Programmobergrenze festlegen
1360 '
1370 PRINT:PRINT"Alte Programmobergrenze";HIMEM
1380 INPUT"Neue Programmobergrenze (HIMEM)";w
1390 IF w<0 THEN w=w+65536
1400 MEMORY w
1410 PRINT:GOTO 590
1420 '
1430 ' Zahlen konvertieren
1440 '
1450 PRINT:INPUT"Zu konvertierende Zahl";w
1460 IF w<0 THEN w=w+65536
1470 IF w>255 THEN 1510
1480 PRINT"  dezimal   hexa       dual"
1490 PRINT"    "+RIGHT$("   "+MID$(STR$(w),2),3)+"       "+RIGHT$("
0"+HEX$(w),2)+"       "+RIGHT$("0000000"+BIN$(w),8)
1500 PRINT:GOTO 590
1510 PRINT"  dezimal     hexa          dual"
1520 PRINT"    "+RIGHT$("     "+MID$(STR$(w),2),5)+"       "+RIGHT$(
"000"+HEX$(w),4)+"    "+RIGHT$("000000000000000"+BIN$(w),16)
1530 PRINT
1540 GOTO 590
1550 '
1560 ' Speicherbereich loeschen
1570 '
1580 INPUT"Erstes zu loeschendes Byte";an
1590 INPUT"Letztes zu loeschendes Byte";en
1600 IF an<0 THEN an=an+65536
1610 IF en<0 THEN en=en+65536
1620 PRINT"Loeschen von";an;"bis";en;"j/n"
```



```
1630 z$=LOWER$(INKEY$)
1640 IF z$="n" THEN 590
1650 IF z$<>"j" THEN 1630
1660 FOR i=an TO en:POKE i,0:NEXT i:GOTO 590
1670 '
1680 ' Help
1690 '
1700 CLS:PRINT"Befehlsvorrat:":PRINT
1710 PRINT"A Ausgabegeraet Drucker/Schirm"
1720 PRINT"B Blocktransfer"
1730 PRINT"C Zahlensysteme konvertieren"
1740 PRINT"D Programm disassemblieren"
1750 PRINT"G Programm anspringen"
1760 PRINT"H Hilfsliste ausgeben"
1770 PRINT"I Speicherstellen eingeben"
1780 PRINT"K Speicherbereiche loeschen"
1790 PRINT"L Speicherbereich laden"
1800 PRINT"M Speicherbereich darstellen"
1810 PRINT"P Programmobergrenze festlegen"
1820 PRINT"R ROM/RAM-Umschaltung"
1830 PRINT"S Speicherbereich sichern"
1840 PRINT"T Text eingeben"
1850 PRINT"X CPCMON verlassen"
1860 PRINT
1870 GOTO 590
1880 '
1890 ' Speicherbereiche eingeben
1900 '
1910 INPUT"Ab welcher Speicherstelle";w
1920 IF w<0 THEN w=w+65536
1930 PRINT CHR$(24)+"h"+CHR$(24)+"ex oder "+CHR$(24)+"d"+CHR$(24)+
"ezimal ?";
1940 z$=LOWER$(INKEY$):IF z$="h" THEN hex=1 ELSE IF z$="d" THEN he
x=0 ELSE 1940
1950 PRINT:PRINT"Ende mit 'x'"
1960 PRINT
1970 PRINT"Inhalt Speicherstelle";w;:INPUT n$
1980 IF LOWER$(n$)="x" THEN PRINT:GOTO 590
1990 IF hex=1 THEN POKE w,VAL("&"+n$):w=w+1:GOTO 1970
2000 POKE w,VAL(n$):w=w+1:GOTO 1970
2010 '
2020 ' Blocktransfer
2030 '
2040 PRINT:INPUT"Ab welcher Speicherstelle";s1:INPUT"Bis zu welche
r Speicherstelle";s2:INPUT"Neuer Anfang";s
2050 IF s<0 THEN s=s+65536
2060 IF s1<0 THEN s1=s1+65536
2070 IF s2<0 THEN s2=s2+65536
2080 IF s1>=s2 THEN PRINT"Bereich falsch!!":GOTO 2040
2090 IF s<s1 THEN FOR i=s TO s+s2-s1:POKE i,PEEK(s1+i-s):NEXT:GOTO
 590
2100 FOR i=s+s2-s1 TO s STEP-1:POKE i,PEEK(s1+i-s):NEXT:GOTO 590
2110 GOTO 590
2120 '
2130 ' Maschinenprogramm anspringen
2140 '
```

```
2150 INPUT"Ansprunstelle";s
2160 IF s<0 THEN s= s+65536
2170 s1=INT(s/256):s2=s-256*s1
2180 POKE 40068,s2:POKE 40069,s1:CALL 40060
2190 machstack=PEEK(40055)+256*PEEK(40056)
2200 z$=""
2210 FOR i=machstack TO machstack+10 STEP 2
2220 z$=RIGHT$("0"+HEX$(PEEK(i+1)),2)+RIGHT$("0"+HEX$(PEEK(i)),2)+
"  "+z$
2230 NEXT
2240 z$=z$+RIGHT$("0"+HEX$(PEEK(40058)),2)+RIGHT$("0"+HEX$(PEEK(40
057)),2)
2250 n$=" AF    BC    DE    HL    IX    IY    SP "
2260 LOCATE#1,1,2:PRINT#1,n$;:LOCATE#1,1,3:PRINT#1,z$;
2270 GOTO 590
2280 '
2290 ' Programm disassemblieren
2300 '
2310 REM
2320 REM ******************
2330 REM ** Disassembler **
2340 REM ******************
2350 ZONE 18
2360 LOCATE#1,15,1:PRINT#1,"DISASSEMBLER"
2370 PRINT:INPUT "Beginn in Hex";ic$:ic=VAL("&"+ic$):PRINT
2380 IF ic<0 THEN ic=ic+65536
2390 IF rom=-1 THEN 2410 ELSE h1$=HEX$(PEEK(ic),2):h2$=HEX$(PEEK(i
c+1),2):h3$=HEX$(PEEK(ic+2),2):h4$=HEX$(PEEK(ic+3),2)
2400 b1$=BIN$(PEEK(ic),8):b2$=BIN$(PEEK(ic+1),8):GOTO 2450
2410 s2=INT((ic)/256):s1=ic-256*s2
2420 POKE romval,s1:POKE romval+1,s2:CALL romread
2430 h1$=HEX$(PEEK(40050),2):h2$=HEX$(PEEK(40051),2):h3$=HEX$(PEEK
(40052),2):h4$=HEX$(PEEK(40053),2)
2440 b1$=BIN$(PEEK(40050),8):b2$=BIN$(PEEK(40051),8)
2450 GOSUB 2600:PRINT#stream*-8,HEX$(ic,4)+"  ";:z$=h1$:IF u>1 THE
N z$=z$+h2$:IF u>2 THEN z$=z$+h3$:IF u>3 THEN z$=z$+h4$
2460 IF INSTR(b$,"ö")<>0 THEN b$=LEFT$(b$,INSTR(b$,"ö")-1)+","+MID
$(b$,INSTR(b$,"ö")+1)
2470 PRINT# stream*-8, z$,b$:ic=ic+u
2480 IF INKEY$="^" THEN LOCATE#1,15,1:PRINT#1,SPACE$(15):PRINT:GOT
O 590 ELSE 2390
2490 '
2500 '
2510 REM ***************
2520 REM ** proc:DISI **
2530 REM ***************
2540 REM
2550 REM --------------------------
2560 REM IN: b1$,b2$,h1$,h2$,h3$,h4$
2570 REM OUT:b$,u
2580 REM --------------------------
2590 REM
2600 IF h1$="CB" THEN GOSUB 3380:RETURN
2610 IF h1$="CB" THEN GOSUB 3380:RETURN
2620 IF h1$="DD" THEN R$="IX":GOSUB 3470:RETURN
2630 IF h1$="ED" THEN GOSUB 3650:RETURN
```



```
2640 IF h1$="FD" THEN R$="IY":GOSUB 3470:RETURN
2650 b0$=b1$:GOSUB 2710:REM Binteil
2660 ON b1%+1 GOSUB 2790,3030,3080,3130
2670 RETURN
2680 REM ******************
2690 REM ** sub:Binteil **
2700 REM ******************
2710 b1%=VAL("&x"+LEFT$(b0$,2)):b2%=VAL("&x"+MID$(b0$,3,3)):b3%=VA
L("&x"+MID$(b0$,6))
2720 RETURN
2730 REM ************************
2740 REM ** Ein-Byte-OP-Codes **
2750 REM ************************
2760 REM ****************
2770 REM ** sub:DISI 0 **
2780 REM ****************
2790 IF b3%<4 OR b3%>6 THEN GOSUB 2860 ELSE GOSUB 2810
2800 RETURN
2810 RESTORE 2850:GOSUB 3300
2820 IF b3%=6 THEN b$="LD "+z$+","+h2$:u=2:RETURN
2830 IF B3%=4 THEN B$="INC " ELSE B$="DEC "
2840 B$=B$+Z$:U=1:RETURN
2850 DATA B,C,D,E,H,L,(HL),A
2860 DATA NOP,"EX AFöAF'",DJNZ,JR,"JR NZö","JR Zö","JR NCö","JR Cö
"
2870 DATA "LD BCö","ADD HLöBC","LD DEö","ADD HLöDE","LD HLö","ADD
HLöHL","LD SPö","ADD HLöSP"
2880 DATA INC BC,DEC BC,INC DE,DEC DE,INC HL,DEC HL,INC SP,DEC SP
2890 DATA RLCA,RRCA,RLA,RRA,DAA,CPL,SCF,CCF
2900 DATA "LD(BC)öA","LD Aö(BC)","LD(DE)öA","LD Aö(DE)"
2910 IF B3%=0 THEN RESTORE 2860:GOSUB 3300:B$=Z$:IF B2%<2 THEN U=1
:RETURN ELSE B$=B$+STR$(VAL("&"+H2$)+256*(VAL("&"+H2$)>127))+" = "
+RIGHT$("0000"+HEX$(ic+2+VAL("&"+H2$)+256*(VAL("&"+H2$)>127)),4):U
=2:RETURN
2920 IF b3%=1 THEN RESTORE 2870:GOSUB 3300:b$=z$:IF INT(b2%/2)<>b2
%/2 THEN u=1:RETURN ELSE b$=b$+h3$+h2$:u=3:RETURN
2930 IF B3%=3 THEN RESTORE 2880:GOSUB 3300:B$=Z$:u=1:RETURN
2940 IF b3%=7 THEN RESTORE 2890:GOSUB 3300:b$=z$:u=1:RETURN
2950 IF B2%<4 THEN RESTORE 2900:GOSUB 3300:b$=z$:u=1:RETURN
2960 u=3:IF b2%=4 THEN b$="LD("+h3$+h2$+")öHL":RETURN
2970 IF b2%=5 THEN b$="LD HLö("+h3$+h2$+")":RETURN
2980 IF b2%=6 THEN b$="LD("+h3$+h2$+")öA":RETURN
2990 b$="LD Aö("+h3$+h2$+")":RETURN
3000 REM ************
3010 REM ** DISI 1 **
3020 REM ************
3030 u=1:IF h1$="76" THEN b$="HALT":RETURN
3040 RESTORE 2850:GOSUB 3300:b$="LD "+z$:b2%=b3%:RESTORE 2850:GOSU
B 3300:b$=b$+"ö"+z$:RETURN
3050 REM ************
3060 REM ** DISI 2 **
3070 REM ************
3080 u=1:RESTORE 3090:GOSUB 3300:b$=z$:b2%=b3%:RESTORE 2850:GOSUB
3300:b$=b$+z$:RETURN
3090 DATA "ADD Aö","ADC Aö","SUB ","SBC Aö","AND ","XOR ","OR ","C
P "
```

```
3100 REM ************
3110 REM ** DISI 3 **
3120 REM ************
3130 IF b3%=7 THEN u=1:b$="RST "+HEX$(8*b2%):RETURN
3140 IF b3%=0 THEN u=1:RESTORE 3240:GOSUB 3300:b$="RET "+z$:RETURN
3150 IF B3%=1 THEN RESTORE 3250:GOSUB 3300:b$=z$:u=1:RETURN
3160 IF b3%=6 THEN RESTORE 3090:GOSUB 3300:b$=z$+h2$:u=2:RETURN
3170 IF b3%=2 THEN RESTORE 3240:GOSUB 3300:b$="JP "+z$+"ö"+h3$+h2$
:u=3:RETURN
3180 IF b3%=4 THEN RESTORE 3240:GOSUB 3300:b$="CALL "+z$+"ö"+h3$+h
2$:u=3:RETURN
3190 IF b3%=3 AND b2%=0 THEN b$="JP "+h3$+h2$:u=3:RETURN
3200 IF b3%=5 AND b2%=1 THEN b$="CALL "+h3$+h2$:u=3:RETURN
3210 IF b3%=5 THEN RESTORE 3260:b2%=b2%/2:GOSUB 3300:b$="PUSH "+z$
:u=1:RETURN
3220 IF b2%>3 THEN RESTORE 3260:GOSUB 3300:b$=z$:u=1:RETURN
3230 u=2:IF b2%=2 THEN b$="OUT("+h2$+")öA":RETURN ELSE b$="IN Aö("
+h2$+")":RETURN
3240 DATA NZ,Z,NC,C,PO,PE,P,M
3250 DATA "POP BC","RET","POP DE","EXX","POP HL","JP(HL)","POP AF"
,"LD SPöHL"
3260 DATA BC,DE,HL,AF,"EX(SP)öHL","EX DEöHL","DI","EI"
3270 '
3280 ' Befehl aus DATA lesen
3290 '
3300 FOR I=0 TO B2%:READ Z$:NEXT I:RETURN
3310 REM ****************************
3320 REM ** Erweiterungskommandos **
3330 REM ****************************
3340 '
3350 REM *************
3360 REM ** DISI CB **
3370 REM *************
3380 DATA RLC,RRC,RL,RR,SLA,SRA,???,SRL
3390 b0$=b2$:GOSUB 2710:u=2:IF b1%=0 THEN RESTORE 3380:GOSUB 3300:
b$=z$:RESTORE 2850:b2%=b3%:GOSUB 3300:IF B$<>"???" THEN b$=b$+" "+
z$:RETURN ELSE RETURN
3400 V=B2%:B2%=B3%:RESTORE 2850:GOSUB 3300
3410 IF b1%=1 THEN b$="BIT"+STR$(V)+"ö"+z$:RETURN
3420 IF b1%=2 THEN b$="RES"+STR$(V)+"ö"+z$:RETURN
3430 IF b1%=3 THEN b$="SET"+STR$(V)+"ö"+z$:RETURN
3440 REM ****************
3450 REM ** DISI DD+FD **
3460 REM ****************
3470 IF h2$<>"CB" THEN 3550
3480 U=1:b0$=RIGHT$("00000000"+BIN$(VAL("&"+h4$)),8):GOSUB 2710:IF
 b3%<>6 THEN b$="???":RETURN
3490 IF H4$="7E" THEN B$="???":RETURN
3500 DATA RLC,RRC,RL,RR,SLA,SRA,???,SRL
3510 U=4:IF B1%=0 THEN RESTORE 3500:GOSUB 3300:IF h4$="36" THEN b$
=z$:u=1:RETURN ELSE B$=Z$+"("+R$+"+"+H3$+")":RETURN
3520 IF B1%=1 THEN B$="BIT"+STR$(B2%)+"ö("+R$+"+"+H3$+")":RETURN
3530 IF B1%=2 THEN B$="RES"+STR$(B2%)+"ö("+R$+"+"+H3$+")":RETURN
3540 B$="SET"+STR$(B2%)+"ö("+R$+"+"+H3$+")":RETURN
```



```
3550 b0$=B2$:GOSUB 2710:IF B1%=3 THEN U=2:IF H2$="E5" THEN B$="PUS
H "+R$:RETURN ELSE IF H2$="E1" THEN B$="POP "+R$:RETURN ELSE IF H2
$="E9" THEN B$="JP("+R$+")":RETURN ELSE IF H2$="E3" THEN B$="EX(SP
)ö"+R$:RETURN ELSE IF H2$="F9" THEN B$="LD SPö"+R$:RETURN
3560 IF B1%=2 THEN IF B3%=6 THEN RESTORE 3090:GOSUB 3240:B$=Z$+"("
+R$+"+"+H3$+")":U=3:RETURN
3570 IF B1%=1 THEN U=3:IF B2%=6 AND B3%=6 THEN U=1:B$="???":RETURN
 ELSE IF B2%=6 THEN RESTORE 2850:B2%=B3%:GOSUB 3240:B$="LD("+R$+"+
"+H3$+")ö"+Z$:RETURN ELSE IF B3%=6 THEN RESTORE 2850:GOSUB 3240:B$
="LD "+Z$+"ö("+R$+"+"+H3$+")":RETURN
3580 IF B1%<>0 THEN 3610 ELSE U=2:IF H2$="09" THEN B$="ADD "+R$+"ö
BC":RETURN ELSE IF H2$="19" THEN B$="ADD "+R$+"öDE":RETURN ELSE IF
 H2$="29" THEN B$="ADD "+R$+"ö"+R$:RETURN ELSE IF H2$="39" THEN B$
="ADD "+R$+"öSP":RETURN
3590 IF H2$="23" THEN B$="INC "+R$:RETURN ELSE IF H2$="2B" THEN B$
="DEC "+R$:RETURN ELSE IF H2$="34" THEN B$="INC("+R$+"+"+H3$+")":U
=3:RETURN ELSE IF H2$="35" THEN B$="DEC("+R$+"+"+h3$+")":u=3:RETUR
N ELSE U=4
3600 IF H2$="36" THEN B$="LD("+R$+"+"+H3$+")ö"+H4$:RETURN ELSE IF
H2$="22" THEN B$="LD("+H4$+H3$+")ö"+R$:RETURN ELSE IF H2$="2A" THE
N B$="LD "+R$+"ö("+H4$+H3$+")":RETURN ELSE IF H2$="21" THEN B$="LD
 "+R$+"ö"+H4$+H3$:RETURN
3610 U=1:B$="???":RETURN
3620 REM *************
3630 REM ** DISI ED **
3640 REM *************
3650 B0$=B2$:GOSUB 2710:IF B1%>2 OR B1%=0 THEN 3760
3660 IF B1%=2 AND(B2%<4 OR B3%>3) THEN 3760
3670 IF B1%=2 THEN B2%=B2%-4+4*B3%:RESTORE 3680:GOSUB 3240:U=2:B$=
Z$:RETURN
3680 DATA LDI,LDD,LDIR,LDDR,CPI,CPD,CPIR,CPDR,INI,IND,INIR,INDR,OU
TI,OUTD,OTIR,OTDR
3690 IF B3%>3 THEN U=2:IF H2$="44" THEN B$="NEG":RETURN ELSE IF H2
$="4D" THEN B$="RETI":RETURN ELSE IF H2$="45" THEN B$="RETN":RETUR
N ELSE IF H2$="46" THEN B$="IM 0":RETURN ELSE IF H2$="56" THEN B$=
"IM 1":RETURN ELSE IF H2$="5E" THEN B$="IM 2":RETURN
3700 IF B3%>3 THEN IF H2$="57" THEN B$="LD AöI":RETURN ELSE IF H2$
="47" THEN B$="LD IöA":RETURN ELSE IF H2$="6F" THEN B$="RLD":RETUR
N ELSE IF H2$="67" THEN B$="RRD":RETURN ELSE IF h2$="4F" THEN b$="
LD R,A":RETURN ELSE IF h2$="5F" THEN b$="LD A,R":RETURN
3705 IF b3%>3 THEN 3760
3710 DATA "SBC HLöBC","ADC HLöBC","SBC HLöDE","ADC HLöDE","SBC HLö
HL","ADC HLöHL","SBC HLöSP","ADC HLöSP"
3720 IF B3%=2 THEN RESTORE 3710:GOSUB 3300:B$=Z$:U=2:RETURN
3730 IF B3%=3 THEN U=4:IF B2%=0 THEN B$="LD("+H4$+H3$+")öBC":RETUR
N ELSE IF B2%=1 THEN B$="LD BCö("+H4$+H3$+")":RETURN ELSE IF B2%=2
 THEN B$="LD("+H4$+H3$+")öDE":RETURN ELSE IF B2%=3 THEN B$="LD DEö
("+H4$+H3$+")":RETURN
3740 IF B3%=3 THEN IF b2%=4 THEN B$="LD("+H4$+H3$+")öHL":RETURN EL
SE IF B2%=5 THEN B$="LD HLö("+H4$+H3$+")":RETURN ELSE B2%=6 THEN B
$="LD("+H4$+H3$+")öSP":RETURN ELSE IF B2%=7 THEN B$="LD SPö("+H4$+
H3$+")":RETURN ELSE 3760
3750 IF B2%=6 THEN 3760 ELSE U=2:RESTORE 2850:GOSUB 3240:IF B3%=0
THEN B$="IN "+Z$+"ö(C)":RETURN ELSE B$="OUT (C)ö"+Z$:RETURN
3760 U=1:B$="???":RETURN
3770 DATA öööööö"
```

# 7 Die Unterteilung des Firmware-Speichers

Mit der Entwicklung des Disassemblers im letzten Kapitel haben wir nun endlich alle Vorarbeiten abgeschlossen. Wir sind grundsätzlich über den Aufbau der einzelnen Bausteine des Systems und ihre Funktionen informiert. Wir verfügen über Grundkenntnisse und einige nützliche Routinen im Bereich der Prozessorprogrammierung und kennen uns auch mit dem Innenleben des Z80-Prozessors aus.

Mit dem Monitor verfügen wir des weiteren über ein geeignetes Werkzeug, das uns bei unseren weiteren Untersuchungen sehr gute Hilfestellung leisten wird. Somit sind wir nun in der Lage, uns mit dem System von der Anwenderseite her etwas näher zu beschäftigen. Anwenderseite, das soll hier im wesentlichen heißen: Firmware-Routinen und Firmware-Sprungtabellen.

Der CPC verfügt über ungefähr 300 verschiedene Ansprungpunkte in das Betriebssystem. Bei jedem dieser Betriebssystemeingänge handelt es sich um eine kleine Softwareschnittstelle. Der Prozessor erwartet beim Ansprung eine vorgegebene Belegung der Universalregister, anhand derer er eine genau definierte Funktion, z.B. das Zeichnen einer Linie oder das Laden eines Soundregisters mit bestimmten Werten ausführt und dann an das aufrufende Programm zurückgibt.

Der Aufruf einer solchen Betriebssystemroutine erfolgt nicht durch direkten Ansprung einer Adresse im ROM, sondern über einen Zeiger. Alle Zeiger sind in verschiedenen Sprungtabellen im oberen Bereich des RAM von B900 bis BDFF abgelegt. Springt man einen dieser Zeiger an, so wird die benötigte ROM-Konfiguration hergestellt und dann die Betriebssystemroutine ausgeführt.

Den genauen Ablauf der Umschaltung und des Aufrufs haben wir ja schon in der Anwendung zu Kapitel 5.5 betrachtet. Hier wird es nun darum gehen, eine Reihe weiterer neuer Routinen mit ihren Funktionen und Eingangsbedingungen vorzustellen. Darüber hinaus wollen wir aufzeigen, wie diese vom Benutzer in eigene Programme und Anwendungen integriert werden können.

Hier muß jedoch gleich eine Warnung angebracht werden, um späteren Enttäuschungen vorzubeugen. 300 Routinen würden bei ausführlicher Erklärung sämtlicher Möglichkeiten und gar der Kombination mit anderen Programmen den Rahmen dieses Buches bei weitem sprengen und wohl vom Umfang her einen kleinen Brockhaus ergeben.

Es bleiben daher nur zwei Möglichkeiten: Tabellarische Übersicht in Stichworten über alle Routinen oder Beschränkung auf einige interessante und allgemein anwendbare Einsprungpunkte, die dann ausführlicher vorgestellt werden können. Wir wollen im nun Folgenden den letztgenannten Weg beschreiten.

Wir werden unsere Erörterung zu den einzelnen Routinen mit einem Streifzug durch die einzelnen Teile des Computers, wie Tastatur und Tastaturabfrage, Kassettenspeicherung und Druckeransteuerung sowie eine Reihe anderer Anwendungsfelder verbinden. Dabei werden wir zunächst die einzelnen Systemfunktionen vorstellen und darauf folgend dann auf die Eingriffsmöglichkeiten via Firmwareroutinen näher eingehen.

Soviel zur inhaltlichen Übersicht. Zunächst jedoch noch ein wenig zum Speicheraufbau und der Gliederung der einzelnen Bereiche.

Mit dem Benutzerspeicher und seiner Belegung durch BASIC-Quelltext und Variable haben wir uns schon zu Beginn dieses Buches in den Kapiteln 1 bis 3 relativ ausführlich beschäftigt. Der oberhalb liegende Firmwarespeicher war uns dagegen bisher nur einige wenige Worte wert. Dies soll jetzt anders werden.

Beginnen wir einmal mit einigen grundsätzlichen Überlegungen zur **Funktion des Firmware-Speichers.** Dieser wird von zwei Teilen unseres Computers relativ intensiv genutzt: dem BASIC-Interpreter und den Betriebssystemroutinen.



Der **BASIC-Interpreter** interpretiert den BASIC-Quelltext. Anhand der Codes unseres BASIC-Programms führt er dann Operationen wie das Setzen eines BASIC-Registers mit einem Wert, den Ansprung einer Speicherstelle im Programm und ähnliches durch. Er benötigt den Firmwarespeicher für zwei Zwecke:

- als Zwischenspeicher für Daten und Zeiger
- als Sprungverteiler zum Ansprung von Betriebssystemroutinen mittels der Jumpblöcke

Das **Betriebssystem** enthält grundsätzliche, von jeder Hochsprache benötigte Unterprogramme, wie Routinen zur Bildschirmansprache und zum Datentransfer auf Massenspeicher (Kassette), zur Tastaturabfrage und zur zeitlichen Koordination von Ereignissen. Das Betriebssystem des CPC enthält darüber hinaus einen kompletten Satz von Routinen für die Integerarithmetik. Es benötigt den Firmwarespeicher ebenfalls für zwei Zwecke:

- zur Zwischenspeicherung von Daten und Rücksprungadressen
- als Ablagebereich für die Sprungverteiler und Ereignisse

Jeder Bereich (BASIC und Firmware=Betriebssystem) verfügt über einen eigenen Stack. Der **Betriebssystem-Stack** läuft von Adresse C000 nach unten. Daneben existiert ein eigener Stack für das BASIC zur Ablage von GOSUB-Rücksprüngen und FOR-TO-Schleifen. Hierfür sind die Adressen AE8B bis B09B vorgesehen. Als Stackpointer für diesen Bereich dienen dabei die Speicherstellen B08B und B08C. Der CPC arbeitet also mit zwei Stacks gleichzeitig.

Neben der kurzfristigen Zwischenspeicherung auf dem Stack verfügen beide Bereiche noch über eine Reihe von festen Speicherplätzen, die für ganz bestimmte Zeiger und Daten vorgesehen sind. In diesen Bereich gehören zum Beipiel die Pointer für die Speicheraufteilung zwischen den einzelnen Bereichen des Benutzerspeichers, die wir in Kapitel 1 benutzt haben. Ablagebereiche für die Zwischenspeicherung der Tastatur und der Ausgabeparameter bei Benutzung des Datacorders (siehe nächstes Kapitel) sind weitere interessante Beispiele.

Einen größeren Teil des Firmwarespeichers nimmt darüber hinaus die Ablage der Firmwaresprungblöcke ein. Wir können drei verschiedene Sprungblöcke unterscheiden:

**Kernel-Jumpblock:** Dieser enthält eine Reihe von Routinen zur Umschaltung von Speicherbereichen und die Übertragung von Speicherbereichen bei ausgeschalteten ROMs. Wir haben diese Routinen bereits in Kapitel 2 ausgiebig betrachtet. Die Kernel-Routinen liegen von B900 bis B923. Es existieren noch eine Reihe weiterer Routinen im Kernel-Bereich, die in den untersten Byte des RAM-Speichers abgelegt sind, die RESTARTs.

**Indirections:** Dieser Sprungverteiler besteht aus einer Reihe von Unterprogrammaufrufen, die von den Routinen des Haupt-Firmware-Jumpblocks benutzt werden. Die hier enthaltenen Routinen arbeiten alle auf relativ niedrigem Niveau und ohne Sicherungen in Form der Maskierung von Daten etc. Für den Anwender sind sie daher nur bedingt geeignet. Die Zeiger auf die Indirections belegen die Adressen BDCD bis BDF3 im RAM.

**Haupt-Firmware-Sprungtabelle:** Diese Tabelle enthält eine ganze Reihe von wichtigen Einsprungpunkten in das untere und obere ROM. Teilweise benutzen die dort gespeicherten Routinen Kernel-Routinen und die Indirections als Hilfs- beziehungsweise Unterprogramme.

Die Hauptsprungtabelle liegt von hex BB00 bis hex BD39. Jeder Eintrag in der Sprungtabelle belegt 3 Byte und ist für die Verwendung von LOW-JUMP-RESTARTS mit Hilfe von RST1 und der Routine ab hex B982 (vergleiche Anwendung zu Kapitel 5.4) beziehungsweise von RST 5 vorgesehen.

Während RST 1 die gewünschte Speicherkonfiguration herstellt, schaltet RST 5 nur das untere ROM ein und nach der Rückkehr aus der Routine wieder aus. Das obere ROM wird dagegen nicht verändert. RST 5 bietet



also weniger Möglichkeiten, ist aber dafür bedeutend schneller, weshalb dieser Ansprung bei geschwindigkeitsabhängigen Routinen, die nicht auf die oberen Speicherbereiche (ROM oder Bildschirmspeicher) zurückgreifen, verwandt wird. Der prinzipielle Ablauf ist aber mit RST 1 identisch.

Innerhalb der Sprungtabelle kann man bestimmte Anwendungsbereiche unterteilen. Jede Routine in der Sprungtabelle kann durch ihren Namen und ihre Ansprungadresse beschrieben werden. Da jeder Ansprung genau 3 Byte in Anspruch nimmt, folgen die einzelnen Routinen in der Tabelle im Abstand von jeweils 3 Adressen aufeinander. Der Name jeder Routine besteht aus zwei Teilen, einem Kürzel für den Anwendungsbereich, dem die Routine zuzuordnen ist, und einer Beschreibung ihrer Funktion. Wir können dabei acht Abschnitte unterscheiden:

**1. Die Tastaturverwaltung (KEY MANAGER=KM)**

Die Tastaturverwaltung ist für die Abfrage von Tastatur- und Joysticks, das Drücken bestimmter Tasten, das Zulassen von Repeat und das Lesen von Zeichen zuständig. Adressen: BB00 bis BB4B.

**2. Der Text-VDU (TXT)**

Diese Abteilung des Betriebssystems enthält all jene Routinen, die mit der Darstellung und dem Lesen von Zeichen vom Bildschirm, dem Setzen des Cursors und der Schriftfarben sowie der Definition von Windows und Ausgabekanälen zu tun haben. Die Einträge im Text-VDU finden sich mit Adressen von BB4E bis BBB7.

**3. Das Grafik-VDU (GRA)**

In diesem Teil des Betriebssystems finden sich die Routinen, die mit einzelnen Pixeln arbeiten, wie das Setzen des Grafikcursors und des Grafikfensters , das Zeichnen und Testen von Punkten und Linien. Die Grafikroutinen sind in den Sprungadressen von BBBA bis BBFC gespeichert.

### 4. Das Bildschirmpaket (SCREEN=SCR)

Das Bildschirmpaket stellt eine Schnittstelle zwischen den Softwareroutinen des Grafik- und des Text-VDU auf der einen Seite und der Bildschirmhardware auf der anderen Seite dar. Bildschirmfunktionen, die sowohl vom Textbereich als auch von der Grafik benötigt werden, befinden sich hier. Dazu gehören z.B. die uns schon bekannten Routinen für das Setzen des Bildschirmoffsets, die Definition der Modes und der Inks sowie das Rollen des Bildschirms, die Auflösung eines Zeichens in eine Zeichenmatrix und umgekehrt. Dieser Teil läuft von BBFF bis BC62.

### 5. Die Kassettenverwaltung (CASSETTE=CAS)

Dieser Teil der Betriebssystemroutinen beschäftigt sich mit dem Lesen von Dateien vom Band und dem Schreiben auf das Band. Die Einträge laufen von BC65 bis BCA4.

### 6. Die Tongeneratorverwaltung (SOUND)

Die in diesem Teil abgelegten Routinen steuern den Tongeneratorchip. Sie haben Adressen von BCA7 bis BCC5.

### 7. Der Betriebssystemkern (KERNEL=KL)

Diese Routinen behandeln synchrone und asynchrone Ereignisse und überwachen die Speicherbelegung und die Aufteilung des Speichers in ROM und RAM. Das Kernel nimmt also im Rahmen des Betriebssystems im wesentlichen Koordinationsfunktionen wahr. In der Hauptfirmware-Sprungtabelle sind nur ein Teil der Kernelroutinen mit Adressen zwischen BCC8 und BD10 abgelegt. Es existiert noch eine weitere Sprungtabelle im Bereich B900 bis B921 (siehe oben). Außerdem können in diesen Bereich noch die Restartanweisungen bzw. ihre Funktion für das Umschalten von ROMs und das Anspringen von Betriebssystemroutinen eingeordnet werden (Adresse 0000 bis 003B).



### 8. Das Maschinenpaket (MC)

Dieses enthält all jene Routinen, die eine Schnittstellenfunktion zwischen dem System Hardware und den mehr softwaremäßig orientierten Routinen der anderen Pakete wahrnehmen. Viele andere Routinen greifen auf die Programme des Maschinenpakets zurück. So enthält z.B. das Maschinenpaket die schon bekannte Routine MC SOUND REGISTER. Die Einträge im Maschinenpaket führen meist zu wenig komplexen, allgemein verwendbaren Funktionen. Adressen: BD13 bis BD36.

Die Jumpblöcke existieren alle als Kopie im unteren ROM. Beim Einschalten werden sie in ihre angestammten RAM-Bereiche kopiert. Da das BASIC ebenso wie andere Hochsprachen auf diese Routinen zurückgreift, da die hier vorhandenen definierten Ein- und Aussprungkriterien eine leichte Benutzung garantieren, ist es möglich, durch Verbiegen dieser Routinen auf ein eigenes Maschinenprogramm bestimmte Systemfunktionen umzubauen, zu verändern oder zu erweitern. Die Original-Speicheraufteilung läßt sich dabei jederzeit mit der Routine JUMP RESTORE (BD37) wieder herstellen. Schauen wir uns nun einzelne Systemfunktionen und die zugehörigen Anwendungsmöglichkeiten der Firmwareroutinen näher an.

# 8 Systemfunktionen und Jumpblockbenutzung

## 8.1 Tastaturabfrage und Dateneingabe über Tastatur

Einer der wohl komplexesten Abläufe im Inneren des Rechners ist ein Vorgang, der auf den ersten Blick relativ einfach aussieht: die Eingabe einer BASIC-Zeile . Hier sind auf einem langen Weg eine Vielzahl von Programmen zu durchlaufen. Wir werden uns die einzelnen Stationen nacheinander anschauen. Zunächst einmal eine Grobübersicht. Beginnen wir bei den Grundlagen. Das ULA (vergleiche Kapitel 3) liefert einen Unterbrechungstakt von 300 Hertz, den sogenannten FAST TICKER. Eine Teilung durch 6 liefert hieraus den normalen Hauptabfragetakt (TICKER) von 50 Hertz.

In diesem Rhythmus, also 50mal pro Sekunde, fragt der CPC die Tastatur ab und überprüft, ob Tasten gedrückt wurden (vergleiche Kapitel 3, I/O-Bereich). Wurde ein Zeichen über Tastatur eingegeben, so findet ein Eintrag in den **Tastatureingabepuffer** statt.

Neben dem Eingabepuffer existieren noch zwei weitere kleinere Speicher für Sonderzwecke. Es handelt sich um eine Abspeicherungsmöglichkeit für **Erweiterungszeichen** (Expansionstrings) und einen **Put-Back-Puffer.** Der Erweiterungsspeicher nimmt ein Erweiterungszeichen, das heißt eine vorher zum Beispiel mit dem BASIC-Kommando KEY definierte Zeichenkombination, auf.

Der Put-Back-Puffer ermöglicht es, ein gerade gelesenes Zeichen wieder an die Tastaturverwaltung zurückzugeben, wenn dieses momentan nicht benötigt wird. Ein Grund dafür könnte sein, daß dieses Zeichen zu einem anderen (folgenden) Wort gehört, welches erst in Folge analysiert werden soll. Ein solches Zeichen würde dann in den Put-Back-Puffer geschrieben.

Ein eingegebenes Zeichen kann sich also nun in drei Speichern befinden: im Tastatureingabepuffer, als Erweiterungszeichen im Expansion-Puffer und natürlich auch in dem zuletzt genannten PutBack-Puffer. Auf diese drei Adressen oder besser gesagt Adressräume können wir dann zurückgreifen, um ein oder mehrere Zeichen zu lesen.

Einen wichtigen Zwischenschritt haben wir bei dieser Betrachtungsweise allerdings außer acht gelassen; die Zuordnung von Tasten zu Buchstaben, Zahlen oder allgemein grafischen Symbolen. Wir müssen nämlich zwischen der physischen Tastaturabfrage und den darauf folgend abgebildeten Zeichen deutlich unterscheiden. Zunächst also zur eigentlichen **Tastaturabfrage.**

Wie wir schon aus Kapitel 3 wissen, erfolgt die Erkennung einer gedrückten Taste beim CPC im Wechselspiel zwischen dem I/O-Baustein 8255 und dem Soundchip. Port C des Schnittstellen-ICs legt über einen Decoder eine von 10 Leitungen der 8*10-Tastaturmatrix, die sogenannten Y-Leitungen auf Aktiv-Pegel. Diese 10 Leitungen werden von 8 weiteren (den X-Leitungen) gekreuzt, die mit dem Eingangsport des Soundchips verbunden sind. Die Tasten des CPC sind nun so angeordnet, daß sie jeweils eine X- mit einer Y-Leitung verbinden können.

Bei der Tastaturabfrage, dem SCANNEN, tastet der CPC nun nacheinander alle 10 Y-Leitungen auf und überprüft das Ergebnis auf den X-Leitungen. Wenn nun eine Taste gedrückt ist, so wird die zugehörige X-Leitung ebenfalls auf Aktivpegel gesetzt. Damit kann der Computer feststellen, welche Taste gedrückt war.

Jeder Taste ist eine Nummer zugeordnet, die sich aus ihrer Position in der Matrix ergibt. Sie errechnet sich zu:

10 * X-Position + Y-Position

Die Zuordnung von Tasten zu Nummern finden Sie im Anhang 3, Seite 16n des Bedienerhandbuches.

Hat der CPC beim Scannen festgestellt, daß eine Taste gedrückt ist, so folgt der nächste Schritt, die Zuordnung von Zeichen. Dies geschieht mit Hilfe von drei Übersetzungstabellen, die im Firmwarespeicher ab 45900 nach oben abgelegt sind. Jede Tabelle enthält dabei 80 Byte für die 80 verschiedenen möglichen Tasten in jeder der drei Ebenen 'Normal', 'SHIFT' und 'CTRL'. Die Anordnung gibt Tabelle 8.1 wieder.



Die **Zahlenwerte in den Übersetzungstabellen** lassen sich in drei Gruppen einteilen. Die erste Gruppe bilden die Zeichencodes. Trifft der CPC auf eine Zahl aus diesem Bereich, so übernimmt er das zugehörige Zeichen in den Eingabepuffer.

Mit Nummern zwischen 128 und 159 werden die 32 Expansionstrings angesprochen. Drückt man also auf eine Taste, die mit einem Expansion-Code belegt ist, so wird die auf diesen Code definierte Zeichenkombination in den Puffer übernommen. Die dritte Gruppe bilden Steuercodes. Trifft der CPC auf sie, so führt er die zugehörige Operation aus. Tabelle 8.2 gibt ihre Funktionen wieder.

**Tabelle 8.1:** Die Tastaturübersetzungstabellen ab Adresse 45900

| Adresse | Funktion | |
|---|---|---|
| 45900 | Belegung Normalebene Taste Nr. 0 | |
| 45901 | Belegung Normalebene Taste Nr. 1 | |
| : | | |
| : | | |
| : | | |
| 45979 | Belegung Normalebene Taste Nr. 79 | |
| 45980 | Belegung Shiftebene Taste Nr. 0 | |
| : | | |
| : | | |
| 46059 | | |
| 46060 | Belegung Controlebene Taste 0 | |
| : | | |
| : | | |
| 46139 | Belegung Controlebene Taste 79 | |
| 46140 | Bitweise Abspeicherung Repeat: | Repeat=1<br>kein Repeat=0 |
| : | | |
| 46149 | Ende Abspeicherung Repeat | |
| 46150 | Sprungzeiger Exp. 128 | |
| 46151 | Zeichen 1 Exp. 128 | |
| : | | |
| : | | |
| 46150+n | Zeichen n | |
| 46151+n | Sprungzeiger Exp. 129 | |
| : | | |
| : | | |

**Tabelle 8.2:** Bedeutung der Werte in der Übersetzungstabelle

| Nummer | Bedeutung |
|---|---|
| 1-31 | Druck der Kontrollzeichen-Kommandos, dabei werden die Nummern 13=Enter und 16=CLR ausgeführt |
| 32-126 | Grafiksatz (Anhang III) |
| 127 | DEL |
| 128-159 | Erweiterungszeichen (expansion characters) |
| 160-223 | Grafiksatz |
| 224 | Copy |
| 225 | CTRL Copy |
| 226-238 | Grafiksatz (Firmwarezeichensatz nach Anhang III) |
| 240 | CSR hoch ausführen |
| 241 | CSR runter |
| 242 | CSR links |
| 243 | CSR rechts |
| 244 | Copy CSR hoch |
| 245 | Copy CSR runter |
| 246 | Copy CSR links |
| 247 | Copy CSR rechts |
| 248 | CTRL+CSR hoch |
| 249 | CTRL+CSR runter |
| 250 | CTRL+CSR links |
| 251 | CTRL+CSR rechts |
| 252 | ESC |
| 253 | CAPS LOCK |
| 254 | SHIFT LOCK |
| 255 | frei SHIFT/CTRL |

**Beispiele:** Wir wollen einmal die Taste 0 in der Normalebene auf den Wert 65 ("A") definieren. Dies können wir relativ einfach mit dem Kommando

```
POKE 45900,65
```

erreichen. Sie sollten diese Änderungsmethoden jetzt einmal anhand anderer Tasten und Ebenen durchspielen. Die allgemeine Formel zur Definition einer Taste in einer Ebene lautet:

```
80 * Ebene + Tastennummer + 45900
```



wobei die Ebenen folgende Nummern haben:

- o Normalebene = 0
- o SHIFT = 1
- o CTRL = 2

Wir haben nun die Behandlung der eigentlichen Tastaturabfrage abgeschlossen. Als Abschluß stellt sich die Frage, wo die Zeichen nach der Abfrage und Übersetzung abgelegt werden und welche Eingriffsmöglichkeiten wir besitzen. Nach der Eingabe befinden sich die Zeichen im Tastatureingabepuffer. Dieser liegt ab Speicherstelle 44196 im Firmwarespeicher des Computers. Mit den folgenden Befehlen können Sie ihn untersuchen:

```
10 FOR i=44196 TO 45000:PRINT i,PEEK(i),CHR$(PEEK(I)):NEXT
```

Sie werden nun nach Eingabe von RUN die folgenden Ausgaben erhalten:

```
44196    114    r
44197    117    u
44198    110    n
44199    0
44200    111    o
44201    114    r
44202    32
44203    105    i
44204    61     =
44205    52     4
44206    52     4
44207    49     1
44208    57     9
44209    54     6
44210    32
44211    116    t
44212    111    o
44213    32
44214    52     4
44215    53     5
.        .      .
.        .      .
```

Jede neue Eingabe überschreibt die vorhergehende. Das Ende der aktuellen Eingabe wird dabei durch CHR$(0) (vergleiche 44202) gekennzeichnet. Daher finden Sie zuerst den RUN-Befehl und danach den Rest der Untersuchungszeile. Bei der Interpretation eines Befehls im Direkt-Modus ist die Tastaturabfrage und -übersetzung mit diesem Schritt abgeschlossen. Soll der eingegebene Text allerdings als BASIC-Zeile übernommen werden, so wird ein weiterer Schritte notwendig, die Übersetzung in die BASIC-Zeile.

Dies leistet ein Teil des Betriebssystems, der Editor. Er wandelt die eingegebenen Zeichen in die Befehls-TOKENs um und legt diese dann nach einer Zwischenspeicherung in den unteren Bytes der Maschine, im Benutzerspeicher, ab. Dazu wird der BASIC-Quelltext mit höheren Zeilennummern nach oben, das heißt in höhere Adressbereiche, verschoben und die neue Zeile eingefügt.

Kommen wir nun zu den **Eingriffsmöglichkeiten**. Einige erste Änderungen haben wir ja schon in BASIC vorgenommen. Gerade im Bereich der Tastatur stellt uns das Betriebssystem aber eine ganze Reihe nützlicher Routinen zur Verfügung. Zum Auslesen von Zeichen aus den drei Tastaturzwischenspeichern existieren vier Einsprungpunkte:

**KM WAIT CHARACTER BB06**

Diese Routine wartet so lange, bis ein Zeichen in irgendeinem der drei Puffer zur Verfügung steht. Dieses befindet sich nach dem Rücksprung in Register A. Das Carry-Flag ist bei der Rückkehr aus dem Unterprogramm eingeschaltet.

**KM READ CHAR BB09**

Diese Routine wartet nicht, bis ein Zeichen verfügbar ist, sondern kehrt direkt zurück. War ein Zeichen vorhanden, ist das Carry gesetzt, und A enthält das Zeichen. Ansonsten ist es rückgesetzt und der Inhalt von A zerstört.



**KM WAIT KEY BB18**
**KM READ KEY BB1B**

Diese beiden Einsprünge arbeiten wie KM WAIT CHAR und KM READ CHAR, nur daß hier nur der Tastaturpuffer, jedoch nicht Funktionszeichen (Expansions) oder das Rückgabezeichen beachtet werden.

**KM CHAR RETURN BB0C**

stellt ein Zeichen in den Put-Back-Puffer ein. Da dieser bevorzugt durch KM CHAR WAIT und KM CHAR READ abgefragt wird, wird dieses Zeichen beim nächsten Lesen zurückgegeben.

Soviel zum Auslesen von Zeichen aus den Puffern. Für die Arbeit mit den Erweiterungszeichen existieren drei weitere Einsprünge:

**KM EXP BUFFER BB15**

Dieser Betriebssystemteil richtet einen Puffer für Expansionstrings ein. HL enthält dabei die Länge des Puffers (min.44), DE seine Anfangsadresse. Der Puffer wird auf die Normbelegung der Zeichen 128 bis 140 definiert. Das sind die Zahlen im abgesetzten Zahlenfeld, der Zehnertastatur.

Normalerweise liegt der Expansion-Puffer hinter den Tastaturübersetzungstabellen (vergleiche Tabelle 8.1). Als erster Code ist hier Erweiterungsstring 128 abgelegt. Den Anfang jeder Definition bildet dabei die Angabe der Länge des Strings, anhand derer die Verzeigerung zum nächsten String (129 und so weiter) hergestellt wird. Ein Wert von 0 besagt dabei, daß das Zeichen nicht definiert war. Die nächste Adresse ist dann die Längenangabe des folgenden Strings. Die Zeichen, die den Erweiterungsstring bilden, sind dabei im ASCII-Code abgelegt, so daß wir uns die Belegung mit der Routine, die wir schon bei der Untersuchung des Tastaturspeichers benutzt haben, anschauen können. Sie werden dann nacheinander die Zahlen und dann "echte" Erweiterungszeichen wie 'RUN' (CTRL + kleine ENTER-Taste) finden.

Der normale Expansion-Puffer ist aber nur 128 Zeichen lang. Wollen wir alle 32 möglichen Zeichen belegen, so ergeben sich relativ schnell Platzprobleme. Wir brauchen mehr Speicher. Mit KM EXP BUFFER lassen sich die dazu notwendigen Änderungen relativ einfach realisieren. Zuerst ver-

schieben wir die Speicherobergrenze mit MEMORY nach unten. Zwischen altem und neuem HIMEM haben wir dann wiederum einen geschützten Speicherbereich. Dessen Anfangsadresse laden wir in DE, die Länge in HL und rufen das Programm auf. Danach können wir wie gewohnt auf die verschiedenen Erweiterungszeichen mit KEY und KEY DEF zurückgreifen. Das zugehörige Maschinenprogramm sähe zum Beispiel so aus:

```
LD DE, 40000              11 40 9C
LD HL, 1000               21 E8 03
CALL KM EXP BUFFER        CD 15 BB
RET                       C9
```

Wenn Sie dieses Programm ab 42000 ablegen und dann durchstarten, erhalten Sie den gewünschten neuen Speicherbereich. Zur Überprüfung der Funktion sollten Sie sich das Ganze dann wieder einmal mit unserem Hilfsprogramm anschauen.

Das Setzen und die Abfrage von Erweiterungszeichen geschieht mit

**KM SET EXPAND** **BB0F und**
**KM GET EXPAND** **BB12**

Die erste definiert eine neue Erweiterungszeichenkette. B enthält die Nummer des Funktionszeichens, C die Länge der Zeichenkette und HL ihre Adresse. Die adressierte Kette wird dann in den Funktionszeichenpuffer übernommen. Ist sie zu lang, wird das Carry-Flag gelöscht. Die Routine benötigt und zerstört A, HL, BC, DE und die anderen Flags.

KM GET EXPAND liest ein Zeichen aus einem Erweiterungsstring aus. A enthält dabei das angesprochene Zeichen (also z.B. 128), L die Stelle, an der aus dem durch A definierten String gelesen werden soll (z.B. fünftes Zeichen lesen). Damit ist ein gezielter Zugriff auf einzelne Teile eines Erweiterungsstrings möglich. Erweiterungsstrings können dadurch sukzessive abgearbeitet werden.

**KM GET STATE** **BB21**

Neben dem Lesen von Zeichen ist natürlich auch noch die Abfrage von SHIFT und CAPS LOCK sehr nützlich. Dies erreicht man mit dieser Routine. Nach der Rückkehr enthält L die SHIFT-Taste (00 nicht gedrückt, FF gedrückt). H gibt analog den Zustand von CAPS-LOCK an.



**KM GET JOYSTICK** **BB24**

Die Abfrage der Joysticks, für Spiele besonders interessant, ist mit diesem Einsprung möglich. Die Antwort ist dabei bitweise gespeichert. H enthält Joystick 0, L Nummer 1. Die Routine ist das Gegenstück zum BASIC-Kommando JOY. Die einzelnen Bits bedeuten dabei:

| Bit | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| | auf | ab | links | rechts | Feuer 2 | Feuer 1 | frei | immer 0 |

Der KEY MANAGER enthält auch einige Routinen, die es ermöglichen, die Einträge in den Tastaturübersetzungstabellen abzufragen oder zu verändern. Es sind dies:

| | | | |
|---|---|---|---|
| **KM SET TRANSLATE** | **BB27** | **KM GET TRANSLATE** | **BB2A** |
| **KM SET SHIFT** | **BB2D** | **KM GET SHIFT** | **BB30** |
| **KM SET CONTROL** | **BB33** | **KM GET CONTROL** | **BB36** |

Bei den SET-Routinen muß A die Tastennummer und B den zugehörigen Wert enthalten. Bei GET enthält A beim Einsprung die Tastennummer und liefert beim Aussprung den dazugehörenden Tastaturwert als Antwort. Statt dem direkten POKE oder PEEK nach der oben angegebenen Formel kann man also, speziell in Maschinenprogrammen, auf diese Routinen zurückgreifen, um Tasten zu definieren oder abzufragen.

## 8.2 Die Kassettenspeicherung

Der Firmwareblock des CPC stellt uns auch im Bereich der Kassette, oder besser des Datacorders, einige Routinen zur Verfügung, deren Kennt-nis nicht nur beim Maschinenspracheprogrammieren nützlich sein kann. Bevor wir jedoch zu den Anwendungen übergehen, noch ein paar Worte zum Ablauf und den verschiedenen Möglichkeiten beim Arbeiten mit dem Gerät.

Beginnen wir einmal mit den **Speicherformaten**. Die normalen LOAD- und SAVE-Befehle laufen im wesentlichen automatisch. Interessanter wird es schon, wenn wir an den SAVE-Befehl ein Kürzel anhängen. Dabei stehen drei Varianten zur Auswahl: P, A und B.

Das einfachste dabei ist **P**. Es setzt ein Bit im Dateivorspann, dem HEADER des Programms. Durch Setzen dieses FLAGs wird der Computer angewiesen, nach dem Abbruch eines mit RUN geladenen Programms ein RST 0 (das heißt eine Neu-Initialisierung des kompletten Systems) durchzuführen, womit natürlich auch Programm und auch etwaige Dateien und andere Informationen gelöscht sind.

Das nächste Kürzel ist das A. Es speichert, an SAVE angehängt, ein Programm als ASCII-Textdatei; auf diese Art speichert der CPC auch Daten, z.B. bei

OPENOUT

Die letzte Speicherart erreichen wir durch das angehängte B. Es wird zum Benutzen eines binären FILEs, das heißt normalerweise für Maschinenspracheprogramme und deren Daten verwandt. Während die anderen drei Kürzel Daten und Programme aus dem Benutzerspeicher sichern, ist mit ",B" ein Zugriff auf den Rest des Rechners und damit auch auf die Firmwareroutinen, den Grafikspeicher und andere, normalerweise nicht zugängliche Bereiche, offen. Allerdings gilt dies nur für das Schreiben von Dateien, den FILEs.

Das Lesen, das heißt die Dateneingabe vom Kassettenrecorder in den CPC, ist dagegen an einige Bedingungen gebunden. Ein solches FILE muß nämlich oberhalb der Variablen HIMEM (also in dem für Maschinenprogramme geschützten Bereich) abgelegt sein.

Doch nun zurück zum Ablegen binärer FILEs. Wie wir schon gesagt hatten, bereitet das Speichern keinerlei Probleme. Das Kommando dafür ist relativ einfach (angehängtes B und Anfügen der ersten zu speichernden Stelle sowie der Anzahl der zu speichernden Bytes, gegebenenfalls auch noch einer Ausführungsadresse).

Wird ein solches Programm mit RUN geladen, so lädt es sich in den alten Speicherbereich und beginnt bei der Ausführungsadresse zu laufen. Schwieriger dagegen wird es mit einem LOAD-Befehl. Der CPC benötigt nämlich einen Kassettenpuffer von exakt 4K, das heißt 4096 Byte. Er wird zwischen HIMEM und dem Start der STRINGs (siehe Kapitel 2) bei jedem Lade- oder Schreibvorgang eröffnet, außer, er war noch vom letzten Ladevorgang vorhanden.

Für das Einladen in niedrige Speicherbereiche ergeben sich daraus einige Probleme. Verlegt man nun zum Beispiel HIMEM auf 1000, so fehlt der



nötige Platz für die Zwischenspeicherung, und der CPC gibt ein verärgertes MEMORY FULL aus. Somit kann erst ab ca. 5500 nach oben ein großes Maschinenprogramm abgelegt werden.

Nach soviel LOAD und SAVE in den verschiedenen Formaten ist es nun an der Zeit, die verschiedenen Speicherformate kombiniert einzusetzen. Möglich wird dies durch die Kombination von ASCII- FILEs und BASIC-Programmen. Der Trick besteht darin, daß bei der internen Abspeicherung ein BASIC-Programm im Endeffekt wie ein ASCII-FILE behandelt wird. Daraus leiten sich natürlich sofort interessante Möglichkeiten ab. So ist es nämlich möglich, ein Text-FILE als Programm wieder einzuladen, was den breiten Anwendungsbereich von Programmgeneratoren bis hin zur künstlichen Intelligenz eröffnet.

Wir wollen dies einmal anhand eines bei anderen Homecomputern oft schwierig zu lösendes Problem demonstrieren, nämlich der **Abspeicherung von Maschinenprogrammen** im Rahmen eines BASIC-Programms, das heißt in DATA-Zeilen und möglichst auch noch mit dem entsprechendem LOADER. Dies läßt sich beim CPC relativ einfach lösen, indem man eine Textdatei mit

OPENOUT und PRINT#9

wegschreibt und danach wieder als BASIC-Programm lädt. Wie dies genau geht, zeigt das nebenstehende kleine Programm Datagenerator.

Nach den Anfangsabfragen (Programmname und Speicherstellen) geht es in Zeile 200 in die eigentliche Ausgaberoutine. Hier wird Speicherstelle für Speicherstelle ausgeladen, in einen hex-Code umgewandelt und dann, durch Komma getrennt, zu dem STRING Z$ zusammengefügt. Allen zehn Speicherstellen fügt das Programm dabei eine konstruierte Zeilennummer (beginnend von 1 nach oben) und natürlich den DATA-Befehl vorab an und gibt dann nach der Addition diesen STRING in den Kassettenpuffer ein (Zeile 260). Das Kommando LEFT$(Z$) dient dabei dazu, das letzte Komma vor der Abspeicherung zu unterdrücken.

Auf zwei Besonderheiten muß bei dieser Methode noch hingewiesen werden. STRINGs werden bekanntlich dadurch begonnen oder beendet, daß Anführungsstriche eingegeben werden. Somit ist es natürlich nicht mehr möglich, Anführungsstriche in einem STRING abzuspeichern. Das ist aber bei manchen Programmenzeilen dringend notwendig; beispielsweise bei der POKE-Schleife, die nach den DATA-Zeilen abgelegt wird und dazu dient,

die DATA-Zeilen wieder in ihren angestammten Speicherbereich zu überführen. Diese Schleife wird in Zeile 310 zusammengebaut.

Die Lösung läuft relativ einfach ab. Wir geben diese Zeichen einfach mit der CHR$-Funktion ein. Damit können wir die Halbstrings auf beiden Seiten unseres Problemzeichens wieder verbinden und haben trotzdem das gewünschte Zeichen im STRING gespeichert.

```
10 , ********************
20 , ** Datagenerator **
30 , ********************
40 ,
50 , Anfangsabfragen
60 ,
70 , INPUT"Wie soll das Programm heissen";n$
80 IF n$="" THEN n$="Datazeilen"
90 CLOSEOUT:PRINT"Bitte druecken Sie 'PLAY' und 'RECORD' und dann
beliebige Taste!"
100 IF INKEY$="" THEN 100
110 OPENOUT "!"+n$
120 INPUT"Ab welcher Speicherstelle";sa
130 INPUT"Bis zu welcher Speicherstelle";se
140 IF sa<0 THEN sa=sa+65536
150 IF se<0 THEN se=se+65536
160 IF se<sa THEN PRINT"falscher Bereich!!":GOTO 120
170 '
180 ' Zeilen zusammenfuegen
190 '
200 FOR i=0 TO (se-sa)/10
210 z$=STR$(i+1)+" DATA "
220 FOR j=0 TO 9:w$=HEX$(PEEK(sa+10*i+j)):z$=z$+w$+CHR$(44):NEXT j
230 '
240 ' ...und ausgeben
250 '
260 PRINT#9,LEFT$(z$,LEN(z$)-1)
270 NEXT i
280 '
290 ' POKE-Schleife zusammensetzen
300 '
310 z$=STR$(i+1)+" FOR i="+STR$(sa)+" TO"+STR$(se)+":READ a$:POKE
i"+CHR$(44)+"VAL("+CHR$(34)+"&"+CHR$(34)+"+a$):NEXT i"
320 '
330 ' ...und ausgeben
340 '
350 PRINT#9,z$
360 CLOSEOUT
```



Die notwendigen Codes lauten dabei

CHR$(44)

für die Abspeicherung der Anführungsstriche, beziehungsweise

CHR$(34)

für das Komma. Um sich mit diesen Möglichkeiten vertraut zu machen, sollten Sie einfach einmal das kleine Programm eintippen, sichern und dann laufen lassen, um zu sehen, wie es funktioniert. Unter dem angegebenen Namen wird dann das Datazeilenprogramm inclusive LOADER als Textdatei abgespeichert. Diese können Sie dann mit einem ganz normalen LOAD wieder hereinholen und sich anschauen.

Nachdem wir uns bereits relativ intensiv mit den Anwendungsmöglichkeiten der verschiedenen Speicherformate via BASIC beschäftigt haben, wollen wir uns nun einmal anschauen, wie das Ganze vom **internen Ablauf der Systemfunktionen** und vom Speicheraufbau her abläuft.

Die **Hardwareseite** der Kassettenspeicherung haben wir schon in Kapitel 3 ausreichend behandelt. Pin 4 von Port C des 8255 liefert das Ausgabesignal zum Aufspeichern von Daten auf Kassette, C5 steuert den Motor, über Pin 7 von Port b ist das Einlesen von Daten von der Kassette möglich. In diesem Zusammenhang ist nur zu sagen, daß stärker als bei anderen Computern die Kassettenspeicherung von Software gesteuert ist, sogar die Auswahl der verschiedenen Tonhöhen, die die Nullen und Einsen eines Bytes decodieren erfolgt über Software.

Damit hätten wir auch schon die wichtigste **Grundlage** der Kassettenspeicherung kennengelernt: die Decodierung von Daten mittels verschiedener Tonhöhen. Der CPC verwendet dabei zwei exakte Rechteckschwingungen, von denen die eine (die dem Wert Null entspricht) genau die doppelte Frequenz wie die der Eins entsprechenden Schwingung aufweist. Beide Schwingungen haben ein Tastverhältnis von eins zu eins, das heißt, die Zeit, in der das Signal high ist, entspricht der Länge der Low-Flanke. Um sich einen akustischen Eindruck von den beiden verwendeten Frequenzen zu verschaffen, können Sie einmal den folgenden **Test** durchführen:

Zunächst setzen Sie nach dem Einschalten des Rechners H mit MEMORY 40000 herab. Der Speicherbereich zwischen 40000 und dem alten HIMEM

enthält nun eine Reihe von Nullbit. Mit dem Kommando save "Test", b, 40000, 2048 können Sie diesen Bereich nun wegschreiben. Spulen Sie nun das Band zurück, und hören Sie sich das ganze einmal an. Sie sollten dabei auch einmal die benötigte Zeit für das Anhören eines Blocks stoppen. Als nächstes schreiben wir den gesamten Speicherbereich mit Einserbit voll, was relativ einfach mit

```
FOR i=40000 TO 42048:POKE i,&FF:NEXT i
```

möglich ist, und wiederholen die gesamte Prozedur noch einmal. Nochmaliges Anhören des Bandes liefert uns dann das Testergebnis. Die Frequenz ist deutlich niedriger. Und noch etwas anderes werden Sie feststellen. Die Abspeicherung hat erheblich mehr Zeit in Anspruch genommen. Dies rührt daher, daß der CPC ein Bit immer genau durch eine Schwingung der entsprechenden Frequenz codiert. Schreibt man also mehr Einsen, so dauert die Abspeicherung länger.

Unsere bisherigen Experimente sind im Geschwindigkeitsmode 0 abgelaufen. Mit SPEED WRITE 1 können Sie nun auf die höhere Geschwindigkeit durchschalten und sich das Ganze noch einmal anhören. Wenn wir nun gemischte Daten abspeichern, so benötigt die Abspeicherung irgendeine Zeit zwischen der für die Abspeicherung von Null und der Abspeicherung von nur Eins benötigten Größen. Die einzelnen Bits und Byte sind dabei nicht unterscheidbar, da ihr Wechsel zu schnell stattfindet.

Die im Handbuch angegebenen **Datenübertragungsraten** von 1000 Baud (=1000 Bit pro Sekunde) und 2000 Baud (=2000 Bit pro Sekunde) stellen also nur Mittelwerte dar. Die Änderung der Geschwindigkeit erfolgt via Software, ist also vom Benutzer frei vorgebbar. Die bei der Initialisierung vom Computer eingestellten Werte von 1000 bzw. 2000 Baud sind hier nur Richtwerte. Wir können diese weiter erhöhen. Allerdings sind hier einige Einschränkungen zu machen. Die Fehlerquote bei der Aufzeichnung wird nicht nur durch die Übertragungsfrequenz, sondern natürlich auch durch die verwendete Bandqualität bestimmt. Bei guten Bändern sind Übertragungsraten von 4000 Baud relativ einfach zu realisieren.

Die Einstellung der verschiedenen Parameter kann dabei mit einer Firmwareroutine erfolgen. Sie heißt **CAS SET SPEED** und hat die Adresse BC68. Diese Routine erwartet in Registerpaar HL die Länge für ein halbes Nullbit. Der Akkumulator muß die zugeordnete Vorprüflänge enthalten. Beide Angaben sind dabei in Mikrosekunden festgelegt.



Bei der Vorprüflänge handelt es sich um einen Korrekturfaktor, der durch die Kassettenelektronik bedingt ist. Diese verschleift die Signale, was dazu führt, daß Nullbit länger und Einserbit kürzer gelesen werden, als sie geschrieben wurden. Beim Schreiben wird daher eine Vorkompensation vorgenommen. Ihre zeitliche Größe wird eben durch den Inhalt von Register A bestimmt. Die Standardwerte für HL und A vom System her sind:

SPEED WRITE 0: 333 Mikrosekunden und 25 Mikrosekunden Vorprüflänge

SPEED WRITE 1: 167 Mikrosekunden und 50 Mikrosekunden Vorprüflänge

Bei gleicher Verteilung zwischen Null- und Einsbit in dem zu speichernden Datenfeld können wir von der folgenden Beziehung zwischen Baudrate und halber Nullbitlänge ausgehen:

**Durchschnittliche Baudrate = 1000000 /3mal halbe Nullbitlänge.**

Um 4000 Baud zu erreichen, müßte HL mit 83 geladen werden. Die Vorprüflänge in A wäre dann auszuprobieren. Um dies zu realisieren, benötigen wir ein kleines Maschinenprogramm, welches aus vier Befehlen besteht:

```
LD HL, 59 hex              21 59 00
LD A, 07 hex               3E 07
CALL CAS SET SPEED         CD 68 BC
RET                        C9
```

```
10 MEMORY 39999
20 DATA 21, 59, 00, 3e, 07, cd, 68, bc, c9,x
30 FOR i=40000 TO 50000: READ a$: IF a$<>"x" THEN POKE i,
VAL ("&"+a$): NEXT
40 CALL 40000
```

Wenn Sie nun die Werte für HL und A variieren, können Sie die für Ihren Recorder und vor allem das von Ihnen verwendete Bandmaterial optimale Schreibgeschwindigkeit bestimmen.

Beim Lesen gibt Ihnen dabei der CPC durch die Angabe der Fehler meldung wertvolle Hilfestellung. Es bedeuten:

| | |
|---|---|
| WRITE ERROR A: | Die in HL angegebene Länge war zu kurz; der Rechner konnte das Bit nicht schreiben. |

READ ERROR A: Ein Bit war zu lang.
READ ERROR B: Paritätsprüfung unstimmig. Daten wurden falsch vom Band gelesen.

READ ERROR D: Ein Block enthielt mehr als 2048 Byte.

Mit der letzten Fehlermeldung haben wir den Bereich der Abspeicherung einzelner Bits auf das Band verlassen und den Kontakt mit der **Kassettenverwaltung** bzw. den Speicherformaten aufgenommen.

Ihnen ist sicherlich bekannt, daß der Computer Daten und Programme in Form von Blöcken wegschreibt. Jeder Block kann dabei maximal 2K oder 2048 Byte aufnehmen. Aber auch innerhalb der Blöcke wird Ihnen beim Anhören eine gewisse Struktur aufgefallen sein.

Zunächst hört man nichts (sog. Motorstartlücke). Auf sie folgen zwei Sätze, - der Kopf- und der Datensatz. Der Kopfsatz enthält alle Nebeninformationen, die für die Dateiabspeicherung benötigt werden; das sind Name der Datei, Anzahl der Blöcke, Nummer des abgespeicherten Blocks sowie Informationen darüber, ob es sich um eine geschützte Datei handelt und ähnliches.

Jeder Satz besteht aus drei Teilen:

**Satzvorspann:** Er enthält 2048 Einsbit und (nach einem Trennbit) ein Synchronisationsbyte, das angibt, ob es sich bei dem Satz um einen Kopf- oder um einen Datensatz handelt.

**Datenteil:** Er besteht aus einem bis acht Segmenten mit jeweils 256 Byte (getrennt durch eine bestimmte Bitkonfiguration) und enthält die eigentlichen zu speichernden Daten.

**Satznachspann:** Er besteht aus 32 Nullbit, die die Endemarkierung darstellen.

Ein Kopfsatz besteht dabei aus einem Segment, ein Datensatz aus bis zu acht. Wenn wir uns nun einmal eine Datei vom Band anhören, können wir die einzelnen Teile relativ gut herausfinden.



Zu Beginn des Blocks hört man eine tiefe Frequenz: die 2048 Einsbit. Es folgt das eine Segment für den Kopfsatz. Da es zu einem großen Teil Nullen enthält, geht die Frequenz hoch. Am Ende, in einem kurzen Piepser, wieder von tieferer Frequenz, erhalten wir den Satznachspann von 32 Einsbit. Es folgt wiederum eine kurze Anfangslücke, wonach dann, zunächst auf tieferer Frequenz (2048 Einsbit), maximal acht Segmente für den Datensatz folgen.

Bei der Abspeicherung unserer Testdateien zu Beginn dieses Unterkapitels können Sie die Trennung zwischen den einzelnen Segmenten gut ausmachen. In der gleichförmigen Frequenz tauchen einzelne kurze Piepser auf. Diese stellen die Trennmarkierungen dar. Soviel zur physikalischen Seite der Abspeicherung.

Schauen wir uns nun einmal an, welche Informationen im Vorspann bzw. im Kopfsatz abgelegt sind. Bei der Abspeicherung legt der CPC einen Datenein- bzw. Datenausgabepuffer direkt unterhalb der Speicherobergrenze, daß heißt unterhalb von HIMEM an. In diesem Bereich wird der Datensatz vor dem Schreiben auf das Band abgelegt.

Gleichzeitig wird im Bereich der Firmwaredaten ab Adresse 47180 ein **Vorspann** für die Datei abgelegt. Dieser besteht aus jeweils 64 Zeichen. Da der CPC gleichzeitig sowohl eine Eingabe- als auch eine **Ausgabedatei** verarbeiten kann, muß dieser Block von 64 Zeichen zweimal angelegt werden. Wird eine **Ausgabedatei** eröffnet, so wird er ab 47180 abgelegt, ansonsten 69 Byte tiefer, das heißt ab Adresse 47111. Wird eine Datei dann tatsächlich eingelesen, wird deren Header an der Adresse 47244 abgelegt und überprüft, ob er mit dem Header an der Adresse 47111 übereinstimmt, also ob tatsächlich die gesuchte Datei gefunden wurde.

Der Vorspannblock kann in zwei Felder unterteilt werden. Das **Systemfeld** umfaßt die Byte 0 bis 23 und enthält im wesentlichen vom System gesetzte Daten, die sich auf die Anzahl der Datenbytes, die Gesamtzahl der Blöcke, Anfangs- und Endemarkierungen einer Datei etc. beziehen.

Das **Anwenderfeld** umfaßt Daten, die durch Aktionen des Anwenders vorgegeben wurden, wie die totale Länge der Datei in Bytes, eine Ausführungsadresse für Maschinencodeprogramme und einen größeren Freiraum nicht belegter Bytes, die vom Benutzer verwandt werden können. Die nachfolgende Tabelle gibt eine übersicht über die Bedeutung der einzelnen Bytes im Dateivorspann. Der Dateityp (Byte 18) ist dabei bitorientiert abgespeichert. Hier decodieren einzelne Bits verschiedene Angaben.

**Tabelle 8.2:** Vorspannfelder in der Kassettenverwaltung

**Systemfeld**

| | | | |
|---|---|---|---|
| Byte | 0..15 | Dateiname | 16 Byte (Rest mit Nullen aufgefüllt) |
| Byte | 16 | Blocknummer | |
| Byte | 17 | Letzter Block | Ein Wert ungleich Null besagt, daß dies der letzte Block einer Datei ist (Typisch 255=FF) |
| Byte | 18 | Dateityp | |

Der Dateityp ist in eine Anzahl Felder unterteilt:

| | | | |
|---|---|---|---|
| Bit | 0 | Schutz | Datei geschützt. Wenn dieses Bit gesetzt ist, ist die Datei geschützt. |
| Bit | 1..3 | Dateiinhalt | 0=Internes BASIC<br>1=Binär<br>2=Bildschirm-Darstellung (Bild)<br>3=ASCII<br>4..7 = nicht festgelegt |
| Bit | 4..7 | Version | ASCII-Dateien sollten auf 1, alle anderen auf 0 gesetzt sein |
| Byte | 19,20 | Länge | Anzahl der Datenbytes im Datensatz |
| Byte | 21,22 | Datenherkunft | enthält die Adresse der Stelle, von der die Daten ursprünglich stammen |
| Byte | 23 | Merker für ersten Block | Ein Wert ungleich Null besagt, daß dies der erste Block ist |

**Anwenderfeld**

| | | | |
|---|---|---|---|
| Byte | 24,25 | Logische Länge | totale Länge der Datei in Bytes |
| Byte | 26,27 | Einsprung-adresse | die Ausführungsadresse für Maschinencodeprogramme (SAVE "<Name>", b) |
| Byte | 28..63 | Nicht festgelegt | können beliebig verwendet werden |



Schauen wir uns das Ganze einmal anhand eines praktischen Beispiels an. Laden Sie einmal ein kleineres, das heißt wenige Blocks umfassendes, Programm in den Speicher, und lassen Sie sich dann einmal mit unserer üblichen FOR-TO-Schleife die Werte ab 47244 ausgeben.

```
10 FOR i=47244 TO 47310:PRINT i, PEEK (i),:IF PEEK (i) > 32 THEN
PRINT CHR$(PEEK(i)) ELSE PRINT
20 NEXT i
```

Sie werden nun auf dem Schirm erst den Namen Ihrer Datei wiederfinden. Er ist in den ersten 16 Byte gespeichert. Nicht benutzte Stellen werden dabei mit Null aufgefüllt. Aus diesem Grund sollten Sie niemals in einem Dateinamen CHR$(0) benutzen. Der CPC würde dies als Endemarkierung des Namens interpretieren.

Byte 16, das heißt Adresse 47260 enthält die Blocknummer. Da Ihre Eingabedatei Block für Block eingelesen wurde, finden Sie hier den letzten Block abgespeichert, bei z.B. drei Blöcken erscheint also hier eine 3. Es folgt in Byte 18 der Dateityp, hier auf Null gesetzt, da sowohl die Datei internes BASIC enthält als auch ungeschützt abgespeichert wurde. Probieren Sie das Ganze einmal mit dem Befehl CAT oder einer geschützten Datei oder mit dem Einlesen einer ASCII-Datei mit dem Kommando openin aus. Sie werden dann hier andere Werte erhalten.

Nach dieser Übersicht über den Ablauf des Speichervorgangs wollen wir uns nun mit den Eingriffsmöglichkeiten via Maschinensprache näher beschäftigen. Der Firmwarejumpblock (Unterabteilung CAS) bietet uns hierzu eine reichhaltige Auswahl.

CAS SET SPEED haben wir schon kennengelernt. Sie diente dazu, die Geschwindigkeit bei der Abspeicherung vorzugeben. Die Motorsteuerung ist mit 3 Routinen möglich:

| | | |
|---|---|---|
| **CAS START MOTOR** | BC6E | startet den Kassettenmotor, A enthält den vorigen Zustand des Motors |
| **CAS STOP MOTOR** | BC71 | Stoppt den Kassettenmotor, A enthält vorigen Zustand des Motors |

**CAS RESTORE MOTOR** BC74 Diese Routine setzt den Kassettenmotor auf den vorigen Zustand zurück. Dieser ist dem Akkumulator zu übergeben.

Die ersten beiden Routinen können wir auch von BASIC aus benutzen. Da bei der Interpretation von BASIC-Befehlen nach dem Rücksprung aus einem Maschinenprogramm der Akkumulator für andere Zwecke wieder benutzt wird und somit nicht mehr den alten Wert enthält, ist CAS RESTORE MOTOR nur in der Maschinenebene einsetzbar.

**Beispiel:** Geben Sie einmal CALL & BC6E ein. Der Kassettenmotor beginnt zu laufen.

Eine weitere interessante Routine ist CAS NOISY BC6B. Sie läßt Bereitschaftsmeldungen zu bzw. sperrt sie. Die Bereitschaftsmeldungen, wie PRESS PLAY etc. werden dann nicht mehr ausgegeben. Die Fehlermeldungen bleiben jedoch erhalten. Die Routine hat dieselbe Funktion wie ein vorangestelltes Ausrufezeichen im Dateinamen. Die Angabe, ob Meldungen zugelassen werden sollen, erfolgt wiederum im Akkumulator. Sollen die Meldungen erfolgen, so muß A Null enthalten. Ansonsten muß A ungleich Null sein.

Wir kommen nun zu den Kommandos, mit denen wir Dateien aus- bzw. eingeben können. Wir können dabei zwei Bereiche unterscheiden:

Vorbereitungsoperationen, die Dateien für Eingabe- oder Ausgabegeräte öffnen oder schließen.

Datentransferoperationen, die für das Schreiben in oder Lesen aus einer Datei zuständig sind.

Bei beiden müssen wir immer noch zwischen Ein- und Ausgabe unterscheiden, das heißt, es stehen hier verschiedene Routinen zur Verfügung. Das Eröffnen einer Datei geschieht mit

**CAS IN OPEN BC77 CAS OUT OPEN BC8C**

Register B enthält die Länge des Dateinamens, HL seine aktuelle Position. In DE ist die Adresse eines 2K-Puffers gespeichert, der für die Zwischenspeicherung der Daten verwendet werden soll.



Das Schließen einer Datei geschieht mit den nachfolgenden Routinen:

**CAS IN CLOSE BC7A CAS IN ABANDON BC7D**
**CAS OUT CLOSE BC8F CAS OUT ABANDON BC92**

Der Unterschied zwischen ABANDON und CLOSE besteht darin, daß CLOSE ein Eingabegerät als geschlossen markiert und damit die Eingabedatei abschließt. Wenn das Eingabegerät nicht eröffnet war, ist das Carry auf Null, ansonsten auf 1. ABANDON bricht in jedem Fall das Lesen vom Eingabegerät ab und schließt die Eingabe. Diese Routine ist im wesentlichen für den Fehlerfall und ähnliche Umstände vorgesehen. Hier hat das Flag keinerlei Funktionen. Nach dem Aufruf einer der vier Routinen kann der Schreib- bzw. Lesepuffer wieder für andere Zwecke verwendet werden. Er wird dann freigegeben.

Nach den Befehlen zur Dateivorbereitung kommen wir nun zu den Kommandos, mit denen wir auf eine Datei zurückgreifen können, den eigentlichen Datentransferbefehlen. Wir können bei den Kassettenoperationen grundsätzlich zwei Arten von Datentransfers unterscheiden, die Ein- bzw. Ausgabe von Zeichen soweit die Direkteingabe bzw. -ausgabe eines ganzen Blocks. Dementsprechend existieren auch zwei verschiedene Unterroutinen.

**CAS IN CHAR BC80 CAS OUT CHAR BC95**

geben ein Zeichen aus bzw. lesen es ein. Der richtige Datentransfer wird dabei durch ein Carry "ein" zurückgemeldet. Bei EOF oder nicht eröffneter Datei ist das Carry "aus". Ein gesetztes Zeroflag zeigt an, daß während der Abfrage ESCAPE gedrückt wurde.

**CAS IN DIRECT BC83**

Diese Routine liest eine Eingabedatei in den Speicher ein. HL enthält dabei die Adresse (irgendwo im RAM), ab der die Datei abgelegt werden soll. Dabei ist zu beachten, daß eine einmal mit CAS IN CHAR aufgerufene Datei nicht mehr mit CAS IN DIRECT gelesen werden kann.

**CAS OUT DIRECT BC98**

stellt den Datentransfer in Gegenrichtung dar. Das benötigt eine ganze Reihe von Registern. HL enthält die Adresse der auszugebenden Daten, DE ihre Länge. BC enthält die Einsprungadresse (Byte 26 und 27 im Kopfsatz),

A die Dateiart (Byte 18 im Kopfsatz). Die Bedeutung der Flags ist wie bei CAS IN CHAR bzw. CAS OUT CHAR beschrieben.

Beim zeichenweisen Einlesen existieren noch zwei besondere Routinen. Es ist nämlich möglich, ein gelesenes Zeichen in den Eingabepuffer rückzuübertragen. Das Prinzip ist dabei mit der Zeichenrückgabe bei der Tastaturabfrage identisch (vergleiche Kapitel 8.1). Die Zeichenrückgabe erfolgt mit der Routine

**CAS RETURN BC86**

Die Routine benötigt keine Register, und auch die FLAGs bleiben unverändert. Bedingung dafür ist allerdings, daß bereits ein Zeichen gelesen wurde. Zum Feststellen des Dateiendes (EOF) dient ein weiterer Einsprung

**CAS TEST EOF BC89**

Bei Dateiende ist das Carry-FLAG gesetzt, das Zero-FLAG zeigt wiederum an, daß eine Unterbrechung mit Escape erfolgt ist. Bei der Anwendung dieser Routine muß in diesem Fall noch darauf hingewiesen werden daß es nach dieser Routine nicht mehr möglich ist, CAS RETURN aufzurufen, bevor nicht ein weiteres Zeichen gelesen wurden. Auch ist ein Direktzugriff nicht mehr möglich.

Wie setzen wir nun diese Routinen in eigenen Maschinenprogrammen ein? Dies ist realtiv einfach und ergibt sich eigentlich schon aus dem bisher Gesagten. Als erster Schritt muß immer eine Datei eröffnet werden, wahlweise zum Lesen oder zum Schreiben. Dabei kann maximal eine Datei in jeder Richtung gleichzeitig bestehen. Es folgt der Datentransfer direkt oder zeichenweise und abschließend das Schließen der Datei mit CLOSE oder ABANDON.

Um ein zwischenzeitliches Überlaufen des Puffers braucht man sich nicht zu kümmern. Die Betriebssystemroutinen stellen sicher, daß, sobald ein Block voll ist, dieser zunächst aufs Band geschrieben wird, bevor ein neues Zeichen an den Puffer weitergegeben wird.



## 8.3 Die Druckersteuerung

Wir wollen uns nun mit einem weiteren Anwendungsbereich im Gebiet der Ein-/Ausgabesteuerung beschäftigen, der Ansprache unseres Druckers. Die hardwaremäßige Seite haben wir schon in Kapitel 3 ausreichend behandelt. An dieser Stelle soll auf einige Firmwareroutinen eingegangen werden, die uns den Zugriff auf den Drucker ermöglichen.

Davor sollte man auf einige grundsätzliche Probleme zu sprechen kommen. Wie wir schon in Kapitel 3 gesehen haben, findet die Übergabe von Daten an den Drucker beim CPC im 7-Bit-Format statt, das heißt, das 8. Datenbit wird nicht benutzt. Es ist auf LOW-Signal gelegt. Dies führt zu der etwas unangenehmen Konsequenz, daß eine ganze Reihe von Grafikbefehlen, über die viele Drucker verfügen und die normalerweise mit Codes von über 128 (was also ein Setzen des 8. Bits impliziert) angesprochen werden, nicht ausgeführt werden können.

Eine sehr lästige Eigenschaft des CPC besteht darin, daß er bei nicht angeschlossenem oder nicht aktiviertem Drucker nicht mit einer Fehlermeldung zurückkehrt, sondern - gegebenenfalls ewig - wartet. Zwar kann man den Computer durch Drücken von ESC dazu bringen, das Programm bzw. die Ausgabeoperation abzubrechen. Dies ist aber weder komfortabel, noch kann diese Methode bei PROTECT-gespeicherten Programmen verwandt werden, da in diesem Fall der Rechner den kompletten Speicher löscht.

Der Grund dafür ist relativ einfach. Im Betriebssystem, genauer gesagt im Maschinenpaket, existiert eine Reihe von Routinen für die Druckeransprache. Die interessanteste dabei ist

**MC PRINT CHAR BD2B**

Mit dieser Routine ist es möglich, ein Zeichen an den Centronixausgang zu senden. Der Akkumulator muß dabei das zu sendende Zeichen enthalten, wobei Bit 7 ignoriert wird. Es werden also nur die unteren 7 Bit ausgesandt; das 8. Bit ( vergleiche Kapitel 3) wird also STROBE-Signal, das heißt für die Aktivierung des Druckers benutzt. Wenn das Zeichen richtig übersandt wurde, ist das Carry-FLAG an.

War der Drucker länger als ca. 0.4 Sekunden beschäftigt, erfolgt eine Zeitabschaltung (vgl. Routine MC BUSY PRINTER), und das Zeichen wird nicht gesendet. Das Carry-FLAG ist dann aus.

**MC BUSY PRINT BD2E:** Mit dieser Routine ist es möglich zu überprüfen, ob der Centronixausgang beschäftigt ist. Sie wird als Unterprogramm von MC PRINT CHAR angesprungen, um diese Aussage abzutesten. Wenn der Centronixausgang beschäftigt ist, ist das Carry an, bei freiem Centronix ist es aus. Sonst hat diese Routine keine Auswirkungen.

Will man nur ein Zeichen an einen freien Printer übergeben, so kann man auf die Routine MC SEND PRINTER BD31 zurückgreifen. A enthält wieder das zu sendende Zeichen, das Carry-FLAG ist beim Aussprung an.

Für die Benutzung von BASIC aus ist besonders die Routine MC BUSY PRINTER interessant, da wir mit ihr abtesten können, ob ein Drucker überhaupt angeschlossen ist und somit gegebenenfalls eine Einschaltmeldung auf den Bildschirm bringen können (z.B. "bitte Drucker einschalten", wenn der Drucker nicht eingeschaltet ist). Dazu genügt ein relativ kleines Maschinenprogramm.

PRINTERKONTROLLE

```
CALL MC BUSY PRINTER    CD 2E BD
JVC+4                   38 04
LD (42020),A            32 24 A4
RET                     C9
LDA, 1                  3E01
LD (42020), A           32 24 A4
RET                     C9
```

Der Ablauf: Zunächst wird MC BUSY PRINTER angesprungen. Es folgt eine bedingte Verzweigung in Abhängigkeit vom Carry. Ist das Carry nicht gesetzt, setzt sich der Programmablauf nahtlos fort. A wird mit Null geladen, danach in die Speicherstelle 42020 (A424 hex) übertragen. Es erfolgt der Rücksprung in das aufgerufene Programm. War dagegen das Carry gesetzt, wird A mit 1 geladen, und mit diesem neuen Wert läuft das Programm wie oben beschrieben ab. Von BASIC aus können wir nun mit PRINT PEEK von 42020 feststellen, ob der Drucker empfangsbereit ist oder nicht. Verwenden wir das PEEK-Kommando in einer IF-Abfrage, so können wir daran anschließend eine Einschaltmeldung ausgeben oder diese unterdrücken.



## 8.4 Grafikroutinen

Zum Abschluß dieses Buches wollen wir uns nun noch mit einigen Anwendungen aus dem Bereich der Grafik beschäftigen. Hier muß allerdings eine Warnung vorab gegeben werden. Die Grafik- und Textroutinen sind der wohl größte Bereich im Firmware-Jumpblock. Es stehen über 100 Routinen zur Verfügung, mit denen sich sehr schöne Effekte und Anwendungen erzielen lassen.

Hier eine umfassende und gegliederte Einführung in alle Routinen zu geben, ist weder Sinn dieses Buches noch vom Umfang her realisierbar. Wer sich also mit diesen Routinen weiterbeschäftigen möchte, sei hier auf das CPC-Grafikbuch verwiesen, das im selben Verlag erschienen ist.

Wir wollen an dieser Stelle nur einige Spezialitäten und einfache, schnell nutzbare Routinen vorstellen. Zunächst einmal zum grundsätzlichen Aufbau. Es existieren 3 Untersprungblöcke (vergleichen Sie dazu Kapitel 7). Dies sind die Textroutinen (TXT), die Grafikroutinen (GRA) und eine Gruppe weiterer Routinen, die von den vorgenannten beiden benötigt werden, das Bildschirmpaket (SCR).

Wir wollen uns das Ganze von der Anwendungsseite her anschauen. Die **Cursordarstellung** können wir mit den Unterroutinen

**TXT PLACE CURSOR BB8A** bzw.**TXT REMOVE CURSOR BB8D**

beeinflussen. Die erste Routine setzt das Cursorzeichen auf den Bildschirm, die zweite löscht es. Dabei ist zu beachten, daß jede dieser Routinen nur einmal hintereinander aufgerufen werden sollte; nach TXT PLACE CUR SOR muß also erst TXT REMOVE CURSOR aufgerufen werden, bevor man ein neues PLACE durchführen kann.

Ändert man nämlich zwischenzeitlich die aktuelle Cursorposition, so bleibt ansonsten das Cursorzeichen auf dem Bildschirm stehen. Außerdem ist auch noch zu beachten, daß TXT REMOVE CURSOR gegebenenfalls ein Cursorzeichen auf dem Bildschirm darstellt, wenn sich an dieser Stelle keines befindet. Bei falscher Anwendung erscheinen also Cursorsymbole auf dem Bildschirm.

Das Setzen der Cursorposition kann man mit dem BASIC-Kommando LOCATE erreichen. Will man an einer bestimmten Stelle des Bildschirms Zeichen ausgeben oder von dieser lesen, so muß zunächst der Bildschirmbereich als Ein- und Ausgabegerät adressiert sein.

Beim CPC wird dabei das STREAM-Konzept verwandt. Alle Änderungen, wie z.B. das Setzen von Schrift oder Hintergrundfarbe, die Ausgabe von Fensterbegrenzungen (Windows), das Zulassen oder Sperren des Cursors oder auch des Zeichendarstellungsmodus bedingen, daß das gewünschte Ein-/Ausgabegerät als aktuelles E/A-Gerät definiert wurde. Dies geschieht mit der Routine

**TXT STR SELECT BBB4**

Diese wählt aus den 8 möglichen Windows (0 bis 7) das aktuelle Ein-/Ausgabegerät aus. Alle weiteren Ansprachen laufen dann auf dieses Gerät. Die Nummer des auszuwählenden Windows ist dabei in A mitzugeben. Beim Aussprung enthält A das vorher ausgewählte E/A-Gerät, HL und die Flags sind zerstört. Man sollte diese Routine immer vor einer Ausgabe anspringen, um sicherzustellen, daß das gewählte Ein-/Ausgabegerät auch wirklich als aktuelles Ein-/Ausgabegerät definiert ist.

Somit ist es uns nun auch möglich, mehrere Cursor gleichzeitig (maximal 8) auf dem Bildschirm darzustellen. Dazu definieren wir 8 verschiedene Windows, sprechen diese als aktuelle Geräte nacheinander an und setzen den Cursor bzw. setzen ihn zurück. Das Setzen der Schriftfarbe im ausgewählten E/A-Gerät geht mit der Routine

**TXT SET PEN BB90**

A enthält dabei die zu benutzende INK. Von Maschinensprache aus läßt sich auch etwas darstellen, was via BASIC nicht möglich ist: nämlich die aktuelle Schriftfarbe innerhalb eines E/A-Geräts abfragen.

Dies geschieht mit der Routine

**TXT GET PEN BB93**

A enthält bei Aussprung die INK, mit der in diesem Gerät gerade geschrieben wird. In Analogie zu den oben genannten Routinen existieren auch zwei weitere Unterprogramme für die Hintergrundfarbe. Es sind dies die Einsprünge



**TXT SET PAPER BB96 und TXT GET PAPER BB99**

Die Ein- und Aussprungbedingungen sind hier analog zur Behandlung der Schriftroutinen.

Daneben gibt es noch die Möglichkeit einer Inversdarstellung, das heißt des Austauschens von Stift- und Papierfarbe. Dies geschieht mit der Routine

**TXT INVERSE BB9C**

Um eine schnelle Änderung verschiedener Werte zwischen zwei Ein-/Ausgabegeräten erzeugen zu können, existiert die Routine

**TXT SWAP STREAMS BBB7**

Diese vertauscht die Werte für PEN, PAPER, die Cursorposition und die Bildschirm- bzw. WINDOW-Grenzen sowie den Zeichendarstellungsmode mit den Werten des anderen Windows. B enthält dabei die Nummer eines E/A-Gerätes, C das andere. Beim Aussprung sind alle Registerpaare bis auf die Indexregister zerstört. Diese Operation ist nur zwischen Windows, nicht jedoch zwischen Textfenstern und wirklichen externen Geräten wie Drucker oder Datacorder möglich.

Zum Schluß dieses Kapitels kommen wir nun noch zu einigen Routinen aus dem Grafik-Bereich. Die Grafik-Routinen haben ausschließlich das Arbeiten mit einzelnen Bildpunkten, den Pixeln, zur Aufgabe. Ebenso wie bei den TXT-Einsprüngen sind auch die Aufrufe in diesem Bereich meist ausführende Programme oder Unterprogramme, die bei der Interpretation von BASIC-Befehlen benötigt werden.

Alle Grafikroutinen sind abhängig von der aktuellen Stellung des Grafik-Cursors. Dieser läßt sich absolut oder relativ bewegen. An seiner aktuellen Position können Punkte gezeichnet oder von dort Linien gezogen werden. Bei allen Operationen ist darüber hinaus zu berücksichtigen, daß eine Ausgabe auf dem Bildschirm nur dann erfolgt, wenn der Grafik-Cursor innerhalb eines vom Benutzer vorgegebenen Bereiches des Bildschirms liegt, dem Grafik-Fenster.

Zum Setzen des Grafik-Cursors stellt uns der Firmware-Jumpblock den Einsprung

**GRA MOVE ABSOLUTE BBC0**

zur Verfügung. Dieser setzt den Grafik-Cursor auf den durch DE (X-Koordinate) und HL (Y-Koordinate) definierten Bildschirmpunkt. Die Angabe erfolgt dabei relativ zum Benutzerursprung. Diese Routine stellt also das Gegenstück zu dem BASIC-Befehl MOVE dar.

Eine relative Bewegung kann man mit

**GRA MOVE RELATIVE BBC3**

erreichen. Die Register sind dieselben. Allerdings wird hier die Position relativ zum Ursprung, dem BASIC-Befehl MOVER entsprechend, angegeben. Wollen wir die aktuelle Position des Cursors abfragen, so können wir die Routine

**GRA ASK CURSOR BBC6**

benutzen. Diese liefert uns wiederum in DE und HL die entsprechenden Werte. Das Setzen des Ursprungs können wir mit

**GRA SET ORIGIN BBC9**

erreichen. DE und HL enthalten hier die Koordinaten, wie beim BASIC-Befehl ORIGIN gewohnt. Mit den nachfolgenden Befehlen ist es uns nun möglich, Punkte zu setzen und Linien zu ziehen:

**GRA PLOT ABSOLUTE BBEA** und **GRA PLOT RELATIVE BBED**

**GRA LINE ABSOLUTE BBF6** und **GRA LINE RELATIVE BBF9**

Die ersten beiden Kommandos sind dabei für das Setzen eines Punktes, die unteren beiden für das Zeichnen von Linien zuständig. In allen Routinen erfolgt die Angabe von absoluter oder relativer Position in DE (X-Koordinate bzw. X-Abstand) und HL (Y-Koordinate bzw. Y-Abstand). Die Funktion und auch die Definition sind dabei mit den BASIC-Kommandos PLOT, PLOTR, DRAW und DRAWR identisch.



**Beispiele:** Als erstes wollen wir eine Linie vom Punkt (0,100) zum Punkt (200,100) ziehen, also einen waagrechten Strich. Dazu setzen wir den Grafikcursor auf (0,100) und springen dann mit DE= dezimal 200 und HL= dezimal 100 in die Routine GRA LINE ABSOLUT.

```
LD DE, 0000                 11 00 00
LD HL, 0064                 21 64 00
CALL GRA MOVE ABSOLUTE      CD C0 BB
LD DE, 00C8                 11 C8 00
CALL GRA LINE ABSOLUTE      CD F6 BB
RET                         C9
```

Nach der Ausführung dieses Programms erhalten Sie die gewünschte Linie. Wie würde das Ganze nun für eine relative Bewegung lauten? Der Ablauf der Routinen wäre derselbe. Einzige Änderung: Wir müßten HL auf 0 löschen, weil ansonsten ein Strich um 200 Einheiten in Y-Richtung, also nach (200,300), gezogen würde. Danach könnten wir dann GRA LINE RELATIVE aufrufen. Probieren Sie es doch einmal aus!

# 9 Anhang

## CODES beim CPC 464

| dezimal | binär | hex | TOKEN | Funktions-TOKEN | ASCII/Text |
|---|---|---|---|---|---|
| 00 | 00000000 | 00 | Zeilenende | ABS | NUL |
| 01 | 00000001 | 01 | Statementende | ASC | SOH |
| 02 | 00000010 | 02 | Kennzeichen % | ATN | STX |
| 03 | 00000011 | 03 | Kennzeichen $ | CHR$ | ETX |
| 04 | 00000100 | 04 | Kennzeichen ! | CINT | EOT |
| 05 | 00000101 | 05 | "SYNTAX ERROR" | COS | ENQ |
| 06 | 00000110 | 06 | "SYNTAX ERROR" | CREAL | ACK |
| 07 | 00000111 | 07 | "SYNTAX ERROR" | EXP | BEL |
| 08 | 00001000 | 08 | "SYNTAX ERROR" | FIX | BS |
| 09 | 00001001 | 09 | "SYNTAX ERROR" | FRE | HT |
| 10 | 00001010 | 0A | "SYNTAX ERROR" | INKEY | LF |
| 11 | 00001011 | 0B | "SYNTAX ERROR" | INP | VT |
| 12 | 00001100 | 0C | "SYNTAX ERROR" | INT | FF |
| 13 | 00001101 | 0D | Variable ohne Kennzeichen | JOY | CR |
| 14 | 00001110 | 0E | Ganzzahl 0 | LEN | SO |
| 15 | 00001111 | 0F | Ganzzahl 1 | LOG | SI |
| 16 | 00010000 | 10 | Ganzzahl 2 | LOG10 | DLE |
| 17 | 00010001 | 11 | Ganzzahl 3 | LOWERS$ | DC1 |
| 18 | 00010010 | 12 | Ganzzahl 4 | PEEK | DC2 |
| 19 | 00010011 | 13 | Ganzzahl 5 | REMAIN | DC3 |
| 20 | 00010100 | 14 | Ganzzahl 6 | SGN | DC4 |
| 21 | 00010101 | 15 | Ganzzahl 7 | SIN | NAK |
| 22 | 00010110 | 16 | Ganzzahl 8 | SPACE | SYN |
| 23 | 00010111 | 17 | Ganzzahl 9 | SQ | ETB |
| 24 | 00011000 | 18 | "SYNTAX ERROR" | SQR | CAN |
| 25 | 00011001 | 19 | Ein Byte Zahl | STR$ | EM |
| 26 | 00011010 | 1A | Zwei Byte Zahl dezimal | TAN | SUB |
| 27 | 00011011 | 1B | Zwei Byte Zahl binär | UNT | ESC |
| 28 | 00011100 | 1C | Zwei Byte Zahl hex | UPPER$ | FS |

| | | | | | | |
|---|---|---|---|---|---|---|
| 29 | 00011101 | 1D | Zeilenadresse | VAL | | GS |
| 30 | 00011110 | 1E | Zeilennummer | n.b. | | RS |
| 31 | 00011111 | 1F | Realvariable | n.b. | US | |
| 32 | 00100000 | 20 | "SYNTAX ERROR" | n.b. | | SP |
| 33 | 00100001 | 21 | '' | n.b. | | ! |
| 34 | 00100010 | 22 | '' | n.b. | | " |
| 35 | 00100011 | 23 | '' | n.b. | | # |
| 36 | 00100100 | 24 | '' | n.b. | | $ |
| 37 | 00100101 | 25 | '' | n.b. | | % |
| 38 | 00100110 | 26 | '' | n.b. | | & |
| 39 | 00100111 | 27 | '' | n.b. | | ' |
| 40 | 00101000 | 28 | '' | n.b. | | ( |
| 41 | 00101001 | 29 | '' | n.b. | | ) |
| 42 | 00101010 | 2A | '' | n.b. | | * |
| 43 | 00101011 | 2B | '' | n.b. | | + |
| 44 | 00101100 | 2C | '' | n.b. | | , |
| 45 | 00101101 | 2D | '' | n.b. | | - |
| 46 | 00101110 | 2E | '' | n.b. | | . |
| 47 | 00101111 | 2F | '' | n.b. | | / |
| 48 | 00110000 | 30 | '' | n.b. | | 0 |
| 49 | 00110001 | 31 | '' | n.b. | | 1 |
| 50 | 00110010 | 32 | '' | n.b. | | 2 |
| 51 | 00110011 | 33 | '' | n.b. | | 3 |
| 52 | 00110100 | 34 | '' | n.b. | | 4 |
| 53 | 00110101 | 35 | '' | n.b. | | 5 |
| 54 | 00110110 | 36 | '' | n.b. | | 6 |
| 55 | 00110111 | 37 | '' | n.b. | | 7 |
| 56 | 00111000 | 38 | '' | n.b. | | 8 |
| 57 | 00111001 | 39 | '' | n.b. | | 9 |
| 58 | 00111010 | 3A | '' | n.b. | | : |
| 59 | 00111011 | 3B | '' | n.b. | | ; |
| 60 | 00111100 | 3C | '' | n.b. | | < |
| 61 | 00111101 | 3D | '' | n.b. | | = |
| 62 | 00111110 | 3E | '' | n.b. | | > |
| 63 | 00111111 | 3F | '' | n.b. | | ? |
| 64 | 01000000 | 40 | '' | EOF | | §/@ |
| 65 | 01000001 | 41 | '' | ERR | | A |
| 66 | 01000010 | 42 | '' | HIMEM | | B |
| 67 | 01000011 | 43 | '' | INKEY$ | | C |
| 68 | 01000100 | 44 | '' | PI | | D |
| 69 | 01000101 | 45 | '' | RND | | E |
| 70 | 01000110 | 46 | '' | TIME | | F |



| | | | | | |
|---|---|---|---|---|---|
| 71 | 01000111 | 47 | '' | XPOS | G |
| 72 | 01001000 | 48 | '' | YPOS | H |
| 73 | 01001001 | 49 | '' | n.b. | I |
| 74 | 01001010 | 4A | '' | n.b. | J |
| 75 | 01001011 | 4B | '' | n.b. | K |
| 76 | 01001100 | 4C | '' | n.b. | L |
| 77 | 01001101 | 4D | '' | n.b. | M |
| 78 | 01001110 | 4E | '' | n.b. | N |
| 79 | 01001111 | 4F | '' | n.b. | O |
| 80 | 01010000 | 50 | '' | n.b. | P |
| 81 | 01010001 | 51 | '' | n.b. | Q |
| 82 | 01010010 | 52 | '' | n.b. | R |
| 83 | 01010011 | 53 | '' | n.b. | S |
| 84 | 01010100 | 54 | '' | n.b. | T |
| 85 | 01010101 | 55 | '' | n.b. | U |
| 86 | 01010110 | 56 | '' | n.b. | V |
| 87 | 01010111 | 57 | '' | n.b. | W |
| 88 | 01011000 | 58 | '' | n.b. | X |
| 89 | 01011001 | 59 | '' | n.b. | Y |
| 90 | 01011010 | 5A | '' | n.b. | Z |
| 91 | 01011011 | 5B | '' | n.b. | Ä/[ |
| 92 | 01011100 | 5C | '' | n.b. | Ö/\ |
| 93 | 01011101 | 5D | '' | n.b. | Ü/] |
| 94 | 01011110 | 5E | '' | n.b. | ^ |
| 95 | 01011111 | 5F | '' | n.b. | _ |
| 96 | 01100000 | 60 | '' | n.b. | ' |
| 97 | 01100001 | 61 | '' | n.b. | a |
| 98 | 01100010 | 62 | '' | n.b. | b |
| 99 | 01100011 | 63 | '' | n.b. | c |
| 100 | 01100100 | 64 | '' | n.b. | d |
| 101 | 01100101 | 65 | '' | n.b. | e |
| 102 | 01100110 | 66 | '' | n.b. | f |
| 103 | 01100111 | 67 | '' | n.b. | g |
| 104 | 01101000 | 68 | '' | n.b. | h |
| 105 | 01101001 | 69 | '' | n.b. | i |
| 106 | 01101010 | 6A | '' | n.b. | j |
| 107 | 01101011 | 6B | '' | n.b. | k |
| 108 | 01101100 | 6C | '' | n.b. | l |
| 109 | 01101101 | 6D | '' | n.b. | m |
| 110 | 01101110 | 6E | '' | n.b. | n |
| 111 | 01101111 | 6F | '' | n.b. | o |
| 112 | 01110000 | 70 | '' | n.b. | p |

| | | | | | |
|---|---|---|---|---|---|
| 113 | 01110001 | 71 | '' | BIN$ | q |
| 114 | 01110010 | 72 | '' | DEC$ | r |
| 115 | 01110011 | 73 | '' | HEX$ | s |
| 116 | 01110100 | 74 | '' | INSTR | t |
| 117 | 01110101 | 75 | '' | LEFT$ | u |
| 118 | 01110110 | 76 | '' | MAX | v |
| 119 | 01110111 | 77 | '' | MIN | w |
| 120 | 01111000 | 78 | '' | POS | x |
| 121 | 01111001 | 79 | '' | RIGHT$ | y |
| 122 | 01111010 | 7A | '' | ROUND | z |
| 123 | 01111011 | 7B | '' | STRING$ | ä/{ |
| 124 | 01111100 | 7C | '' | TEST | ö/\| |
| 125 | 01111101 | 7D | '' | TESTR | ü/} |
| 126 | 01111110 | 7E | '' | 'IMPROPER ARGUMENT' | ß/~ |
| 127 | 01111111 | 7F | '' | VPOS | DEL |
| 128 | 10000000 | 80 | AFTER | n.b. | n.b. |
| 129 | 10000001 | 81 | AUTO | n.b. | n.b. |
| 130 | 10000010 | 82 | BORDER | n.b. | n.b. |
| 131 | 10000011 | 83 | CALL | n.b. | n.b. |
| 132 | 10000100 | 84 | CAT | n.b. | n.b. |
| 133 | 10000101 | 85 | CHAIN | n.b. | n.b. |
| 134 | 10000110 | 86 | CLEAR | n.b. | n.b. |
| 135 | 10000111 | 87 | CLG | n.b. | n.b. |
| 136 | 10001000 | 88 | CLOSEIN | n.b. | n.b. |
| 137 | 10001001 | 89 | CLOSEOUT | n.b. | n.b. |
| 138 | 10001010 | 8A | CLS | n.b. | n.b. |
| 139 | 10001011 | 8B | CONT | n.b. | n.b. |
| 140 | 10001100 | 8C | DATA | n.b. | n.b. |
| 141 | 10001101 | 8D | DEF | n.b. | n.b. |
| 142 | 10001110 | 8E | DEFINT | n.b. | n.b. |
| 143 | 10001111 | 8F | DEFREAL | n.b. | n.b. |
| 144 | 10010000 | 90 | DEFSTR | n.b. | n.b. |
| 145 | 10010001 | 91 | DEG | n.b. | n.b. |
| 146 | 10010010 | 92 | DELETE | n.b. | n.b. |
| 147 | 10010011 | 93 | DIM | n.b. | n.b. |
| 148 | 10010100 | 94 | DRAW | n.b. | n.b. |
| 149 | 10010101 | 95 | DRAWR | n.b. | n.b. |
| 150 | 10010110 | 96 | EDIT | n.b. | n.b. |
| 151 | 10010111 | 97 | ELSE | n.b. | n.b. |
| 152 | 10011000 | 98 | END | n.b. | n.b. |
| 153 | 10011001 | 99 | ENT | n.b. | n.b. |



| | | | | | |
|---|---|---|---|---|---|
| 154 | 10011010 | 9A | ENV | n.b. | n.b. |
| 155 | 10011011 | 9B | ERASE | n.b. | n.b. |
| 156 | 10011100 | 9C | ERROR | n.b. | n.b. |
| 157 | 10011101 | 9D | EVERY | n.b. | n.b. |
| 158 | 10011110 | 9E | FOR | n.b. | n.b. |
| 159 | 10011111 | 9F | GOSUB | n.b. | n.b. |
| 160 | 10100000 | A0 | GOTO | n.b. | n.b. |
| 161 | 10100001 | A1 | IF | n.b. | n.b. |
| 162 | 10100010 | A2 | INK | n.b. | n.b. |
| 163 | 10100011 | A3 | INPUT | n.b. | n.b. |
| 164 | 10100100 | A4 | KEY | n.b. | n.b. |
| 165 | 10100101 | A5 | LET | n.b. | n.b. |
| 166 | 10100110 | A6 | LINE | n.b. | n.b. |
| 167 | 10100111 | A7 | LIST | n.b. | n.b. |
| 168 | 10101000 | A8 | LOAD | n.b. | n.b. |
| 169 | 10101001 | A9 | LOCATE | n.b. | n.b. |
| 170 | 10101010 | AA | MEMORY | n.b. | n.b. |
| 171 | 10101011 | AB | MERGE | n.b. | n.b. |
| 172 | 10101100 | AC | MID$ | n.b. | n.b. |
| 173 | 10101101 | AD | MODE | n.b. | n.b. |
| 174 | 10101110 | AE | MOVE | n.b. | n.b. |
| 175 | 10101111 | AF | MOVER | n.b. | n.b. |
| 176 | 10110000 | B0 | NEXT | n.b. | n.b. |
| 177 | 10110001 | B1 | NEW | n.b. | n.b. |
| 178 | 10110010 | B2 | ON | n.b. | n.b. |
| 179 | 10110011 | B3 | ON BREAK | n.b. | n.b. |
| 180 | 10110100 | B4 | ON ERROR GOTO 0 | n.b. | n.b. |
| 181 | 10110101 | B5 | ON SQ | n.b. | n.b. |
| 182 | 10110110 | B6 | OPENIN | n.b. | n.b. |
| 183 | 10110111 | B7 | OPENOUT | n.b. | n.b. |
| 184 | 10111000 | B8 | ORIGIN | n.b. | n.b. |
| 185 | 10111001 | B9 | OUT | n.b. | n.b. |
| 186 | 10111010 | BA | PAPER | n.b. | n.b. |
| 187 | 10111011 | BB | PEN | n.b. | n.b. |
| 188 | 10111100 | BC | PLOT | n.b. | n.b. |
| 189 | 10111101 | BD | PLOTR | n.b. | n.b. |
| 190 | 10111110 | BE | POKE | n.b. | n.b. |
| 191 | 10111111 | BF | PRINT | n.b. | n.b. |
| 192 | 11000000 | C0 | "'" = REM | n.b. | n.b. |
| 193 | 11000001 | C1 | RAD | n.b. | n.b. |
| 194 | 11000010 | C2 | RANDOMIZE | n.b. | n.b. |

| | | | | | |
|---|---|---|---|---|---|
| 195 | 11000011 | C3 | READ | n.b. | n.b. |
| 196 | 11000100 | C4 | RELEASE | n.b. | n.b. |
| 197 | 11000101 | C5 | REM | n.b. | n.b. |
| 198 | 11000110 | C6 | RENUM | n.b. | n.b. |
| 199 | 11000111 | C7 | RESTORE | n.b. | n.b. |
| 200 | 11001000 | C8 | RESUME | n.b. | n.b. |
| 201 | 11001001 | C9 | RETURN | n.b. | n.b. |
| 202 | 11001010 | CA | RUN | n.b. | n.b. |
| 203 | 11001011 | CB | SAVE | n.b. | n.b. |
| 204 | 11001100 | CC | SOUND | n.b. | n.b. |
| 205 | 11001101 | CD | SPEED | n.b. | n.b. |
| 206 | 11001110 | CE | STOP | n.b. | n.b. |
| 207 | 11001111 | CF | SYMBOL | n.b. | n.b. |
| 208 | 11010000 | D0 | TAG | n.b. | n.b. |
| 209 | 11010001 | D1 | TAGOFF | n.b. | n.b. |
| 210 | 11010010 | D2 | TRON | n.b. | n.b. |
| 211 | 11010011 | D3 | TROFF | n.b. | n.b. |
| 212 | 11010100 | D4 | WAIT | n.b. | n.b. |
| 213 | 11010101 | D5 | WEND | n.b. | n.b. |
| 214 | 11010110 | D6 | WHILE | n.b. | n.b. |
| 215 | 11010111 | D7 | WIDTH | n.b. | n.b. |
| 216 | 11011000 | D8 | WINDOW | n.b. | n.b. |
| 217 | 11011001 | D9 | ZONE | n.b. | n.b. |
| 218 | 11011010 | DA | WRITE | n.b. | n.b. |
| 219 | 11011011 | DB | DI | n.b. | n.b. |
| 220 | 11011100 | DC | EI | n.b. | n.b. |
| 221 | 11011101 | DD | "SYNTAX ERROR" | n.b. | n.b. |
| 222 | 11011110 | DE | "SYNTAX ERROR" | n.b. | n.b. |
| 223 | 11011111 | DF | "SYNTAX ERROR" | n.b. | n.b. |
| 224 | 11100000 | E0 | "SYNTAX ERROR" | n.b. | n.b. |
| 225 | 11100001 | E1 | "SYNTAX ERROR" | n.b. | n.b. |
| 226 | 11100010 | E2 | "SYNTAX ERROR" | n.b. | n.b. |
| 227 | 11100011 | E3 | ERL | n.b. | n.b. |
| 228 | 11100100 | E4 | FN | n.b. | n.b. |
| 229 | 11100101 | E5 | SPC | n.b. | n.b. |
| 230 | 11100110 | E6 | STEP | n.b. | n.b. |



| | | | | | |
|---|---|---|---|---|---|
| 231 | 11100111 | E7 | SWAP | n.b. | n.b. |
| 232 | 11101000 | E8 | "SYNTAX ERROR" | n.b. | n.b. |
| 233 | 11101001 | E9 | "SYNTAX ERROR" | n.b. | n.b. |
| 234 | 11101010 | EA | TAB | n.b. | n.b. |
| 235 | 11101011 | EB | THEN | n.b. | n.b. |
| 236 | 11101100 | EC | TO | n.b. | n.b. |
| 237 | 11101101 | ED | USING | n.b. | n.b. |
| 238 | 11101110 | EE | > | n.b. | n.b. |
| 239 | 11101111 | EF | = | n.b. | n.b. |
| 240 | 11110000 | F0 | >= | n.b. | n.b. |
| 241 | 11110001 | F1 | < | n.b. | n.b. |
| 242 | 11110010 | F2 | <> | n.b. | n.b. |
| 243 | 11110011 | F3 | <= | n.b. | n.b. |
| 244 | 11110100 | F4 | + | n.b. | n.b. |
| 245 | 11110101 | F5 | - | n.b. | n.b. |
| 246 | 11110110 | F6 | * | n.b. | n.b. |
| 247 | 11110111 | F7 | / | n.b. | n.b. |
| 248 | 11111000 | F8 | ^ | n.b. | n.b. |
| 249 | 11111001 | F9 | 'Backslash' | n.b. | n.b. |
| 250 | 11111010 | FA | AND | n.b. | n.b. |
| 251 | 11111011 | FB | MOD | n.b. | n.b. |
| 252 | 11111100 | FC | OR | n.b. | n.b. |
| 253 | 11111101 | FD | XOR | n.b. | n.b. |
| 254 | 11111110 | FE | NOT | n.b. | n.b. |
| 255 | 11111111 | FF | Funktion | n.b. | n.b. |

# Index

Absolute Adressierung 140
Absoluter Sprung 191
ADC 167
ADD 167
Adressen 49
Adressierung 140
Adreßleitungen 83
Adreßregister 114
Adreßwort 16
Akkumulator 120, 187
ALU 120
AND 169
Anwenderspeicher 51
Arithmetikbefehle 166
Arithmetik-Logik-Einheit 120
ARRAY 36
Arrays 41, 43, 88
Arrays abspeichern 43, 46
ASCII-Code 25
ASCII-Werte 29
ASC() 28
Austauschvorgänge 147, 151
AY 8912 (Soundgenerator) 100, 107

BASIC-Schlüsselwörter 28
Baudrate 249
BCD-Operationen 187
Befehlsabfrage 70
Befehlscodes 32, 38, 138
Befehlsdecodierung 208
Benutzerspeicher 53
Betriebssystemkern 232
Betriebssystem-Stack 229
Bildschirmbasis 89
Bildschirmmodus 89
Bildschirmoffset 89
Bildschirmpaket 232
Binäre Files speichern 73
Binäres File laden 74
Binärsystem 18
BIN$ 18
Bit 15
BIT 185
Bitbefehle 167
Bitmanipulation 185, 186
Block 16
Blockausgabebefehle 163
Blockladebefehle 157
Blockschaltbild 80
Byte 16

CALL 57
Carry-Bit 166
Carry-Flag 122
CAS 253
Central Processing Unit 83
Centronixport 100
Codes 24
Code-Tabellen 32
Compare 166, 172
CP 166, 172
CPC-MON (Programm 2) 218
CPU 83
CPU-Register 160
CRTC 88
C-Flag 122

Datagenerator (Programm) 246
Datenaustausch 155
Datenbus 84
Datenformate 15
Datentransfer 146, 160
Datentransferbefehle 138
Datenübertragungsraten 248
DEC 167
Dezimalsystem 16
DI 203
DIM 41
DISABLE 58



Disassembler 142, 207
Disassembler (Programm) 213
Displacement 141
Doppelbyte 16
Druckersteuerung 257
Dualsystem 17

EI 203
Ein-/Ausgabe 103
ENABLE 58
ERROR-Handling 72
Erweiterungsbyte 137
Erweiterungscode 137
Erweiterungstabelle 144
Erweiterungszeichen 235
Exklusives ODER 170
Expansionstrings 235
Externe Ladebefehle 154
Externer Datentransfer 147, 153
E/A-Anfrage 84

Farbbyte 99
FAST TICKER 235
Firmware 50, 227
Firmware-Speicher 51, 228
Firmware-Sprungtabelle 230
Flags 120, 187, 196
Fließkommazahlen 39

Gate Array 88
Gleitkommawert 22
GOMACH (Programm) 125, 130
GRA 231, 262
Grafikroutinen 259
Grafik-VDU 231

Halbübertrags-Flag 121
HALT 205
Handshaking 103
Hardware 79
Hauptplatine 80
Hexadezimalsystem 17
HEX$ 18
HIMEM 52, 61
H-Flag 121

IM 204
Implizite Adressierung 141
IN 161
INC 167
Indirections 230
Indirekte Adressierung 140
Indirekter Sprung 192
Indizierte Adressierung 143
Informationseinheit 15
Initialisierung 70
INT 203
Integer-Variable 22, 35
Intel 8080 119
Intergrationsfähigkeit 208
Interner Datentransfer 146, 147
Interpretation 70
Interrupt 124, 203
INTERRUPT REQUEST 84
IO REQUEST 84
IORQ 84
IO-Bereich 100
IRQ 84
IX (Indexregister) 117
IY (Indexregister) 117
I-Register 124
I/O-Bausteine 83

Jumpblock 50

Kassettenspeicherung 243
Kassettenverwaltung 232
Kernel Jumpblock 50, 230
KL 232
KM 231
Komplementieren 189
Konvertieren zwischen Zahlensystemen 18-24

Ladevorgänge 147, 148
LEFT$ 20
Lesen von externen Bausteinen 84
Logikbefehle 166, 169
LOW JUMP RESTART 200
LOWJUMP 50

Mantisse 39
Maschinenpaket 233
Maschinenprogramme 56, 83
MC 233, 258
MC SOUND REGISTER 109
MEMORY 52, 87
MEMORY FULL 72
MEMORY REQUEST 84
MID$ 20
Mnemonics 142, 207
Modes 99
MODE-Register 89
Monitor 66
Monitor-Programm CPC MON 75
MREQ 84

Nicht maskierbarer Interrupt 203
NMI 203
NOP 204
Nullseitenadressierung 141
Numerische Arrays 45
N-Flag 122

ODER-Verknüpfung 171
OFFSET-Zeiger 90
OP CODE 138
Operation Code 138
OR 171
OUT 93, 161

Paritätsbit 25
Paritäts-Flag 122
PC (Programm Counter) 114
PEEK 19
PIO 8255 100
Pointer 50
POP 117
PORT 105
Printer-Port 100
Programm Counter 114
Programm CPC MON 75
Programm CPC-MON 2 218
Programm Datagenerator 246
Programm Disassembler 213
Programm GOMACH 125, 130
Programm ROMREAD 63, 65
Programm Sound-Register 111
Programm Transformer 40
Programmzähler 114
PROGSTART 36
Prozessor 83
PSG 100, 107
PUSH 116
Put-Back-Puffer 235
P/V-Flag 122

RAM 49, 58, 83, 87
RD 84
READ 84
Realvariable 35
REFRESH-Register 124
Register 113
Relative Adressierung 141
Relativer Sprung 190
RES 185
Restart 53
Restart-Kommandos 194
RET 57
RETURN 116
RIGHT$ 19
RL 178
RLC 178
ROM 49, 58, 83, 87
ROMCHANGE 72
ROMREAD (Programm) 63, 65
Rotation 181
Rotieren 178



RR 179
RRC 178
RST 53, 194

Satznachspann 250
Satzvorspann 250
SAVE 73
SBC 167
Schreiben an externe Bausteine 84
SCR 232
SCR SET BASE 90
SCR SET OFFSET 90
Scrollen 98
SET 185
SLA 179
SOUND 232
Soundgenerator 100, 106
Sound-Register (Programm) 111
SP 115
Speicheranalyse-Programm 55
Speicheranfrage 84
Speicheraufbau 49
Spezialregister 124
Sprungbefehle 139, 190
Sprungbefehle (Tabelle) 195
Sprungverteiler 210
SRA 179
SRL 179
Stack 151
Stack Pointer 115
Stapelzeiger 115
Steuerbefehle 139
Steuerbus 84
STRING 40
Stringarrays 42
Strings 35
STROBE OUTPUT 100
SUB 167
Subtraktions-Flag 122
SYMBOL AFTER 52
Systemsteuerbefehle 203
Systemtakt 89
S-Flag 121

Tastaturabfrage 236
Tastatureingabepuffer 235
Tastatur-Übersetzungstabellen 237
Textanalyse-Programm 56
Texteingabe 73
Text-VDU 231
TICKER 235
Token 28, 208
Tongeneratorverwaltung 232
Transformer (Programm) 40
TXT 231

Übersetzungstabellen 237
ULA 87
Uncommitted Logic Array 88
UND-Verknüpfung 169
Universalregister 119
Unmittelbare Adressierung 139
Unterbrechungsanfrage 84

Variable 35
VARSTART 30
Verschiebebefehle 166, 178, 182
Verzweigungsbefehle 138
Videocontroller (6845) 88
Vorspannfelder (Tabelle) 252
Vorzeichenflag 121

WAIT 85
Warte-Befehl 85
WINDOW 95
WR 84
WRITE 84

XOR 170

Z80A-Prozessor 83, 113
Zahlensysteme 16
ZERO FLAG 121
Z-Flag 121

16-Bit-Arithmetikbefehle 168
16-Bit-Register 113
16-Bit-Registerladebefehle 151
8-Bit-Arithmetik-Befehle 168, 175
8-Bit-Logikbefehle 175
8-Bit-Register 113
8-Bit-Registerladebefehle 149
@ (Funktion) 61