DULLIN · STRASSENBURG

DAS MASCHINENSPRACHEBUCH ZUM CPC

FÜR DEN CPC 464, 664 & 6128

EIN DATA BECKER BUCH

DULLIN · STRASSENBURG

DAS MASCHINENSPRACHEBUCH ZUM CPC

FÜR DEN CPC 464, 664 & 6128

EIN DATA BECKER BUCH

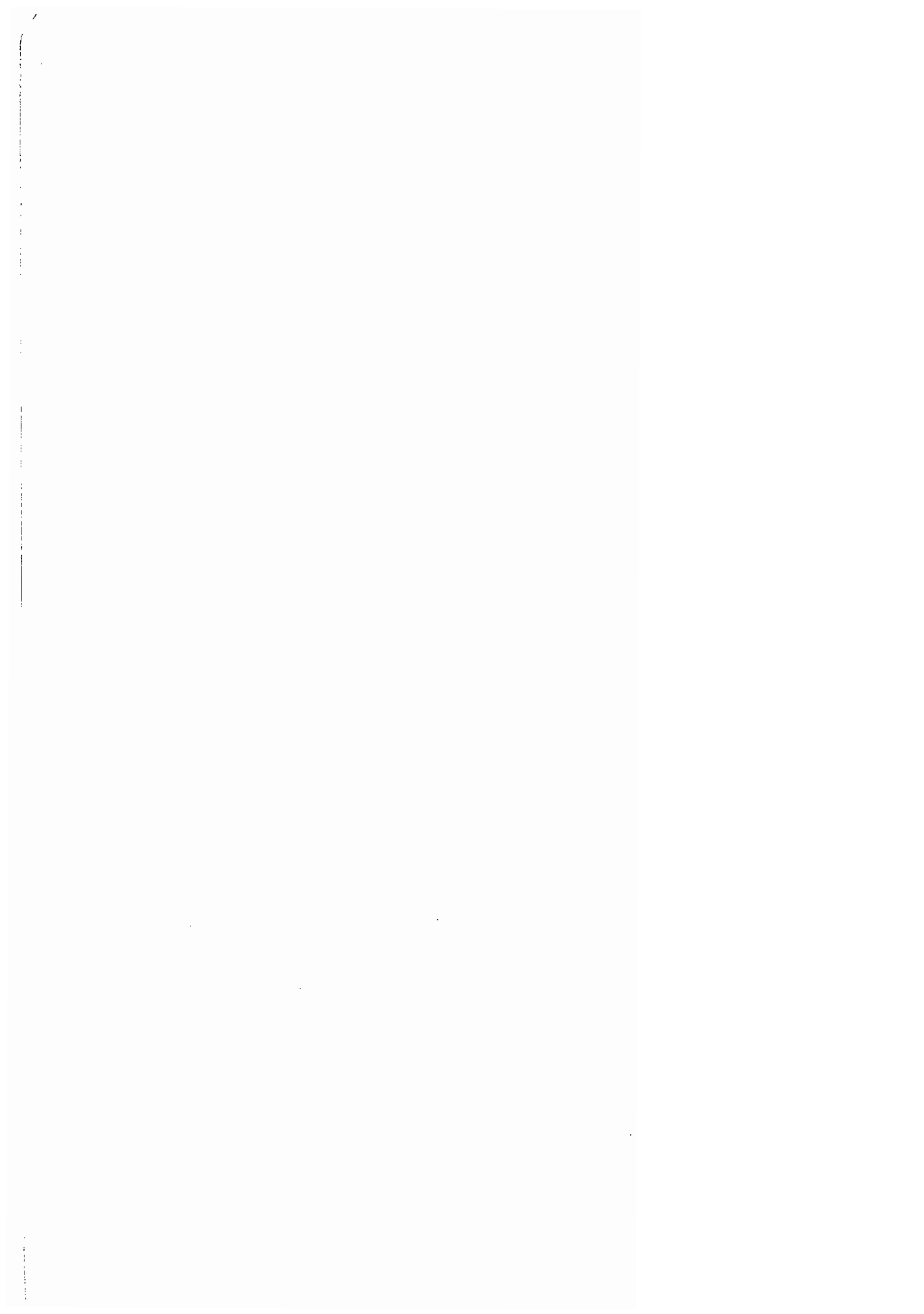
ISBN 3-89011-070-3

Copyright © 1985 DATA BECKER GmbH
Merowingerstraße 30
4000 Düsseldorf

Alle Rechte vorbehalten. Kein Teil dieses Programms darf in irgendeiner Form (Druck, Fotokopie oder einem anderen Verfahren) ohne schriftliche Genehmigung der DATA BECKER GmbH reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.*

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

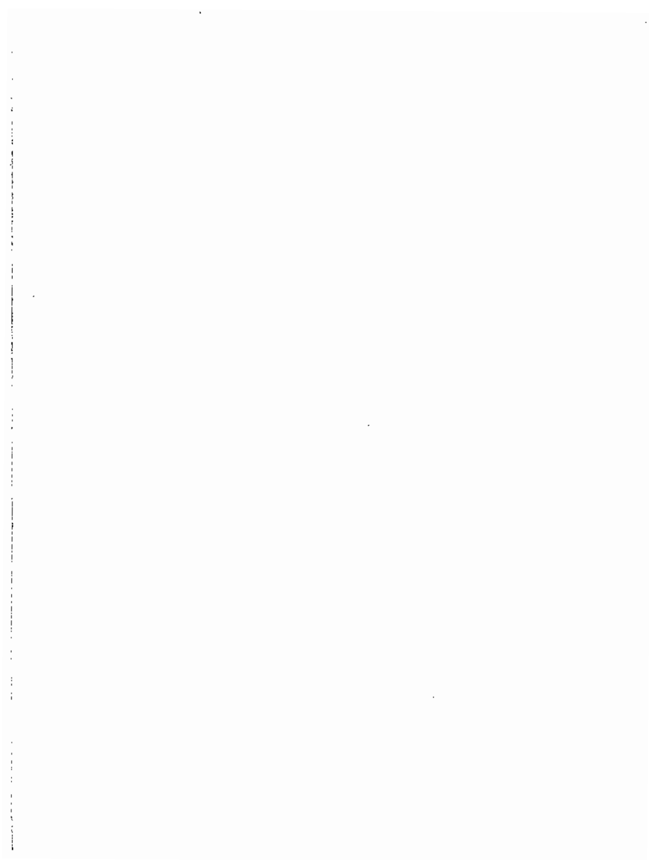
Alle Schaltungen, technischen Angaben und Programme in diesem Buch wurden von dem Autoren mit größter Sorgfalt erarbeitet bzw. zusammengestellt und unter Einschaltung wirksamer Kontrollmaßnahmen reproduziert. Trotzdem sind Fehler nicht ganz auszuschließen. DATA BECKER sieht sich deshalb gezwungen, darauf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



DANKWORT

Das Schreiben eines solchen Buches macht zugegebenermaßen meistens nur den Autoren Spaß. Für Mitbewohner, Freundinnen und Freunde kann aber leider "Frust" auftreten, da wir die Autoren, in einem sog. "Bücherarbeitsrausch und Computerwahn" stecken müssen, um ein Buch zu vollenden. Wir möchten uns bei all denen bedanken, die durch ihre Geduld und Unterstützung dieses Buch möglich gemacht haben. Insbesondere danken wir Birgitt Mikutta, Kristin Grünewald, Andreas Bethmann für die unschätzbare Hilfe bei der Korrektur des Manuskriptes und Ralph Dullin für das Anfertigen der Zeichnungen und einiger Tabellen.

Weiterhin möchten wir unseren Dank an die Firma Zilog Inc, USA richten, die die Befehlslisten zum Z80A zur Verfügung gestellt hat.



VORWORT zur 2ten AUFLAGE

Die zweite Auflage des Maschinensprache-Buches behandelt die Programmierung in Maschinensprache für drei Schneider-Computer: Den CPC 464, CPC 664 und das neueste der Firma, den CPC 6128. Da diese Rechner im großen und ganzen kompatibel sind, ist die Zusammenfassung in einem Buch möglich.

Wir haben in der überarbeiteten Auflage versucht, Verbesserungsvorschläge unserer Leser zu berücksichtigen. Auch ist es dank der Firma Zilog möglich geworden, die umständlichen und für unsere Begriffe etwas unübersichtlichen Befehlstabellen durch die Standardtabellen der Firma zu ersetzen. Dadurch haben wir Platz für Systemroutinen, Listings wie z.B. den Simulator und für zusätzliche Informationen geschaffen.

Die Programmierung in Maschinensprache bringt einige entscheidende Vorteile in bezug auf Geschwindigkeit und Speicherbedarf gegenüber BASIC mit sich. Ziel dieses Buches ist es, dem CPC-Benutzer den Einstieg in die Maschinensprache zu ermöglichen und ihm dadurch die oben genannten Vorzüge für seine Programme nutzbar zu machen.

Doch ist das Erlernen der Maschinensprache gar nicht so einfach, denn wer kann schon mit folgendem etwas anfangen:

21,00,C0,36,CC,23,BC,20,FA,C9

Aber legen Sie das Buch nicht gleich wieder aus der Hand. Ihnen wird das Erlernen der Maschinensprache leicht fallen, wenn Sie das Buch folgendermaßen handhaben:

- Arbeiten Sie das Buch Kapitel für Kapitel durch
- Versuchen Sie, die Aufgaben zu lösen
- Fällt Ihnen die Lösung der Aufgaben schwer, arbeiten Sie das Kapitel ruhig noch einmal durch.

Doch damit genug der guten Ratschläge; stürzen Sie sich hinein in das Abenteuer MASCHINENSPRACHE.

Die Autoren

INHALTSVERZEICHNIS

Dankwort
Vorwort
Inhaltsverzeichnis

KAPITEL I: EINFÜHRUNG

1.1 Was ist Maschinensprache 11
1.2 Das erste Maschinenprogramm 16
1.3 Zahlensysteme 19
Das Dezimalsystem 21
Das Dualsystem 22
Bit und Byte 23
Das Hexadezimalsystem 26
Aufgaben und Lösungen 30
1.4 Rechneraufbau 31

KAPITEL II: DER Z80 PROZESSOR

2.1 Aufbau der CPU 35
2.2 Der Akkumulator 38
2.3 Die Flags 38
2.4 Die "verknüpfbaren Sechs" 8-Bit Register 39
2.5 Die "unzertrennlichen Vier" 16-Bit Register 40
2.6 Interrupt-/ Refresh-Register 41

KAPITEL III: DER BEFEHLSSATZ DES Z80

3.1 Einleitung:Eingabe von Maschinenprogrammen 43
3.2 Transfer von Daten 45
3.3 Bearbeitung von Daten und Tests 46
3.4 Sprünge 47
3.5 Steuerbefehle 48
3.6 Ein/Ausgabebefehle 48

KAPITEL IV: DIE BEFEHLE

4.1 8-Bit Transferbefehle 49
Unmittelbare Adressierung................50
Implizite- und Registeradressierung................51
Absolute Adressierung................53
Indizierte Adressierung................53
Indirekte Adressierung................55
Befehlsliste................57
Anwendung (z.B. Aufgaben, Beispiele, Programme usw.).....58
4.2 16-Bit Transferbefehle 60
Unmittelbare Adressierung................61
Implizite Adressierung................61
Absolute Adressierung................62
Befehlsliste................64
Anwendung................65
4.3 Stapelbefehle 68
4.4 Austauschbefehle 73
Befehlsliste................75
4.5 Blocktransfer- und Blocksuchbefehle 76
Blocksuchbefehle................79
Anwendung................80
4.6 Arithmetische Befehle 84
Addition (Anwendung)................85
Subtraktion (Anwendung)................88
Was ist das Zweierkomplement? 89
8-Bit Arithmetische und Zählbefehle................94
Flagbeeinflussung................95

Befehlsliste (8-Bit) 102
16-Bit Arithmetische- und Zählbefehle 103
Befehlsliste (16-Bit) 105
Anwendung 106
4.7 Logische Befehle und CP (Compare) 107
Anwendung 112
Der Vergleichsbefehl CP (Flagbeeinflussung) 113
Anwendung 116
4.8 Rotations- und Schiebe Befehle 118
Befehlsliste 125
Anwendung 126
4.9 Bit-Manipulationsbefehle 133
Die Spezialbefehle SCF und CCF 135
Befehlsliste 137
Anwendung 138
4.10 Sprünge 139
JUMP/JP 143
CALL/RET 144
RESTART/RST 146
JUMP RELATIV/JR 146
Befehlsliste 149
Anwendung 151
4.11 Steuerbefehle 153
Befehlsliste 155
4.12 Ein- Ausgabebefehle 156
Befehlsliste 159

KAPITEL V: PROGRAMMIERUNG DES Z80

5.1 Der Assembler 161
Listing 168
Programmbeschreibung 182
Variablenliste 190

5.2 Programmierung 194
Monitorroutine BASIC 199
Fillroutine 202
Transferroutine 207
Compareroutine 210

KAPITEL VI: BENUTZUNG VON SYSTEMROUTINEN

6.1 Disassembler und Einzelschrittsimulator 217
Programmbenutzung/Dissasembler u.a 217
Programmbenutzung/Einzelschrittsimulator 221
Assemblerlisting Simulator 228
Listing 231
Querverweis(XREF)-Tabelle der Variablen 244
Programmbeschreibung 247
Variablenliste und Tabellen 259
6.2 Systemroutinen 261
Einführung 261
Der Monitor 265
Der Breakpoint 278
Suchroutine 282
Eingabe von Daten 284
Befehlserweiterung mit RSX 287
Assemblerlisting (DOKE) 292
Assemblerlisting (RPEEK) 295
Assemblerlisting (LINER) 299
Listing (BASIC) (Baslamak) 302

KAPITEL VII: PERSPEKTIVEN

7.1 Perspektiven 305

ANHANG

1. Systemroutinen 308
2. Umrechnungstabelle dez,hex,bi 311
3. Tabellen 316
4. Erklärung zu den Befehlstabellen 323
5. Befehlstabellen 324
6. Flagbeeinflussungstabelle 333

KAPITEL I: EINFÜHRUNG

1.1 Was ist Maschinensprache?

Maschinensprache ist die Programmiersprache, die der Computer direkt verarbeiten kann. Was ist darunter zu verstehen?

Wie Sie sicher wissen, besitzt jeder Computer einen Mikroprozessor, den man als das "Gehirn" des Rechners bezeichnen kann. Diesen IC (Integrierter Schaltkreis) nennt man CPU (Central Prozessing Unit) oder Zentraleinheit. Die CPU führt Maschinenbefehle aus, steuert den Ablauf im Rechner und die extern angeschlossenen Geräte (Peripherie). Die Zentraleinheit ist der wichtigste Baustein in einem Computer. Wenn wir in Maschinensprache programmieren, benutzen wir Befehle, die die CPU direkt ansprechen und die sie sofort ausführen kann. Damit ist die Maschinensprache vom jeweiligen Prozessortyp abhängig.

Die Schneider CPC's besitzen einen Z80A Prozessor, der auch in vielen anderen Mikrocomputern Verwendung findet. Der Z80A ist eine sehr leistungsfähige Zentraleinheit, welche über 600 Befehle versteht, die bei den CPC's mit sehr hoher Geschwindigkeit verarbeitet werden.

Warum eigentlich Maschinensprache?

Die meisten Homecomputer sind mit BASIC ausgerüstet. Wie Sie sicher gemerkt haben, ist diese Sprache nicht schwer zu erlernen. Besonders das Schneider BASIC fällt durch seine Vielzahl von Befehlen auf. Es entsteht der Eindruck, daß mit diesem BASIC keine Wünsche offen bleiben und alle Programmierprobleme damit gut gelöst werden können.

Um zu verstehen, wo die Vorteile der Maschinensprache liegen, müssen wir erst einmal wissen, wie der Rechner BASIC verarbeitet.

Stellen Sie sich vor:

Außenminister S.Basic verhandelt mit seinem Amtskollegen Mr.CPU im Maschinenspracheland. Leider sind seine Kenntnisse dieser Sprache sehr gering, so daß er auf die Hilfe der Dolmetscherin Frau Interpreter angewiesen ist, die seine Sätze in Maschinensprache übersetzt. Wie Sie sich sicher vorstellen können, ist Frau Interpreter, obwohl eine hervorragende Dolmetscherin, immer ein wenig später fertig, als der Politiker spricht. Dadurch wird diese Verhandlung unnötig verlängert.

Genau dasselbe Problem finden wir bei der Programmierung in BASIC vor. Der Computer muß zuerst das vom Programmierer geschriebene BASIC Programm durch den Interpreter interpretieren. Der BASIC Interpreter ist ein Teil des Betriebssystems. Er interpretiert das Programm Befehl für Befehl. Dann bewirkt er die sofortige Ausführung. Genauer: Der Interpreter erkennt den BASIC Befehl und löst dann die Ausführung des BASIC Befehls durch den Aufruf der zu dem jeweiligen Befehl gehörenden Maschinenroutine aus.

Betrachten wir ein Beispiel:

MODE 2

Der Interpreter liest diesen Befehl Zeichen für Zeichen, wobei z.B. Space (Leerzeichen), Doppelpunkte, Klammern und Kommata ihm sagen, daß ein Wort beendet ist. Dieses Wort (>MODE<) vergleicht er mit den Eintragungen in der BASIC Befehlstabelle im Betriebssystem. Findet er es nicht, so wird versucht, das Wort als Variable zu interpretieren. Funktioniert auch dies nicht, wird eine Fehlermeldung ausgegeben.

Findet der Interpreter das Wort, so verzweigt er an die dem Wort zugeordnete Sprungadresse. Dort wird der nachfolgende Wert (bei unserem Beispiel 2) eingelesen, die Zulässigkeit dieses Argumentes überprüft und der Befehl ausgeführt. Dann wird zurück in den Interpreter gesprungen: Der oben beschriebene

Vorgang beginnt von Neuem. Die Aufgabe, die in unserem Beispiel Frau Interpreter übernommen hat, benötigt natürlich einige Zeit. Diese Zeit wird gespart, wenn wir direkt in Maschinensprache programmieren.

Leider hat die Maschinensprache den Nachteil sehr abstrakt zu sein. Der Mensch hat grundsätzlich einige Schwierigkeiten, sich Zahlen vorzustellen. Diese Unanschaulichkeit ist auch der Grund für die Entwicklung sogenannter "Höherer Programmiersprachen", wie Logo, BASIC, usw., die mit Begriffen und nicht mit Zahlen operieren. Diese Sprachen stellen einen Kompromiß in der Kommunikation zwischen Mensch und Maschine dar. Leider sind damit erhebliche Nachteile in Bezug auf Geschwindigkeit, Speicherplatzbedarf und oft auch auf Programmiermöglichkeiten verbunden.

Alle höheren Programmiersprachen wie auch Cobol, Pascal, Fortran etc. müssen übersetzt werden, bevor der Rechner sie ausführen kann. Man unterscheidet hierbei zwischen Interpreter und Compiler:

Ein Interpreter, wie z.B. der der CPC's, übersetzt während jeder Ausführung des Programms schrittweise alle Befehle und führt sie gleich aus. Der Interpreter ist gemäß unserem Beispiel also ein Simultanübersetzer, d.h. beim Programmablauf wird jeder Befehl immer wieder neu interpretiert. Daher ist das Ändern eines BASIC Programms so unproblematisch.

Im Gegensatz dazu übersetzt ein Compiler das jeweilige Programm nur einmal und erzeugt dabei ein äquivalentes in Maschinensprache. Dann erst kann das erzeugte Maschinenprogramm ausgeführt werden. Der Vorgang des Compilierens dauert normalerweise recht lang, dafür läuft der dann erzeugte Maschinencode auch viel schneller. Wird das Programm geändert, so muß die neue Version erst wieder compiliert werden. Dadurch ist das Ändern solcher Programme langwierig. In diesem Buch stellen wir Ihnen einen Compiler vor, der von Assemblersprache in Maschinencode übersetzt. Einen solchen Compiler nennt man ASSEMBLER.

Hier erkennen Sie schon einen grundsätzlichen Vorteil der Maschinensprache: Maschinenprogramme erreichen bis zu 1000 mal höhere Ausführungsgeschwindigkeiten als BASIC Programme. Auch gegenüber den von Compilern erstellten Maschinenprogrammen sind die von Hand für ein spezielles Problem geschriebenen Maschinenprogramme schneller. Der >RETURN< Befehl in BASIC hat eine Ausführungszeit von ca. 0.6 Millisekunden, der entsprechende Befehl in Maschinensprache RET dauert jedoch nur 2.5 Mikrosekunden. Damit ist die Maschinensprache beim RET Befehl ca. 240 mal, bei dem Äquivalent zum >POKE< Befehl in Maschinensprache sogar knapp 1000 mal schneller. Wichtig sind diese Unterschiede z.B. beim Sortieren und Durchsuchen von großen Datenmengen, für das Verschieben von Speicherinhalten, wie es für das Scrolling oder auch für Textprogramme notwendig ist. Weiterhin ist die Programmierung von hochauflösender Grafik in BASIC zu langsam, d.h. für Spiele oder Businessgrafik ist die Maschinensprache unerläßlich.

Außerdem gibt es noch andere Vorteile.

In der Regel sind Maschinenprogramme kürzer als BASIC Programme, wodurch wichtiger Speicherplatz eingespart wird. Sobald Sie Ihre ersten Maschinenprogramme geschrieben haben, werden Sie feststellen, daß ein Maschinenprogramm von über 500 Bytes schon sehr lang ist und damit eine Menge gemacht werden kann. Dagegen würde man für ein BASIC Programm mit ähnlichen Fähigkeiten viel mehr Speicherplatz verbrauchen.

Anmerkung: Die Länge eines BASIC Programms in Bytes kann bei den CPC's mit >PRINT HIMEM-FRE(0)-370< berechnet werden.
Ein weiterer Vorteil der Maschinensprache liegt darin, daß nur mit ihr die Möglichkeiten eines Rechners vollständig ausgeschöpft werden können. Mit Maschinensprache ist man erst in der Lage, z.B. Ein- bzw. Ausgabebausteine zu programmieren. Man kann also mit Hilfe eigener Programme Ein- bzw. Ausgabegeräte bedienen oder von ihnen Daten empfangen.

Auch die Entwicklung eigener Datenstrukturen, die oft sehr viel platzsparender sind als die vom BASIC vorgegebenen, ist nur in

Maschinensprache möglich. Große Datenmengen, wie sie u.a. in der Textverarbeitung auftreten, können damit besser in dem zur Verfügung stehenden Speicherplatz untergebracht werden.

Diese Beispiele sollten genügen, um die Notwendigkeit der Maschinensprache, auch bei Rechnern mit sehr gutem BASIC, wie den CPC's darzustellen. Allerdings muß auch gesagt werden, daß die Programmierung in Maschinensprache einen großen Nachteil hat.

Maschinensprache ist die Sprache der CPU des Computers und damit die am weitesten maschinenorientierte Sprache. Eine starke Maschinenorientierung hat aber für den Programmierer zur Folge, daß er, um diese Sprache zu verstehen, sehr abstrakt denken muß. Der Mensch denkt vorrangig in Worten und Assoziationen, d.h. eine problem- bzw. menschorientierte Sprache verwendet anschauliche Begriffe und Strukturen. Dies ist bei der Maschinensprache nicht der Fall. Prinzipiell versteht die CPU nur Zahlen, d.h. ein Maschinenprogramm ist einfach eine Reihe von Zahlen und nicht eine Folge von Begriffen. In dieser Form wäre die Programmierung in Maschinensprache bei umfangreichen Programmen beinahe ein Ding der Unmöglichkeit. Deshalb wurde schon von den "Pionieren der Computerei" eine Art Zwischensprache entwickelt, die Maschinenprogramme anschaulicher und verständlicher macht. Diese Sprache nennt man Assembler. Die Assemblersprache ordnet jedem Maschinencode (also einer Zahl) eine Reihe von Symbolen zu. Diese Symbole bestehen aus:

1. Befehlswort, d.h. meist einer Abkürzung des englischen Wortes für den Befehl, auch Mnemonic genannt.

2. Operandem, der z.B. Adressen, Konstanten o.ä. (das Befehlswort betreffend) angibt.

Damit vereinfacht sich das Erstellen eines Maschinenprogramms auf das Schreiben in Assemblersprache. Diese Assemblersprache wird dann von einem sogenannten Assemblerprogramm automatisch in den Maschinencode übersetzt. Einen solchen Assembler (einen Compiler für Assemblersprache) stellen wir

Ihnen in diesem Buch vor, und werden ihn benutzen, um in Assembler (jetzt ist die Assemblersprache gemeint) zu programmieren. Aus diesem Grund werden wir nur kurz und beispielhaft in der wirklichen Maschinensprache, in Form von Zahlen programmieren, dann aber zur Programmierung in Assembler übergehen, und die Arbeit des Übersetzens dem Assembler (Compiler) überlassen.

Nun geht es aber richtig los!!!

1.2 Das erste Maschinenprogramm

Um Ihnen zu zeigen, daß sich das Erlernen der Maschinensprache lohnt, folgt ein Vergleich zwischen einem BASIC- und Ihrem ersten MASCHINENPROGRAMM:

Bitte geben Sie folgende BASIC Zeilen ein:

```
10 HL=&C000
20 POKE HL,&CC
30 HL=HL+1
40 IF HL<=&FFFF THEN 20
50 RETURN
```

Geben Sie jetzt im Direktmodus >MODE 2< und anschließend >GOSUB 10< ein und schauen Sie sich an was geschieht!

Das nächste Programm lädt das Maschinenprogramm mit der gleichen Aufgabe, wie das BASIC Programm:

```
10 MEMORY &9FFF
20 FOR I=&A000 TO &A009
30 READ a
40 POKE i,a
50 NEXT I
60 END
70 DATA &21,&00,&C0,&36,&CC,&23,&BC,&20,&FA,&C9
```

Nun geben Sie wieder im Direktmodus >MODE 2< ein, laden das Maschinenprogramm mit >RUN<, rufen dann das so geladene Maschinenprogramm mit >CALL &A000< auf und wundern sich!

Wie Sie gesehen haben, läuft das:

- BASIC Programm : ca.1 Minute
- Maschinenprogramm : ca.1/10 Sekunden
  Man kann die Ablaufzeit theoretisch berechnen.
  Sie beträgt bei dem Beispielprogramm 0.1106 Sekunden.

Die Länge beträgt für das:

```
- BASIC Programm          : 88 Bytes
- Maschinenprogramm       : 10 Bytes
  nämlich von &A000 bis &A009.
```

Wir hoffen, daß Sie nicht zu sehr von der Vielzahl der Neuigkeiten "geschockt" sind. In den folgenden Kapiteln werden wir alles ausführlich erklären.

Zur Analogie der Programme:

```
   BASIC                      Assemblersprache

10 HL=&C000               -   LD HL,C000
20 POKE (HL),&CC          -   LD (HL),&CC
30 HL=HL+1                -   INC HL
                          -   CP H
40 IF HL<=&FFFF THEN 20   -   JR NZ,$-6>A006
50 RETURN                 -   RET
```

ERKLÄRUNG:

Zeile 10: Hier wird der Wert für die VARIABLE HL bzw.dasREGISTER HL auf den Anfang des Bildschirmspeichers gesetzt. (LD=engl.load=lade)

Zeile 20: In dieser Zeile wird an der Adresse HL der Wert &CC gespeichert. Da der Bildschirmspeicher von &C000 bis &FFFF liegt, bewirkt dieser Befehl eine Veränderung der Bildschirmanzeige.

Probieren Sie doch einfach einmal unterschiedliche Werte im Direktmodus für die Adresse HL im Bildschirmspeicher (HL darf zwischen &C000 und &FFFF liegen !!) und für das Argument (in unserem Programm &CC) Werte zwischen &00 und &FF einzusetzen (z.B.: POKE &C100,&AA).

Zeile 30: Erhöht die Variable HL bzw. das Register HL um 1. (INC=engl. increase= erhöhe)

Zeile 40: Abfrage ob HL größer als &FFFF ist, also ob das Ende des Bildschirmbereichs erreicht ist. Diese Abfrage muß in Maschinensprache in zwei Befehle aufgeteilt werden: CP (engl.compare=vergleiche); JR (jump relativ= relativer Sprung); NZ (engl. non zero= nicht Null). Man kann also sagen: "Springe, wenn nicht Null (NZ)." Diese Darstellung ist so nicht ganz richtig. Eine exakte Erklärung erfolgt später.

Im Folgenden zeigen wir das Assemblerlisting, um Ihnen ein Beispiel zu geben:

ASSEMBLERLISTING zum Maschinenprogramm

```
Adresse Code    BASIC-Nr.  Assemblerbefehl  Kommentar


A000 2100C0     10         LD     HL,C000 ;Start Bildschirmsp
eicher
A003 36CC       20         LD     (HL),&CC ;&CC ist der Wert,
der in den Bildschirmspeicher geschrieben wird
A005 23         30         INC    HL ;HL=HL+1
A006 BC         40         CP     H ;Vergleich mit 0
A007 20FA       50         JR     NZ,$-6>A003 ;Wenn nicht 0 (
NZ=Non-Zero ), dann 6 Programmschritte zurück, wenn 0,nächst
er Befehl
A009 C9         60         RET    ;Return zu Basic
```

Wir hoffen, daß wir Ihre Neugierde erwecken konnten, da wir jetzt zur systematischen Erarbeitung der Maschinesprache übergehen, und die Beispiele, die wir oben gegeben haben, genau erklären werden.

1.3 Zahlensysteme

Im vorhergehenden Kapitel wurde das &-Zeichen als Kennzeichen für eine Zahl im Hexadezimalsystem (Hexadezimal - 16) benutzt. Was hat es damit auf sich?

Bei der Realisierung elektronischer Rechenanlagen gab es zwei Möglichkeiten der Zahlendarstellung.

> Analog: Bei einem Analogrechner wird eine Zahl durch eine entsprechend hohe Spannung dargestellt, z.B. 1=1 Volt und 100=100 Volt. Eine Armbanduhr mit Zeigern ist demnach eine Analoguhr. Die kontinuierliche Zunahme der Zeit entspricht, ist analog zu, der Zahl der Umdrehungen der Zeiger.

Digital: Bei Digitalcomputern liegt die Idee zugrunde, nicht das Maß der Spannung, sondern nur die beiden Zustände" es fließt Strom" und "es fließt kein Strom", zu betrachten. Digital bedeutet: Darstellung von Größen mit Hilfe von Ziffern. Die Zustände EIN und AUS entsprechen also den Ziffern 1 und 0.

Damit hat ein Digitalcomputer nur zwei Ziffern zur Verfügung. Mit Hilfe dieser beiden erfolgt die Zahlendarstellung im Rechner.

Für Aufgaben, die fest vorgegeben sind, ist die Bearbeitung mit einem Analogrechner unter Umständen sinnvoller (z.B. Maschinensteuerung). Sollen jedoch verschiedenste Probleme auf einem Computer gelöst werden, ist der Digitalcomputer dem Analogrechner weit überlegen, da eine Programmierung eines Analogrechners in der uns bekannten Form nicht möglich ist. Das heißt, daß sämtliche Home- und Personalcomputer Digitalcomputer sind und damit im Dualsystem mit den Ziffern 1 und 0 Daten verarbeiten.

Für den Programmierer sind folgende Zahlensysteme von Bedeutung:

1. Dezimalsystem
2. Dualsystem
3. Hexadezimalsystem

Zahlensysteme sind nach einem bestimmten Prinzip aufgebaute Ordnungsschemata der Ziffern. Jede Zahl kann in andere Zahlensysteme umgerechnet werden. In allen Zahlensystemen steigt der Stellenwert einer Ziffer von rechts nach links.
Um die anderen Zahlensysteme zu erklären, gehen wir von dem bekannten Dezimalsystem aus.

Das Dezimalsystem

Tausender	Hunderter	Zehner	Einer	- Stellenwert
7	3	5	6	- Ziffern

<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

Der Stellenwert steigt von rechts nach links !

Potenz	Zahl	Bezeichnung
10^0	1	E-iner
10^1	10	Z-ehner
10^2	100	H-underter
10^3	1000	T-ausender
10^4	10000	Zehntausender
10^6	1000000	Million

Die Dezimalzahl 1335 kann man auch folgendermaßen schreiben:

1335 bedeutet: 1T + 3H + 3Z + 5E - Der niedrigste Stellenwert(Einer) steht am
435 bedeutet: 4H + 3Z + 5E - weitesten rechts.

1335 ist : 1*1000 + 3*100 + 3*10 + 5*1

1335 ist auch: $1*10^3 + 3*10^2 + 3*10^1 + 5*10^0$

Man definiert eine Potenz mit dem Exponenten 0 als 1.

Z.B.: $10^0=1$, $2^0=1$, $x^0=1$

Das Dualsystem

Das Dualsystem ist nach dem gleichen Prinzip aufgebaut. Der Unterschied besteht nur darin, daß der Stellenwert der einzelnen Ziffern nicht durch Zehnerpotenzen, sondern durch Zweierpotenzen dargestellt wird.
Die Basis des Dualsystems ist 2.

Binär 10101101 = Dezimal 173

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	- Stellenwert
1	0	1	0	1	1	0	1	- Ziffer

$$173 = 1*2^7 + 0*2^6 + 1*2^5 + 0*2^4 + 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0$$

173 = 1*128 + 0*64 + 1*32 + 0*16 + 1*8 + 1*4 + 0*2 + 1*1

Bis jetzt haben Sie die Umrechnung vom Dualsystem in das Dezimalsystem gelernt. Dieser Vorgang läßt sich natürlich auch umkehren. Zur Erläuterung der Umkehrung, betrachten wir die oben errechnete Dezimalzahl 173.

Wir überlegen, welche 2er Potenz gerade noch in dieser Zahl enthalten ist. Zur Hilfe: Im Prinzip kann man das Dualsystem auf n-stellige Zahlen anwenden. Im Computerbereich werden aber nur 8-stellige Binärzahlen verwendet. Folgende Potenzen von 2 können vorkommen.

Potenzen von 2	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
umgerechnete Werte	128	64	32	16	8	4	2	1

In diesem Fall ist also 2^7=128 die höchste vorkommende 2-er Potenz. Jetzt bilden wir die Differenz zwischen 173 und 128. Das Ergebnis lautet 45. Bei diesem Rest wird nun in gleicher Weise wie oben verfahren. Wir suchen also wieder die höchste Potenz von 2, die in diesem Wert steckt. Anhand der Tabelle läßt sie sich leicht ermitteln und beträgt 2^5=32. Anschließend bilden wir wieder die Differenz: (45-32=13).

Das beschriebene Verfahren wird solange angewendet, bis der Rest Null beträgt.

```
2^3=8    (13-8=5)
2^2=4    ( 5-4=1)
2^0=1    ( 1-1=0)
```

Wir haben folgende Potenzen von 2 ermittelt:
2^7 , 2^5 , 2^3 , 2^2 und 2^0
Unter jede vorkommende 2er Potenz schreiben wir eine Eins und unter die fehlenden eine Null:

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

1 0 1 0 1 1 0 1  = 173
```

Die Dezimalzahl 173 wird also im Dualsystem durch 10101101 dargestellt. Im Folgenden wollen wir Binärzahlen durch das Voranstellen von &X kennzeichnen.

z.B.   173= &X 10101101

Bit und Byte

Ein bit ist die kleinste Informationseinheit, aus der alle anderen Informationen zusammengesetzt sind. In dieser Bedeutung steht bit für "basic indissoluble information unit = grundlegende nicht zerlegbare Informationseinheit. Das bit ist ein Begriff, der aus der Kybernetik kommt. Wir werden nicht diese theoretische Bedeutung des bit's sondern die folgende benutzen.

BIT ist die Abkürzung für "binary digit", was soviel heißt, wie Binärziffer. Es wird von einem gesetzten BIT gesprochen, wenn das BIT den Zustand 1, oder von einem rückgesetzten BIT, wenn es den Zustand 0 hat.

Die CPC's haben einen 8-BIT Prozessor, d.h. er kann 8-BIT lange Dualzahlen verarbeiten, was den Dezimalwerten von 0 bis 255 entspricht.

Binärzahl:

```
1 0 1 1 0 1 1 1

g r g g r g g g     g=gesetztes BIT; r=rückgesetztes BIT

7 6 5 4 3 2 1 0     Nummer des BITs
```

Jedem Bit, bzw. jeder Ziffer einer Binärzahl ist eine Bitnummer zugeordnet. Das Bit mit dem niedrigsten Stellenwert, d.h. daß am weitesten rechts stehende, hat die Nummer 0. Von rechts nach links wird dann fortlaufend numeriert. Die Bitnummer entspricht dem Exponenten der Zweierpotenz, die den jeweiligen Stellenwert darstellt.

Beim Computer ist es oft sinnvoll, sich die BIT Zustände als einen Schalter vorzustellen.

SCHALTER EIN = 1
SCHALTER AUS = 0

Bei einer Zahl von 8 Schaltern lassen sich Werte von 0-255, also 256 Schaltzustände darstellen.

Acht Schalter (BITs) zusammengefaßt nennt man ein BYTE. Ein Byte kann vom Computer in einer Speicherstelle abgelegt werden. Wie werden aber Zahlen gespeichert, die größer als 255 sind?

Zu diesem Zweck teilt man die Zahl in zwei Hälften, nämlich dem LOW Byte (engl. low: niedrig; niederwertiges Byte) und dem HIGH Byte (engl. high: hoch; höherwertiges Byte). Diese Bytes werden nun in zwei aufeinanderfolgenden Speicherzellen abgelegt.

Das HIGH und LOW Byte läßt sich folgendermaßen berechnen:

Zahl dividiert durch 256=(HIGH Byte)+Rest
Der Rest der Division entspricht dem LOW Byte.

Zur Erinnerung: Die Zahl 255 ist der maximal darstellbare Wert in einem Byte, da es sich aus 8 BITs zusammensetzt.

Beispiel: Die Zahl 34065 soll in ein LOW und ein HIGH Byte zerlegt werden.

```
34065 / 256=133 Rest 17
34065       =133*256+17
```

```
133=High Byte
 17=Low Byte
```

Die allgemeinen Formeln, in BASIC geschrieben, lauten:

```
1. HB=INT(Zahl/256)        HB=High Byte
   LB=Zahl-HB*256          LB=Low Byte
```

Obige Formel ist bei Zahlen beliebiger Größe anzuwenden.

```
2. HB=Zahl«256             HB=High Byte
   LB=Zahl MOD 256         LB=Low Byte
```

Die zweite Formel ist bei Zahlen, die in dem Bereich von -32768 bis 32767 liegen, anzuwenden.

Damit benötigt eine Zahl, die im Bereich von 256 bis 65535 liegt und im Speicher abgelegt wird, 2 Bytes.

Zur vereinfachten Darstellung von Zahlen, die in dieser Form im Speicher abgelegt sind, ist die Einführung eines weiteren Zahlensystems sinnvoll.

Das Hexadezimalsystem

Die Basis des Hexadezimalsystems ist 16.
Zur Erinnerung:
Die Basis des Dezimalsystems ist 10.
Die Basis des Dualsystems ist 2.

Zur Darstellung von Ziffern, deren Wert größer als 10 ist, werden im Hexadezimalsystem die Buchstaben A bis F verwendet.

Dezimalsystem:
0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,...

Hexadezimalsystem:
0,1,2,3,4,5,6,7,8,9, A, B, C, D, E, F,10,11,12,...

Zuerst wandeln wir Hexadezimalzahlen in Dezimalzahlen um:

Potenz	Wert
16^0	1
16^1	16
16^2	256
16^3	4096

```
&3ABF=3*16^3 + 10*16^2 + 11*16^1 + 15*16^0
&3ABF=3*4096 + 10*256  + 11*16   + 15*1
&3ABF=12288  + 176     + 15
&3ABF=15039
```

Noch ein Beispiel:

```
&1A3E=1*16^3 + 10*16^2 + 3*16^1 + 14*16^0
&1A3E=1*4096 + 10*256  + 3*16   + 14*1
&1A3E=4096   + 2560    + 48     + 14
&1A3E=6718
```

Jetzt folgt die Umrechnung von Dezimalzahlen in Hexadezimalzahlen:

Der Vorgang der Umrechnung gleicht dem auf den vorigen Seiten beschriebenen. Nehmen wir an, die Dezimalzahl 45380 soll im Hexadezimalsystem dargestellt werden:

1. Schritt: Wir überlegen, welche größtmögliche Potenz von 16 in dieser Zahl vorkommen kann (zur Verfügung stehen die Zahlen in obiger Umrechnungstabelle: Hexadezimal in Dezimalzahlen).

2. Schritt: Wir teilen unsere oben genannte Zahl (45380) durch diesen Wert (4096) und wandeln die so erhaltene Dezimalzahl in eine Hexadezimalzahl um.

> 45380/4096=11 Rest 324
> Dezimal=11 => Hexadezimal B

3. Schritt: Nun folgt der gleiche Vorgang mit dem Rest (324). Diesen dividieren wir nun durch die nächst kleinere passende Zahl aus der Tabelle, nämlich 256.

> 324/256=1 Rest 68
> Dezimal=1 => Hexadezimal 1

Die oben beschriebenen Berechnungen werden so lang weitergeführt, bis der Rest der Division 0 ergibt.

68/16=4 Rest 4
Dezimal=4 => Hexadezimal 4

4/1=4 Rest 0
Dezimal=4 => Hexadezimal 4

Unsere umgewandelte Zahl heißt &B144.

Der Vorteil des Hexadezimalsystems liegt darin, daß man das Low und das High Byte direkt ablesen kann.
Für &3ABF gilt:

- das High Byte setzt sich aus den ersten beiden Hexadezimalziffern (3 und A) zusammen. Es hat den Dezimalwert (3*16^1+10*16^0)=58.

-das Low Byte setzt sich aus den letzten beiden Hexadezimalziffern (B und F) zusammen. Es hat den Dezimalwert (11*16^1 + 15*16^0)=191.

Geben Sie einmal folgendes ein:

```
PRINT PEEK(9),PEEK(10)
```

An den beiden Adressen 9 und 10 steht die Sprungadresse, an die das Betriebssystem verzweigt, wenn eine Routine, im unteren ROM aufgerufen werden soll. Für eine Sprungadresse ist ein Wert von 0 bis 65535 (also bis &FFFF) möglich. Diese Zahl ist mit Hilfe von High Byte und Low Byte abgespeichert. Wir wollen die Sprungadresse nun berechnen. Mit dem obigen BASIC Befehl erhalten wir an Adresse 9 den Wert 130 (CPC 664 Wert=138 / CPC 6128 Wert=138) und an Adresse 10 den Wert 185 (CPC 664 Wert=185 / CPC 6128 Wert=185). Dezimal ergibt sich die Sprungadresse also aus 185*256+130=47490 (CPC 664 aus CPC 185*256+138=47498 / 6128 aus 185*256+138=47498).
Nun wollen wir im Hexadezimalsystem die gleiche Rechnung durchführen:

130=&82 (CPC 664 138=&8A /CPC 6128 138=&8A) und 185=&B9 (CPC 664 185=&B9 / CPC 6128 185=&B9), wie Sie leicht nachprüfen können. Den Wert der Sprungadresse erhalten wir einfach durch das Hintereinanderschreiben von High Byte und Low Byte: 47490=&B982 (CPC 664 47498=&B98A / CPC 6128 47498=&B98A).

Es ist also genauso einfach eine Hexadezimalzahl in High Byte und Low Byte zu zerlegen, wie sie aus High Byte und Low Byte zusammenzusetzen. Im Allgemeinen steht das Low Byte einer Zahl an der niedrigeren Speicheradresse, darauf folgt dann das High Byte.

Hiermit haben Sie den ersten Vorteil des Hexadezimalsystems kennengelernt. Außerdem läßt sich die Umwandlung vom Dualsystem in das Hexadezimalsystem sehr leicht durchführen. Dazu unterteilt man eine Dualzahl in zwei Blöcke zu je 4 Bit. Den Block vom 0ten bis 3ten Bit nennt man Low Nibble und den anderen Block vom 4ten bis 7ten Bit High Nibble. Jedes Nibble entspricht genau einer Hexadezimalziffer. Das ist leicht einsichtig, da eine 4 Bit Dualzahl maximal den Wert 15 annehemen kann (15=8+4+2+1). Alle Werte von 0 bis 15 können aber auch durch eine Hexadezimalziffer (0, 1,...9, A, B, C, D, E, F) dargestellt werden. Betrachten wir ein Beispiel:

```
1 1 0 1  1 0 0 1
High N.  Low Nibble
8+4+1    8+1
  13       9
  &D      &9

Also: &X11011001=&D9
```

Mit einiger Übung können Sie direkt aus einer 4-Bit Zahl die dazugehörige Hexadezimalziffer und umgekehrt ablesen. Dabei soll Ihnen folgende Tabelle helfen:

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

Entsprechend läuft die Umwandlung von Hexadezimal nach Dual. Jede Hexadezimalziffer wird durch die entsprechende vier Bit Kombination ersetzt, z.B. &C7=&X1100 0111.

Das Verstehen der Umwandlung zwischen den unterschiedlichen Zahlensystemen ist eine Grundlage für die Programmierung in Maschinensprache.

Aufgaben:

1. Füllen Sie folgende Tabelle aus:

Dezimal	Binär	Hexadezimal
130	?	?
?	&X1001001	?
57312	—	?
?	—	&C0B6
?	?	&37

2. Ab Speicherstelle &A000 soll der Wert 37315 gespeichert werden. Berechnen Sie das High Byte und das Low Byte, und geben Sie die BASIC Befehle an, mit denen die Zahl gespeichert werden kann.

3. Ab Speicherstelle &0006 steht eine wichtige Sprungadresse des Betriebssystems. Welchen Wert hat sie?

Lösungen:

1.

Dezimal	Binär	Hexadezimal
130	&X10000010	&82
147	&X10010011	&93
57312	—	&DFE0
49334	—	&C0B6
55	&X00110111	&37

```
 2. High Byte=145=&91; Low Byte=195=&C3
POKE &A000,&C3: POKE &A001,&91

 3. Low Byte=PEEK(&0006), High Byte=PEEK(&0007)
Sprungadresse=&0580 für den CPC 464
Sprungadresse=&057B für den CPC 664
Sprungadresse=&0591 für den CPC 6128
```

Im Anhang finden Sie eine Tabelle, in der Zahlen von 0-255 (1-Byte) in den drei Zahlensystemen angegeben sind.

1.4 Rechneraufbau

Wenn wir uns mit der Programmierung in Maschinensprache beschäftigen, müssen wir eine Vorstellung vom internen Aufbau und der internen Organisation des Rechners haben. Im Folgen-

den soll hiervon eine Vorstellung entwickelt werden, die unseren Ansprüchen vorerst genügt.

Die meisten von Ihnen besitzen einen 64K (K Kilobyte=1024 Bytes) Computer. Das heißt, daß die Speicherkapazität des Rechners 64*1024=65536 Bytes ist. Da sich ein Byte aus 8 Bit zusammensetzt und das die interne Speicherdarstellung von Daten ist, besteht Ihr Computer quasi aus 64*1024*8 Bits, also ca. 0.5 Millionen Schaltern, die entweder EIN- oder AUSgeschaltet sind. Diese Vorstellung ist jedoch für die konkrete Arbeit mit dem Computer nicht sinnvoll. Aus diesem Grund sind 8 Bit zu einem Byte zusammengefaßt. Diese 64*1024 Bytes stehen im RAM des Rechners. RAM heißt Random Access Memory, zu deutsch Schreib- und Lesespeicher oder auch Arbeitsspeicher. Die 65536 Bytes des RAM sind von &0000 bis &FFFF durchnumeriert. Die dem Byte entsprechende Nummer ist seine Adresse. Diese Adresse wird normalerweise als eine Hexadezimalzahl angegeben. Vom BASIC aus können wir direkt auf den RAM zugreifen. Hierzu dienen die Befehle >PEEK< und >POKE<. >PEEK (Adresse)< liest den Wert des an der angegebenen Adresse stehenden Bytes, und >POKE Adresse,Wert<, speichert den angegebenen Wert an der angegebenen Adresse. Da jede Adresse einem Byte zugeordnet ist und ein Byte aus 8 Bit besteht, also im Bereich von 0-255 (&00-&FF) liegt, darf der zu speichernde Wert auch nur in diesem Bereich liegen. Natürlich muß auch die Adresse zwischen &0000 und &FFFF liegen.

Der RAM dient der Speicherung der von Ihnen eingegebenen BASIC Programme. Weiterhin wird der codierte Bildschirminhalt von &C000 abgespeichert, wobei in >MODE 2< ein Punkt einem gesetzten Bit und umgekehrt entspricht. Außerdem befinden sich einige wichtige Routinen des Betriebssystems und Informationen über aktuelle Farben, Keybelegung, selbstdefinierte Zeichen etc. im RAM. Da sich Systemroutinen und wichtige Informationen im RAM befinden, können unvorsichtige POKEs den Rechner zum Absturz bringen. Versuchen Sie zum Beispiel nie >POKE &8,0<.

Die Aufteilung des RAM ist folgendermaßen:

```
&0000 - &0170       vom System benutzt
&0171 - &AB7F       für BASIC Programme
&AB80 - &BFFF       vom System benutzt
&C000 - &FFFF       Bildschirmspeicher
```

Die Adressen von &AB80 bis &BFFF gelten für den CPC 464 ohne Floppy. Bei Verwendung eines Diskettenlaufwerkes ändert er sich zu &A6FB. Beim CPC 664 wird dieser Wert zu &A67B und beim CPC 6128 zu &A67B.

Durch den >MEMORY Adresse< Befehl können wir den Platz, der für BASIC Programme reserviert ist, begrenzen. Damit steht uns der Bereich von der im MEMORY Befehl angegebenen Adresse bis &AB7F für das Abspeichern unserer Maschinenprogramme zur Verfügung. In unserem Beispiel haben wir durch >MEMORY &9FFF< den Bereich von &A000 bis &AB7F für unser Maschinenprogramm reserviert und es dann ab &A000 mit Hilfe von >POKE< Befehlen abgespeichert.

Nun werden Sie sich wundern, daß nur etwas mehr als 5K des RAMs für Systemroutinen benutzt wird:

Wo befinden sich der Interpreter und das Betriebssystem, die es uns möglich machen, in BASIC zu programmieren?
Sie vermuten richtig:

Es gibt noch einen weiteren wichtigen Speicher, den ROM (Read Only Memory=Nur Lese Speicher oder Festwertspeicher).
Im ROM befinden sich alle Daten und Programme, die es uns ermöglichen so auf einfache Weise in BASIC zu programmieren. Da ein ROM ein Festwertspeicher ist, wird er, mit Daten und Programmen (in Maschinensprache !) beschrieben, vom Werk in den Rechner eingebaut. Leider ist es uns vom BASIC aus nicht möglich, den Inhalt des ROMs zu lesen. Sobald wir ein Maschinenprogramm für diese Aufgabe erstellt haben, ergibt sich folgendes Bild:

Die CPC's besitzen zwei 16K ROMs (plus 16K Disketten ROM, falls vorhanden), deren Adressen sich mit denen des RAMs überlagern. Das ist notwendig, da der Z80A Prozessor nur 16 Adressleitungen besitzt, d.h. die Adresse eines Bytes kann nicht länger als 16 Bit sein. Mit 16 Bit ist genau der Bereich von &0000 bis &FFFF abgedeckt. Zum Lesen der ROMs wird also erst der CPU mitgeteilt, daß der ROM gelesen werden soll, und danach können dieselben Adressen wie für das RAM benutzt werden. Dieses Verfahren bezeichnet man als "Bank Switching". Beim CPC 6128 wird er benutzt, um 128K RAM zu verwalten. Die ROMs belegen folgende Bereiche:

```
1. ROM &0000 - &3FFF   Betriebssystem
2. ROM &C000 - &FFFF   BASIC
```

Das Betriebssystem enthält, wie der Name schon sagt, die Routinen, die grundsätzlich notwendig sind, damit der Rechner arbeitet. Es ist für die Steuerung der externen Geräte, für die Verwaltung der Daten, Datenverkehr usw., zuständig. Im unteren ROM Bereich befinden sich auch die Kopien der Systemroutinen, die im RAM stehen. Beim Einschalten oder Reset des Rechners werden diese Routinen vom ROM ins RAM kopiert. Außerdem befindet sich der Zeichenspeicher im ROM (&3800-&3FFF), wo jedes Zeichen des Computers in einer Bitmatrix (d.h. 0-kein Punkt, 1-Punkt) dargestellt ist.

Die von uns programmierten Basic Befehle, werden durch die im BASIC ROM stehenden Programme ausgeführt. Die Befehlswort-tabelle steht z.B. ab &E388 beim CPC 464 (ab &E451 beim CPC 664, ab &E44C beim CPC 6128. Soviel zu den Speichern der CPC's.

Natürlich enthält unser Computer noch viele andere ICs, wie den Z80-A Prozessor oder den Sound Chip. Den Z80A Prozessor werden wir im nächstem Kapitel genau beschreiben. Falls Sie Interesse an weiteren Informationen über den internen Aufbau Ihres Rechners haben, greifen Sie bitte auf die Bücher "CPC 464 Intern" bzw. "CPC 664 & 6128 Intern" von DATA BECKER zurück.

KAPITEL II: DER Z80 PROZESSOR

2.1 AUFBAU DER CPU

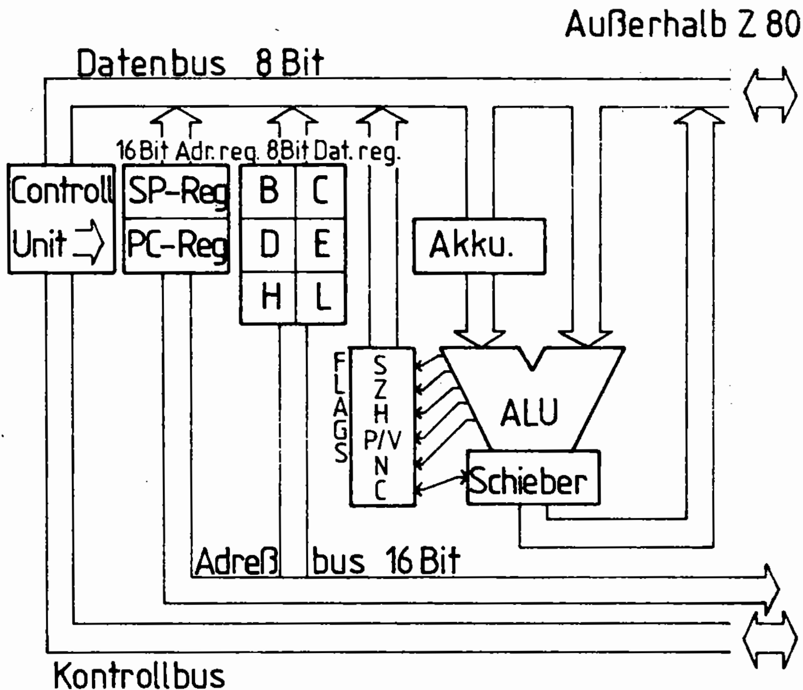


Abb.1 Aufbau des Z 80 2.1

Die Schneider CPC's besitzen eine Z80-A CPU (Zentraleinheit). Wir erinnern uns, daß die CPU als "das Gehirn" des Rechners bezeichnet werden kann. Damit ist die Bedeutung dieses MPU (MPU:engl.Micro Prozessing Unit- Mikroprozessor) keine Frage.

In diesem Kapitel wollen wir uns mit dem Aufbau und der Funktion der einzelnen, in der CPU enthaltenen Bausteine befassen. Die Grafik auf dieser Seite soll uns helfen, daß Innenleben einer Zentraleinheit zu verstehen. Wenn wir die Zeichnung von links nach rechts betrachten, erkennen wir folgendes:

1. Cu (CU:engl.Control Unit- Kontrolleinheit)
   Alle Abläufe in einem Computer werden durch die CU kontrolliert und gesteuert.

2. Kontrollbus
   Der Kontrollbus ist der "lange Arm" der CU. Durch ihn werden Bausteine außerhalb der CPU gelenkt und überwacht.

3. Stapelzeiger SP (SP:engl. Stack Pointer)
   Mit Hilfe des SPs werden Daten und Unterprogrammrücksprungadressen im RAM zwischengespeichert. Da im SP Adressen gespeichert werden, ist er ein 16-Bit Register.

4. Programmzeiger PC (PC:engl.Programm Counter-eigentlich Programmzähler)
   Der PC zeigt auf die Speicheradresse, an der der jeweils zu verarbeitende Befehl steht.

5. Register B bis L (Register registrieren)
   Die Schneider CPU besitzt mehrere Register, in denen Daten gespeichert werden.

6. Flags (Flag:engl.flag- Flagge, Fahne; hier besser Kennzeichen)
   Flags dienen als Anzeiger für bestimmte Ereignisse, die bei Rechenoperationen in der CPU enstehen. Flags können gesetzt (Flagge oben) oder nicht gesetzt bzw. rückgesetzt (Flagge unten) sein.

7. Adreßbus (liegt außerhalb der CPU)
   Der Adreßbus stellt die Verbindung zu anderen MPUs des Computers her. Er zeigt auf die Speicherstelle im ROM bzw. RAM, deren Inhalt gelesen oder beschrieben werden soll. Der Adreßbus ist 16-Bit breit. Das ist notwendig, um 64K Speicherplatz adressieren zu können.

8. Datenbus (liegt außerhalb der CPU)
   Datenbusse "befördern" die gelesenen bzw. zu schreibenden Daten. Der Adreßbus zeigt dabei auf die Adresse der Daten. Der Datenbus ist 8-Bit breit.

9. Akkumulator (lat.akkumulieren:ansammeln)
   Der Akkumulator (Akku) ist das wichtigste Register der CPU. Man kann ihn auch als das Rechenregister bezeichnen.

10. ALU (ALU:engl. Arithmetical Logical Unit- arithmetisch logische Einheit, Recheneinheit, Rechenwerk)
    Die ALU führt sämtliche arithmetischen und logischen Operationen durch. Abhängig vom Ergebnis der Operationen werden die Flags beeinflußt.

11. Schieber
    Der Schieber führt die Rotier- und Schieberoutinen aus.

Wie in Punkt 4 schon erwähnt, enthält die CPU mehrere Register. Zum Verständnis der Funktionen, haben wir sie in fünf Gruppen eingeteilt.

1. Der Akkumulator
2. Die Flags
3. Die "verknüpfbaren Sechs" 8-Bit Register
4. Die "unzertrennlichen Vier" 16-Bit Register
5. Interrupt-/Refreshregister

2.2 Der Akkumulator

Der Akku bzw. das A Register ist das wichtigste Register des Z80A. Die meisten arithmetischen und logischen Befehle benutzen dieses Register. Bei der Ausführung eines Vergleichbefehls wird grundsätzlich mit dem Inhalt des Akkus verglichen. Wie alle Register, bis auf SP, PC, IX und IY ist das A Register ein 8-Bit Register.

2.3 Die Flags

Das F bzw. Flagregister ist 8 Bit breit (wie A,B,C,D,E,H und L). Es hat jedoch eine andere Funktion als diese. Im Flagregister werden die einzelnen Bits als Anzeiger für bestimmte Ereignisse, die bei Operationen des ALUs (Rechenwerk) entstehen, benutzt. Die einzelnen Bits des F Registers haben folgende Bedeutung:

```
S   Z       H       P/V  N   C  -Flagbezeichnung
7   6   5   4   3   2    1   0  -Bitnummer
```

```
C   - Carry-Übertrag
N   - Subtraktion
P/V - Parität/Überlauf
H   - Halbübertrag
Z   - Zero-Null
S   - Sign bzw. Vorzeichen
```

C Flag (Bit 0)
Tritt bei einer Addition oder Subtraktion ein Übertrag auf, wird dieses Bit gesetzt, sonst rückgesetzt.

N und H Flag (Bit 1, Bit 4)
Diese Flags werden intern vom Z80A benutzt. Sie haben für unsere Zwecke keine Bedeutung.

P/V Flag (Bit 2)
Dieses Flag hat eine doppelte Funktion:
Es wird gesetzt, wenn ein Überlauf (V) (engl.:overflow) auftritt, sonst rückgesetzt. Weiterhin zeigt es die Parität (P) eines Bytes an.

Z Flag (Bit 6)
Dieses Flag wird gesetzt, wenn das Ergebnis einer Subtraktion Null ist, sonst rückgesetzt. Bei einem Vergleich wird dieses Bit gesetzt, wenn Gleichheit vorliegt.

S Flag (Bit 7)
Ist das Ergebnis einer Addition bzw. Subtraktion größer als 127, wird dieses Bit gesetzt. Wie wir später sehen werden, bedeuten bei der Arithmetik der CPU Bytes, die größer als 127 sind, negative Zahlen.

Die Bits 3 und 5 des Flagregisters sind ungenutzt.

2.4 Die "verknüpfbaren sechs" 8-BIT Register

Zu dieser Gruppe gehören sechs 8-Bit Register:
B, C, D, E, H, L
Diese Register sind in der Lage, Registerpaare zu bilden, um ein 16-Bit breites Register darzustellen. In C, E, L wird jeweils das Low und in B, D, H das High Byte gespeichert.

B/C (Byte Counter)
Das B Register bzw. BC Registerpaar wird häufig als Zähler z.B. für Schleifen verwendet.

Das DE Registerpaar ist frei verfügbar.
Dieses Registerpaar wird oft zur Zwischenspeicherung von Adressen oder Daten verwendet.

H/L (High/Low)
Das Registerpaar HL wird oft zur Speicherung von Adressen verwendet.

Eine Gewöhnung an die Benennung der Register in dieser Weise ist sinnvoll, da einige Befehle die Register in der oben beschriebenen Weise benutzen. Prinzipiell kann man natürlich auch das L oder E Register als Zähler verwenden.
Eine Besonderheit des Z80A ist, daß alle oben genannten Register mit gleicher Funktion noch einmal vorhanden sind. Dieser Zweitregistersatz steht uns zur Verfügung. Allerdings kann immer nur ein Satz zur Zeit benutzt werden.

2.5 Die "unzertrennlichen vier" 16-BIT Register

Zu dieser Gruppe gehören vier 16-Bit Register:
SP, PC, IX, IY
Das SP Register ist ein festes 16-Bit Register, d.h. es kann nicht in zwei 8-Bit breite Register zerlegt werden. Der Stack Pointer zeigt auf die jeweilige Adresse im Speicher, an der Rücksprungadressen oder zwischengespeicherte Daten stehen. Die Adresse bezieht sich auf eine Speicherstelle, die in einem Bereich des RAMs liegt, den man Stack oder Stapel nennt. Die Benutzung des Stacks zur Datenspeicherung geht folgendermaßen vor sich:

Beim Einschalten des Rechners wird der SP auf die höchste Adresse im Stack gesetzt (&C000). Soll nun ein Byte auf den Stack gelegt werden, so wird SP automatisch um eins erniedrigt und dieses Byte in der Adresse, die SP dann anzeigt, abgespeichert. Er zeigt also immer auf die letzte Eintragung im Stapel. Beim "Holen vom Stack" läuft der Vorgang umgekehrt ab. Erst wird das Byte an der Adresse, auf die SP zeigt, gelesen, dann wird SP um eins erhöht. Auf diese Weise ist es möglich, Unterprogrammaufrufe beliebig ineinander zu verschachteln.

Der PC ist ein besonderes Register. Er kann vom Programm aus weder beschrieben noch geändert werden. Der PC wird intern verwaltet und zeigt immer auf die Adresse des aktuellen Befehls.

Das Byte, an der Adresse auf die der PC zeigt, wird gelesen und der PC wird um eins erhöht (d.h. er zeigt auf das nächstfolgende

Byte). Das gelesene Byte wird als Befehl interpretiert. Dann werden eventuell zu dem Befehl gehörende Daten gelesen (PC wird dann wieder erhöht). Danach erfolgt die Ausführung des Befehls und der Vorgang beginnt von Neuem.

IX/IY Register werden hauptsächlich zur Speicherung von Adressen bzw. relativen Adressen benutzt. Auch diese beiden Register gehören, wie alle unter drittens aufgeführten, zu den 16-Bit Registern. Bei diesen ist es nicht möglich, getrennt auf High bzw. Low Byte (wie bei BC, DE, HL) zuzugreifen. Die Benutzung der Indexregister ist der des HL Registerpaares ähnlich. Den Unterschied werden wir bei der indizierten Adressierung kennenlernen.

2.6 Interrupt-/ UND Refresh-Register

Diese beiden Register sind der CU zugeordnet.

I bzw.Interruptregister
(engl.interrupt: unterbrechen)
Tritt ein Interrupt auf, d.h. eine Programmunterbrechung, so enthält dieses 8-Bit Register den oberen Teil der Adresse, an die verzweigt werden soll. Der untere Teil wird von dem Baustein des Computers geliefert, der den Interrupt ausgelöst hat.

R bzw. Refreshregister (engl.:refresh: auffrischen)
Dieses Register wird von der Hardware als Zähler benutzt, um in regelmäßigen Abständen den Inhalt der dynamischen Speicher aufzufrischen. Damit soll verhindert werden, daß gespeicherte Informationen verlorengehen. Durch ständiges Neuladen des gleichen Speicherinhaltes innerhalb sehr kurzer Zeit wird ein Verlust der Daten verhindert.

Eine Befehlsausführung durch die CPU sieht dann folgendermaßen aus:

Nachdem wir nun die Z80-A CPU kennengelernt haben, werden wir uns jetzt den eigentlichen Maschinenbefehlen zuwenden.

KAPITEL III: DER BEFEHLSSATZ DES Z80

3.1 Einleitung: Eingabe von Maschinenprogrammen

Damit wir die Befehle des Z80A gleich ausprobieren können, müssen wir uns zuerst darüber Gedanken machen, auf welche Weise ein Maschinenprogramm vom BASIC aus eingegeben und abgespeichert wird. Ähnlich wie beim BASIC, wo eine Zeilennummer einem Befehl zugeordnet ist, wird jedem Maschinenbefehl eine Adresse zugeordnet.

BASIC		Maschinensprache		
Zeilennr.	Befehl	Adresse	Befehl	Code
9	HL=HL+1	&A009	INC HL	&23
10	RETURN	&A00A	RET	&C9

- Beim BASIC wird eine Zeilennummer einem Befehl zugeordnet.

- Bei der Maschinensprache gehört zu jedem Befehl eine Adresse.

Ein Maschinenprogramm ist damit eine Folge von Befehlscodes, die in aufeinanderfolgenden Adressen im Speicher stehen.

Vom BASIC aus haben wir die Möglichkeit, mit Hilfe des >POKE< Befehls die Codes an die entsprechenden Adressen zu schreiben. Ein Aufruf der Maschinenprogramme geschieht dann mit >CALL Adresse<, wobei die Adresse den Speicherplatz kennzeichnet, der den ersten Befehl enthält. Damit unser Maschinenprogramm nicht versehentlich überschrieben wird, müssen wir einen Speicherbereich mit dem >MEMORY< Befehl reservieren. Wir werden durch >MEMORY &9FFF< immer den Bereich von &A000 bis &AB7F reservieren, damit stehen also &B80 Bytes (entspricht 3K) für Maschinenprogramme zur Ver-

fügung. Ein typisches BASIC Programm, zum Laden von Maschinenprogrammen hat folgenden Aufbau:

```
10 MEMORY &9FFF
20 FOR I=Startadresse TO Endadresse
30 READ A
40 POKE I,A
50 NEXT I
60 DATA .....
70 DATA .....
    .
    .
```

In den DATA Zeilen stehen die Codes, die das eigentliche Maschinenprogramm bilden werden. Die Endadresse (V=Variable; diese Abkürzung werden wir in Zukunft immer hinter Wörter schreiben, die Variablen sind) muß natürlich größer als &9FFF und Startadresse (V) muß kleiner als &AB80 sein. Der Aufruf des geladenen Programmes erfolgt mit >CALL Startadresse<.

Normalerweise werden wir &A000 als Startadresse benutzen. Endadresse (V) ergibt sich aus Startadresse (V) plus Länge des Programms in Bytes minus 1. Die Länge eines Programms entspricht der Anzahl der Eintragungen in den DATA Zeilen.

Für die Eingabe von kleinen Programmen ist folgendes BASIC Programm sinnvoll:

```
10 CLS
20 MEMORY &9FFF
30 LOCATE 10,10:INPUT "Startadresse";adr
40 IF adr <&A000 OR adr>&ABFF THEN 30
50 PRINT
60 PRINT HEX$(adr,4);":";
70 INPUT Wert$
80 IF Wert$="" THEN END
90 Wert=VAL("&"+Wert$)
```

```
100 POKE adr,wert
110 adr=adr+1
120 IF adr>&AB7F THEN PRINT "Speicher voll": END
130 GOTO 60
```

Sie geben die Hexadezimalcodes direkt ein, und das Programm wird das "Poken" für Sie erledigen. Bei der Startadresse brauchen Sie das Hexzeichen (&) nicht mit einzugeben. Wollen Sie das Programm beenden, geben Sie >ENTER< ein.

Nachdem wir nun die Eingabe von Maschinenprogrammen kennengelernt haben, wollen wir uns die Befehle des Z80A ansehen.

Anmerkung: Bei der Befehlserklärung werden wir oft mit Analogien zu den BASIC Befehlen arbeiten. Dazu stellen wir uns ein Register im BASIC als eine Variable mit demselben Namen vor (Register HL in Maschinensprache entspricht Variable HL in BASIC).

Die Befehle des Z80A lassen sich in 5 Gruppen unterteilen:

1. Transfer von Daten
2. Bearbeitung von Daten und Tests
3. Sprünge
4. Steuerbefehle
5. Ein- und Ausgabe

3.2 Transfer von Daten

Diese Befehle dienen der Übertragung von Daten.
Daten können übertragen werden von:

a) *Register zu Register*

Das entspricht einer Zuweisung im BASIC, wie z.B. A=B oder SP=HL. Der Maschinenbefehl hat folgendes Format:

LD Ziel,Quelle z.B. LD A,B (LD-lade)

b) *Register zur Speicherstelle*

Bei der Übertragung vom Register zur Speicherstelle ist der BASIC Befehl >POKE Speicheradresse, Variable<, z.B. >POKE &A000,HL< entsprechend dem Maschinensprachebefehl LD (&A000),HL.

c) *Speicherplatz zu Register*

Die Datenübertragung vom Speicher in ein Register, z.B. LD H,(&A005), entspricht dem BASIC Befehl: >H=PEEK (&A005)<.

3.3 Bearbeitung von Daten und Tests

Die Befehle zur Bearbeitung von Daten kann man wiederum in 5 Gruppen einteilen:

- arithmetische Operationen (z.B. ADDition, SUBtraktion)
- logische Operationen (z.B. AND, OR)
- Zählbefehle (INCrease = erhöhen, DECrease = erniedrigen)
- Bitmanipulation (SET, RESet)
- Vertauschen und Schieben von Bits (Rotate = rotieren, Shift = schieben)

Bei der Ausführung dieser Befehle werden Register- oder Speicherinhalte (im RAM) verändert. Viele Befehle sind denen des BASIC ähnlich:

Assembler	BASIC
SUB A,B (SUBtraktion)	A=A-B
ADD HL,BC (ADDition)	HL=HL+BC
AND C	A=A AND C
OR &HL	A=A OR PEEK(HL)

Getestet werden entweder einzelne Bits in Registern bzw. Speicherstellen (BIT Befehl), oder es werden Register- oder Speicherinhalte mit dem Akku verglichen (CP Befehl= compare). Je nach dem Ausgang dieser Tests werden von der ALU die jeweiligen Flags im F Register gesetzt oder gelöscht.

3.4 Sprünge

Mit Hilfe dieser Befehle ist es möglich, Verzweigungen in Maschinenprogramme einzubauen.
Man unterscheidet drei Sprungarten:

- direkter Sprung an eine 16-Bit Adresse (JP=Jump)
- relativer Sprung zur aktuellen Adresse (JR=Jump relativ)
- Unterprogrammsprünge (CALL und RET-Rücksprünge)

Man bezeichnet einen Sprung als bedingt, wenn die Entscheidung darüber, ob gesprungen wird, vom Status eines Flags abhängt. Ein bedingter Sprung, d.h. einer, bei dem der Sprung vom Status eines Flags abhängt, ist z.B. JR NZ,$-6>A000.

Analogien:

Assembler	BASIC
JP	GOTO
CALL	GOSUB
RET	RETURN
JR	---- (ähnlich der FOR-NEXT Schleife)

3.5 Steuerbefehle

Mit diesen Befehlen kann beispielsweise ein Programm unterbrochen werden. Auch Interruptsteuerung ist mit diesen Befehlen möglich.

3.6 EIN/AUSGABEBEFEHLE (Input/Output)

Die I/O Befehle dienen der Kommunikation mit Ein/Ausgabegeräten.Abhängig von der jeweilig angesprochenen I/O Portadresse werden die verschiedensten Aufgaben mit diesen Befehlen bewältigt. Häufig über I/O Befehle ansprechbare ICs sind:

PPI (Programmable Peripheral Interface),
PSG (Programmable Sound Generator),
CTRC (Cathode Ray Tube Controller) und damit die Peripheriegeräte: Tastatur, Lautsprecher, Monitor, Drucker und Kassettenrecorder.

KAPITEL IV: DIE BEFEHLE

4.1 8-Bit Transferbefehle

Alle Transferbefehle dieser Art werden durch den Ladebefehl LD dargestellt.
Ein Ladebefehl hat folgendes Format:

LD Ziel,Quelle

Bei den 8-Bit Transferbefehlen werden je 8-Bit von der Quelle in das Ziel geladen. Am Beispiel dieser Befehle wollen wir die Adressierungsarten des Z8OA kennenlernen.

Jeder Maschinenbefehl besteht grundsätzlich aus einem Operationscode (Opcode), auf den ein Operanden- oder Adressenfeld folgen kann. Der Opcode legt fest, welche Operation ausgeführt werden soll. Manchmal enthält ein Opcode Bits, die als Zeiger auf ein Register benutzt werden. Genaugenommen gehören diese Bits nicht zum Opcode. Zur Vereinfachung wollen wir aber die eventuell vorhandenen Zeiger zum Opcode dazuzählen. Bei einigen Befehlen folgen auf den Opcode Daten- oder Adressbytes. Außerdem gibt es Befehle, deren Opcode zwei Bytes lang ist. Damit kann ein Befehl eine Länge von 1 bis 4 Bytes haben.

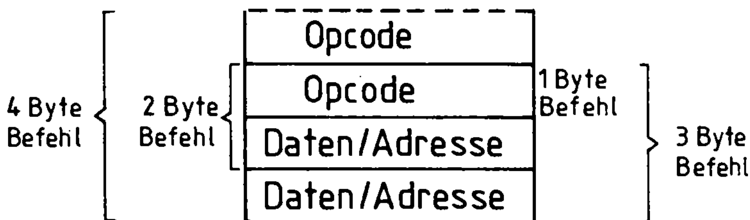


Abb. 2 4.1

Zum Interpretieren der einem Befehl folgenden Daten bzw. Adressen, ist es notwendig, die verschiedenen Adressierungsarten zu kennen.

Unmittelbare Adressierung (Immediately Adressing)

(engl.immediately: unverzüglich, unmittelbar)
Das ist die einfachste Art der Adressierung.

Format:

LD r,n

Bei diesem Befehl stellt "r" ein Register (A,B,C,D,E,H oder L) und "n" eine 8-Bit Zahl (Konstante) dar; d.h. das angegebene Register r wird mit der "unmittelbar" dahinterstehenden Konstante geladen. Eine solche Konstante bezeichnet man auch als Literal. Die unmittelbare Adressierung ist in Abbildung 3 dargestellt. Auf den 8-Bit Opcode folgt ein 8 oder 16-Bit Literal (die Konstante).

Beispiel:

```
LD C,&7F                 BASIC: C=&7F
                        (Bedeutet: lade Register C mit &7F)
```

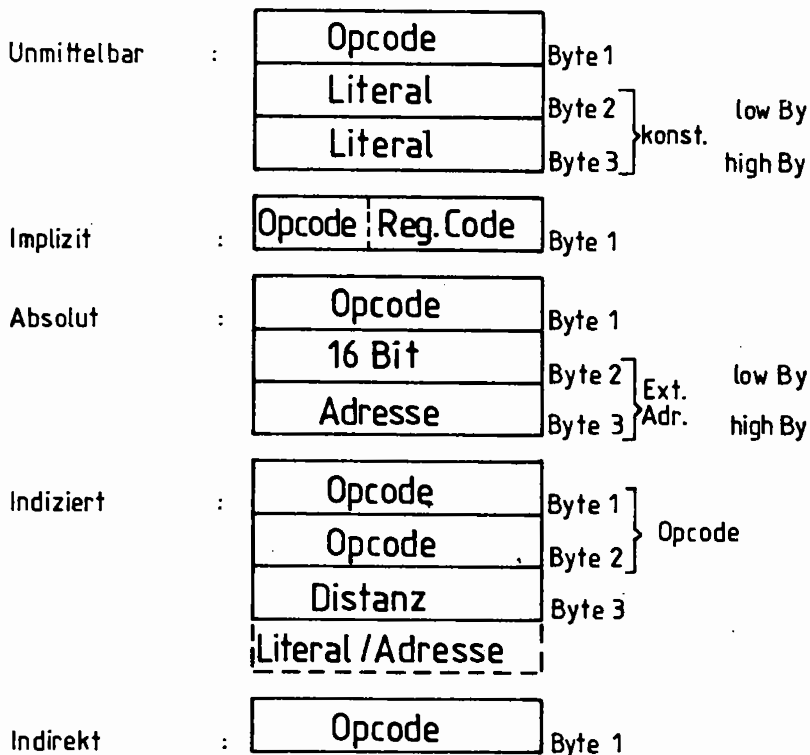


Abb. 3 4.1

Implizite- und Registeradressierung (engl. Implied Register Addressing)

Befehle, die ausschließlich mit Registern arbeiten, verwenden die implizite Adressierung.(engl. implied: implizit- mit inbegriffen, einschließlich)

Format:

LD r,r'

Übertrage den Inhalt des Quellregisters r' nach r oder lade r aus r'. Register können A, B, C, D, E, H oder L sein.

Der Name dieser Adressierungsart ergibt sich aus der Tatsache, daß der Operand (d.h. die beiden betroffenen Register) nicht extra angegeben ist. Vielmehr enthält der Opcode des Befehls die betroffenen Register, (er impliziert sie).
Der Opcode dieses Befehls in Binärform ist:

01ZZZQQQ

Jeder der Buchstaben Z und Q steht hierbei für ein Bit. Weiterhin stehen die drei Z's zusammen für das Zielregister r und die Q's für das Quellregister r'. Der Code für die Register ist:

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

Beispiel: LD B,C = 01 000 001 = &41
                   LD  B   C
```

Damit ist es möglich, die implizit adressierten Befehle, als 1-Byte Opcode darzustellen. Aus diesem Grund ist ihre Ausführungsdauer sehr gering.

Beispiel:

LD A,B BASIC: A=B

Bedeutet: Übertrage den Inhalt von B nach A oder lade A aus B.

Zilog Inc. (Der "Erfinder" des Z80A) bezeichnet obige Adressierungsart als Registeradressierung und definiert die implizite Adressierung etwas abweichend. Demnach wären nur die Befehle

LD I,A ; LD R,A ; LD A,R und LD A,I implizit adressiert. Wir werden diesen Unterschied jedoch nicht machen und beide Begriffe, implizite Adressierung und Registeradressierung synonym benutzen.

Absolute oder "äußere" Adressierung (External Addressing)

(engl.external: außerhalb, äußerlich)
Als absolute Adressierung bezeichnet man das Verfahren, Daten aus dem Speicher zu holen oder dort abzulegen. Bei diesem Verfahren wird die 16-Bit Adresse der Speicherstelle komplett angegeben (die "absolute" Adresse).

Format:

LD (nn),r oder LD r,(nn)
(nn:ist die Adresse der Speicherstelle.)

Das angegebene Register r wird mit dem Inhalt der Speicherstelle nn geladen und umgekehrt. Aus Abbildung 3 können Sie ersehen, daß die Adresse auf den Opcode folgt. Die absolute Adressierung braucht drei Bytes, damit sind die Befehle dieser Klasse relativ langsam.

Beispiel:

```
LD A,(&BF93)         BASIC:A=PEEK(&BF93)
LD (&A001),A         BASIC:POKE &A001,A
```

Indizierte Adressierung (Indexed Addressing)

(engl.index: Hinweis)
Bei der indizierten Adressierung wird die Adresse der Speicherstelle nicht absolut angegeben, sondern aus dem Inhalt eines Indexregisters und einer angegebenen Distanz berechnet.

Format:

```
LD r,(IX+d) oder LD (IX+d),r
LD r,(IY+d) oder LD (IY+d),r
(d=Distanz)(IX oder IY sind Register)
```

Laden des Registers r mit der Speicherstelle, die folgende Adresse hat (und umgekehrt): Die Adresse ergibt sich aus dem Inhalt vom Indexregister und der angegebenen Distanz.

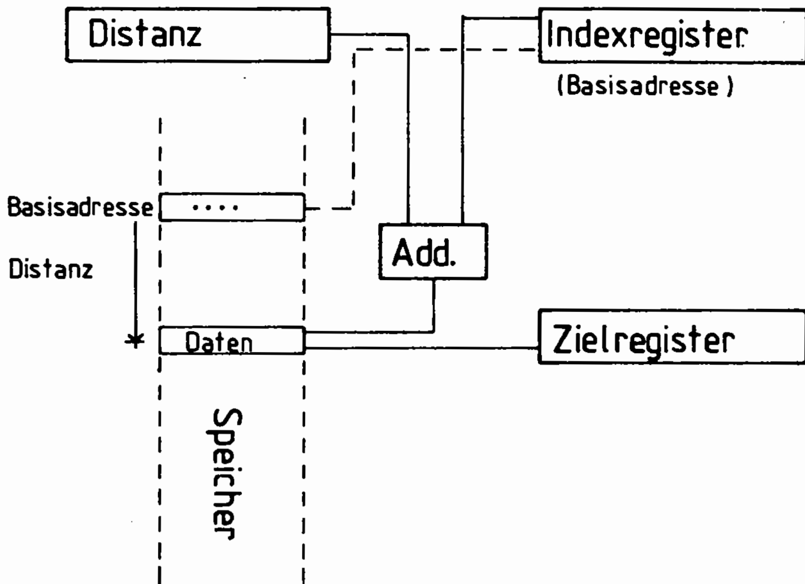


Abb. 4 Indizierte Adressierung / LD reg,(XY+dis)

Die indizierten Befehle besitzen einen 2-Byte Opcode, auf den die Distanzangabe folgt. Das erste Byte des Opcodes ist:

&DD - wenn das IX Register gemeint ist
&FD - wenn das IY Register gemeint ist

Die restlichen Bytes des Codes sind identisch, unabhängig davon, ob IX oder IY gemeint ist. Die Technik der indizierten Adressierung verwendet man, um nacheinander auf die Elemente eines Datenblocks zuzugreifen. Die Distanz kann positiv oder negativ sein, d.h. das Distanzbyte wird im Zweierkomplement angegeben.

Beispiel:

```
LD E,(IX+&32)       BASIC: E=PEEK (IX+&32)
LD (IY+&12),A       BASIC: POKE IY+&12,A
```

Indirekte Adressierung (Register indirekt)

Diese Adressierungsart ist der indizierten Adressierung ähnlich, nur wird hierbei die Speicherstelle durch den Inhalt eines der Registerpaare HL, BC oder DE adressiert.

Format:

```
LD r,(HL)     oder    LD (HL),r
LD A,(BC)     oder    LD (BC),A
LD A,(DE)     oder    LD (DE),A
```

Laden des Registers r bzw. A mit dem Inhalt der Speicherstelle, die durch den Inhalt des Registerpaares HL bzw. BC, DE adressiert ist. Diese Adressierungstechnik hat gegenüber der indizierten und absoluten Adressierung den Vorteil, daß sie nur 1-Byte lange Befehle braucht, d.h. Register r und Registerpaar rps sind im Opcode enthalten und müssen nicht extra angegeben werden. Damit ist dieser Befehl schneller, und bietet trotzdem die Möglichkeit auf die kompletten 64K zuzugreifen.

Beispiel:

```
LD B,(HL)       BASIC: B=PEEK (HL)
LD (BC),A       BASIC: POKE BC,A
```

Damit haben wir alle bei den 8-Bit Transferbefehlen vorkommenden Adressierungsarten besprochen. Im Laufe dieses Kapitels werden wir noch einige andere Adressierungsarten kennenlernen und die jetzt bekannten auf andere Befehle übertragen. Im Anhang finden Sie Tabellen, in denen sich alle Befehle, sortiert nach Aufgaben (Transfer, Sprünge, etc.) und Adressierungsarten befinden. In diesen Tabellen können Sie die Opcodes aller Befehle nachschlagen. Im folgenden wollen wir noch einmal alle 8-Bit Ladebefehle zusammenstellen. Eine Tabelle für die Symbole der Flagbeeinflussung befindet sich im Anhang.

Mnemonic	Symbolic Operation	Flags C	Z	P/V	S	N	H	OP-Code 76 543 210	No. of Bytes	No. of M Cycles	No. of T Cycles	Comments	
LD r, r'	r ← r'	•	•	•	•	•	•	01 r r'	1	1	4	r, r'	Reg.
LD r, n	r ← n	•	•	•	•	•	•	00 r 110	2	2	7	000	B
								← n →				001	C
LD r, (HL)	r ← (HL)	•	•	•	•	•	•	01 r 110	1	2	7	010	D
LD r, (IX+d)	r ← (IX+d)	•	•	•	•	•	•	11 011 101	3	5	19	011	E
								01 r 110				100	H
								← d →				101	L
LD r, (IY+d)	r ← (IY+d)	•	•	•	•	•	•	11 111 101	3	5	19	111	A
								01 r 110					
								← d →					
LD (HL), r	(HL) ← r	•	•	•	•	•	•	01 110 r	1	2	7		
LD (IX+d), r	(IX+d) ← r	•	•	•	•	•	•	11 011 101	3	5	19		
								01 110 r					
								← d →					
LD (IY+d), r	(IY+d) ← r	•	•	•	•	•	•	11 111 101	3	5	19		
								01 110 r					
								← d →					
LD (HL), n	(HL) ← n	•	•	•	•	•	•	00 110 110	2	3	10		
								← n →					
LD (IX+d), n	(IX+d) ← n	•	•	•	•	•	•	11 011 101	4	5	19		
								00 110 110					
								← d →					
								← n →					
LD (IY+d), n	(IY+d) ← n	•	•	•	•	•	•	11 111 101	4	5	19		
								00 110 110					
								← d →					
								← n →					
LD A, (BC)	A ← (BC)	•	•	•	•	•	•	00 001 010	1	2	7		
LD A, (DE)	A ← (DE)	•	•	•	•	•	•	00 011 010	1	2	7		
LD A, (nn)	A ← (nn)	•	•	•	•	•	•	00 111 010	3	4	13		
								← n →					
								← n →					
LD (BC), A	(BC) ← A	•	•	•	•	•	•	00 000 010	1	2	7		
LD (DE), A	(DE) ← A	•	•	•	•	•	•	00 010 010	1	2	7		
LD (nn), A	(nn) ← A	•	•	•	•	•	•	00 110 010	3	4	13		
								← n →					
								← n →					
LD A, I	A ← I	•	↕	IFF	↕	0	0	11 101 101	2	2	9		
								01 010 111					
LD A, R	A ← R	•	↕	IFF	↕	0	0	11 101 101	2	2	9		
								01 011 111					
LD I, A	I ← A	•	•	•	•	•	•	11 101 101	2	2	9		
								01 000 111					
LD R, A	R ← A	•	•	•	•	•	•	11 101 101	2	2	9		
								01 001 111					

Notes: r, r' means any of the registers A, B, C, D, E, H, L

IFF the content of the interrupt enable flip-flop (IFF) is copied into the P/V flag

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,

↕ = flag is affected according to the result of the operation.

Aufgabe:

Sie haben nun die 8-Bit Ladebefehle kennengelernt. Als Anwendung werden wir mit Hilfe dieser Befehle einige Speicherstellen des Betriebssystems von Maschinensprache aus ändern.

Betrachten wir zunächst die Speicherstelle &B289 (664:&B72A / 6128:&B72A). An dieser Adresse ist die 1te linke Spalte des aktuellen Bildschirmwindows, die vom BASIC aus mit dem >WINDOW 10,80,1,25< Befehl eingestellt werden kann, abgespeichert. Wir wollen die linke Bildschirmbreite auf 10 setzen. Vom BASIC aus können wir dies mit >WINDOW 10,80,1,25< erreichen. Es besteht aber auch die Möglichkeit, mit einem >POKE< Befehl den gewünschten Wert direkt in die Speicherstelle &HB289 (664: &B72A / 6128:&B72A) zu schreiben.

Probieren Sie also >POKE &B289,10< (664:>POKE &B72A,10< / 6128: >POKE &B72A,10<) und die linke Bildschirmspalte ist die 10te.
Versuchen Sie den BASIC Befehl

>POKE &B289,10< (664: >POKE &B72A,10< / 6128: >POKE &B72A,10<)

in Maschinensprache zu schreiben. Beenden Sie Ihr Programm mit RET Befehl (Code &C9).

Diskussion des Lösungsweges

Analysieren wir zuerst die gestellte Aufgabe:
Eine Speicherstelle (Adresse &B289 (664: &B72A/ 6128: &B72A)) soll mit einem 1-Byte Wert, also einer Zahl zwischen 0 und 255 geladen werden. Da der Wert eine 1-Byte Zahl ist, benutzen wir einen 8-Bit Ladebefehl.
Jetzt gibt es verschiedene Lösungsmöglichkeiten, die von der gewählten Adressierungsart abhängen. Um eine Speicherstelle mit einem Byte zu laden, muß natürlich erst einmal die Adresse

der Speicherstelle festgelegt werden. Die naheliegendste Möglichkeit ist die absolute Adressierung, bei der die Adresse komplett angegeben wird:

```
LD (&B289),A  (absolut adressiert) für CPC 464
LD (&B72A),A  (absolut adressiert) für CPC 664
LD (&B72A),A  (absolut adressiert) für CPC 6128
```

d.h.: Lade die Speicherstelle &B289 (664: &B72A / 6128: &B72A) mit dem Wert, der im Akku enthalten ist. Das bedeutet, daß der Akku vorher mit dem von uns gewünschtem Wert geladen werden muß:

LD A,10 (unmittelbar adressiert)

Beide Befehle hintereinander ausgeführt erzielen das gewünschte Ergebnis.

```
LD A,10
LD (&B289),A (664: LD (&B72A),A /
              6128: LD (&B72A),A)
RET
```

Der RET Befehl muß am Ende eines jeden Maschinenprogramms stehen. Er bewirkt, daß die Programmausführung im BASIC fortgesetzt wird.

Bisher haben wir das Maschinenprogramm nur in Assembler, d.h. in symbolischer Schreibweise dargestellt. Damit der Prozessor unsere Befehle ausführen kann, müssen diese zuvor noch in Zahlen übersetzt werden. Aus den voranstehenden Befehlslisten oder aus den Tabellen am Ende des Buches, erhalten Sie für den unmittelbar adressierten 8-Bit Ladebefehl LD A,10 die Codes &3E,10, wobei der zweite Code der von uns gewünschte Wert ist. Für den absolut adressierten Befehl LD (&B289),A (664: LD (&B72A),A / 6128: LD (&B72A),A) erhalten wir &32, &89, &B2 (664: &32, &2A, &B7 / 6128: &32, &2A, &B7). Auch hier

stellen die beiden letzten Codes die von uns vorgegebene Adresse dar. Beachten Sie, daß zuerst das Low Byte und dann das High Byte angegeben wird.
Der Code für RET ist, wie oben schon erwähnt, &C9.

Aus diesen Codes ergibt sich nun die DATA Zeile für den BASIC Lader im Kapitel 3:

```
DATA &3E,10,&32,&89,&B2,&C9 für CPC 464
DATA &3E,10,&32,&2A,&B7,&C9 für CPC 664
DATA &3E,10,&32,&2A,&B7,&C9 für CPC 6128
```

Unser Programm ist 6 Bytes lang, d.h. die FOR-NEXT Schleife muß von &A000 bis &A005 laufen. Nachdem wir den BASIC Lader mit >RUN< ausgeführt haben, steht nun unser erstes Maschinenprogramm im Speicher. Sie können nun Ihr erstes Programm starten indem Sie >CALL &A000< eingeben. Wenn Sie alles richtig gemacht haben, sollte sofort der Cursor auf die 10te Bildschirmspalte springen. Soll die linke Spalte wieder auf einen anderen Wert gesetzt werden, so braucht nur die 2te Zahl in der DATA Zeile (die "10") geändert werden, das Programm durch >RUN< erneut in den Speicher geschrieben und mit CALL &A000 aufgerufen werden.

Probieren Sie auch folgende Adressen:

464	664	6128	
&B288	&B729	&B729	oberste Zeile
&B28A	&B72B	&B72B	rechteste Spalte
&B28B	&B72C	&B72C	unterste Zeile

4.2 16-Bit Transferbefehle

Auch die 16-Bit Ladebefehle haben das allgemeine Format:

LD Ziel,Quelle

Jedoch werden hierbei 16-Bit übertragen. Damit werden durch diese Befehle die Registerpaare BC,DE,HL,SP,IX und IY angesprochen.

Unmittelbare Adressierung

Da hier nun 16-Bit Register geladen werden, muß die Konstante, die auf den Opcode folgt, 16-Bit lang sein. Daher enthalten die zwei auf den Opcode folgenden Bytes das Low und High Byte der Konstante (in dieser Reihenfolge!). Im Gegensatz zur unmittelbaren Adressierung mit 1-Byte Konstanten, nennt man diese Technik die unmittelbar erweiterte Adressierung (engl. immediately extended).

Format:

LD x,nn
(x: Eines der 16-Bit Register SP,BC,DE,HL,IX,IY)
(nn: 16-Bit Konstante)

Durch diesen Befehl wird Register x mit der Konstanten nn geladen.

Beispiel:
LD HL,&C000 BASIC: HL=&C000

Implizite Adressierung

Bei den 16-Bit Ladebefehlen gibt es nur drei Befehle dieser Art, die alle das SP Register betreffen:

LD SP,HL LD SP,IX LD SP,IY

Diese Befehle bedeuten:
Laden des Stapelzeigers mit dem Inhalt des HL, IX bzw. IY Registers.

BASIC Analog:
SP=HL SP=IX SP=IY

Absolute Adressierung

Die absolute Adressierung bei den 16-Bit Befehlen müssen wir wieder etwas genauer besprechen:

Format:

```
LD x,(nn)    oder LD (nn),x
(x: BC,DE,HL,SP,IX oder IY)
```

Da nn auf eine Adresse zeigt, also nur ein Byte adressiert, x jedoch ein 16-Bit Register ist, hat man folgende Vereinbarung getroffen:
Zuerst wird das Low Byte an der Adresse nn, dann das High Byte an der Adresse nn+1 in das Register geladen.

```
z.B.: LD HL,(&AB80) bedeutet:
      L Register = Low  Byte aus Adresse &AB80
      H Register = High Byte aus Adresse &AB81
```

Bei dem umgekehrten Befehl der Form LD (nn),x wird entsprechend das Low Byte in Adresse nn abgespeichert und das High Byte in Adresse nn+1.

```
z.B. LD (&CB00),IX
Adresse &CB00 = Low  Byte von IX
Adresse &CB01 = High Byte von IX
```

Ein Befehl dieser Art entspricht also zwei 8-Bit Ladebefehlen.

```
16-Bit Befehl:                 8-Bit Befehle:
LD BC,(&FC05)  entspricht      LD C,(&FC05) (Low  Byte)
                               LD B,(&FC06) (High Byte)
```

Wie Sie wissen, kann man eine 16-Bit Zahl aus High Byte und Low Byte in folgender Weise darstellen:

Zahl=256*(High Byte)+(Low Byte)

Damit ergibt das Basicäquivalent zu:

```
Maschinesprache:        BASIC:
LD DE,(&4000)           DE=256*PEEK(&4001)+PEEK(&4000)
```

Machen Sie sich klar, daß man unter Verwendung des Hexadezimalsystems auch folgendes schreiben kann:

DE=VAL("&"+HEX$(PEEK(&4001))+HEX$(PEEK(&4000)))

Um den umgekehrten Befehl, also z.B. LD (&6800),IY im BASIC zu schreiben, braucht man zwei Befehle:

```
POKE &6800, IY-INT (IY/256)*256    (Low Byte)
POKE &6801,     INT (IY/256)       (High Byte)
```

Falls Ihnen diese Analogien nicht klar sind, sehen Sie sich noch einmal das Kapitel über Zahlendarstellungen an. Setzen Sie dann für DE und IY jedesmal Zahlen ein, und führen Sie die Berechnungen selbstständig durch!
Die Befehlslisten in diesem Buch sind alle von der Firma Zilog für uns zur Verfügung gestellt worden.
Sie geben eine Übersicht über alle verfügbaren Befehle. Für jede Gruppe von Befehlen werden wirjeweils die dazugehörige Tabelle abdrucken. Benutzen Sie die Tabellen, um die für die Beispiele benötigten Opcodes nachzuschlagen.

Mnemonic	Symbolic Operation	Flags						Op-Code	No. of Bytes	No. of M Cycles	No. of T States	Comments
		C	Z	P/V	S	N	H	76 543 210				
LD dd, nn	dd ← nn	•	•	•	•	•	•	00 dd0 001	3	3	10	dd Pair
								← n →				00 BC
								← n →				01 DE
LD IX, nn	IX ← nn	•	•	•	•	•	•	11 011 101	4	4	14	10 HL
								00 100 001				11 SP
								← n →				
								← n →				
LD IY, nn	IY ← nn	•	•	•	•	•	•	11 111 101	4	4	14	
								00 100 001				
								← n →				
								← n →				
LD HL, (nn)	H ← (nn+1)	•	•	•	•	•	•	00 101 010	3	5	16	
	L ← (nn)							← n →				
								← n →				
LD dd, (nn)	dd_H ← (nn+1)	•	•	•	•	•	•	11 101 101	4	6	20	
	dd_L ← (nn)							01 dd1 011				
								← n →				
								← n →				
LD IX, (nn)	IX_H ← (nn+1)	•	•	•	•	•	•	11 011 101	4	6	20	
	IX_L ← (nn)							00 101 010				
								← n →				
								← n →				
LD IY, (nn)	IY_H ← (nn+1)	•	•	•	•	•	•	11 111 101	4	6	20	
	IY_L ← (nn)							00 101 010				
								← n →				
								← n →				
LD (nn), HL	(nn+1) ← H	•	•	•	•	•	•	00 100 010	3	5	16	
	(nn) ← L							← n →				
								← n →				
LD (nn), dd	(nn+1) ← dd_H	•	•	•	•	•	•	11 101 101	4	6	20	
	(nn) ← dd_L							01 dd0 011				
								← n →				
								← n →				
LD (nn), IX	(nn+1) ← IX_H	•	•	•	•	•	•	11 011 101	4	6	20	
	(nn) ← IX_L							00 100 010				
								← n →				
								← n →				
LD (nn), IY	(nn+1) ← IY_H	•	•	•	•	•	•	11 111 101	4	6	20	
	(nn) ← IY_L							00 100 010				
								← n →				
								← n →				
LD SP, HL	SP ← HL	•	•	•	•	•	•	11 111 001	1	1	6	
LD SP, IX	SP ← IX	•	•	•	•	•	•	11 011 101	2	2	10	
								11 111 001				
LD SP, IY	SP ← IY	•	•	•	•	•	•	11 111 101	2	2	10	
								11 111 001				qq Pair
PUSH qq	(SP-2) ← qq_L	•	•	•	•	•	•	11 qq0 101	1	3	11	00 BC
	(SP-1) ← qq_H											01 DE
PUSH IX	(SP-2) ← IX_L	•	•	•	•	•	•	11 011 101	2	4	15	10 HL
	(SP-1) ← IX_H							11 100 101				11 AF
PUSH IY	(SP-2) ← IY_L	•	•	•	•	•	•	11 111 101	2	4	15	
	(SP-1) ← IY_H							11 100 101				
POP qq	qq_H ← (SP+1)	•	•	•	•	•	•	11 qq0 001	1	3	10	
	qq_L ← (SP)											
POP IX	IX_H ← (SP+1)	•	•	•	•	•	•	11 011 101	2	4	14	
	IX_L ← (SP)							11 100 001				
POP IY	IY_H ← (SP+1)	•	•	•	•	•	•	11 111 101	2	4	14	
	IY_L ← (SP)							11 100 001				

Notes: dd is any of the register pairs BC, DE, HL, SP
qq is any of the register pairs AF, BC, DE, HL
$(PAIR)_H$, $(PAIR)_L$ refer to high order and low order eight bits of the register pair respectively.
E.g. BC_L = C, AF_H = A

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
‡ flag is affected according to the result of the operation.

Aufgabe:

Bevor wir zur weiteren Besprechung der Befehle übergehen, wollen wir die bisher gelernten anwenden. Wie Sie wissen, liegt der Bildschirmspeicher der CPC's ab Adresse &C000. In diesem Bereich entsprechen je 8-Bit (ein Byte), acht nebeneinanderliegenden Punkten (in >MODE 2<). Adresse &C000 ist den ersten 8 Punkten, angefangen in der oberen linken Ecke des Bildschirms, zugeordnet. Die 8 darunterliegenden Punkte (=ein Byte) sind an Adresse &C800 abgelegt, die darunterliegenden an Adresse &D000 usw...(in &800er Schritten). Geben Sie einmal ein:

```
10 POKE &C000,&FF
20 POKE &C800,&FF
30 POKE &D000,&FF
40 POKE &D800,&FF
50 POKE &E000,&FF
60 POKE &E800,&FF
70 POKE &F000,&FF
80 POKE &F800,&FF
MODE 2
RUN
```

Wie Sie sehen, ist das obere linke Kästchen mit der aktuellen Farbe gefüllt worden.

(&FF=&X11111111=8 gesetzte Punkte)

Dieses Programm sollen Sie nun mit Hilfe der jetzt gelernten Befehle in Maschinensprache übersetzen. Beenden Sie Ihr Maschinenprogramm mit RET (&C9).

Diskussion des Lösungsweges zum
selbsterstellten Maschinenprogramm

Zunächst brauchen wir einen Befehl , der eine Speicherstelle mit einem Wert lädt (>POKE<). Es kommen hierfür die Befehle mit indirekter, indizierter und absoluter Adressierung in Frage (siehe Definition). Um genau unser BASIC Beispiel zu übersetzen, wählen wir die absolute Adressierung, d.h. wir geben, wie im

BASIC Programm, die Adresse jeweils vollständig an. Es ist natürlich auch möglich, die Adresse in einem Register zu speichern und dann die indirekte oder indizierte Adressierung zu verwenden.

Beispiel für die indirekte Adressierung:

```
           BASIC: HL=&C000:POKE HL,&FF
Maschinensprache: LD HL,&C000 bzw. LD (HL),&FF
```

Da bei den 16-Bit Befehlen immer zwei aufeinanderfolgende Speicherstellen beschrieben werden, wählen wir den 8-Bit Befehl:

LD (nn),A

Vor der Ausführung dieses Befehls muß im Akku noch der Wert &FF gespeichert werden. Hierfür verwendet man die unmittelbare Adressierung:

LD A,&FF

Danach sieht unser Programm folgendermaßen aus:

```
LD A,&FF
LD (&C000),A
LD (&C800),A
LD (&D000),A
LD (&D800),A
LD (&E000),A
LD (&E800),A
LD (&F000),A
LD (&F800),A
RET
```

Nun suchen wir uns die Codes für die entsprechenden Befehle heraus:

```
LD   A,n    : &3E,n
LD  (nn),A : &32,nl,nh   :l: Low, h: High
RET         : &C9
```

Damit ergeben sich die DATA Zeilen unseres BASIC Laders von Kapitel 3.1 zu:

```
10 MEMORY &9FFF
20 FOR i=&A000 TO &A01A
30 READ a
40 POKE i,a
50 NEXT i
60 END
60 DATA &3E,&FF,&32,&00,&C0,&32,&00,&C8
70 DATA &32,&00,&D0,&32,&00,&D8,&32,&00,&E0
80 DATA &32,&00,&E8,&32,&00,&F0,&32,&00,&F8
90 DATA &C9
```

Wir wollen dieses Programm ab Adresse &A000 (=Startadresse (V)) speichern. Unser Programm ist 27 Bytes lang. Daraus läßt sich die Endadresse (V) zu &A000 + 27-1=&A000+&1A=&A01A berechnen. Also lautet Zeile 20:
20 FOR I=&A000 TO &A01A
Nachdem das Maschinenprogramm durch >RUN< in den Speicher "gepoked" wurde, kann es nach Eingabe von >MODE 2< mit >CALL &A000< gestartet werden. Wie Sie sehen, färbt sich augenblicklich das linke obere Feld im Bildschirm. Sie können dieses Programm mit dem Direktlader eingeben. Dazu starten Sie den Direktlader und geben die Startadresse &A000 ein. Darauf folgend die Codes (z.B.&3E,&FF,usw.).
Das war nun Ihr erstes eigenes Maschinenprogramm. Sie werden dieses Programm verändern und verbessern können, sobald Sie einige neue Befehle kennengelernt haben.

4.3 STAPELBEFEHLE

Zum Verständnis der Funktionsweise des Stapels, ist es notwendig zu wissen, was im Inneren des Z80A abläuft, wenn in ein Unterprogramm gesprungen wird. Der dazu nötige Assemblerbefehl lautet >CALL adresse<. Das grundsätzliche Problem ist, daß die CPU sich die Adresse des nächstfolgenden Befehls "merken" muß, da bei einem Rücksprung ins Hauptprogramm (RET) die Programmausführung dort fortgesetzt wird.

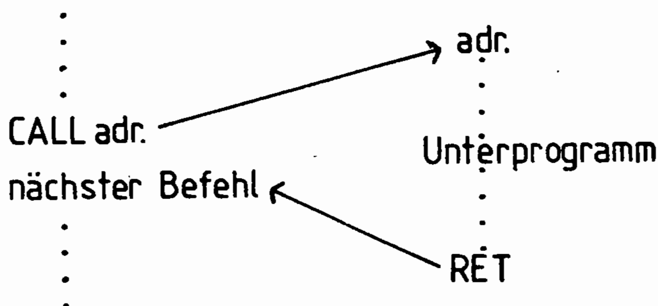


Abb 5 Unterprogramm Aufruf

Da die Register für andere wichtige Aufgaben gebraucht werden, müssen die Rücksprungadressen außerhalb der CPU, also im RAM, gespeichert werden. Mit diesem Verfahren könnte jedoch nur eine Rücksprungadresse gespeichert werden. Das bedeutet, daß eine Verschachtelung von Unterprogrammen nicht möglich wäre. Das ist der Grund dafür, warum ein Bereich des RAM für diese Aufgabe reserviert wird. Diesen Bereich nennt man Stack oder Stapel. Stellen wir uns diesen Stapel als einen Stapel Teller vor:

Eine Rücksprungadresse wird durch das Notieren auf einem Teller gespeichert. Der so "adressierte" Teller wird auf den Stapel gelegt. So können viele Unterprogrammaufrufe stattfinden, der Stapel wird dadurch einfach höher. Bei einem Rück-

sprung wird nun der oberste Teller genommen und an die auf ihm stehende Adresse verzweigt. Auf diese Weise wird in der richtigen Reihenfolge solange zurückgesprungen, bis der Tellerstapel abgebaut ist, d.h. man befindet sich wieder im Hauptprogramm. Wichtig ist, daß immer der Teller, der zuletzt auf den Stapel gelegt wurde, auch als erstes wieder heruntergenommen wird (sonst kippt der Stapel um).

Da im Computer keine Teller gestapelt werden, muß ein Register des Z80A als "Höhenmesser" des Stapels benutzt werden. Dieses Register zeigt immer auf den letzten Teller im Stapel. Es wird Stack Pointer (SP) genannt. Allerdings "hängt" unser Stapel im Computer von der Decke, d.h. der erste Teller wird an der höchsten und der letzte Teller an der niedrigeren Adresse im Stack abgelegt. Dieser Bereich (des Stacks) liegt beim Schneider ab &BFFF abwärts.
Damit sieht der Ablauf des CALL Befehls so aus:
Auschnitt aus dem Stapel:

```
Ausgangsposition:           .      .
                            .      .
                            .      .
                     Stapel &BFF4 : (frühere Eintragung)
                            &BFF3 : (frühere Eintragung)
                            &BFF2 : (frühere Eintragung)
                            &BFF1 : (letzte  Eintragung)
                            &BFF0 : (Platz f.neue Eintragungen)
                            &BFEF : (Platz f.neue Eintragungen)
                            .      .
                            .      .

Stack Pointer SP:
   &BFF1
```

Das SP Register zeigt auf die letzte Eintragung im Stack. Nehmen wir an: Bei der Programmabarbeitung stößt der Prozessor auf einen CALL &B267 Befehl an Adresse &780.

&780 CALL &B267
&783 nächster Befehl

Nach dem Einlesen des Befehls steht der PC auf &783. Das ist die zu speichernde Rücksprungadresse. Die Adresse wird als Low Byte und High Byte auf den Stapel gelegt. Also wird SP erniedrigt, das High Byte an Adresse SP gespeichert und SP nochmals erniedrigt und das Low Byte an der neuen Adresse SP gespeichert. Dann wird der PC mit der angegebenen Unterprogrammstartadresse (&B267) geladen, d.h. die Programmausführung wird dort fortgesetzt. Es ergibt sich folgende Konstellation:

```
          .      .
          .      .
          .      .
Stapel: &BFF0   &07
        &BFEF   &83   : (letzte Eintragung)
          .      .
          .      .
SP:&BFEF
```

Wie Sie sehen, zeigt SP wieder auf die letzte Eintragung. Beim RET Befehl läuft der ganze Vorgang nun umgekehrt ab:

Das Byte an der Speicherstelle, auf die SP zeigt, wird als Low Byte in den PC geladen. Der SP wird um eins erhöht und das High Byte der Rücksprungadresse nach PC geladen. Danach wird SP nochmals um eins erhöht, d.h. er zeigt wieder auf die jetzt aktuelle Rücksprungadresse im Stack. Die Programmausführung wird jetzt an Stelle PC fortgesetzt, also an der korrekten Rücksprungadresse.

```
          .      .
          .      .
Stapel: &BFF1   ...        SP: &BFF1
        &BFF0   &07
        &BFEF   &83
          .      .
```

Die beschriebenen Vorgänge laufen automatisch im Z80A ab, sobald ein CALL oder RET erfolgt. Das gewährleistet, daß die Reihenfolge im Stapel immer korrekt ist und SP auf die richtige

Stelle zeigt. Verändern Sie SP direkt vom Programm aus, kann die Reihenfolge leicht durcheinander geraten und der Rechner abstürzen. Verwenden Sie also die LD SP,x Befehle mit Vorsicht.

Zusätzlich können auf dem Stapel auch Daten abgelegt und von dort abgerufen werden. Dazu dienen die Befehle:

```
PUSH (auf den Stapel legen)
und
POP  (vom Stapel holen).
```

PUSH funktioniert analog zum CALL Befehl. Die zu speichernden Daten werden nach dem Erniedrigen des SP auf den Stack geschrieben. Beim POP werden die Daten gelesen und SP automatisch erhöht. Auch hierbei werden sämtliche Operationen durch die CPU übernommen. Mit PUSH und POP können sämtliche 16-Bit Register (-paare), natürlich außer SP selber, "gestapelt" werden.

Format:

```
PUSH qq        POP qq
PUSH IX        POP IX
PUSH IY        POP IY

(qq: BC, DE, HL, AF)
```

Da der Akku immer ein 8-Bit Register ist und es auch sinnvoll ist, das F (Flag) Register auf den Stapel zu retten, werden A und F zusammen behandelt.

Die Technik der Zwischenspeicherung auf dem Stapel ist dann sinnvoll, wenn die Register zur Speicherung nicht mehr ausreichen.

Beispiel:

HL enthält einen Summanden
BC enthält zweiten Summanden

Nun wird ein Unterprogramm aufgerufen, daß HL und BC addiert. Dabei wird das Ergebnis der Addition in HL gespeichert. Wird der erste Summand noch benötigt, so sollte er rechtzeitig auf den Stapel gelegt werden.

```
LD HL,Summand-eins
LD BC,Summand-zwei
PUSH HL
CALL Addresse-Addition
   .
   .
   .
POP HL
   .
   .
```

Wird dieser Summand benötigt, kann er mit POP HL vom Stapel geholt werden.

Zu beachten ist, daß der zu einem PUSH gehörende POP Befehl immer im selben Unterprogramm stehen muß. Sonst werden die durch PUSH gespeicherten Daten als die Rücksprungadresse für den RET Befehl interpretiert, was aller Wahrscheinlichkeit nach zum Absturz des Rechners führt. Der PUSH bzw. POP Befehl besitzt keinen direkt ähnlichen Befehl im Schneider BASIC. Diese Befehle können im BASIC folgendermaßen geschrieben werden.

BASIC Beispiel:

```
PUSH AF        BASIC: POKE SP-1,A:(High Byte)
                      POKE SP-2,F
                      SP=SP-2

POP BC         BASIC: BC=PEEK(SP)+256*PEEK(SP+1)
                      SP=SP+2
```

Da PUSH und POP SP als Adresszeiger benutzen, zählen sie zur indirekten Adressierung.

Beispiel:

```
PUSH HL                     SP=&BE05
                            HL=&1234

Nach der Ausführung:  Speicherstelle &BE04:&12
                      Speicherstelle &BE03:&34

Sp = &BE03
HL = &1234
```

Beispiel:

```
POP HL                      SP=&BE03
                            HL=&FFFF

Nach der Ausführung:

SP = &BE05
HL = &1234
```

Die Befehlsliste zu den Stapelbefehlen befindet sich am Ende des Kapitels 4.2: 16-Bit Ladebefehle

4.4 Austauschbefehle

Beim Z80A gibt es neben den Befehlen zur einfachen Datenübertragung (LD) auch einen Befehl, der den Inhalt zweier Plätze miteinander vertauscht. Diese Befehle werden durch EX (engl. exchange: vertauschen) dargestellt.

Ein Befehl dieser Art, EX DE,HL, vertauscht z.B. den Inhalt des DE mit dem des HL Registers. Der EX Befehl mit indirekter Adressierung vertauscht den Inhalt des HL, IX oder IY Registers mit dem obersten Stapelelement (SP bleibt dabei gleich).

Format:

```
EX (SP),HL
EX (SP),IX
EX (SP),IY
```

Weiterhin gibt es Austauschbefehle, die mit dem Inhalt des Zweitregistersatzes vertauschen. Wie schon erwähnt, gibt es zu jedem der Register A, BC, DE, HL, F ein entsprechendes Register A', BC', DE', HL' und F'. Gearbeitet wird jeweils nur mit einem Registersatz. Bei Bedarf kann nun der Inhalt der beiden Sätze miteinander vertauscht werden.
Die Zweitregistersätze können bei den Schneider Rechnern leider nicht so ohne weiteres benutzt werden, da sie intern vom Computer gebraucht werden.

Der Befehl EX AF,AF' vertauscht den Inhalt des Akkus und den des Flagregisters mit den entsprechenden Registern A' und F'. Der EXX Befehl vertauscht die anderen Registerpaare BC, DE und HL jeweils mit BC', DE' und HL'.
Diese Befehle sind implizit adressiert.
Beispiel:

```
EX DE,HL       BASIC:ZWI=HL:HL=DE:DE=ZWI
EX (SP),HL     BASIC:ZWI=HL:HL=256*PEEK(SP+1)+PEEK(SP):
                     POKE SP+1,INT(ZWI/"X&"):POKE SP,
                     ZWI-INT(ZWI/256)*256
```

Mnemonic	Symbolic Operation	C	Z	P/V	S	N	H	Op-Code 76 543 210	No. of Bytes	No. of M Cycles	No. of T States	Comments
EX DE, HL	DE ↔ HL	•	•	•	•	•	•	11 101 011	1	1	4	
EX AF, AF'	AF ↔ AF'	•	•	•	•	•	•	00 001 000	1	1	4	
EXX	(BC, DE, HL) ↔ (BC', DE', HL')	•	•	•	•	•	•	11 011 001	1	1	4	Register bank and auxiliary register bank exchange
EX (SP), HL	H ↔ (SP+1) L ↔ (SP)	•	•	•	•	•	•	11 100 011	1	5	19	
EX (SP), IX	IX_H ↔ (SP+1) IX_L ↔ (SP)	•	•	•	•	•	•	11 011 101 11 100 011	2	6	23	
EX (SP), IY	IY_H ↔ (SP+1) IY_L ↔ (SP)	•	•	•	•	•	•	11 111 101 11 100 011	2	6	23	
LDI	(DE) ← (HL) DE ← DE+1 HL ← HL+1 BC ← BC-1	•	•	↕ ①	•	0	0	11 101 101 10 100 000	2	4	16	Load (HL) into (DE), increment the pointers and decrement the byte counter (BC)
LDIR	(DE) ← (HL) DE ← DE+1 HL ← HL+1 BC ← BC-1 Repeat until BC = 0	•	•	0	•	0	0	11 101 101 10 110 000	2 2	5 4	21 16	If BC ≠ 0 If BC = 0
LDD	(DE) ← (HL) DE ← DE-1 HL ← HL-1 BC ← BC-1	•	•	↕ ①	•	0	0	11 101 101 10 101 000	2	4	16	
LDDR	(DE) ← (HL) DE ← DE-1 HL ← HL-1 BC ← BC-1 Repeat until BC = 0	•	•	0	•	0	0	11 101 101 10 111 000	2 2	5 4	21 16	If BC ≠ 0 If BC = 0
CPI	A - (HL) HL ← HL+1 BC ← BC - 1	•	↕ ②	↕ ①	↕	1	↕	11 101 101 10 100 001	2	4	16	
CPIR	A - (HL) HL ← HL+1 BC ← BC-1 Repeat until A = (HL) or BC = 0	•	↕ ②	↕ ①	↕	1	↕	11 101 101 10 110 001	2 2	5 4	21 16	If BC ≠ 0 and A ≠ (HL) If BC = 0 or A = (HL)
CPD	A - (HL) HL ← HL-1 BC ← BC-1	•	↕ ②	↕ ①	↕	1	↕	11 101 101 10 101 001	2	4	16	
CPDR	A - (HL) HL ← HL-1 BC ← BC-1 Repeat until A = (HL) or BC = 0	•	↕ ②	↕ ①	↕	1	↕	11 101 101 10 111 001	2 2	5 4	21 16	If BC ≠ 0 and A ≠ (HL) If BC = 0 or A = (HL)

Notes: ① P/V flag is 0 if the result of BC-1 = 0, otherwise P/V = 1
② Z flag is 1 if A = (HL), otherwise Z = 0.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

4.5 Blocktransfer- und Blocksuchbefehle

Die Blocktransferbefehle übertragen, nicht wie LD, nur ein oder zwei Bytes, sondern einen ganzen Block von Daten. Sie stellen eine Besonderheit des Z80A dar. Üblicherweise sind diese Befehle nicht in Mikroprozessoren verfügbar, da sie für den Hersteller recht aufwendig zu realisieren sind. Für den Programmierer hingegen sind diese Befehle sehr nützlich. Sie erhöhen die Leistungsfähigkeit eines Programms.

Ein Block von Daten wird durch folgende Angaben charakterisiert:

- Die Anfangsadresse oder Endadresse des Blocks. Sie wird in HL gespeichert.
- Die Länge des Blocks in Bytes. Sie wird in BC (Byte Counter) gespeichert.

Mit diesen beiden Größen ist es möglich, Blöcke von bis zu 64K Länge, die an beliebiger Stelle (HL) im Speicher beginnen, zu definieren. Da der so definierte Block übertragen werden soll, muß noch die Anfangs- bzw. Endadresse des Zielblocks angegeben werden. Sie wird in DE gespeichert. Nachdem diese Daten in den Registern abgelegt wurden, kann der eigentliche Blocktransferbefehl erfolgen.

Es gibt vier Blocktransferbefehle:

LDD, LDDR, LDI, LDIR

Jeder Blocktransferbefehl decrementiert (erniedrigt) den Zähler BC nach jeder Übertragung eines Bytes. Zwei von ihnen, LDI und LDIR, incrementieren (erhöhen) dann die Zeiger HL und DE, die dann auf Quell- und Zieladresse des nächsten zu übertragenen Bytes zeigen.

Bei LDD und LDDR werden im Gegensatz dazu die Zeiger decrementiert , d.h. der Block wird sozusagen "von oben angefangen" übertragen. Für diese Befehle müssen HL und DE anfangs

natürlich auch mit der Quell- bzw. Zielendadresse des Blocks geladen werden. Das R am Ende der Befehle steht für Repeat (engl.: wiederhole). Diese Befehle werden automatisch solange wiederholt, bis BC=0 ist, d.h. bis der gesamte Block übertragen ist. Im Einzelnen gilt für die Befehle folgendes:

LDI : Lade und (I)ncrementiere

Dieser Befehl überträgt ein Byte von Adresse HL nach Adresse DE. Danach wird BC decrementiert. Die Adresszeiger HL und DE werden incrementiert, so daß alles für eine eventuelle Fortsetzung der Übertragung vorbereitet ist. Dazu muß dann dieser Befehl wieder angesprungen werden.

LDIR: Lade, incrementiere und wiederhole

Der Vorgang der Übertragung läuft wie bei LDI ab. Danach wird zusätzlich der PC automatisch wieder auf diesen Befehl gesetzt. Dann wird er erneut ausgeführt, solange bis BC=0 ist. Anschließend wird mit dem nächsten Befehl die Programmabarbeitung wieder aufgenommen.

LDD : Lade und (D)ecrementiere

Ähnlich wie bei LDI, nur wird der Block bei der Endadresse angefangen übertragen, d.h. HL und DE werden decrementiert. Wichtig ist dieser Unterschied, wenn sich Ziel- und Quellblock überschneiden. Benutzt man hier den falschen Befehl, würden unter Umständen Daten des Quellblocks vor ihren Übertragungen schon überschrieben werden.

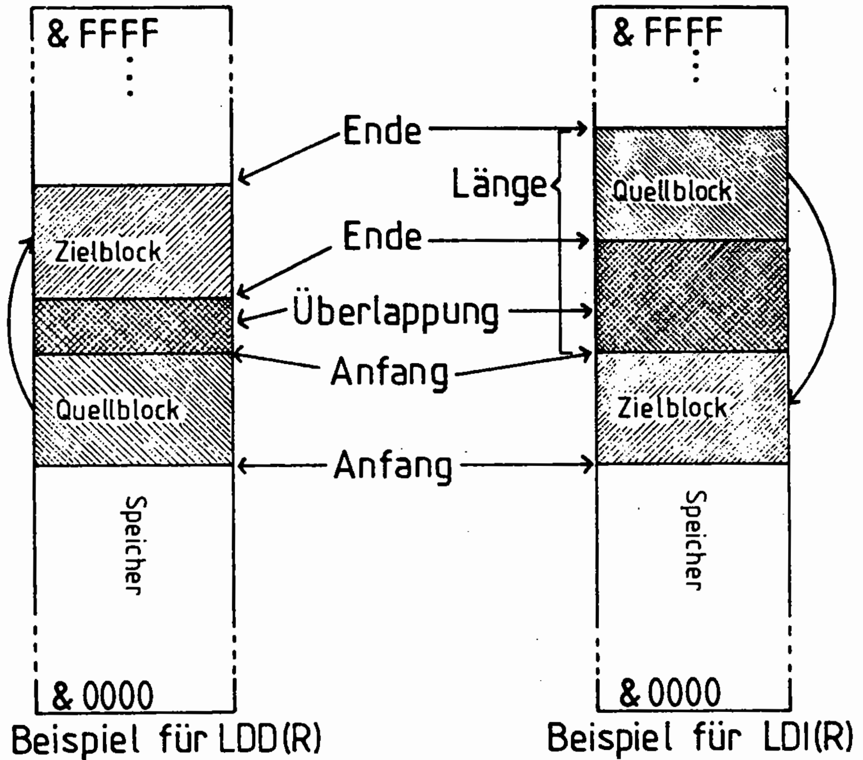


Abb. 6 Blocktransferbefehle

LDDR: Lade, decrementiere und wiederhole

Ähnlich wie LDD, nur daß, wie bei LDIR der Befehl wiederholt wird, bis der gesamte Block übertragen ist.

Beispiel:

```
LDIR        BASIC: 10 POKE DE,PEEK(HL)
                   20 HL=HL+1
                   30 DE=DE+1
                   40 BC=BC-1
                   50 IF BC<>0 THEN 10

LDD         BASIC: POKE DE,PEEK(HL):
                   DE=DE-1:HL=HL-1:BC=BC-1
```

Überlegen Sie sich die BASIC Analogie zu LDDR und LDI.

An der Größe des BASIC Programms können Sie sehen, daß es sich um einen sehr leistungsstarken Befehl handelt.

Flagbeeinflussung: Wenn BC nach der Ausführung =0 ist, ist P/V=0.
Die Repeatbefehle LDDR und LDIR setzen das P/V Flag immer auf 1.

Blocksuchbefehle

Mit Hilfe der Blocksuchbefehle kann ein Datenblock nach einem bestimmten Byte durchsucht werden. Das gesuchte Byte wird vorher im Akku gespeichert. Trifft der Befehl während der Suche auf ein Byte, das gleich dem Akkuinhalt ist, wird das Z Flag gesetzt, und die Repeatbefehle werden nicht mehr wiederholt. Die Register werden wie bei den Blocktransferbefehlen benutzt.

```
HL- Start bzw. Endadresse des Blocks
BC- Byte Counter: Länge des Blocks
DE- hat keine Funktion.
 A- Der Akku enthält das zu suchende
    Byte.
```

CPIR vergleicht bei jedem Durchlauf den Inhalt der Speicherstelle HL mit dem Akkuinhalt. Dann wird HL incrementiert und

BC decrementiert. Ist BC=0, wird das P/V Flag auf 0 gesetzt, ansonsten auf eins. Liegt beim Vergleich von A und (HL) Gleichheit vor, wird das Z Flag gesetzt, sonst rückgesetzt.

Das S Flag entspricht, wie bei CP, dem 7ten Bit des Ergebnisses der Subtraktion A-(HL). Das Carry wird nicht beeinflußt. Vier Blocksuchbefehle sind möglich:

CPI, CPIR, CPD, CPR

Ihre Funktionsweise ist denen der jeweiligen Blocktransferbefehle entsprechend.
Alle Blockbefehle sind 2 Byte Befehle, ihr erstes Opcode Byte ist &ED. Wie auch durch die Blocktransferbefehle wird mit den Suchbefehlen die Programmierung in vielen Bereichen einfacher und schneller.

Die Befehlslisten zu den Blocktransfer- und Blocksuchbefehlen befindet sich am Ende des vorigen Kapitels.

Aufgabe

Um den Befehl LDDR vollständig zu verstehen, werden wir ihn gleich ausprobieren. Wir wollen den Bildschirminhalt um ein Zeichen nach rechts verschieben. Da ein Byte genau der Breite eines Zeichens entspricht, müssen wir also den Block von &C000 bis &FFFF um ein Byte nach oben verschieben.
Schreiben Sie hierfür mit Hilfe der Blocktransferbefehle ein Maschinenprogramm.

Lösung

Analysieren wir zunächst unser Problem: Der Quellblock liegt im Bereich &C000- &FFFE.

Dieser Block soll um ein Byte nach oben verschoben werden, also in den Bereich &C001-&FFFF. Die beiden Blöcke überlappen sich offensichtlich. Da die Endadresse des Quellblocks &FFFE überlappt ist, muß der LDDR Befehl gewählt werden.

Berechnen wir nun die Registerinhalte von HL, DE und BC. HL soll die Endadresse des Quellblocks, also &FFFE, enthalten. BC enthält die Anzahl der zu verschiebenden Bytes. Sie beträgt &4000-1 (Der Bildschirmbereich von &C000-&FFFF ist &4000 Bytes groß) also: BC=&3FFF. DE enthält die Endadresse des Zielblockes, also &FFFF.
Damit ergibt sich das folgende Assemblerprogramm:

```
LD HL,&FFFE
LD DE,&FFFF
LD BC,&3FFF
LDDR
RET
```

Nach der Übersetzung dieses Programmes ergeben sich die DATA Zeilen des BASIC Laders zu:

```
DATA &21,&FE,&FF,&11,&FF,&FF
DATA &01,&FF,&3F,&ED,&B8
DATA &C9
```

(Startadresse ist &A000 und Endadresse ist &A00B).
Geben Sie nun >MODE 2< ein, laden Sie das Maschinenprogramm mit >RUN< und starten es mit >CALL Adresse<.

Unser Programm hat einen kleinen Schönheitsfehler:
Das linke obere Kästchen enthält oben einen Punkt. Damit dieser verschwindet, laden wir die entsprechende Speicherstelle &C000 mit 0.

```
LD A,00
LD(&C000),A
Code: &3E,&00,&32,&00,&C0
```

Diese Befehle fügen wir nach dem LDDR Befehl ein. Die letzte DATA Zeile lautet dann:

DATA &3E,&00,&32,&00,&C0,&C9

(Die Endadresse ändert sich zu &A010).
Nachdem Sie dieses Programm getestet haben, geben Sie folgendes ein:

FOR I=1 TO 80:CALL &A000:NEXT

Das Ergebnis dieser Anweisung ist, daß der Bildschirm um eine Zeile nach unten geschoben wird. Der Zeitaufwand dafür ist allerdings relativ groß, da die 16K des Bildschirms 80 mal verschoben werden müssen. In BASIC würde diese Verschiebung ca. eine Stunde benötigen. Wenn der Bildschirmblock gleich um 80 Zeichen verschoben wird, wäre die Ausführungszeit 80 mal kleiner. Dazu müssen wir die Registerinhalte in unserem Maschinenprogramm verändern:

HL soll &FFFF-80 an Stelle von &FFFF-1 enthalten, also &FFAF. DE bleibt auf &FFFF

Die Anzahl der zu verschiebenden Bytes ist &4000-80=&3FB0

Ändern Sie die DATA Zeilen entsprechend, und unser Programm schiebt den Bildschirm eine Zeile nach unten. Leider sind jedoch die ersten 80 Bytes des Bildschirmspeichers noch auf ihrem alten Stand. Sie müssen gelöscht werden! Auch hierfür wollen wir den Blocktranferbefehl benutzen. Damit ein Bereich durch ihn gelöscht wird, müssen wir ihn absichtlich falsch benutzen:

Zuerst speichern wir an Stelle &C000 das Nullbyte ab. Nun verschieben wir den Block von &C000 bis &C000+80=&C050 nach &C001. Da sich die Bereiche an der Endadresse des Quellblockes überlappen, müßten wir eigentlich LDDR benutzen.

Nehmen wir jedoch LDIR, HL=&C000, DE=&C001, BC=&4F, so wird immer die Speicherstelle, die als nächstes übertragen wird,

mit dem Wert der gerade übertragenen überschrieben. Da &C000 den Wert 0 hat, haben dann alle Bytes dieses Blocks den Wert Null!

Das komplette Programm hat folgende Form:

Adresse/Code/ BASIC-Zeilennr./Assemblerbefehl

```
A000   21AFFF    10    LD HL,&FFAF
A003   11FFFF    20    LD DE,&FFFF
A006   01B03F    30    LD BC,&EFB0
A009   EDB8      40    LDDR
A00B   3200C0    50    LD (&C000),A
A00E   2100C0    60    LD HL,&C000
A011   1101C0    70    LD DE,&C001
A014   014F00    80    LD BC,&4F
A017   EDB0      90    LDIR
A019   C9       100    RET
```

Erklärung zum Assemblerlisting:

Die Adresse wird fortlaufend nach der Anzahl der Bytes im Code numeriert. Da ein Byte immer durch 2 Hexadezimalziffern angezeigt wird, ergibt sich der zuerst unerklärlich erscheinende Sprung von &A000 zu &A003.

Der Code besteht hier aus 3 Bytes, nämlich aus: &21, &00, &C0. Da jedes Byte die Adresse um den Wert eins erhöht, ist die Anfangsadresse des nächsten Befehls &A003 (&A000+3=&A003). Aus der Anzahl der Codes läßt sich leicht die Befehlslänge ermitteln. Die Assemblerbefehle stehen hinter den Codes.

Wenn Ihnen durch das "Arbeiten" am Computer der Bildschirm gescrollt ist, treten Unregelmäßigkeiten bei dem Ablauf des Maschinenprogramms auf. Dieses Phänomen tritt aber nur dann in Erscheinung, wenn Sie vor dem Aufrufen des Programms nicht mit dem >MODE 2< Befehl den Bildschirm gelöscht haben.
Probieren Sie außerdem einmal folgendes:

FOR I=1 to 26:CALL &A000: NEXT

Eigentlich sollte durch diesen Befehl der gesamte Bildschirm (25 Zeilen) gelöscht sein. Die am unteren Rand verschwindenden Zeilen tauchen jedoch wieder am oberen Rand, in der Mitte der Zeile, auf.

Das liegt einmal an dem Aufbau des Bildschirmspeichers und weiterhin an der Tatsache, daß das eingebaute Scrolling auf andere Weise funktioniert. Wir werden uns mit diesem Problem weiter beschäftigen, sobald wir einige neue Befehle kennengelernt haben.

Probieren Sie mit den Blocktransferbefehlen noch ein wenig herum: Verwenden Sie verschiedene Werte für HL, DE und BC. Achten Sie auf jeden Fall darauf, daß der Zielblock nicht aus dem Bereich von &C000 - &FFFF herausragt. Dies führt zum Absturz des Computers, da Systemroutinen überschrieben werden.
Auch Folgendes ist einen Versuch wert:

HL=&C000, DE=&FFFF , BC=&3FFE

4.6 Arithmetische Befehle

Die ersten, in den 50er Jahren entstandenen Digitalcomputer, waren vorrangig als Rechenmaschinen ausgelegt. Obwohl die damaligen Computer mit den heutigen nur noch wenig gemeinsam haben, sind die Befehle zur Arithmetik ähnlich. Es gibt zwei grundsätzliche arithmetische Operationen, Addition und Subtraktion, die den Maschinenbefehlen ADD und SUB entsprechen. Da der Computer im Dualsystem rechnet, sehen wir uns zunächst an, wie diese Rechenoperationen in diesem Zahlensystem durchgeführt werden.

Addition:

Beim Dezimalsystem addiert man zwei übereinanderstehende Ziffern. Die Einerstelle des Ergebnisses wird notiert und eventuell auftretende Zehnerstellen (der Übertrag) werden für die Addition der nächsten Ziffern gemerkt.

Beispiel:

```
   3573
+  7154      (* Hier mußten Sie sich bei der Addition
 ------       eine 1 merken. Diese Ziffer entspricht
  10727       dem Übertrag.)
  *  *
```

Ein Übertrag entsteht, sobald die Summe zweier Ziffern größer als 9 (10-1) ist. Im Dualsystem entsteht ein Übertrag, wenn die Summe zweier Ziffern größer als 1 (2-1) ist.

Regeln:

```
0 + 1 = 1
1 + 0 = 1
0 + 0 = 1
1 + 1 = 0 <--(bei der letzten Rechnung
               müssen Sie sich einen merken !)
```

Anwendung:

```
 1 0 0 1 0 1 1 0  = &96 = 150
+0 0 1 1 1 0 0 1  = &39 =  57
 ---------------    ---   ---
 1 1 0 0 1 1 1 1  = &CF = 207
   * *
(* bedeutet: 1 gemerkt !!)
```

Im Hexadezimalsystem gilt ähnliches (s.o.):

Ein Übertrag entsteht, wenn das Ergebnis größer als 15 ist.

```
  0 0 1 0 1 1 1 0  = &2E =  46
+ 0 0 0 1 0 1 1 1  = &17 =  23
  ---------------    ---    --
  0 1 0 0 0 1 0 1  = &45 =  69
```

&E+&7= 14+7= 21 =&15

d.h.: 5 notieren, 1 gemerkt!

Außerdem ist im obigen Beispiel bei der Binäraddition noch ein Fall dazugekommen:

```
 11
+11
---
110
```

Bei der zweiten Stelle gilt folgende Regel:

1 + 1 + 1 = 1, und 1 gemerkt!

Aufgaben:

```
1)    1 0 1 0 1 1 1 0  = & ?  =  ?
    + 0 0 1 0 1 1 1 1  = & ?  =  ?
      ---------------    ---    --
             ?         = & ?  =  ?
```

```
2)    0 0 1 1 1 1 1 1   = & ?   =  ?
    + 0 0 1 0 1 1 1 1   = & ?   =  ?
      ---------------     ---     ---
             ?          = & ?   =  ?


3)    1 1 1 1 1 1 1 1   = & ?   =  ?
    + 1 1 0 0 1 0 1 0   = & ?   =  ?
      ---------------     ---     ---
             ?          = & ?   =  ?
```

Lösung:

```
1)    1 0 1 0 1 1 1 0   = &AE   = 174
    + 0 0 1 0 1 1 1 1   = &2F   =  47
      ---------------     ---     ---
      1 1 0 1 1 1 0 1   = &DD   = 221


2)    0 0 1 1 1 1 1 1   = &3F   =  63
    + 0 0 1 0 1 1 1 1   = &2F   =  47
      ---------------     ---     ---
      0 1 1 0 1 1 1 0   = &6E   = 110


3)    1 1 1 1 1 1 1 1   = &FF   = 255
    + 1 1 0 0 1 0 1 0   = &CA   = 202
    -----------------     ---     ---
    1 1 1 0 0 1 0 1 1   =&1C9   = 457
```

Zu 3). Bei dieser Addition tritt ein Übertrag von Stelle 8 (Bit 7) nach Stelle 9 (Bit 8) auf. Ein Byte hat jedoch nur 8 Stellen (8 Bits). Daher wird dieses Übertragsbit, das Carry, im Bit 0 des

Flag Registers gespeichert. Prinzipiell können natürlich auch mehrstellige Ziffern addiert werden. Im Rechner muß dafür jedoch anders vorgegangen werden.

Subtraktion

Die Subtraktion im Dualsystem ist der im Dezimalsystem analog. Es gelten folgende Regeln:

```
0-1=1   1 gemerkt
1-0=1
0-0=0
1-1=0
```

Betrachten wir ein Beispiel:

```
  01101110=&6E=110
- 00110101=&35= 53
  --------  ---  ---
  00111001 &39  57
    **     *
```

Wir erkennen die Sonderregeln für das Weiterrechnen mit dem Übertrag:

```
1-(1+1)=1   1 gemerkt
0-(1+1)=0   1 gemerkt
```

Aufgaben:

Führen Sie die Aufgaben zur Addition als Subtraktionen durch. Prüfen Sie selber Ihre Ergebnisse anhand der Umwandlung ins Dezimalsystem.

Zu 2.) Nach der Umrechnung stellt das Ergebnis eine negative Zahl dar. Das richtige Ergebnis wäre 63-157=-84. Binär ergibt sich:

```
 00111111
- 10011101
 --------
110100010=&1A2
```

Das ist offensichtlich das falsche Ergebnis. Bei der dualen Subtraktion durch den Computer tritt das Problem auf, negative Zahlen darzustellen. Dazu hat man folgende Vereinbarung getroffen:

Das 7. Bit einer Binärzahl wird als Vorzeichenbit benutzt. 0 kennzeichnet positive und 1 kennzeichnet negative Zahlen. Damit begrenzt sich der Zahlenbereich, der durch ein Byte darstellbar ist, auf -128 bis +127. Die Subtraktion von Dualzahlen führt damit auf die Addition von vorzeichenbehafteten Zahlen (5-2=5+(-2)!). Die vorzeichenbehaftete Darstellung, die bei der Subtraktion Verwendung findet, nennt man Zweierkomplement.

Was ist das Zweierkomplement?

In der Zweierkomplementdarstellung werden positive Zahlen weiterhin wie bisher dargestellt, z.B. 5=&X00000101, 126=&X01111110.

Eine negative Zahl wird dargestellt, indem man zunächst ihr Komplement berechnet. Das Komplement ist die Binärzahl, bei der alle Bits genau gegenteilig gesetzt sind, aus 0 wird 1 und aus 1 wird 0. Die erhaltene Binärzahl nennt man das Einerkomplement oder einfach Komplement.

Beispiel:

```
Zahl       : 7=&X00000111
Komplement:   &X11111000
```

Um das Zweierkomplement der Zahl zu erhalten, muß 1 addiert werden.

Beispiel:

```
Komplement         &X11111000
plus 1           +          1
                 ___________
Zweierkomplement  &X11111001
```

Dies ist die Darstellung von -7 im Zweierkomplement.

Das Zweierkomplement wird also auf folgende Weise gebildet:

- ine positive Zahl bleibt unverändert
- on einer negativen Zahl wird das Komplement gebildet nd 1 addiert.

Zweierkomplementdarstellung:

Dezimal	Zweierkomplement
+127	&X01111111
+126	&X01111110
+125	&X01111101
.	.
.	.
.	.
+ 2	&X00000010
+ 1	&X00000001
0	&X00000000
- 1	&X11111111
- 2	&X11111110
- 3	&X11111101
.	.
.	.
.	.
-126	&X10000010
-127	&X10000001
-128	&X10000000

Um den Wert einer negativen Zahl in Zweierkomplementdarstellung zu erhalten, bildet man von ihr wiederum das 2er Komplement.

Beispiel:

```
&X00000111   Komplement
+        1   plus 1
----------
&X00001000

             &X00001000=8
```

Das heißt der Wert von &X11111000 ist -8!
Eine zweimalige 2er Komplementbildung führt wieder auf die Ausgangszahl zurück.

Der Z80A stellt Befehle für die Umwandlung des Akkuinhaltes in das Komplement (CPL) und in das Zweierkomplement (NEG) zur Verfügung. Wir wollen die Funktion dieser Befehle in BASIC nachvollziehen:

Betrachten wir zunächst die Komplementbildung:
A enthalte eine Zahl zwischen 0 und 255 (1Byte). Der >BIN$< Befehl wandelt eine Zahl in einen String um, der der Binärzahl entspricht! Diesen String werden wir Bit für Bit "komplementieren".

```
 10 A=&X11011
 20 abin$=BIN$(a,8)
 30 PRINT "Binaerzahl :";abin$
 40 FOR i=0 TO 7
 50 bit$=MID$(abin$,8-i,1):REM Bit Nr.i
 60 IF bit$="1" THEN bit$="0" ELSE bit$="1"
 70 akpl$=bit$+akpl$ : REM akpl$ ist Komplement $ von a
 80 NEXT
 90 PRINT "Komplement :";akpl$
100 A=VAL ("&X"+akpl$)
```

Zeile 50 extrahiert jeweils das i-te Bit aus abin$. In Zeile 60 wird das Komplement des Bits gebildet, also aus 0 wird 1 und aus 1 wird 0. In Zeile 70 werden die komplementierten Bits in akpl$ gesammelt. Dieses Programm ist allerdings recht langsam. Der XOR Befehl führt die Komplementierung im BASIC schneller aus. Hier geben wir Ihnen nur das Programm, die Funktionsweise dieses logischen Befehls erklären wir im nächsten Kapitel.

```
10 A=&X11011
20 abin$=BIN$(a,8)
30 PRINT "Binaerzahl :";abin$
40 a=a XOR 255
50 akpl$=BIN$(a,8)
90 PRINT "Komplement:";akpl$
```

Zeile 40 führt die eigentliche Komplementbildung aus.

Der NEG Befehl verwandelt eine positive Zahl in eine negative in Zweierkomplementdarstellung. Im BASIC sieht dies dann so aus:

```
10 a=&X11011
20 abin$=BIN$(a,8)
30 PRINT "Binaerzahl      :";abin$
40 a=a XOR 255
45 a=a+1
50 akpl$=BIN$(a,8)
60 PRINT "Zweierkomplement:";akpl$
```

Fügen Sie nun noch folgende Zeile ein:

100 GOTO 30

Nach der Unterbrechung dieses Endlosprogramms werden Sie feststellen, daß eine zweimalige Zweierkomplementbildung wieder auf den Ausgangspunkt zurückführt.

Mit der Zweierkomplementdarstellung kann man nun eine Subtraktion zweier Zahlen als Addition der einen, mit dem Zweier-

komplement der anderen, betrachten. Weiterhin wird das Ergebnis einer Subtraktion als negative Zahl (in Zweierkomplementdarstellung) betrachtet, wenn Bit 7 gesetzt ist (Vorzeichenbit).

Beispiel:

```
120-63=57
120=&X01111000
 63=&X00111111
```

Das Zweierkomplement von 63 ist &X11000001
Nun addieren wir:

```
   01111000 =120
 + 11000001 =Zweierkomplement von 63
  ---------
  100111001
```

Beachten wir zunächst nicht den Übertrag von Bit 7 nach Bit 8 (Carry). Unser Ergebnis ist korrekt: &X00111001=57

Bit 7 ist nicht gesetzt, d.h. das Ergebnis ist positiv. Demnach sollte eigentlich das Carry nicht gesetzt sein.

Da wir mit dem Zweierkomplement rechnen, wird das Carry sozusagen auch komplementiert. In diesem Fall braucht das Carry nicht beachtet werden. Unser Ergebnis stimmt trotzdem.

Die genaue Betrachtung der Arithmetik mit vorzeichenbehafteten Zahlen zeigt, daß mehrere Spezialfälle berücksichtigt werden müssen. Dabei ist das Zusammenspiel der Flags wichtig.

Aufgabe:

Berechnen Sie das Zweierkomplement von:

1) -60
2) -120
3) +5
4) -6

Lösungen:

```
1) &X11000100(=196)
2) &X10001000(=136)
3) &X00000101(=5)
4) &X11111010(=250)
```

8-Bit Arithmetische und Zählbefehle

Es gibt je zwei Befehle zur Addition und Subtraktion:

ADD;ADC und SUB;SBC

Bei den auf C (Carry) endenden Befehlen wird jeweils das Carry Flag bei der Operation in entsprechender Weise berücksichtigt. Bei Verwendung einer dieser beiden Befehle, wird Bit 0 des F Registers (das Carry!!) addiert bzw. subtrahiert.
Die Operanden dieser Befehle haben das Format:

A,x wobei x für r,n,(HL),(IX+d) oder (IY+d) steht.

Daraus ergeben sich folgende Anweisungsarten:

```
A,r         - implizit
A,n         - unmittelbar
A,(HL)      - indirekt
A,(IX+d)    - indiziert
A,(IY+d)    - indiziert
```

Beim SUB Befehl wird nur r, n, (HL), (IX+d) oder (IY+d) als Operand angegeben. "A" wird ausgelassen, da sich alle Befehle dieser Art auf den Akku beziehen.
Diese Befehle sind 8-Bit Operationen. Der Z80A enthält außerdem noch 16-Bit arithmetische Befehle.

Flagbeeinflussung

Bei der Ausführung von Befehlen der Datenbearbeitung werden die Flags beeinflußt:

Carry Flag

Das Carry wird gesetzt, wenn ein Übertrag von Bit 7 nach Bit 8 auftritt. Da ein Byte nur aus Bit 0 bis Bit 7 besteht, ist dieser Übertrag im C Flag abgespeichert. Ansonsten wird das Carry Flag rückgesetzt.

N und H Flag

Diese Flags werden beeinflußt, haben aber für uns keine Bedeutung.

P/V Overflow Flag

Ein Überlauf ist folgendermaßen definiert:

- Wenn ein interner Übertrag von Bit 6 nach Bit 7 vorliegt, aber kein Übertrag von Bit 7 nach Bit 8 (sog. externer Übertrag, wird durch das Carry ange zeigt

- Wenn kein interner Übertrag, dafür aber ein externer Übertrag vorliegt.

Wie diese Definitionen entstehen, wollen wir nicht aufzeigen. Wichtig ist, daß dieses Flag gesetzt ist, wenn bei einer arithmetischen Operation das Vorzeichen des Ergebnisses (Bit 7) fehlerhaft geändert wurde. Das V Flag wird gesetzt, wenn ein Überlauf eintritt, sonst rückgesetzt.

Zero Flag

Dieses Flag wird gesetzt, wenn das Ergebnis der Operation 0 war, ansonsten ist es rückgesetzt.

Sign Flag

Dieses Flag entspricht Bit 7 des Ergebnisses. In der vorzeichenbehafteten Zahlendarstellung ist dies das Vorzeichen, daher der Name Sign Flag.

Eine Tabelle zu der Flagbeeinflussung finden Sie im Anhang.

Bei der Erklärung der Befehle werden wir im folgenden für den Status eines Flags nach einer Operation schreiben:

```
1- Flag ist gesetzt nach der Operation
0- Flag ist rückgesetzt nach der Operation
|- Flag wird je nach Ausgang der Operation gesetzt bzw.
   rückgesetzt
P- P/V Flag zeigt Parität an
V- P/V Flag zeigt Overflow an
 - Kein Einfluß
X- Flag nach der Operation unbekannt

Beispiel:   Flags   S Z V C
                    X | 1
            bedeutet:
                    S  - unbekannt
                    Z  - wenn 0 dann 1 und umgekehrt
                    V  - 1 Overflow
                    C  - kein Einfluß
```

BASIC Analogien zu den Befehlen:

```
ADD A,H          BASIC: A=A+H

ADC A,&A9        BASIC: A=A+&A9+CF
```

CF ist das Carry Flag, sein Wert wird zusätzlich addiert.

```
SUB A,(HL)       BASIC: A=A-PEEK(HL)
SBC A,L          BASIC: A=A-L-CF
```

Beispiele:

```
ADD A,(HL)       A =&1F
                 HL=&B1C9

Speicherstelle &B1C9: &43

  &1F =   0 0 0 1 1 1 1 1
+ &43 =   0 1 0 0 0 0 1 1
          _______________
          0 1 1 0 0 0 1 0
        8 7 6 5 4 3 2 1 0 - Bitnummer

Bit 8= 0 => Carry Flag =0
Bit 7= 0 => Sign  Flag =0
Ergebnis <>0 => Zero Flag=0
Externer Übertrag = 0 und interner Übertrag = 0 => overflow (P/V) Flag =
0
```

Akkuinhalt nach Operation: &X011000110=&62

```
ADD A,D       A enthält &E1
              D enthält &A2
```

```
  &E1 =   1 1 1 0 0 0 0 1
+ &A2 =   1 0 1 0 0 0 1 0
-----   -----------------
 &183 = 1 1 0 0 0 0 0 1 1
        8 7 6 5 4 3 2 1 0 - Bitnummer

Bit 8=1 => Carry Flag = 1
Bit 7=1 => Sign  Flag = 1
Ergebnis nicht nicht Null      => Zero Flag = 0
externer und interner Übertrag => Overflow (P/V) Flag = 0
```

Akkuinhalt nach Ausführung: &X10000011=&83

Wie Sie sehen, enthält der Akku nicht das richtige Ergebnis. Erst wenn man das Carry Flag als 8tes Bit dazunimmt, ergibt sich das korrekte Ergebnis. Aus diesem Grund ist es wichtig, nach arithmetischen Operationen den Status der Flags zu prüfen, um eventuell falsche Ergebnisse entsprechend zu korrigieren.

Beachten Sie zusätzlich, daß bei einer Addition, deren Ergebnis genau 256 ist, das Zero Flag gesetzt wird, obwohl das Ergebnis nicht Null ist.

```
ADC  A,&19            A=&5A
                      Carry Flag=1 (gesetzt)

   &5A = 0 1 0 1 1 0 1 0
 + &19 = 0 0 0 1 1 0 0 1
         ---------------
   &74 = 0 1 1 1 0 1 0 0

Flags:  S Z V C          Akku = &X01110100 = &74
        0 0 0 0
```

Merke: Wurde vor einem ADC Befehl das Carry gelöscht, entspricht er genau dem ADD Befehl.

SUB A,(HL)

```
  A enthält &3C
 HL enthält &BC19
&BC19 enthält &15

  0 0 1 1 0 1 1 0    &36
  1 1 1 0 1 0 1 1    2er Komp.von &15
 ----------------
1 0 0 1 0 0 0 0 1

Bit 7=0 => Sign Flag = 0
Bit 8=1 => Carry Flag= 0
```

Beachten Sie, daß hier das Komplement des wirklichen Carrys genommen wurde (Spezialfall!).

Kein Überlauf V=0
Ergebnis <> 0 =>Z=0

Akkuinhalt nach Ausführung &X00100001=&21

```
SBC A,B
    A=&57
    B=&73

   CF=1

    0 1 0 1 0 1 1 1   = &57
  + 1 0 0 0 1 1 0 1   = 2er Komplement von &73
  + 1 1 1 1 1 1 1 1   = 2er Komplement von &1(CF)
  -----------------
  1 1 1 1 0 0 0 1 1

Flag: S Z V C
      1 0 0 1
```

```
Akkuinhalt &X11100100=&E4
ist das Zweierkomplement von 29
d.h. das Ergebnis ist -29 (87-115-1=-29).
```

Neben der Binärarithmetik gibt es noch eine weitere Möglichkeit Zahlen im Rechner zu verarbeiten:

BCD Arithmetik

Hierbei wird jede Ziffer des Dezimalsystems durch einen Block von 4 Bit dargestellt. Wichtig ist diese Anwendung bei der Behandlung kaufmännischer Probleme, bei denen eine genau vorgegebene Stellenzahl und Genauigkeit eingehalten werden muß. Für die BCD Operationen gibt es den Spezialbefehl DAA, der den Akkuinhalt nach arithmetischen Operationen wieder auf das BCD Format aufbereitet.

Außerdem gibt es noch die besprochenen Spezialbefehle CPL und NEG.

CPL komplementiert den Akkuinhalt und NEG negiert, d.h. wandelt ihn in ein Zweierkomplement um.
Auch einige "normale" Befehle werden zu Spezialbefehlen entfremdet, z.B. kann man SUB A benutzen, um den Akku zu löschen. Das ist fast doppelt so schnell und halb so kurz wie LD A,0.

Zu dieser Gruppe von Befehlen gehören noch die Zählbefehle. Sie erhöhen oder erniedrigen den Wert eines Speichers. Für die Zählbefehle stehen die implizite-, register- und indizierte Adressierung zur Verfügung. Befehle dieser Art werden oft für die Programmierung von Schleifen benutzt. Ihre Funktionsweise ist einfach:

```
INC x erhöht x und
DEC x erniedrigt x, wobei x folgendes sein kann:
```

```
r, (HL), (IX+d), (IY+d)

INC r        BASIC: r=r+1
DEC (HL)     BASIC: POKE HL,PEEK(HL)-1
```

Das Sign-, Zero- und das V-Flag werden je nach dem Ausgang der Operation gesetzt bzw. rückgesetzt. Das Carry bleibt unverändert. Wichtig ist, daß nur die 8-Bit Zählbefehle die Flags beeinflussen. Bei den 16-Bit Zählbefehlen muß extra ein Vergleich gezogen werden.

Mnemonic	Symbolic Operation	Flags C	Z	P/V	S	N	H	Op-Code 76 543 210	No. of Bytes	No. of M Cycles	No. of T States	Comments
ADD A, r	A ← A + r	↕	↕	V	↕	0	↕	10 [000] r	1	1	4	r Reg.
ADD A, n	A ← A + n	↕	↕	V	↕	0	↕	11 [000] 110	2	2	7	000 B
								← n →				001 C 010 D
ADD A, (HL)	A ← A + (HL)	↕	↕	V	↕	0	↕	10 [000] 110	1	2	7	011 E
ADD A, (IX+d)	A ← A + (IX+d)	↕	↕	V	↕	0	↕	11 011 101	3	5	19	100 H 101 L
								10 [000] 110				111 A
								← d →				
ADD A, (IY+d)	A←A+(IY+d)	↕	↕	V	↕	0	↕	11 111 101	3	5	19	
								10 [000] 110				
								← d →				
ADC A, s	A ← A + s + CY	↕	↕	V	↕	0	↕	[001]				s is any of r, n, (HL), (IX+d), (IY+d) as shown for ADD instruction
SUB s	A ← A − s	↕	↕	V	↕	1	↕	[010]				
SBC A, s	A ← A − s − CY	↕	↕	V	↕	1	↕	[011]				
AND s	A ← A ∧ s	0	↕	P	↕	0	1	[100]				The indicated bits replace the 000 in the ADD set above.
OR s	A ← A ∨ s	0	↕	P	↕	0	0	[110]				
XOR s	A ← A • s	0	↕	P	↕	0	0	[101]				
CP s	A − s	↕	↕	V	↕	1	↕	[111]				
INC r	r ← r + 1	•	↕	V	↕	0	↕	00 r [100]	1	1	4	
INC (HL)	(HL) ← (HL)+1	•	↕	V	↕	0	↕	00 110 [100]	1	3	11	
INC (IX+d)	(IX+d) ← (IX+d)+1	•	↕	V	↕	0	↕	11 011 101	3	6	23	
								00 110 [100]				
								← d →				
INC (IY+d)	(IY+d) ← (IY+d) + 1	•	↕	V	↕	0	↕	11 111 101	3	6	23	
								00 110 [100]				
								← d →				
DEC m	m ← m−1	•	↕	V	↕	1	↕	[101]				m is any of r, (HL), (IX+d), (IY+d) as shown for INC. Same format and states as INC. Replace 100 with 101 in OP code.

Notes: The V symbol in the P/V flag column indicates that the P/V flag contains the overflow of the result of the operation. Similarly the P symbol indicates parity. V = 1 means overflow, V = 0 means not overflow. P = 1 means parity of the result is even, P = 0 means parity of the result is odd.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown, ↕ = flag is affected according to the result of the operation.

16-Bit Arithmetische- und Zählbefehle

Die 16-Bit Arithmetik Befehle sind prinzipiell den 8-Bit Befehlen ähnlich. 16-Bit Befehle sind eingeschränkter. Nur die Befehle ADD, ADC und SUB sind für einige Registerpaare vorhanden. Das Ergebnis einer Operation wird grundsätzlich im HL Registerpaar (nicht im Akku, wie bei den 8-Bit Befehlen) abgelegt. Beim ADD Befehl besteht die Möglichkeit Ergebnisse auch in den Indexregistern zu speichern.

Die 16-Bit Befehle entsprechen mehreren Hintereinanderausführungen von 8-Bit Befehlen. Da sie diese Befehle automatisch verbinden, sind sie schneller und kürzer.

```
16-Bit          8-Bit

ADD HL,BC        LD A,L
                ADD A,C
                 LD L,A
                 LD A,H
                ADC A,B
                 LD H,A
```

Sämtliche 16 Bit Arithmetikbefehle verwenden die implizite Adressierung. Die Flagbeeinflussung bei ADC und SBC ist der der 8-Bit Befehle analog. Bei ADD wird nur das Carry beeinflußt, und bei den 16-Bit Befehlen INC und DEC werden die Flags gar nicht beeinflußt (oder verändert)!

```
ADD IX,DE        BASIC: IX=IX+DE
ADC HL,BC        BASIC: HL=HL+BC+CF
SBC HL,SP        BASIC: HL=HL-SP-CF
```

```
Beispiel:

HL=&C000
DE=&0800

ADD HL,DE

  &C000 = 1100 0000 0000 0000
+ &0800 = 0000 1000 0000 0000
  -----   -------------------
  &C800 = 1100 1000 0000 0000

Flag: S Z V C
            0

S, Z, V Flag sind unbeeinflußt

HL=&F800
DE=&0800

ADC HL,DE

  &F800 =   1111 1000 0000 0000
+ &0800 =   0000 1000 0000 0000
-------   ---------------------
 &10000 = 1 0000 0000 0000 0000

Flag: S Z V C
      0 0 1 1
```

Auch hier enthält der HL nicht das richtige Ergebnis &10000, sondern &0. Das Carry Flag zeigt diesen Fehler an. Bei den 16-Bit Operationen stellt es Bit Nummer 16 dar.

Die 16-Bit Zählbefehle sind alle implizit adressiert. Sie können sich auf die 16-Bit Register BC, DE, HL, SP, IX und IY beziehen. Diese Befehle beeinflussen, im Gegensatz zu den 8-Bit Zählbefehlen, nicht (!) die Flags.

Mnemonic	Symbolic Operation	Flags C	Z	P/V	S	N	H	Op-Code 76 543 210	No. of Bytes	No. of M Cycles	No. of T States	Comments	
ADD HL, ss	HL ← HL+ss	‡	•	•	•	0	X	00 ss1 001	1	3	11	ss	Reg.
												00	BC
ADC HL, ss	HL←HL+ss+CY	‡	‡	V	‡	0	X	11 101 101	2	4	15	01	DE
								01 ss1 010				10	HL
												11	SP
SBC HL, ss	HL←HL-ss-CY	‡	‡	V	‡	1	X	11 101 101	2	4	15		
								01 ss0 010					
ADD IX, pp	IX ← IX + pp	‡	•	•	•	0	X	11 011 101	2	4	15	pp	Reg.
								00 pp1 001				00	BC
												01	DE
												10	IX
												11	SP
ADD IY, rr	IY←IY+rr	‡	•	•	•	0	X	11 111 101	2	4	15	rr	Reg.
								00 rr1 001				00	BC
												01	DE
												10	IY
												11	SP
INC ss	ss ← ss + 1	•	•	•	•	•	•	00 ss0 011	1	1	6		
INC IX	IX ← IX + 1	•	•	•	•	•	•	11 011 101	2	2	10		
								00 100 011					
INC IY	IY ← IY + 1	•	•	•	•	•	•	11 111 101	2	2	10		
								00 100 011					
DEC ss	ss ← ss - 1	•	•	•	•	•	•	00 ss1 011	1	1	6		
DEC IX	IX ← IX - 1	•	•	•	•	•	•	11 011 101	2	2	10		
								00 101 011					
DEC IY	IY ← IY - 1	•	•	•	•	•	•	11 111 101	2	2	10		
								00 101 011					

Notes: ss is any of the register pairs BC, DE, HL, SP
pp is any of the register pairs BC, DE, IX, SP
rr is any of the register pairs BC, DE, IY, SP.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
‡ = flag is affected according to the result of the operation.

Aufgabe:

Nach dieser Durststrecke wollen wir endlich die neuen Befehle zum ersten Mal anwenden. Schreiben Sie ein kleines Programm für die Addition zweier 8-Bit Zahlen. Die Zahlen werden durch >POKE< Befehle vom BASIC aus ins RAM gespeichert. Das Ergebnis der Addition soll wieder im RAM gespeichert werden. Nach dem Rücksprung ins BASIC kann es dann mit dem >PEEK< Befehl gelesen und ausgegeben werden.

Lösung:

Da 8-Bit Additionen grundsätzlich den Akku benutzen, muß der erste Summand im Akku gespeichert werden:

LD A,Summand1

Der zweite Summand wird in einem der 8-Bit Register gespeichert:

LD H,Summand2

Nun führen wir die Addition aus:

ADD A,H

Das Ergebnis soll in Speicherstelle &A100 abgelegt werden:

LD (&A100),A

Wählen wir als Startadresse &A000, ergibt sich folgendes Bild:

```
A000 3E10     10    LD    A,&10
A002 2620     20    LD    H,&20
A004 84       30    ADD   A,H
A005 3200A1   40    LD    (&A100),A
A008 C9       50    RET
```

Die DATA Zeile des Laders ergibt sich zu:

60 DATA &3E,&10,&26,&20,&84,&32,&00,&A1,&C9

Aus dem Assemblerlisting geht hervor, daß der erste Summand an Adresse &A001, und der zweite an Adresse &A003 gespeichert ist. In unserem Falle haben wir hierfür &10 und &20 gewählt. Das BASIC Programm, welches diese Werte festlegt, das Programm ausführt und das Ergebnis ausgibt, sieht dann folgendermaßen aus:
(Zuvor muß natürlich mit dem BASIC Lader das Maschinenprogramm geladen werden.)

```
10 POKE &A001,Summand1
20 POKE &A003,Summand2
30 CALL &A000,
40 PRINT PEEK (&A100)
```

4.7 Logische Befehle und CP (Compare)

Zu den Befehlen zur Datenbearbeitung gehören auch die Logischen Befehle.

Der Z80A besitzt die logischen Befehle AND, OR und XOR (Exklusiv OR) sowie den Vergleichsbefehl CP. Alle diese Befehle arbeiten mit 8-Bit Daten. Der Akku ist immer das Register, mit dem die logische Operation ausgeführt wird. Der Akku wird deshalb nicht mit im Operanden des Assemblerbefehls (wie z.B. bei ADD A,B) angegeben (z.B. AND B).
Die vier Befehle AND, OR, XOR und CP können mit folgenden Adressierungsarten vorkommen:

- implizit (Register A, B, C, D, E, H, L)
- indirekt : Register (HL)
- indiziert
- unmittelbar

Betrachten wir die Funktionen der logischen Befehle. Jeder kann sich etwas unter folgender logischen Aussage vorstellen:

"Wenn es regnet, dann wird die Straße naß."

Diese Aussage ist eine Folgerung der Form <wenn,...dann...>. Betrachten wir die nächste Aussage:

"Wenn es regnet UND ich auf der Straße bin, dann werde ich naß".

Hier sind zwei Aussagen durch UND verbunden. Das logische UND (engl.AND) sagt aus, daß beide Aussagen, also "es regnet" (1.Aussage) und "ich bin auf der Straße " (2.Aussage) zutreffen müßen, damit das Ergebnis eintritt. Regnet es nicht (die 1.Aussage ist nicht erfüllt), werde ich nicht naß; bin ich in einem Haus (2. Aussage ist nicht erfüllt), werde ich auch nicht naß. Damit die Folgerung stimmt (wahr ist) müssen also beide Aussagen wahr sein. Das ist genau die Eigenschaft der AND (UND) Verknüpfung. Da der Computer mit 0 und 1 arbeitet, vereinbart man folgendes:

1 entspricht Aussage ist wahr
0 entspricht Aussage ist falsch

Damit ergibt sich:

```
1 AND 1= 1 beide Aussagen sind   wahr=> Ergebnis wahr
1 AND 0= 0 eine Aussage ist falsch   => Ergebnis falsch
0 AND 1= 0 eine Aussage ist falsch   => Ergebnis falsch
0 AND 0= 0 beide Aussagen sind falsch=> Ergebnis falsch
```

Das Schneider BASIC beinhaltet die logischen Befehle. Probieren Sie sie aus:

```
PRINT 1 AND 1
PRINT 1 AND 0 usw....
```

Die logischen Operationen sind für die Computertechnik von größter Bedeutung. Sie lassen sich relativ einfach elektronisch verwirklichen. Dabei sind zwei Eingangsleitungen, die Strom führen (=1) oder keinen Strom führen (=0), an einen elektronischen Schaltkreis angeschlossen, dessen Ausgangsleitung, je nach Eingangsbedingungen, Strom oder keinen Strom führt (1 oder 0 ist). Solche Schaltungen werden mathematisch mit Hilfe der Booleschen Algebra erfaßt. Ein Mikroprozessor besteht aus einer Vielzahl von hintereinander geschlossenen logischen Gattern. Die Addition im MPU ist z.B. aus verschiedenen logischen Operationen aufgebaut.

Als Programmierer kommen wir jedoch mit diesen Strukturen nicht in Berührung. Wir wenden die logischen Operationen auf Daten (8-Bit) an. Dabei werden jeweils entsprechende Bits, der beiden Bytes verknüpft.

```
      11111000
AND   01010011
      --------
      01010000

Bit 0: 0 AND 1=0
Bit 1: 0 AND 1=0
Bit 3: 0 AND 0=0
Bit 4: 1 AND 0=0
Bit 5: 1 AND 1=1
       .
       .
```

Eine der wichtigsten Anwendungen des AND Befehl ist das Löschen oder Ausblenden von bestimmten Bits.

A=&X10111001

Nehmen wir an, wir wollen nur die Bits 0 bis 3 betrachten, d.h. Bit 4 bis 7 sollen ausgeblendet werden. Um das zu erreichen, "undieren" (verknüpfen mit UND) wir A mit &X00001111.

```
        10111001      :A
AND     00001111      :Maske
        --------
        00001001
```

Die Maske, die zum Ausblenden der Bits benutzt wird, enthält eine 0 für ein auszublendendes Bit, und eine 1 für ein signifikantes Bit.
Formulieren wir in BASIC:

```
A=&X10111001
A=A AND &X00001111
```

In Maschinensprache erhalten wir:

```
LD A,&X10111001
AND  &X00001111
```

Sehen Sie sich folgende Aussagen an:

"Wenn es regnet ODER ich bade, dann werde ich naß."

Das Ergebnis ist wahr, wenn mindestens eine der Aussagen wahr ist. Damit ergibt sich für die ODER (OR) Verknüpfung:

```
0 OR 0= 0
0 OR 1= 1
1 OR 0= 1
1 OR 1= 1
```

Mit der OR Verknüpfung ist es möglich, bestimmte Bits eines Bytes zu setzen.
A enthalte &X10001011.
Nun sollen die obersten 3 Bit (5, 6, 7) auf 1 gesetzt werden:

```
   10001011   :A
OR 11100000   :Maske
   --------
   11101011
```

Die Maske enthält für jedes Bit, das unbedingt auf 1 gesetzt werden soll, eine 1, und für die Bits, die nicht verändert werden sollen, eine 0.

```
LD A,&X10001011        BASIC: A=&X10001011
OR &X11100000          BASIC: A=A OR &X11100000
```

Das XOR, oder exklusiv ODER, unterscheidet sich in nur einem Punkt vom normalen oder inclusiven ODER. Sind beide Eingangsbits auf 1, so ist der Ausgang 0. Das ausschließende (exclusive) OR liefert eine 1 bei verschiedenen Eingängen und eine 0 bei gleichen Eingängen.

0 XOR 0= 0
1 XOR 0= 1
0 XOR 1= 1
1 XOR 1= 0

Für das XOR gibt es zwei Anwendungen, das Vergleichen und das Komplementieren. Die zu vergleichenden Bytes werden durch XOR verknüpft. Ist das Ergebnis 0, so waren die Bytes gleich. Bei Ungleichheit sind die unterschiedlichen Bits des Ergebnisses gesetzt.

```
    10101010
XOR 10101010    Vergleich!!
    --------
    00000000

    10101010
XOR 10101100    Vergleich!!
    --------
    00000110

=> Bit 1 und Bit 2 sind unterschiedlich.
```

Zum Komplementieren wird wieder mit einer Maske verknüpft. Sie enthält eine 1 für ein zu komplementierendes Bit und eine 0 für ein gleichbleibendes Bit.

Bit 4-7 sollen komplementiert werden.

```
    10101111   :A
XOR 11110000   :Maske
    --------
    01011111
```

Analogien:

Maschinensprache	BASIC
AND H	A=A AND H
OR (HL)	A=A OR PEEK(HL)
XOR &FF	A=A XOR &FF

Bei den logischen Befehlen wird das Carry immer auf 0 gesetzt. Z Flag und S Flag werden, wie üblich, beeinflußt. Das P/V Flag zeigt bei diesen Befehlen die Parität des Ergebnisses an. Die Parität ist 1, wenn die Anzahl der Einsen im Byte gerade ist, und ist 0, wenn sie ungerade ist.

Aufgaben:

1. Was bewirkt ein:

- OR mit &FF ?
- OR mit &0 ?
- AND mit &FF ?
- AND mit &0 ?
- XOR mit &FF ?
- XOR mit &0 ?

2. Im BASIC gibt es den Befehl >NOT<. Setzen Sie diesen Befehl auf zwei verschiedene Weisen in Maschinensprache um (bezüglich des Akku).

Lösung:

zu 1.

```
 OR  &FF  => &FF d.h. alle Bits sind gesetzt
 OR  &0   => keine Veränderung
AND  &FF  => keine Veränderung
AND  &0   => &0 d.h. alle Bits sind rückgesetzt
XOR  &FF  => alle Bits sind komplementiert
XOR  &0   => keine Veränderung
```

zu 2.

```
XOR Befehl :   XOR &FF
CPL Befehl :   CPL
```

Der Vergleichsbefehl CP (Flagbeeinflussung)

Der CP Befehl dient dem Vergleich des Akkuinhaltes mit einem Byte. Dieses Byte kann folgendermaßen adressiert sein:

- implizit : Register A, B, C, D, E, H, L
- indirekt : Registerpaar (HL)
- indiziert
- unmittelbar

Durch den CP Befehl wird das adressierte Byte vom Akku abgezogen, und je nach dem Ausgang der Rechnung werden die Flags beeinflußt. Im Gegensatz zum SUB Befehl wird das Ergebnis jedoch nicht im Akku abgespeichert, d.h. der Akkuinhalt wird durch den Befehl nicht beeinflußt. Abhängig vom Status der Flags kann nach diesem Befehl ein bedingter Sprung ausgeführt werden.
Betrachten wir die möglichen Fälle bei dem Vergleich:

Akkumulatorinhalt ist größer:
- Das Carry ist in diesem Fall immer 0, da das Ergebnis nicht größer als 255 sein kann.

Akkumulatorinhalt ist gleich:
- In diesem Fall ist Z=1, da das Ergebnis der Subtraktion 0 ist. Auch hier ist C=0, da kein Übertrag auftritt.

Akkumulatorinhalt ist kleiner:
- In diesem Fall ist das Carry Flag immer gesetzt, da ein negativer Übertrag auftritt.

Regeln:

```
C=0   bedeutet  >=
Z=0   bedeutet   =
C=1   bedeutet  =<
```

weiterhin erhält man:

```
Z=1            bedeutet  <>
C=0  und Z=1   bedeutet  >
C=1 oder Z=0   bedeutet  =<
```

Diese Regeln gelten nur, wenn die zu vergleichenden Bytes als vorzeichenlose Zahlen zwischen 0 und 255 betrachtet werden.

Stellen die beiden Bytes vorzeichenbehaftete Zahlen in 2er Komplementdarstellung dar, so gelten kompliziertere Regeln, die sich aus den Flagregeln für vorzeichenbehaftete Arithmetik ergeben. In den meisten Fällen ist diese Anwendung nicht notwendig.

Für die Entscheidung auf Gleichheit wird das Z Flag benutzt. Größer bzw. kleiner entscheidet sich nach dem Status von S und V Flag. S und V Flag werden durch XOR verknüpft, d.h. ist V gesetzt (ein Überlauf ist eingetreten), wird S komplementiert, sonst bleibt S auf dem alten Stand.

```
S XOR V =0   bedeutet >=
S XOR V =1   bedeutet <
```

Im folgenden werden wir voraussetzen, daß die Bytes als vorzeichenlose Zahlen zu interpretieren sind.

Beispiel:

```
A =&35
B =&21

CP B

liefert S Z V C
        0 0 0 0    wegen

   00110101  :A
 - 00100001  :B (kein (!) Zweierkomplement)
   ________
   00010100

Kein Übertrag:    => C=0
Bit=0             => S=0
 <>0              => Z=0
kein Überlauf     => V=0
```

Das Carry Flag ist gleich 0. Daraus folgt, daß der Akkuinhalt größer als der des vergleichbaren Bytes ist (Inhalt vom B Register).

```
C =&81

CP C    liefert

Flag:S Z V C
     1 0 1 1   wegen:
```

```
  00000001   :A Register
- 10000001   :C Register
  --------
 110000000

Übertrag von 7 nach 8 => C=1
Bit 7=1               => S=1
     <>               => Z=0
Übertrag von 7 nach 8 und kein
Übertrag von 6 nach 7 => V=1
```

Folglich ist C=1. Daraus läßt sich schließen, daß der Wert, mit dem verglichen wurde (Inhalt vom C Register), größer war als der Akkuinhalt.

Im Zusammenhang mit den Befehlen für Tests und Sprünge, werden wir den CP Befehl später oft benutzen.

Die logischen Befehle befinden sich bei der Liste der 8-Bit arithmetischen Befehle (Kapitel 4.6).

Das Demoprogramm:

```
A000 06FF      10     LD   B,&FF
A002 2100C0    20     LD   HL,&C000
A005 7E        30     LD   A,(HL)
A006 A8        40     XOR  B
A007 77        50     LDd  (HL),A
A008 23        60     INC  HL
A009 3E00      70     LD   A,0
A00B BC        80     CP   H
A00C 20F7      90     JR   NZ,&A005
A00E C9       100     RET
```

Dieses Programm invertiert den gesamten Bildschirm in >MODE 2<.
LD B,&FF ist die Maske, mit der durch den XOR B Befehl der jeweilige Akkuinhalt invertiert wird.

HL wird mit der Startadresse des Bildschirms &C000 geladen (LD HL,&C000). Dann beginnt die Programmschleife. LD A,(HL) liest ein Byte aus dem Bilschirmspeicher. Durch XOR B wird dieses invertiert, und dann mit LD (HL),A wieder in den Bildschirmspeicher geschrieben. Dann wird HL erhöht (INC HL) und geprüft, ob HL noch im Bereich des Bildschirmspeichers liegt.

HL läuft von &C000 bis &FFFF. Wird dann wiederum HL erhöht (&FFFF+1), ergibt sich der Wert 0 für HL. Eigentlich wäre das Ergebnis &10000, da HL jedoch nur 16-Bit Zahlen speichern kann, bleibt das überzählige Bit unberücksichtigt: HL=0.

Mit dem CP Befehl soll festgestellt werden, ob HL bereits 0 ist. Da CP immer mit dem Akkuinhalt vergleicht, muß der Akku zuvor durch LD A,0 mit 0 geladen werden.

Bei dem Vergleich muß nun das High Byte von HL verglichen werden, ist H=0, dann ist auch HL=0. Durch den CP Befehl wird das Z Flag in entsprechender Weise beeinflußt.

Der nachfolgende Sprungbefehl JR NZ,&A005 besagt:
"Springe an Adresse &A005, wenn Z nicht Null ist (Non Zero), sonst nehme den nächsten Befehl."

Ist HL=0, so wird das Programm mit RET abgeschlossen.
Die DATA Zeilen des BASIC Laders sind:

```
DATA &06,&FF,&21,&00,&C0,&7E,&A8,&77
DATA &23,&3E,&00,&BC,&20,&F7,&C9
```

Wählen Sie &A000 als Startadresse, &A000+15-1=&A00E als Endadresse und starten Sie mit >CALL &A000<. (Im MODUS 2)

Anstelle des XOR B Befehls können wir auch CPL (komplementiere den Akku) einsetzen.
Schalten Sie nun in MODUS 1 um und probieren Sie die Routine aus. Das gewünschte Ergebnis kommt nicht zustande. Das liegt im Aufbau des Bildschirmspeichers begründet. Wie Sie wissen, korrespondieren in >MODE 2< gesetzte Bits und gesetzte Punkte direkt miteinander. Daher können im MODUS 2 keine verschiedenen Schriftfarben gewählt werden. Im MODUS 1 stehen vier Farben zur Verfügung. Obwohl nur der Bereich von &C000-&FFFF bereit steht und zusätzlich noch die Information über die Farbe gespeichert werden muß, sind im MODUS 1 die oberen vier Bit jedes Bytes für das Setzen je eines doppelt breiten Punktes zuständig. Die unteren Bits bestimmen die Farbe. Da wir die Punkte und nicht die Farben invertieren, müssen wir die Invertierungsmaske ändern. LD B,&FF (&FF=&X11111111) bedeutet, alle Bits werden durch XOR B invertiert. Durch &X11110000=&F0 werden nur die obersten 4 Bit invertiert.

Um das Programm auch im MODUS 1 zu benutzen, müssen wir also den zweiten Wert von DATA Zeile 60 von &FF nach &F0 ändern.
Im >MODE 0< sind nur Bit 6 und Bit 7 für die Punkte zuständig. Nehmen Sie auch dafür die nötige Änderung im Programm vor.

4.8 Rotations- und Schiebebefehle

Was bedeutet das Verschieben der Ziffern einer Zahl?

```
 4   3   2   1   0
10  10  10  10  10

     3   7   3   0

 3   7   3   0   0  :nach links verschoben !

         3   7   3  :nach rechts verschoben !
```

Im Dezimalsystem bewirkt ein Schieben nach links, eine Multiplikation mit 10 (der Basis des Dezimalsystems) und ein Schieben nach rechts eine Division durch 10. (Ein Verschieben der Ziffern nach links bedeutet ein Verschieben des Kommas um eine Stelle nach rechts.)

Entsprechend bedeutet ein Verschieben im Dualsystem ein Teilen bzw. Malnehmen mit zwei. Im BASIC gibt es für diese Befehle kein direktes Äquivalent. (Es sei denn das Multiplizieren bzw. Dividieren mit bzw. durch 2.)

Der Z80A besitzt 76 Befehle dieser Art, von denen die meisten die implizite-, indirekte- oder indizierte Adressierung benutzen. Es gibt verschiedene Arten des Rotierens und Schiebens. Zuerst wollen wir zwischen den Operationen Schieben und Rotieren unterscheiden.

Schieben: Beim Schieben nach rechts und links wird der Inhalt des Registers Bit für Bit in die jeweilige Richtung bewegt. Das an der Seite herausfallende Bit wird in das Carry übernommen. Die enstandene Freistelle, an der anderen Seite des Bits, wird mit einer 0 gefüllt.

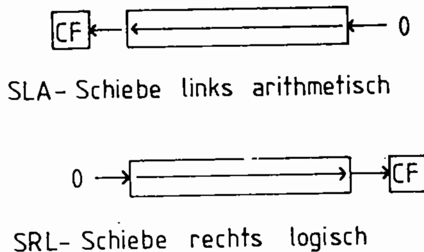
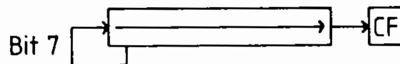


Abb. 7

Beim Anwenden des SRL Befehl auf vorzeichenbehaftete Zahlen tritt ein Fehler auf. Bit 7, das Sign Bit, wird auf den Platz von Bit 6 geschoben. An Stelle von Bit 7 wird eine 0 eingeschoben. Damit wäre aus einer negativen Zahl (Bit 7=1) eine positive (Bit 7=0) geworden. Um diesen Fehler zu umgehen, gibt es den SRA Befehl. Bei diesem Befehl ist das links eingeschobene Bit mit dem Vorzeichenbit identisch. Es ist 0, wenn das linke Bit =0 (+) war und 1 wenn das linke Bit =1 (-) war. Da dieser Befehl die arithmetische Bedeutung des 7ten Bit beachtet, bezeichnet man ihn als arithmetischen und nicht logischen Schiebebefehl.



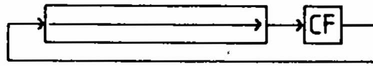
SRA- Schiebe rechts arithmetisch

Abb. 8

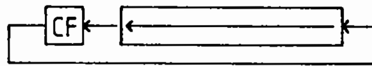
Rotieren: Im Gegensatz zum Schieben ist beim Rotieren das hereinkommende Bit entweder das auf der anderen Seite herausgefallene oder das Carry Bit.
Beim Z80A gibt es zwei Arten der Rotation:

8-Bit Rotation (ohne Carry)
9-Bit Rotation (mit Carry)

Bei einer 9-Bit Rotation nach rechts werden alle acht Bits um eine Stelle nach rechts verschoben. Das rechts herausfallende Bit gelangt ins Carry. Das links hereinkommende ist der alte Inhalt vom Carry (bevor er vom herausfallenden Bit überschrieben wurde). Da hier die 8-Bit des Bytes und das Carry(das 9te Bit!) rotiert werden, bezeichnet man diese Art der Rotation als 9-Bit Rotation.



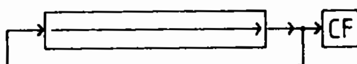
RR- Rotiere rechts durch Carry



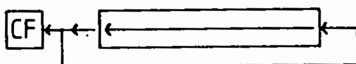
RL- Rotiere links durch Carry

Abb. 9 9- Bit Rotation

Bei der 8-Bit Rotation rotieren nur die 8 Bit des Registers. Im Carry wird nur das herausfallende Bit gespeichert. Der alte Inhalt des Carry wird jedoch nicht mit rotiert. Das herausfallende Bit wird am anderen Ende des Registers wieder aufgenommen.



RRC- Rotiere rechts



RLC- Rotiere links

Abb. 10 8-Bit Rotation

Weiterhin gibt es zwei Spezialbefehle für das Rotieren von Ziffern im BCD Format (=Blöcke von 4 Bit).

RLD und RRD (D:Digit-Ziffer) rotieren zwei Ziffern der Speicherstelle, auf die HL zeigt, und die Ziffer, die durch die untere Hälfte des Akkumulators gegeben ist.

Die Rotier- und Schiebebefehle haben meist einen 2 Byte Opcode. Das erste Byte des Opcodes ist immer &CB. (Bei den indiziert adressierten Befehlen, ist &CB das 2.Byte, da das erste bei dieser Adressierungsart entweder &DD oder &FD ist. Ausnahme: RRD/RLD beginnen mit ED.) Da die Rotationsbefehle für die Arithmetik oft benötigt werden, wurden vier weitere Befehle festgelegt. Diese beziehen sich nur auf den Akku und besitzen einen 1 Byte Opcode. Sie sind genau halb so lang und doppelt so schnell wie die Standardbefehle:

```
"normal"        "Akku-Spezial"

 RLC  A          RLCA
 RRC  A          RRCA
 RL   A          RLA
 RR   A          RRA
```

Durch die normalen Rotations- bzw. Schiebebefehle werden S und Z Flag in der üblichen Weise beeinflußt. P/V Flag zeigt die Parität an. Der Inhalt des Carrys ist das jeweils herausfallende Bit. Die Spezialbefehle für den Akku verändern nur C, dagegen S, Z und P nicht. Die BCD Rotierbefehle RLD/RRD beeinflussen nur S, Z und P Flag in der obigen Weise, dagegen nicht das Carry.

Beispiele:

```
SRL C           C:&36

        00110110         :&36
    0--> 0011011--> 0 ins Carry
        00011011         :C Register nach Ausführung
                    0    :Carry nach der Ausführung

SRL bewirkt eine Division durch 2: &36^2=&1B

SRA (HL)        HL:&B100
                Speicherstelle &B100:&C2

        11000010         :&C2
        *1100001--> 0 :Carry
        11000010    0 :CF:(HL) nach der Ausführung=&E1

(* Bit 7 bleibt an dieser Stelle)

Als Zweierkomplement bedeutet:

&C2 = -62
&E1 = -31
```

Der SRA Befehl führt die Halbierung vorzeichenbehafteter Zahlen richtig durch. SRL (HL) hätte statt dessen &61=97 als Ergebnis gehabt. Das ist jedoch nicht die Hälfte von -62, sondern die Hälfte von 194, was &C2 als vorzeichenloser Zahl entspricht.

```
RLC D
   D: &E4    Carry=1

        &E4= &X11100100
Carry neu <-- 11100100 <-- 1=Carry alt
              11001001
Inhalt von D nach der Ausführung: &C5
     Carry = 1
```

&C5 ist allerdings nicht das Doppelte von &E4. Der Grund dafür ist, daß ein Bit zu Carry rotiert wurde. Also soll &1C5 das Doppelte von &E4 sein. Dies ist nicht ganz richtig, da das alte Carry (=1) hineinrotiert wurde. Also ist &1C9-1=&1C8 das Doppelte von &E4.

Sollen Zahlen, die aus mehreren Bytes bestehen rotiert werden, so wird durch RLC bzw. RRC, daß beim vorher rotierten Byte herausgefallene Bit über das Carry in das nächste Byte hineinrotiert.(Siehe Programm am Ende des Kapitels.)

```
RRA
   Akku: &76

     &76=&X011101110
         &X*01110110 --> Carry
(* hier wird das alte Bit 0 "hineinrotiert")

      Akku: &X00111011       CF=0

Akkuinhalt: &3B
            &3B*2 = &76
```

Mnemonic	Symbolic Operation	C	Z	P/V	S	N	H	Op-Code 76 543 210	No. of Bytes	No. of M Cycles	No. of T States	Comments
RLCA		↕	•	•	•	0	0	00 000 111	1	1	4	Rotate left circular accumulator
RLA		↕	•	•	•	0	0	00 010 111	1	1	4	Rotate left accumulator
RRCA		↕	•	•	•	0	0	00 001 111	1	1	4	Rotate right circular accumulator
RRA		↕	•	•	•	0	0	00 011 111	1	1	4	Rotate right accumulator
RLC r	r, (HL), (IX+d), (IY+d)	↕	↕	P	↕	0	0	11 001 011 00 000 r	2	2	8	Rotate left circular register r
RLC (HL)		↕	↕	P	↕	0	0	11 001 011 00 000 110	2	4	15	r Reg. 000 B 001 C 010 D 011 E 100 H 101 L 111 A
RLC (IX+d)		↕	↕	P	↕	0	0	11 011 101 11 001 011 ← d → 00 000 110	4	6	23	
RLC (IY+d)		↕	↕	P	↕	0	0	11 111 101 11 001 011 ← d → 00 000 110	4	6	23	
RL m	m ≡ r, (HL), (IX+d), (IY+d)	↕	↕	P	↕	0	0	010				Instruction format and states are as shown for RLC,m. To form new OP-code replace 000 of RLC,m with shown code
RRC m	m ≡ r, (HL), (IX+d), (IY+d)	↕	↕	P	↕	0	0	001				
RR m	m ≡ r, (HL), (IX+d), (IY+d)	↕	↕	P	↕	0	0	011				
SLA m	m ≡ r, (HL), (IX+d), (IY+d)	↕	↕	P	↕	0	0	100				
SRA m	m ≡ r, (HL), (IX+d), (IY+d)	↕	↕	P	↕	0	0	101				
SRL m	m ≡ r, (HL), (IX+d), (IY+d)	↕	↕	P	↕	0	0	111				
RLD	A, (HL)	•	↕	P	↕	0	0	11 101 101 01 101 111	2	5	18	Rotate digit left and right between the accumulator and location (HL). The content of the upper half of the accumulator is unaffected
RRD	A, (HL)	•	↕	P	↕	0	0	11 101 101 01 100 111	2	5	18	

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

Anwendung

Die Standardanwendung der Rotations- und Schiebebefehle kommt beim "Rechnen" vor. Wir werden in unserem Beispiel die Befehle "entfremden" und für eine Verschiebung des Bildschirms benutzen.
Mit Hilfe der Blocktransferbefehle war es möglich, den Bildschirm horizontal zeichenweise zu verschieben. Mit den neuen Befehlen können wir eine Bit für Bit-Verschiebung bewirken.

Das Assemblerlisting:

```
A000 97        10    SUB A
A001 2100C0    20    LD HL,&C000
A004 CB3E      30    SRL (HL)
A006 23        40    INC HL
A007 BC        50    CP H
A008 20FA      60    JR NZ,&A004
A00A C9        70    RET
```

Sie erkennen die Grundstruktur der Schleife, mit der HL von &C000 bis &FFFF hochgezählt wird, wieder.
Neu ist der erste Befehl.

SUB A steht an Stelle des sonst verwendeten Befehls LD A,0. SUB A löscht den Akku. Dieser Befehl ist schneller, da er implizit adressiert ist.

LD A,0 ist unmittelbar adressiert, d.h. die Daten (0!) müssen zusätzlich gelesen werden. Nun das Kernstück des Programms:

SRL (HL)

Da HL den gesamten Adressenbereich durchlaufen soll, haben wir die indirekte Adressierung gewählt. SRL verschiebt die 8 Bit jedes Bildschirmbytes um eine Stelle nach rechts.

Setzen Sie mit Hilfe des Assemblerlistings das Programm in DATA Zeilen um, und laden Sie es mit dem BASIC Lader ab

Adresse &A000. Das Programm schiebt jedes Zeichen des Bildschirms nach rechts. Da wir das rechts herausfallende Bit nicht weiter berücksichtigen, sind die Zeichen rechts um ein Bit abgeschnitten.
Geben Sie nun folgendes ein:

```
FOR I=1 TO 8:CALL &A000:NEXT
```

Durch diesen Befehl wird der Bildschirm gelöscht. Die Zeichen verschwinden bitweise nach rechts, da bei dem SRL Befehl das links hereinkommende Bit 0 (=kein Punkt) ist.
Ersetzen wir SRL (HL) durch SLA (HL).

Der Code für diesen Befehl ist &CB,&25. Setzen Sie in den DATA Zeilen für das 5te Element &25 (an Stelle von &3E) ein, und laden Sie erneut mit >RUN<. Dieses Programm bewirkt ähnliches, nur findet die Verschiebung nach links statt.
Probieren Sie auch SRA (HL) Code: &CB,&2E aus. Das 5te Byte in den DATA Zeilen ist dann &2E. Nach der achtmaligen Ausführung durch die FOR-NEXT Schleife ensteht ein merkwürdiges Muster auf dem Bildschirm. Das liegt daran, daß der SRA Befehl das 7te Bit an seiner Stelle stehen läßt. Nach dem mehrfachen Ausführen des Befehls sind also alle Bits auf den vorherigen Wert von Bit 7 gesetzt.

Aus dem Buchstaben R der Readymeldung werden zwei waagerechte Striche (nach 8 maliger Ausführung). Der Grund dafür ist das Bitmuster dieses Buchstaben.

```
          76543210 Bit-Nummer
         1******
         2 **  **
         3 **  **
Zeile    4 *****
         5 ** **
         6 **  **
         7***   *
         8
```

Jedes Zeichen ist auf diese Weise in einem 8x8 Raster dargestellt. Beim R ist Bit 7 nur in Zeile 1 und Zeile 7 gesetzt. Führen Sie die acht Maschinenprogrammaufrufe einzeln hintereinander aus, so können Sie beobachten, daß das R davongeschoben wird, in Zeile 1 und 7 jedoch ein Strich entsteht.

Das "e" der Readymeldung verschwindet ganz, da bei diesem Zeichen kein Bit 7 gesetzt ist. Vom a bleibt ein Strich in Zeile 6, vom d in Zeile 4, 5 und 6 und von y wiederum kein Strich.

Machen Sie sich anhand dieses Ergebnisses klar, warum der SRA Befehl als arithmetisch, dagegen der SRL Befehl als logisch bezeichet wird. Versuchen Sie, auch die anderen Befehle in das Programm einzusetzen.

RRC (HL) hat den Code &CB,&DE; RLC (HL) hat den Code &CB,&06.
Ändern Sie den Lader und führen Sie das Programm 8 mal mit der FOR-NEXT Schleife aus. Wir erkennen hier deutlich, warum diese Befehle als Rotierbefehle bezeichnet werden. Jedes Zeichen rotiert, d.h. die Bits, die rechts bzw. links (für RRC bzw. RLC) herausfallen, werden auf der anderen Seite wieder angefügt. Nach achtmaliger Ausführung befindet sich der Bildschirm wieder in der Ausgangsposition.

Nun bleiben noch die Befehle der 9 Bit Rotation, RL (HL) (Code &CB,&16) und RR (HL) (Code &CB,&1E).
Durch den Aufruf des Programms mit RR erhält der Bildschirm ein Streifenmuster. Nach jedem weiteren Aufruf verbreitern sich diese Streifen, bis schließlich nach 8 Aufrufen der gesamte Bildschirm weiß ist. Das ist aber keinesfalls das erwartet Ergebnis. Durch die 9 Bit Rotation müßte der Bildschirminhalt um ein Bit in die jeweilige Richtung verschoben worden sein.

Vier Bildschirmbytes vor Ausführung

Nach wiederholter Ausführung von RR

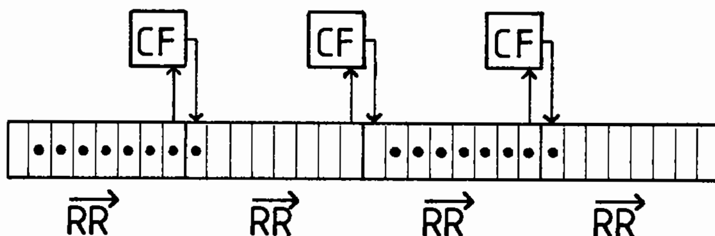


Abb. 11 Anwendung 9-Bit Rotation 4.8

Der Inhalt sollte um 1 Bit nach rechts verschoben sein, da das rotierte Bit im Carry gespeichert wird und dann in das nächste Byte hineinrotiert.
Da aber das erwartete Ergebnis nicht eingetreten ist, liegt offensichtlich ein Programmfehler vor.
Versuchen Sie, diesen Fehler zu finden und überlegen Sie sich eine Lösungsmöglichkeit!
(Tip: Achten Sie auf die Flagbeeinflussung!)

Da jeweils das 1te Bit eines Zeichens nach der Ausführung gesetzt ist (=die "Striche" auf dem Bildschirm) und dieses Bit aus dem Carry Flag geholt wird, war das Carry immer eins. Damit entsprach es nicht dem letzten Bit des vorherigen Bytes. Wie ist das möglich?

Betrachten wir die anderen Befehle des Programms. Nach der Rotation kommt der INC HL Befehl. Die 16 Bit Zählbefehle beeinflussen die Flags nicht. Darauf folgt CP H. Hier liegt der Fehler!

Aufgabe des CP Befehls ist es, Flags zu setzen. Er beeinflußt bei jedem Schleifendurchlauf das Carry. Da H größer als A (A=0) ist, wird das Carry jedesmal gesetzt (nur nicht beim ersten Durchlauf). Das gesetzte Carry Flag wird nun durch RR auf dem Bildschirm rotiert, und dieser wird weiß.

Zur Lösung dieses Problems gibt es zwei Möglichkeiten:

1. - Zwischenspeichern der Flags vor jedem CP Befehl
2. - Umgehen der Flagveränderung

Zu 1.) Mit den Stapelbefehlen ist es möglich, das F Register auf den Stapel zu retten (direkt nach dem Rotierbefehl) und dann wieder (direkt vor dem Rotierbefehl) vom Stapel zu holen. Nach RR muß also PUSH AF (=retten auf den Stapel) und vor RR der Befehl POP AF (=holen vom Stapel) eingefügt werden. Zusätzlich müssen wir beachten, daß der Stapel nicht durcheinander gerät.

Der erste Stapelbefehl in unserem Programm wäre, wie oben beschrieben, POP AF. Das wäre falsch, da dadurch Daten abgelesen werden, die noch gar nicht auf dem Stapel liegen. Vielmehr würde dadurch die Rücksprungadresse geholt werden. Das Programm würde beim Versuch eines Rücksprunges an die falsche Adresse verzweigen. Deshalb muß einmal vor der Schleife PUSH AF und nach der Schleife (vor RET) POP AF eingefügt werden.

Achten Sie bei der Benutzung von PUSH und POP immer auf die richtige Abfolge. Nach diesen Verbesserungen sieht das Programm folgendermaßen aus:

```
A000 97       10    SUB  A
A001 F5       15    PUSH AF
A002 2100C0   20    LD   HL,&C000
A005 F1       25    POP  AF
A006 CB1E     30    RR   (HL)
A008 F5       35    PUSH AF
A009 23       40    INC  HL
A00A BC       50    CP   H
A00B 20FB     60    JR   NZ,&A005
A00D F1       65    POP  AF
A00E C9       70    RET
```

Auch wenn ein BASIC Programm für diese 1 Bit Verschiebung eine Minute braucht, ist dieses Maschinenprogramm schon fast zu langsam. Durch die beiden Stapelbefehle in der Schleife, die 16 000 mal durchlaufen wird, verlängert sich das Programm unnötig. Um diesen Nachteil in Bezug auf die Geschwindigkeit wettzumachen, betrachten wir nun die zweite Möglichkeit.

Zu 2.) Damit der JR NZ Befehl funktioniert und trotzdem das Carry unbeeinflußt bleibt, benötigen wir einen Befehl, der das Z Flag aber nicht das C Flag beeinflußt. Diese Forderung erfüllen die 8-Bit Zählbefehle. Zum Erhöhen des Registerpaares HL sind zwei 8-Bit Zählbefehle notwendig. Zuerst erhöhen wir das Low Byte. Ist L nach der Erhöhung nicht 0, wird die Schleife wiederholt.
Ist L dagegen 0, so muß H um 1 erhöht werden.
Beispiel:

```
                    H=&C0   L=&FE   HL=&C0FE
nach dem Erhöhen:   H=&C0   L=&FF   HL=&C0FF
nach dem Erhöhen:   H=&C1   L=&0    HL=&C100
```

```
Der neue Programmteil:

INC  L
JR   NZ,Adresse
INC  H
JR   NZ,Adresse
RET
```

Außerdem kann der SUB A Befehl weggelassen werden, da der Akku nicht mehr benutzt wird.
Assemblerlisting:

```
A000 2100C0      10      LD  HL,&C000
A003 CB1E        20      RR  (HL)
A005 2C          30      INC L
A006 20FB        40      JR  NZ,&A003
A008 24          50      INC H
A009 20F8        60      JR  NZ,&A003
A00B C9          70      RET
```

Setzen Sie das Programm in DATA Zeilen um:

```
60 DATA &21,&00,&C0,&CB,&1E,&2C,&20,&FB
70 DATA &24,&20,&F8,&C9
```

Laden Sie es durch >RUN< mit dem BASIC Lader und probieren Sie.
Der RRD Befehl: Ändern Sie &CB (Byte 4) zu &ED und &1E zu &67 um. Nach dem Laden führt dieses Programm eine Verschiebung um 4 Bit (1 BCD Ziffer) aus.
Testen Sie folgendes BASIC Programm:

```
 5 MODE 2
10 FOR K=1 TO 4
20 FOR I=0 TO 11
30 LOCATE (K-1)*8+1,12-I:PRINT"HALLO":
35 LOCATE (K-1)*8+1,12+I:PRINT"HALLO"
40 FOR J=1 TO K
50 CALL &A000
60 NEXT J
70 NEXT I
80 NEXT K
```

4.9 BIT-Manipulationsbefehle

Im Kapitel 4.7 wurde gezeigt, wie man die logischen Operationen zum Setzen oder Rücksetzen einzelner Bits oder Gruppen von Bits im Akku benutzen kann. Es ist jedoch nützlich mit einem Befehl ein beliebiges Bit in einem beliebigen Register oder einer Speicherstelle zu setzen oder rückzusetzen. Da das eine erhebliche Anzahl von Befehlen beansprucht, stehen in den meisten CPUs dafür nur wenige oder keine Befehle zur Verfügung. Der Z80A ist in dieser Beziehung sehr gut "versorgt". Die Bit-Testbefehle eingeschlossen, besitzt er 120 Befehle zur Bitmanipulation.

Die Bit-Testbefehle prüfen, ob ein bestimmtes Bit in einem Register oder in einer Speicherstelle gesetzt oder rückgesetzt ist. Je nach Ausgang des Tests, wird das Zero Flag gesetzt oder rückgesetzt. Das Carry bleibt unbeeinflußt, S Flag und P/V Flag sind nach der Ausführung unbestimmt (!).
Die beiden Befehle zum Setzen (SET) und Rücksetzen (RES) von Bits üben keinen Einfluß auf die Flags aus.

Alle Bitbefehle beginnen mit dem Opcode &CB (wie immer mit Ausnahme der indiziert adressierten). Der zweite Opcode ergibt sich aus der Bitnummer und dem Registercode.
Zum Adressieren des betroffenen Bytes stehen folgende Adressierungsarten zur Verfügung:

```
- implizite    :Register A, B, C ,D, E, H, L
- indirekte    :(HL)
- indizierte   :(IX+d) und (IY+d)
```

Format:

```
BIT b,r     BIT b,(HL)    BIT b,(IX+d)  BIT b,(IY+d)
SET b,r     SET b,(HL)    SET b,(IX+d)  SET b,(IY+d)
RES b,r     RES b,(HL)    RES b,(IX+d)  RES b,(IY+d)
```

b=Bitnummer

Die Bitnummer b wird wie folgt codiert:

```
0- 000      4- 100
1- 001      5- 101
2- 010      6- 110
3- 011      7- 111
```

Alle diese Befehle werden auch als bitadressierte Befehle bezeichnet, da das anzusprechende Bit im Opcode angegeben wird.

Beispiele:

```
BIT 6,B        B:&33


   &X00110011  =&33
     *
     76543210  -Bitnummer

*:Bit Nummer 6 ist 0.
Das Z Flag wird, da Bit 6=0 ist, auf 1 gesetzt.
Nach der Ausführung:

    B=&33      Flag: S Z V C
                     X 1 X    X=S-,V-Flag sind
                              unbekannt

RES 1,(HL)    HL:&A975
              &1975=&23

   &X00100011  =&23
          *
     76543210  -Bitnummer

*:Bit Nummer 1 wird rückgesetzt.

   &X00100001  =&21
```

```
Speicherstelle &A975 nach der Ausführung:&21

Flags: S Z V C
                - kein Einfluß
0
SET 7,C        C:&7F

   &X01111111  =&7F
     *
     76543210  -Bitnummer

*-Bit 7 wird gesetzt.

   &X11111111  =&FF

C Flag ist nach der Ausführung:&FF

Flag: S Z V C
               - kein Einfluß
```

Analogien zum BASIC

Versuchen wir den SET b,A Befehl in BASIC nachzuvollziehen: Bit b soll gesetzt werden. Mit dem OR Befehl haben wir die Möglichkeit bestimmte Bits zu setzen. Das b-te Bit hat den Stellenwert 2^b. Es gilt:

```
SET b,A            BASIC: A=A OR (2^b)
```

Für RES gilt ähnliches:

```
RES b,A            BASIC: A=A AND (255-2^b)
```

Die Spezialbefehle SCF und CCF:

Da das Bit 0 im F Register (das Carry) besonders häufig benutzt wird, gibt es dafür zwei Spezialbefehle.

SCF setzt das Carry auf den Wert 1.
CCF komplementiert den Wert des Carry F., d.h. aus C=0 wird C=1 und umgekehrt.
Das sind die einzigen Befehle, mit denen die Flags direkt beeinflußt werden können.
Mit den logischen Befehlen kann das Carry gelöscht werden.
Die Befehle CCF und SCF finden Sie in der Befehlsliste am Ende des Kapitels 4.11.

Mnemonic	Symbolic Operation	Flags C	Z	P/V	S	N	H	Op-Code 76 543 210	No. of Bytes	No. of M Cycles	No. of T States	Comments
BIT b, r	Z ← $\overline{r}_b$	•	↕	X	X	0	1	11 001 011	2	2	8	
								01 b r				
BIT b, (HL)	Z ← $\overline{(HL)}_b$	•	↕	X	X	0	1	11 001 011	2	3	12	
								01 b 110				
BIT b, (IX+d)	Z ← $\overline{(IX+d)}_b$	•	↕	X	X	0	1	11 011 101	4	5	20	
								11 001 011				
								← d →				
								01 b 110				
BIT b, (IY+d)	Z ← $\overline{(IY+d)}_b$	•	↕	X	X	0	1	11 111 101	4	5	20	
								11 001 011				
								← d →				
								01 b 110				
SET b, r	r_b ← 1	•	•	•	•	•	•	11 001 011	2	2	8	
								[11] b r				
SET b, (HL)	$(HL)_b$ ← 1	•	•	•	•	•	•	11 001 011	2	4	15	
								[11] b 110				
SET b, (IX+d)	$(IX+d)_b$ ← 1	•	•	•	•	•	•	11 011 101	4	6	23	
								11 001 011				
								← d →				
								[11] b 110				
SET b, (IY+d)	$(IY+d)_b$ ← 1	•	•	•	•	•	•	11 111 101	4	6	23	
								11 001 011				
								← d →				
								[11] b 110				
RES b, m	s_b ← 0 m≡r, (HL), (IX+d), (IY+d)							[10]				To form new OP-code replace [11] of SET b,m with [10]. Flags and time states for SET instruction

r	Reg.
000	B
001	C
010	D
011	E
100	H
101	L
111	A

b	Bit Tested
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Notes: The notation s_b indicates bit b (0 to 7) or location s.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

Programme zu den Bitbefehlen

Schreiben Sie ein Programm, das den Bildschirm im Modus 2 mit Streifen füllt, die einen Punkt breit sind und in der Mitte eines Zeichens liegen. Benutzen Sie dabei wieder die Schleife aus den vorherigen Programmen.

Assemblerlisting

```
A000 2100C0    10   LD  HL,&C000
A003 CBDE      20   SET 3,(HL)
A005 2C        30   INC L
A006 20FB      40   JR  NZ,&A003
A008 24        50   INC H
A009 20F8      60   JR  NZ,&A003
A00B C9        70   RET
```

BASIC Programm

```
  5 MEMORY &9FFF
 10 FOR i=&A000 TO &A00B
 20 READ a
 30 POKE i,a
 40 NEXT i
 50 MODE 2
 60 CALL &A000
 70 END
100 DATA &21,&00,&C0,&CB,&DE,&2C,&20,&FB
110 DATA &24,&20,&F8,&C9
```

An Stelle von SET 3,(HL) kann auch SET 4,(HL) Code:&CB,&E6) eingesetzt werden. In der DATA Zeile muß dann &DE durch &E6 ersetzt werden.

4.10 Sprünge

Ein Großteil der Sprünge ist bedingt, d.h. vom Status eines Flags abhängig. Wir werden hier die Rolle jedes Flags nochmal zusammenfaßend beschreiben.

Die beiden Flags H und N werden bei der BCD Arithmetik verwendet. Sie können nicht getestet werden. Die anderen Flags (C, P/V, Z, S), können bei einer bedingten Verzweigung getestet werden.

Carry Flag (Übertrag,C)

Das Carry Flag hat zwei Funktionen.
- Es gibt an, ob bei einer Addition oder Subtraktion ein Übertrag auftrat.
- Die Befehle SRL, SRA, SLA, RR, RL, RRC, RLC, RRA, RLA, RRCA und RLCA benutzen das Carry als 9tes Bit.

Eine Ausnahme bilden die BCD-Rotierbefehle RLD, RRD. Sie beeinflussen das Carry nicht.

Die logischen Befehle AND, OR und XOR setzen das Carry immer auf 0. Sie können verwendet werden, um das Carry zu löschen. Folgende Befehle rufen außerdem eine Veränderung des Carry hervor.

NEG: C Flag wird gesetzt, wenn A vor dem Befehl 0 war.

DAA: Die Beeinflussung dieses Befehls ist kompliziert. Weil wir die BCD arithmetischen Befehle nicht behandelt haben, gehen wir nicht näher auf diese Beeinflussung ein.

SCF: Set Carry Flag
Dieser Befehl setzt Carry=1.

CCF: Complement Carry Flag

Dieser Befehl komplementiert das Carry.

Alle anderen Befehle beeinflussen das Carry nicht!

Parity/Overflow (Parität/Überlauf-P/V)

Dieses Flag hat mehrere Funktionen, abhängig von dem ausgeführten Befehl.

- Überlauf

Bei den arithmetischen Operationen 8-Bit ADD, ADC, SUB, SBC, 8-Bit INC, 8 Bit DEC, NEG und bei CP zeigt es einen Überlauf an. Das bedeutet, daß das Vorzeichen einer Zahl fehlerhaft geändert wurde.

Ausnahmen: 16-Bit ADD, 16-Bit INC, 16-Bit DEC
Diese Befehle beeinflussen V nicht.

- Parität

Bei Inputbefehlen (IN), Rotation- und Schiebebefehlen RR, RL ,RRC, RLC, RLD, RRD, SLA, SRA und SRL, logischen Befehlen AND, OR, XOR und bei DAA wird dieser Flag als P Flag benutzt. P ist 1, wenn die Anzahl der Einsen eines Bytes gerade ist und 0, wenn die Anzahl der gesetzten Bits ungerade ist.

Ausnahmen: RLA, RRA, RLCA, RRCA beeinflussen P nicht!

Bei den Blockbefehlen LDD, LDI, CPD, CPI, CPDR und CPIR ist P/V zurückgesetzt, wenn BC=0 war (BC ist das Zählregister), sonst gesetzt.
Aus diesem Grund wird P/V durch LDDR und LDIR immer zurückgesetzt.

- Interrupt Flag

Bei LD A,I und LD A,R wird P/V auf den Wert der Interrupt Enable Flip Flops (IFF) gesetzt. Dieser ist 0, wenn die maskierbaren Interrupts gesperrt sind, und 1, wenn sie zugelassen sind.

ACHTUNG: Der Bitbefehl und alle Block Ein-/ und Ausgabebefehle setzen dieses Flag willkürlich, d.h. sie verändern unter Umständen den vorherigen Wert. Andere Befehle beeinflussen dieses Flag nicht.

Zero Flag (Null,Z)

Das Z Flag zeigt an, ob der Wert eines Bytes Null ist. Ist er 0, wird Z gesetzt, sonst rückgesetzt.
Bei den Vergleichsbefehlen wird Z bei vorliegender Gleichheit auf 1 gesetzt sonst rückgesetzt.
Beim Bitbefehl wird das Zero Flag auf 1 gesetzt, wenn das getestete Bit 0 ist, sonst rückgesetzt.
Folgende Befehle beinflussen das Z Flag:

```
arithmetische     : ADD,ADC,SUB,SBC,INC,DEC,NEG,DAA
         ACHTUNG  : 16-Bit ADD,16-Bit INC,16-Bit DEC:Kein
                    Einfluß!
Vergleich         : CP : Z=1 bei Gleichheit, sonst Z=0
Bit               : BIT
Rotier/Schiebe    : RR,RL,RRC,RLC,SRL,SRA,SLA,RLD,RRD
         ACHTUNG  : RRA,RLA,RRCA,RLCA: Kein Einfluß!
Block/Suchen      : CPI,CPIR,CPD,CPDR: Z=1 bei Gleichheit
Eingabe           : IN
Ladebefehle       : LD A,I bzw. LD A,R
```

Blockein/ausgabe: Z ist gesetzt, wenn nach der Ausführung B=0 ist, d.h. INI,IND,OUTI,OUTD beeinflussen Z in dieser Weise und INIR;INOR;OTIR;OTDR setzen Z auf 1.
Alle anderen Befehle beeinflussen Z nicht!

Sign Flag (Vorzeichen, S)

Das Sign Flag enthält den Wert des höchsten Bit eines Bytes. Dieses Bit entspricht bei der vorzeichenbehafteten Arithmetik dem Vorzeichen.

Folgende Befehle beeinflussen das S Flag:

Alle arithmetisch bzw. logischen Befehle:
ADD,ADC,SUB,SBC,INC,DEC,NEG,DAA,AND,OR,XOR,CP

Die Rotier- und Schiebebefehle:
RL,RR,RLC,RRC,SRL,SRA,SLA,RLD,RRD,

Blocksuchbefehle CPD,CPI,CPDR,CPIR,

Eingabebefehl IN und die Ladebefehle LD A,I und LD A,R

ACHTUNG: 16-Bit ADD,16-Bit INC,16-Bit DEC, RLA, RRA, RLCA, RRCA: Kein Einfluß!

Der Bitbefehl und die Block Ein-/Ausgabebefehle INI,IND,OUTI,OUTD,INIR,INDR,OTIR,OTDR setzen das S Flag willkürlich in einen unbestimmten Zustand.

Die Flagbeeinflussung der einzelnen Befehle, können sie auch im Anhang nachlesen.

Es gibt fünf verschiedenen Arten von Sprüngen beim Z80A.

- Sprünge innerhalb des Hauptprogramms (JUMP), die dem BASIC Befehl >GOTO< entsprechen.
- Unterprogrammverzweigungen (CALL und RET), die den BASIC Befehlen >GOSUB< und >RETURN< entsprechen.
- Relative Sprünge (JUMP RELATIVE), die dem BASIC Befehl
  >FOR-NEXT< ähnlich sind.
- Restart Befehle (RST), die eine Verzweigung zu einer fest vorgegebenen Adresse ausführen. Der RST Befehl besitzt kein BASIC Analog.
- Interruptsprünge (siehe Steuerbefehle)

Die ersten drei Verzweigungsarten sind beim Z80A als unbedingte und bedingte Sprünge, d.h. in Abhängigkeit eines Flag Status, vorhanden. Bei den bedingten Sprüngen kann aufgrund der Flags Z ,C , P/V und S gesprungen werden. Jedes Flag kann entweder auf den Wert 0 oder 1 getestet werden.
In der Assemblersprache gelten folgende Abkürzungen:

```
 Z= Springe wenn Null              (Z=1)
NZ= Springe wenn nicht Null        (Z=0)
 C= Springe wenn Übertrag          (C=1)
NC= Springe wenn kein Übertrag     (C=0)
PO= Springe wenn ungerade Parität (P/V=0)
PE= Springe wenn gerade   Parität (P/V=1)
 P= Springe wenn plus  (+)         (S=0)
 M= Springe wenn minus (-)         (S=1)
```

Zusätzlich kennt der Z80A einen speziellen Schleifenbefehl, der das B Register dekrementiert und dann einen relativen Sprung ausführt, solange B<>0 ist. Dieser Befehl heißt DJNZ (Dekrementiere Jump Non Zero).

JUMP/JP

Die Verzweigungen im Hauptprogramm werden durch den JP Befehl ausgeführt. Die Sprungadresse kann auf zwei Arten adressiert sein.

Absolute Adressierung:

Format:

JP nn   oder   JP cc,nn

cc steht für eine Bedingung (Condition), also für Z, NZ, C, NC, PO, PE, P oder M.

```
JP nn     - springt "unbedingt" an die angegebene Adresse.
JP cc,nn  - springt an die angegebene Adresse,
```

wenn die Bedingung erfüllt ist. Ist die Bedingung nicht erfüllt, wird der nächste Befehl ausgeführt.

Analogie

```
JP nn           BASIC: GOTO Zeilennummer
JP NZ,nn        BASIC: IF Z=0 THEN GOTO Zeilennummer
JP Z,nn         BASIC: IF Z=1 THEN GOTO Zeilennummer
```

Der Prozessor führt einen Sprung aus, indem er die angegebene Adresse in den PC liest. Dann wird an dieser Stelle der nächste Befehlscode gelesen und ausgeführt.

Bei der absoluten Adressierung folgt auf den 1 Byte Opcode die jeweilige Sprungadresse in der Reihenfolge Low Byte, High Byte. Da alle 3-Byte Befehle relativ langsam sind, wurden die relativen Sprünge ermöglicht, da sie nur 2-Byte Befehle sind. Die indirekt adressierten Sprünge haben einen 1-Byte Opcode.

Indirekte Adressierung

Format:

JP (x)

x: HL,IX oder IY

JP (x) springt an die im Register x angegebene Adresse.

CALL/RET

Wie die Rücksprungadressen bei CALL und RET mit Hilfe des Stapels und des SP gespeichert bzw. gelesen werden, haben wir bereits besprochen. Ein Aufruf eines Unterprogrammes ist bedingt oder unbedingt möglich. Die Sprungadresse (=Startadresse des Unterprogrammes) wird absolut angegeben.

Format:

CALL nn oder CALL cc,nn

Bei der Ausführung werden alle notwendigen Operationen am Stapel, SP und PC vorgenommen. Der Ablauf ist folgendermaßen:

Nach dem kompletten Einlesen des Befehls, zeigt PC auf den nächstfolgenden Befehl. Dann folgen die Operationen:

(SP-1)=PC -(High Byte)
(SP-2)=PC -(Low Byte)
SP=SP-2
PC=nn

Der nächste Befehl wird von der Adresse auf die PC dann zeigt, gelesen. Zum Abschluß eines Unterprogramms wird ein RET Befehl gesetzt. Auch das RETURN ist unbedingt oder bedingt möglich.

Format:

RET oder RET cc

Bei der Ausführung des RET Befehls geschieht folgendes:

PC -(Low Byte)=(SP)
PC -(High Byte)=(SP+1)
SP=SP+2

Die Programmausführung wird an der vom Stapel geholten Adresse fortgesetzt.
Der RET Befehl ist im Gegensatz zum CALL Befehl nur 1 Byte lang. Bei CALL muß die 16-Bit Adresse angegeben werden, d.h. dieser Befehl ist 3 Bytes lang.

Es gibt zwei Spezialrücksprünge RETI und RETN, die im Kapitel Steuerbefehle besprochen werden.

RESTART RST

Dieser Typ von Sprungbefehlen hat die minimale Länge von einem Byte und wird daher am schnellsten von allen Sprungbefehlen ausgeführt (ausgenommen RET). Der RST Befehl, den wir in Zukunft als Restart bezeichnen, bewirkt einen Unterprogrammsprung an eine Adresse im unteren Teil des Speichers. Es gibt acht Restartbefehle. Die Verzweigungsadressen der Restarts sind &0, &8, &10, &18, &20, &28, &30 und &38.

Format:

```
RST p
  p:Eine der oben genannten 8-Bit Adressen.
```

Da der Restart der schnellste Sprungbefehl ist, stehen im unteren Teil des Speichers (&0-&40) sehr wichtige, oft benutzte Routinen bzw. Sprünge zu diesen Routinen. Die genaue Funktion der einzelnen Restartbefehle werden wir später untersuchen.

JUMP RELATIV / JR

Die relativen Sprünge springen relativ zur aktuellen Adresse. Die Sprungweite (Distanz) muß angegeben werden. Das erste Byte ist der Opcode und das zweite gibt die Distanz mit Vorzeichen an (im Zweierkomplement). Dieses Verfahren bezeichnet man als relative Adressierung. Die Distanz nennt man in diesem Fall den Offset.

Format:

```
JR e      oder    JR x,e
   e: Offset
   x: Z,NZ,C,NZ
```

Bedingte relative Sprünge sind nur aufgrund des C und Z Flags möglich. Wie wird der Offset berechnet ?

Betrachten wir das letzte Programm von Kapitel 4.9. An Adresse &A006 steht der JR Befehl. Das Sprungziel ist der SET 3,(HL) Befehl an Adresse &A003. Die Differenz ist also &A006-&A003=3. Da es sich um einen "Rückwärtssprung" handelt (Zieladresse ist kleiner als die "Absprungadresse"), ist der Offset -3. Um das 2te Byte des Befehls zu erhalten, müssen wir vom Offset zwei subtrahieren.

Warum ist diese Subtraktion notwendig?

Der Prozessor liest immer erst den kompletten Befehl ein, in diesem Fall also den Opcode (Byte 1) und den Offset (Byte 2). Nach jedem "Lesen" wird PC um eins erhöht. Nachdem der Befehl komplett gelesen wurde, steht der PC auf der Anfangsadresse des nächsten Befehls. Der Programmzeiger ist folglich um 2 höher, als die Adresse des Sprungbefehls. Der Z80A führt den Sprung aus, indem er die Distanz zum PC addiert. Aus diesem Grund müssen wir die Erhöhung des PC um 2 mit berücksichtigen. Bei einem "Rückwärtssprung" ist es notwendig, diese beiden Bytes mit zu überspringen. Die zu speichernde Distanz berechnet sich aus:

-3-2=-5= &FB im Zweierkomplement

Dieses Byte ist im Assemblerlisting an Adresse &A007, auf den Opcode an Adresse &A006 folgend, angegeben. In Assemblersprache wird diese Differenz von 2 nicht angegeben. Der Befehl lautet JR NZ,&A003. Das Assemblerprogramm berechnet automatisch die Sprungdifferenz und führt die Subtraktion von 2 und die Umrechnung ins Zweierkomplemente durch. Obwohl im Assemblerbefehl die 16-Bit Adresse angegeben ist, handelt es sich um einen relativen Sprung. Unter Berücksichtigung der Subtraktion sind Sprünge von +129 bis zu -126 Bytes relativ zur aktuellen Adresse möglich.

Fassen wir die Art und Weise der Berechnung des Offsetbytes zusammen:

Sprungbefehl steht an Adresse ADR
Sprungziel steht an Adresse ADRZ
Offset = ADRZ-ADR
Zu speicherndes Byte: (Offset-2) im Zweierkomplement

Mnemonic	Symbolic Operation	C	Z	P/V	S	N	H	Op-Code 76 543 210	No. of Bytes	No. of M Cycles	No. of T States	Comments
JP nn	PC ← nn	•	•	•	•	•	•	11 000 011	3	3	10	
								← n →				
								← n →				
JP cc, nn	If condition cc is true PC ←nn, otherwise continue	•	•	•	•	•	•	11 cc 010	3	3	10	
								← n →				
								← n →				
JR e	PC ← PC + e	•	•	•	•	•	•	00 011 000	2	3	12	
								← e-2 →				
JR C, e	If C = 0, continue	•	•	•	•	•	•	00 111 000	2	2	7	If condition not met
								← e-2 →				
	If C = 1, PC ← PC+e								2	3	12	If condition is met
JR NC, e	If C = 1, continue	•	•	•	•	•	•	00 110 000	2	2	7	If condition not met
								← e-2 →				
	If C = 0, PC ← PC + e								2	3	12	If condition is met
JR Z, e	If Z = 0 continue	•	•	•	•	•	•	00 101 000	2	2	7	If condition not met
								← e-2 →				
	If Z = 1, PC ← PC + e								2	3	12	If condition is met
JR NZ, e	If Z = 1, continue	•	•	•	•	•	•	00 100 000	2	2	7	If condition not me
								← e-2 →				
	If Z = 0, PC ← PC + e								2	3	12	If condition met
JP (HL)	PC ← HL	•	•	•	•	•	•	11 101 001	1	1	4	
JP (IX)	PC ← IX	•	•	•	•	•	•	11 011 101	2	2	8	
								11 101 001				
JP (IY)	PC ← IY	•	•	•	•	•	•	11 111 101	2	2	8	
								11 101 001				
DJNZ,e	B ← B-1 If B = 0, continue	•	•	•	•	•	•	00 010 000	2	2	8	If B = 0
								← e-2 →				
	If B ≠ 0, PC ← PC + e								2	3	13	IF B ≠ 0

cc	Condition
000	NZ non zero
001	Z zero
010	NC non carry
011	C carry
100	PO parity odd
101	PE parity even
110	P sign positive
111	M sign negative

Notes: e represents the extension in the relative addressing mode.

e is a signed two's complement number in the range <-126, 129>

e-2 in the op-code provides an effective address of pc +e as PC is incremented by 2 prior to the addition of e.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

Mnemonic	Symbolic Operation	Flags C	Z	P/V	S	N	H	Op-Code 76 543 210	No. of Bytes	No. of M Cycles	No. of T States	Comments
CALL nn	(SP-1)←PC_H (SP-2)←PC_L PC←nn	•	•	•	•	•	•	11 001 101 ← n → ← n →	3	5	17	
CALL cc, nn	If condition cc is false continue, otherwise same as CALL nn	•	•	•	•	•	•	11 cc 100 ← n → ← n →	3 3	3 5	10 17	If cc is false If cc is true
RET	PC_L←(SP) PC_H←(SP+1)	•	•	•	•	•	•	11 001 001	1	3	10	
RET cc	If condition cc is false continue, otherwise same as RET	•	•	•	•	•	•	11 cc 000	1 1	1 3	5 11	If cc is false If cc is true
RETI	Return from interrupt	•	•	•	•	•	•	11 101 101 01 001 101	2	4	14	
RETN	Return from non maskable interrupt	•	•	•	•	•	•	11 101 101 01 000 101	2	4	14	
RST p	(SP-1)←PC_H (SP-2)←PC_L PC_H←0 PC_L←P	•	•	•	•	•	•	11 t 111	1	3	11	

cc	Condition	
000	NZ	non zero
001	Z	zero
010	NC	non carry
011	C	carry
100	PO	parity odd
101	PE	parity even
110	P	sign positive
111	M	sign negative

t	P
000	00H
001	08H
010	10H
011	18H
100	20H
101	28H
110	30H
111	38H

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown
‡ = flag is affected according to the result of the operation.

Aufgabe:

Im Assemblerlisting (Kaptil 4.9) steht ein relativer Sprung an Adresse &A009. Sprungziel ist wieder &A003. Berechnen Sie das Offsetbyte und vergleichen Sie Ihr Ergebnis mit dem Assemblerlisting.

Damit haben wir die wichtigsten Befehle behandelt. Greifen wir auf ein Programm aus dem Kapitel über Ladebefehle zurück.

Aufgabe des Programms war es, das linke obere Kästchen des Bildschirms zu füllen. Diese Aufgabe kann besser mit einer Schleife gelöst werden.

Im BASIC erhalten wir:

```
10 FOR I=&C000 TO &FFFF STEP &800
20 POKE I,&FF
30 NEXT
```

Zum Umformulieren des Programms in Maschinensprache laden wir das HL Registerpaar mit der Startadresse &C000. Um die STEP &800 Anweisung zu übersetzen, wird DE mit &800 geladen und eine 16-Bit Addition durchgeführt. Ist nach der Addition das Carry gesetzt, so ist das Programmende erreicht. Schreiben Sie mit Hilfe dieser Angaben das Maschinenprogramm.

Lösung:

```
A000 2100C0    10    LD   HL,&C000
A003 110008    20    LD   DE,&800
A006 36FF      30    LD   (HL),&FF
A008 19        40    ADD  HL,DE
A009 30FB      50    JR   NC,&A006
A00B C9        60    RET
```

Verändern Sie nun dieses Programm derartig, daß das Kästchen nicht gefüllt, sondern das jeweilige Zeichen invers dargestellt wird.

Lösung:

```
A000 2100C0   10   LD   HL,&C000
A003 110008   20   LD   DE,&800
A006 7E       30   LD   A,(HL)
A007 2F       40   CPL
A008 77       50   LD   (HL),A
A009 19       60   ADD  HL,DE
A00A 30FA     70   JR   NC,&A006
A00C C9       80   RET
```

Anstelle der Invertierung des jeweiligen Bytes mit CPL ist natürlich auch ein XOR &FF Befehl möglich. Dieser ist aber länger (2 Bytes) und damit langsamer.

Der DJNZ Befehl ermöglicht eine komfortablere Schleifenprogrammierung. Der Offset wird wie bei JR als zweites Byte angegeben. Das B Register wird als Zähler verwendet. Um acht Schleifenwiederholungen zu erreichen, muß B mit 8 geladen werden, da bei B=0 nicht mehr gesprungen wird. Der JR Befehl wird durch DJNZ ersetzt, und am Anfang wird B mit 8 geladen.

Assemblerlisting

```
A000 0609     10   LD   B,8
A002 2100C0   20   LD   HL,&C000
A005 110008   30   LD   DE,&800
A008 7E       40   LD   A,(HL)
A009 2F       50   CPL
A00A 77       60   LD   (HL),A
A00B 19       70   ADD  HL,DE
A00C 10FA     80   DJNZ &A008
A00E C9       90   RET
```

4.11 Steuerbefehle

Die Steuerbefehle verändern bzw. beeinflussen die Betriebsart oder den Ablauf in der CPU.

Der NOP Befehl

NOP steht für No Operation. NOP ist also ein Befehl ohne Funktion. Das erscheint paradox, hat jedoch seine Berechtigung. Einerseits kann NOP zur absichtlichen Verzögerung benutzt werden (ein NOP Befehl dauert bei den CPC's eine Mikrosekunde, andererseits kann dieser Befehl als Platzhalter in Programmen verwendet werden. Eine Fehlersuche und Fehlerbeseitigung kann dadurch vereinfacht werden. Der Opcode von NOP ist &00, d.h. läuft das Programm versehentlich in einen gelöschten Bereich, wird nichts zerstört oder geändert, da NOP keine Veränderungen verursacht.

HALT Befehl

Dieser Befehl unterbricht die Operationen der CPU solange, bis ein Reset oder ein Interrupt auftritt.

Interrupt-/Steuerbefehle

Ein Interrupt (-Unterbrechung) dient vorrangig der Bearbeitung wichtiger Abläufe im Rechner. Ein Interrupt ist die Meldung eines Bausteins über den Eintritt eines Zustandes, wie z.B. das Warten eines I/O-Gerätes auf Eingabe. Diese Meldungen werden nach Wichtigkeit von der CPU verarbeitet. Ein normal ablaufendes Programm wird durch einen Interrupt unterbrochen. Interrupts spielen bei der Ein- und Ausgabe eine wichtige Rolle. Die Schneider bieten die Möglichkeit, Interrupts auch vom BASIC aus zu programmieren (ERVERY - AFTER). Der Interrupt wird bei diesen Befehlen durch die interne Uhr des Prozessors ausgelöst. Wird ein zugelassener Interrupt angefordert, so verzweigt das Programm an die Startadresse eines Unterpro-

gramms, das die dem jeweiligen Interrupt entsprechenden Aktionen ausführt. Aus diesem Interrupt Bedienprogramm wird mit RETI (Return Interrupt) ins Hauptprogramm zurückgesprungen.

Es wird zwischen maskierbaren und nichtmaskierbaren Interrupts unterschieden. Letztere werden unter allen Umständen ausgeführt. Sie besitzen höchste Priorität. ein Rücksprung von einem Non Mascerable Interrupt ist mit RETN möglich.

DI Disable Interrupt und EI

Der DI Befehl bewirkt, daß ab dem Zeitpunkt seiner Ausführung maskierbarer Interrupts nicht beachtet werden. Die Interrupts sind so lange gesperrt, bis sie durch EI (Enable Interrupt) wieder zugelassen werden.

Der Z80A besitzt drei Interrupt Modi : IM 0, IM 1, IM 2

IM 0 (Interrupt Modus 0)

Mit IM 0 kann vom Standardmodus 1 in den Modus 0 geschaltet werden.
Nach einem Interrupt wartet der Prozessor in diesem Modus auf den Befehl eines externen Gerätes.

IM 1

Das ist der Standardmodus, der nach dem Einschalten des Computers vorliegt.
In diesem Modus wird bei Auftreten eines Interrupts automatisch an eine festgelegte Adresse verzweigt.

IM 2 (Vektorinterrupt)

Im IM 2 wird an eine in einer Tabelle stehenden vom Interrupt abhängigen Adresse verzweigt.

Mnemonic	Symbolic Operation	Flags C	Z	P/V	S	N	H	Op-Code 76 543 210	No. of Bytes	No. of M Cycles	No. of T States	Comments
DAA	Converts acc. content into packed BCD following add or subtract with packed BCD operands	↕	↕	P	↕	•	↕	00 100 111	1	1	4	Decimal adjust accumulator
CPL	A ← $\bar{A}$	•	•	•	•	1	1	00 101 111	1	1	4	Complement accumulator (one's complement)
NEG	A ← 0 − A	↕	↕	V	↕	1	↕	11 101 101 01 000 100	2	2	8	Negate acc. (two's complement)
CCF	CY ← $\overline{CY}$	↕	•	•	•	0	X	00 111 111	1	1	4	Complement carry flag
SCF	CY ← 1	1	•	•	•	0	0	00 110 111	1	1	4	Set carry flag
NOP	No operation	•	•	•	•	•	•	00 000 000	1	1	4	
HALT	CPU halted	•	•	•	•	•	•	01 110 110	1	1	4	
DI	IFF ← 0	•	•	•	•	•	•	11 110 011	1	1	4	
EI	IFF ← 1	•	•	•	•	•	•	11 111 011	1	1	4	
IM 0	Set interrupt mode 0	•	•	•	•	•	•	11 101 101 01 000 110	2	2	8	
IM 1	Set interrupt mode 1	•	•	•	•	•	•	11 101 101 01 010 110	2	2	8	
IM2	Set interrupt mode 2	•	•	•	•	•	•	11 101 101 01 011 110	2	2	8	

Notes: IFF indicates the interrupt enable flip-flop
CY indicates the carry flip-flop.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

4.12 Ein-Ausgabebefehle

I/O Befehle sind eine oft vernachlässigte Gruppe von Befehlen des Z80A. Das liegt daran, daß Wirkung und Einsatzweise der I/O Befehle sehr hardwareabhängig ist. Jeder Rechner benutzt zur Kommunikation mit Peripheriegeräten mindestens einen speziellen I/O IC. Ihr Rechner besitzt einen sog. PPI (Programmable Peripheral Interface (8255)). Auch andere Bezeichnungen sind gebräuchlich, wie PIO, PIA.

Ein Teil der I/O Befehle bezieht sich auf diesen Baustein, der dann selbstständig die Kommunikation mit Tastatur, Kassette und Speichersteuerung durchführt. Außerdem sind der PSG (Programmable Sound Generator), der für die Tonerzeugung und Joystickabfrage zuständig ist, und der CRTC, der das Monitorbild erzeugt, über I/O Leitungen mit dem Prozessor verbunden. Wie Sie sehen, sind alle, über I/O Befehle ansprechbaren Bausteine, wichtige Teile des Rechners. Die Verbindung zwischen diesen sehr selbstständig arbeitenden Bausteinen (d.h. z.B. der CRTC erzeugt ständig und unabhängig von der CPU das Videoausgangssignal) und dem Z80A läuft über besagte I/O Befehle. Mit Hilfe dieser werden Befehle und Daten an die IC Bausteine übermittelt oder von ihnen empfangen. Die I/O Bausteine übernehmen dann die eigentliche Kommunikation mit den externen Geräten.

Für solche Verbindungen wird in der "Computersprache" oft der Begriff "Schnittstelle" verwendet. Wie beschrieben, gibt es also interne Schnittstellen (meist: Prozessor-I/O Bausteine) und externe Schnittstellen (meist: I/O Baustein- Gerät).

Ein I/O Befehl legt (oder schreibt) nun einfach einen Wert auf die jeweilige Schnittstelle. Ist ein Gerät angeschlossen, so wird der Wert empfangen und die entsprechende Aktion ausgeführt. Insgesamt sind im Normalfall 256 verschiedene I/O Adressen möglich. Die Adresse (auch: Adresse des I/O Ports) bestimmt, an welches Gerät die Daten "gesendet" werden.

Aus dem Voranstehenden können Sie entnehmen, daß ein vollständiger Befehl zwei Angaben braucht:

- Die I/O Portadresse
- Den Wert der Daten, die "gesendet" werden bzw. das Register, in dem die empfangenen Daten gespeichert werden sollten.

Die Portadresse steht dabei immer in Klammern. Beim IN Befehl werden Daten vom Port gelesen und in dem angegebenen Register abgespeichert. Der OUT Befehl "sendet" die angegebenen Daten zum jeweiligen Port.
Die I/O Befehle gibt es in zwei Adressierungsarten:

Unmittelbare Adressierung

Format:

OUT (n),A IN A,(n)

Indirekte Adressierung

Format:

OUT (C),r IN r,(C)

Das Schneider BASIC stellt analog zu diesen Befehlen den >INP< und den >OUT< Befehl zur Verfügung. Ihre Wirkungsweise ist den Maschinensprachebefehlen gleich, jedoch sind viele Anwendungen nur von Maschinensprache aus zu realisieren.

Außerdem gibt es noch je vier Befehle zur Block Ein- / Ausgabe. Sie werden ähnlich den Blocktransferbefehlen benutzt, wobei HL auf die jeweilige Adresse im Speicher zeigt, C die Portadresse, und B die Blocklänge darstellt.

Die I/O Befehle beginnen mit dem Code &ED und besitzen einen 2 Byte Code. Nur die unmittelbar adressierten Befehle IN A,(n) und OUT (n),A werden durch einen 1 Byte Code übersetzt.

Bei der Benutzung der I/O Befehle auf den Schneider Rechnern sind einige Einschränkungen des üblichen Z80A Befehlssatzes vorhanden. Normalerweise wirddie Portadresse, z.B. bei unmittelbarer Adressierung als 1-Byte Wert angegeben. Dieses Byte wird auf die Adressleitung &A0 bis &A7 gelegt.

Beim Schneider sind jedoch die Adressleitungen &A8 bis &A15 mit den I/O Bausteinen verbunden. Das bedeutet, daß eine 2-Byte Portadresse angegeben werden muß, wobei nur das High Byte die Portselektion bestimmt. Diese Möglichkeit ist jedoch nur bei der indirekten Adressierung vorhanden, indem das B Register mit der Portadresse geladen wird.

Obwohl der Befehl lediglich:

OUT (C)r bzw. IN r,(C)

lautet, wird intern der Inhalt des B Registers auf die Adressleitung &A8 bis &A15 gelegt. Die Zuordnung der Portadressen zu den Bausteinen ist relativ kompliziert, da oft nur bestimmte Bitkombinationen einen Baustein ansprechen.

Durch diese Methode der Portadressierung bei den Schneider Computern, sind die I/O Befehle der unmittelbaren Adressierung und die Block I/O Befehle nicht benutzbar.

Mnemonic	Symbolic Operation	C	Z	P/V	S	N	H	Op-Code 76 543 210	No. of Bytes	No. of M Cycles	No. of T States	Comments
IN A, (n)	A ← (n)	•	•	•	•	•	•	11 011 011	2	3	11	n to $A_0 \sim A_7$
								← n →				Acc to $A_8 \sim A_{15}$
IN r, (C)	r ← (C)	•	↕	P	↕	0	↕	11 101 101	2	3	12	C to $A_0 \sim A_7$
	if r = 110 only the flags will be affected							01 r 000				B to $A_8 \sim A_{15}$
INI	(HL) ← (C)	X	↕ (1)	X	X	1	X	11 101 101	2	4	16	C to $A_0 \sim A_7$
	B ← B - 1							10 100 010				B to $A_8 \sim A_{15}$
	HL ← HL + 1											
INIR	(HL) ← (C)	X	1	X	X	1	X	11 101 101	2	5 (If B ≠ 0)	21	C to $A_0 \sim A_7$
	B ← B - 1							10 110 010				B to $A_8 \sim A_{15}$
	HL ← HL + 1								2	4 (If B = 0)	16	
	Repeat until B = 0											
IND	(HL) ← (C)	X	↕ (1)	X	X	1	X	11 101 101	2	4	16	C to $A_0 \sim A_7$
	B ← B - 1							10 101 010				B to $A_8 \sim A_{15}$
	HL ← HL - 1											
INDR	(HL) ← (C)	X	1	X	X	1	X	11 101 101	2	5 (If B ≠ 0)	21	C to $A_0 \sim A_7$
	B ← B - 1							10 111 010				B to $A_8 \sim A_{15}$
	HL ← HL - 1								2	4 (If B = 0)	16	
	Repeat until B = 0											
OUT (n), A	(n) ← A	•	•	•	•	•	•	11 010 011	2	3	11	n to $A_0 \sim A_7$
								← n →				Acc to $A_8 \sim A_{15}$
OUT (C), r	(C) ← r	•	•	•	•	•	•	11 101 101	2	3	12	C to $A_0 \sim A_7$
								01 r 001				B to $A_8 \sim A_{15}$
OUTI	(C) ← (HL)	X	↕ (1)	X	X	1	X	11 101 101	2	4	16	C to $A_0 \sim A_7$
	B ← B - 1							10 100 011				B to $A_8 \sim A_{15}$
	HL ← HL + 1											
OTIR	(C) ← (HL)	X	1	X	X	1	X	11 101 101	2	5 (If B ≠ 0)	21	C to $A_0 \sim A_7$
	B ← B - 1							10 110 011				B to $A_8 \sim A_{15}$
	HL ← HL + 1								2	4 (If B = 0)	16	
	Repeat until B = 0											
OUTD	(C) ← (HL)	X	↕ (1)	X	X	1	X	11 101 101	2	4	16	C to $A_0 \sim A_7$
	B ← B - 1							10 101 011				B to $A_8 \sim A_{15}$
	HL ← HL - 1											
OTDR	(C) ← (HL)	X	1	X	X	1	X	11 101 101	2	5 (If B ≠ 0)	21	C to $A_0 \sim A_7$
	B ← B - 1							10 111 011				B to $A_8 \sim A_{15}$
	HL ← HL - 1								2	4 (If B = 0)	16	
	Repeat until B = 0											

Notes: (1) If the result of B - 1 is zero the Z flag is set, otherwise it is reset.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

KAPITEL V: PROGRAMMIERUNG DES Z80

5.1 Der Assembler

Damit wir die nächsten Maschinenprogramme nicht mehr mit der Hand übersetzen müssen, haben wir einen Z80-A Assembler geschrieben.

Der Assembler erzeugt den zu einem in Assemblersprache geschriebenen Programm (Sourceprogramm) dazugehörigen Maschinencode (Objektcode bzw. Objektprogramm). Dabei werden z.B. die Sprungdistanzen automatisch berechnet. Wir brauchen also die lästige Arbeit des Übersetzens per Hand, des Nachschlagens der Opcodes etc. nicht mehr auszuführen.

Für die Z80-A Assemblerprogramme gelten bestimmte Konventionen.

Eine Assemblerzeile sieht folgendermaßen aus:

Label Befehl Operand; Kommentar

Da wir zur Eingabe der Programme den BASIC Editor verwenden wollen, ist jede Assembleranweisung einer Zeilennummer zugeordnet.

Im Folgenden wird das Eingabeformat des Assemblers definiert. Zur Vermeidung von Fehlern beim Benutzen des Assemblers ist der nun folgende Teil sehr wichtig. Bitte bearbeiten Sie ihn besonders gründlich.

Label:

Am Anfang einer Zeile kann ein Label (Marke) stehen. Ein Label ist eine Variable. Die länge des Variablennamens (Labelnamens) darf nicht mehr als 6 Zeichen betragen. Labelnamen

müssen mit einem Buchstaben beginnen. Assemblerbefehle dürfen nicht als Labelnamen benutzt werden. Durch die Benutzung von Labels vereinfacht sich die Programmierung von Sprüngen:

```
         .
         .
ANF      Befehl      ; ANF ist Label
         .
         .
         JR  ANF     ; Springe nach ANF
         .
         .
```

Der Assembler berechnet automatisch die richtige Distanz.

Befehl (Mnemonic):

Nach dem eventuell vorhanden Label muß der Befehl folgen. Label und Befehl müssen durch Leerzeichen voneinander getrennt sein. Der Mnemonic muß ein gültiges Assemblerbefehlswort sein. Gültige Befehlsworte haben wir ständig in den Befehlslisten benutzt z.B.: LD, ADD, INC usw.

Operand:

Auf das Befehlswort folgt, wieder durch Leerzeichen getrennt, der Operand. Bei Sprüngen kann die Sprungadresse als Label angegeben werden (s.o). Die Existenz dieses Labels ist natürlich wichtig.
Anstelle von Konstanten oder Distanzen können Variablennamen oder Labels eingesetzt werden.
Innerhalb des Operanden dürfen niemals Leerzeichen stehen!

Kommentar:

Am Ende der Zeile kann, durch ein Leerzeichen und Semikolon abgetrennt, ein Kommentar folgen. Alle auf ein Semikolon folgenden Zeilen werden bei der Übersetzung nicht beachtet. Das

Kommentieren ist eine nützliche Hilfe zum späteren Verständnis der Programme.

Während der Übersetzung erzeugt der Assembler ein Assemblerlisting, daß auf dem Drucker bzw. Bildschirm ausgegeben werden kann. Außerdem kann der erzeugte Code auf Cassette bzw. Diskette abgespeichert werden.
Das Assemblerlisting hat folgenden Aufbau:

```
&Adr. &Codes BASIC Label Befehl Operand ;Kommentar
             Zeiln.

A003 36CC     50  WEITER LD   (HL),Bitmat ; Adresse...
A005 23       60         INC  HL ; HL erhöhen
```

Zusätzlich zu den Z80-A Befehlen kennt der Assembler eine Reihe von Pseudobefehlen. Sie sind Anweisungen an den Assembler, wie z.B. END. END bedeutet für den Assembler, nicht mehr nach weiteren Befehlen zu suchen und die Übersetzung abzuschließen.
Eine weitere wichtige Anweisung ist EQU (engl.equal: gleich). Mit EQU wird der Wert einer Variablen definiert.

Variablenname EQU Wert

(Weist dem voranstehenden Variablennamen den hinter der EQU Anweisung stehenden Wert zu.)
Die Anweisung ORG (Organisation) gibt an, ab welcher Adresse das Programm gespeichert werden soll. Wir werden meistens &A000 als Startadresse benutzen.
Bei der Angabe von Zahlen sind folgende Vereinbarungen getroffen worden.
Hexadezimale Zahlen werden durch das Voranstellen von "&" gekennzeichnet.
Dualzahlen werden durch das Voranstellen von "&X" gekennzeichnet.
Ist eine Zahl ohne eines dieser Zeichen angegeben, so wird sie als Dezimalzahl interpretiert.

Die Standardvereinbarungen für Z80-A Assembler sind ein H am Ende einer Hex-Zahl und ein B am Ende einer Binärzahl. Wir werden jedoch die oben angegebene Konvention mit & und &X benutzen.

Probieren Sie doch den Assembler durch Eingabe eines kleinen Programms einfach einmal aus.

Das Assemblersourceprogramm (Ausgangsprogramm) kann unabhängig vom Assemblerprogramm eingegeben werden. Das erste Programm aus Kapitel 1.2 würde dann folgendermaßen aussehen:

```
 10 ' org &a000
 20 ' bildad equ &c000 ;start Bildschirmspeicher
 30 ' bitmat equ &cc ;Bildschirmpunkt Matrix
 40 ' ld hl,bildad
 50 ' weiter ld (hl),bitmat
 60 ' inc hl
 70 ' cp h ;vergleich mit 0
 80 ' jr nz,weiter
 90 ' ret
100 ' end
```

Bei der Eingabe können Sie sowohl Klein- als auch Großschrift benutzen. Diese werden jedoch intern nicht unterschieden z.B. bei Labelnamen: weiter=WEITER

Beachten Sie, daß hinter jeder Zeilennummer mit einem Leerzeichen Abstand ein (') eingegeben werden muß. Vergessen Sie dieses Zeichen, kann die jeweilige Zeile später nicht vom Assembler übersetzt werden, und es erscheint die Fehlermeldung:

"Fehler ' missing in ..."

Durch Zeile 10 wird der Programmspeicherplatz ab &A000 aufwärts festgelegt.
In Zeile 20 wird der Variablen Bildad (V) der Wert &C000 zugewiesen.

Anstelle von &C000 kann danach immer "Bildad" (V) geschrieben werden. Beim sinnvollen Einsatz von Variablen wird ein Programm übersichtlicher. In der Programmschleife haben wir das Label "weiter" als Sprungziel verwendet. Ansonsten benutzten wir die normalen Assemblerbefehle.

Folgt ein Kommentar in einer Zeile, so ist er durch Semikolon abgetrennt. Wichtig ist, daß vor dem Semikolon ein Leerzeichen steht. Leerzeichen bedeuten für den Assembler, daß z.B. ein Label endet und darauf der Befehl folgt. Deswegen müssen zwischen Label, Befehl, Operand und Kommentar immer (!!) Leerzeichen stehen und dürfen z.B. innerhalb eines Operanden keine (!!) Leerzeichen verwendet werden.

Beispiel:

```
LD ( HL )      FALSCH!

LD (HL)        RICHTIG!
```

Am Ende des Programms sollte der Pseudobefehl END stehen. Dieser bedeutet für den Assembler, daß die Übersetzung hier beendet werden kann. Der Assembler belgt die Zeilennummern 10000 und die Zeile 1. Diese Zeilennummern dürfen folglich nicht vom Sourceprogramm benutzt werden.

Speichern Sie das eingegebene Sourceprogramm mit >SAVE"Name"< ab laden Sie mit >LOAD< den Assembler und dann mit >MERGE< das vorher gespeicherte Sourceprogramm nach.

ANMERKUNG für Diskettenbenutzer beim 464:

Ein Programm was von Diskette mit >MERGE< geladen werden soll, muß eine ASCII Datei ohne Vorspann sein. Das erreichen Sie durch ein mit Komma abgetrenntes "A" hinter dem >SAVE< Befehl. Da das Laden dieses Dateityps länger dauert, sollten Sie zuerst immer den Assembler (als normales BASIC Programm) laden, und dann das Sourceprogramm (ASCII Datei ohne Vorspann) nachladen. Da das AMSDOS ca. 500 Bytes RAM Speicherplatz belegt, die dynamisch zugeteilt werden, sollten

Diskettenbenutzer Maschinenprogramme bis maximal Adresse &A600 abspeichern, um Komplikationen zu vermeiden.

Beim CPC 664 und 6128 können auch normale BASIC Programme mit >MERGE< verbunden werden. Das langsame Abspeichern als ASCII Datei ist nicht notwendig.

Nun starten Sie einfach mit >RUN<.

Der Assembler fragt nach dem Namen des Programms, ob ein Assemblerlisting der Übersetzung gewünscht wird, und ob dieses Listing gedruckt werden soll. Die vorgegebenen Antworten (j beim Listing und n beim Drucker) können Sie durch >ENTER< bzw. >RETURN< wählen. Wählen Sie zunächst die Standardeingaben.

Jetzt beginnt die eigentliche Übersetzung.
Das Ihnen bekannte Assemblerlisting wird auf dem Schirm ausgegeben. Bei Fehlerauftritt werden die entsprechenden Meldungen vor der jeweiligen Zeile ausgegeben. Am Ende des Listings werden, falls vorhanden, die undefinierten Labels und Variablen angezeigt. Darauf folgt der Programmname, Startadresse, Endadresse, Programmlänge und die Fehlerzahl. Sind Fehler aufgetreten, können sie in der entsprechenden BASIC Zeile korrigiert werden.

Am Schluß des Listings wird eine Tabelle aller Labels und Variablen mit ihren Werten ausgegeben, und zwar in der Reihenfolge ihres Auftretens. Zu guterletzt wird gefragt, ob der Maschinencode aufgezeichnet (gespeichert) werden soll.

Bei Eingabe von "j" wird der erzeugte Code als Binärdatei unter dem eingegebenen Namen mit dem Zusatz ".OBJ" abgespeichert (OBJ: Objektcode). Nach der Assemblierung befindet sich das Maschinenprogramm an der angegebenen Stelle im Speicher und kann mit >CALL< aufgerufen werden.

Sollen weitere Programme in Maschinensprache übersetzt werden, kann mit >DELETE 2-9999< das alte Sourceprogramm gelöscht werden. Das neue Programm kann dann mit >MERGE<

geladen werdem. Als Beispiel führen wir hier das komplette Assemblerlisting von unserem Programmlauf auf.

```
A000          10          ORG  &a000
A000          20   BILDAD EQU  &c000 ;start Bildschirmspeic
her
A000          30   BITMAT EQU  &cc ;Bildschirmpunkt Matrix
A000 2100C0   40          LD   hl,bildad
A003 36CC     50   WEITER LD   (hl),bitmat
A005 23       60          INC  hl
A006 BC       70          CP   h ;vergleich mit 0
A007 20FA     80          JR   nz,weiter
A009 C9       90          RET

Programm:Bild
Start: &A000    Ende: &A009
Laenge: 000A
 0  Fehler
Variablentabelle:
BILDAD C000   BITMAT 00CC   WEITER A003
```

Versuchen Sie, beim Abtippen des Listings die grundsätzliche Struktur des Assemblers anhand der auf das Listing folgenden Erklärungen zu verstehen.

```
1 MEMORY &9EFF:GOTO 10000
10000 REM ********** Z80 Assembler c 1984 by Holger Dullin *
*********
10010 GOTO 14160
10020 LOCATE 20,4:PRINT"Z 8 0 - A s s e m b l e r"
10030 LOCATE 5,8:INPUT"Programmname :",name$
10040 LOCATE 19,11:PRINT"j"
10050 LOCATE 5,11:INPUT"Listing (j/n):",t$
10060 IF t$="n" THEN listflag=0:GOTO 10100 ELSE listflag=-1
10070 LOCATE 19,13:PRINT"n";
10080 LOCATE 5,13:INPUT"Drucker (j/n):",t$
10090 IF t$="j" THEN aus=8:PRINT#aus ELSE aus=0
10100 REM Start Assembly **********
10110 MODE 2
10120 REM Zeilenanfang testen ---------
10130 laze=FNdeek(bpc)
10140 bpc=bpc+2
10150 zenr=FNdeek(bpc)
10160 IF zenr>9999 THEN PRINT#aus,"End Assumed":GOTO 13400
10170 bpc=bpc+2
10180 IF FNdeek(bpc)<>49153 THEN PRINT#aus,"Fehler ' missing
 in";zenr:bpc=bpc+laze-4:feza=feza+1:GOTO 10130
10190 bpc=bpc+2
10200 REM zeile lesen ----------------
10210 POKE vapt,laze-7
10220 POKE vapt+1,bpc-256*INT(bpc/256)
10230 POKE vapt+2,INT(bpc/256)
10240 REM zeile zerlegen -------------
10250 zeia$=zei$
10260 bpc=bpc+laze-6
10270 FOR i=0 TO 3:a$(i)="":NEXT
10280 bepo=INSTR(zei$,";")
10290 IF bepo=0 THEN bemer$="":GOTO 10320
```

```
10300 bemer$=RIGHT$(zei$,LEN(zei$)-bepo+1)
10310 zei$=LEFT$(zei$,bepo-1)
10320 j=0
10330 IF LEFT$(zei$,1)=" " THEN zei$=RIGHT$(zei$,LEN(zei$)-1
):GOTO 10330
10340 sppo=INSTR(zei$," ")
10350 IF zei$="" THEN j=j-1:GOTO 10420
10360 IF sppo=0 THEN 10410
10370 a$(j)=LEFT$(zei$,sppo-1):zei$=RIGHT$(zei$,LEN(zei$)-sp
po)
10380 IF zei$="" THEN j=j-1:GOTO 10420
10390 IF j>3 THEN 10420
10400 j=j+1:GOTO 10330
10410 a$(j)=zei$
10420 IF j>2 THEN 13250
10430 REM Interpretation --------------
10440 j=0
10450 bef$=LEFT$(UPPER$(a$(j))+"     ",4)
10460 po=INSTR(teadr$,bef$)
10470 IF po<>0 THEN lp=0:GOTO 11190
10480 po=INSTR(teb1$,bef$)
10490 IF po<>0 THEN lp=1:GOTO 10810
10500 po=INSTR(teed$,bef$)
10510 IF po<>0 THEN lp=2:pw(1)=&ED:GOTO 10850
10520 REM pseudo bef test -------------
10530 po=INSTR(teps$,bef$)
10540 IF po<>0 THEN 10890
10550 REM a$(j)=label ?  --------------
10560 IF j>0 THEN 13250
10570 IF a$(0)="" THEN 13100
10580 a$=a$(0)
10590 GOSUB 13630
10600 IF nolafl THEN 13280
```

```
10610 label$=UPPER$(lab$)
10620 wert=mpc
10630 lata$(ltp)=label$:wlta(ltp)=mpc:ltp=ltp+1
10640 FOR i=0 TO ultp:IF label$=ulata$(i) THEN 10670
10650 NEXT i
10660 j=j+1:GOTO 10450
10670 ON udata(i,2) GOTO 10680,10700
10680 adr=udata(i,1)-1:ziel=wert:GOSUB 14100
10690 pw(1)=of:GOTO 10720
10700 pw(2)=INT(wert/256)
10710 pw(1)=wert-pw(2)*256
10720 PRINT#aus,"**** Zeile "udata(i,0);" : ";ulata$(i);"=&"
;HEX$(wert,4)
10730 FOR k=1 TO udata(i,2)
10740 POKE udata(i,1)+k-1,pw(k)
10750 NEXT k
10760 FOR k=i TO ultp-1
10770 ulata$(k)=ulata$(k+1)
10780 FOR c=0 TO 2:udata(k,c)=udata(k+1,c):NEXT c:NEXT k
10790 ultp=ultp-1:i=i-1
10800 GOTO 10650
10810 REM bef1 / 1-Byter ohne Operand -
10820 IF a$(j+1)<>"" THEN 13270
10830 pw(1)=wb1((po-1)/4)
10840 GOTO 13100
10850 REM ed / 2 byter ohne operand anfang ed
10860 IF a$(j+1)<>"" THEN 13270
10870 pw(2)=wed((po-1)/4)
10880 GOTO 13100
10890 REM pseudo befehle --------------
10900 j=j+1
10910 ope$=a$(j):op$=UPPER$(ope$)
10920 ON (po-1)/4 GOTO 10980,11040,11060,11080,11100,11160
10930 REM EQU
10940 IF label$="" THEN 13280
```

```
10950 a$=op$:GOSUB 13790
10960 wlta(ltp-1)=wert
10970 GOTO 13100
10980 REM ORG
10990 IF op$="" THEN 13290
11000 a$=op$:GOSUB 13790
11010 lp=0
11020 mpc=wert:mpstart=mpc
11030 GOTO 13100
11040 REM END
11050 GOTO 13400
11060 REM DB
11070 a$=op$:GOSUB 14050:GOTO 13100
11080 REM DW
11090 a$=op$:GOSUB 13860:GOTO 13100
11100 REM DM
11110 IF LEFT$(op$,1)<>CHR$(34) OR RIGHT$(op$,1)<>CHR$(34) T
HEN 13260
11120 zwi$=MID$(ope$,2,LEN(ope$)-2)
11130 lp=LEN(zwi$)
11140 FOR i=1 TO lp:pw(i)=ASC(MID$(zwi$,i,1)):NEXT
11150 GOTO 13100
11160 REM DS
11170 a$=op$:GOSUB 13860
11180 ds=wert:lp=0:GOTO 13100
11190 REM bef Auswertung---------------
11200 j=j+1:ope$=a$(j)
11210 op$=UPPER$(ope$)
11220 IF op$="" AND bef$<>"RET " THEN 13290
11230 GOSUB 11240:GOTO 11340
11240 poko=INSTR(op$,",")
11250 IF poko=0 THEN o1$=op$:koflag=0:GOTO 11280
```

```
11260 koflag=-1
11270 o1$=LEFT$(op$,poko-1):o2$=RIGHT$(op$,LEN(op$)-poko)
11280 pokla=INSTR(op$,"("):poklz=INSTR(op$,")")
11290 IF pokla=0 THEN klaflag=0:klin$="":GOTO 11330
11300 IF pokla>poklz THEN GOTO 13260
11310 klaflag=-1
11320 klin$=MID$(op$,pokla+1,poklz-pokla-1)
11330 RETURN
11340 REM
11350 ipo=INSTR(op$,"IX")
11360 IF ipo<>0 THEN pwi=&DD:ireg$="IX":GOTO 11450
11370 ipo=INSTR(op$,"IY")
11380 IF ipo<>0 THEN pwi=&FD:ireg$="IY":GOTO 11450
11390 zwi=(po+3)/4
11400 ON zwi GOTO 12630,11920,11900,12040,12040,12080,12220,
12240,12340,12320,12380,12430,12430,12520,12560
11410 REM ld0,relativspru(2),spru(3),zahl(2),stapel(2),rst,i
/o,im
11420 IF zwi<24 THEN 11590
11430 IF zwi<32 THEN 11760
11440 GOTO 11830
11450 REM indizierte Befehle ---------
11460 iflag=-1
11470 IF (NOT klaflag) OR (ipo-pokla<>1) THEN op$=LEFT$(op$,
ipo-1)+"HL"+RIGHT$(op$,LEN(op$)-ipo-1):GOTO 11550
11480 zwi$=MID$(klin$,3,1):IF zwi$<>"+" AND zwi$<>"-" THEN I
F bef$="JP  " THEN 11540 ELSE GOTO 13250
11490 a$=RIGHT$(klin$,LEN(klin$)-3)
11500 GOSUB 14050:lp=lp-1
11510 IF fe$<>"" THEN GOTO 13260
11520 disflag=-1
11530 disw=wert:IF zwi$="-" THEN disw=(disw XOR 255) +1
11540 op$=LEFT$(op$,pokla)+"HL"+RIGHT$(op$,LEN(op$)-poklz+1)
```

```
11550 IF (INSTR(op$,"IX")=0)AND(INSTR(op$,"IY")=0)THEN 11570
11560 IF (op$=("HL,"+ireg$))AND(bef$="ADD ") THEN op$="HL,HL
" ELSE GOTO 13260
11570 GOSUB 11240
11580 GOTO 11390
11590 REM arilog ----------------------
11600 IF NOT koflag THEN a$=o1$:GOTO 11620
11610 IF o1$<>"A" THEN 11670 ELSE a$=o2$
11620 lp=1:code=zwi-16
11630 GOSUB 13680
11640 IF rflag THEN pw(1)=128 OR(code*8) OR rrr:GOTO 13100
11650 pw(1)=&X11000110 OR (code*8)
11660 GOSUB 14050:GOTO 13100
11670 IF o1$<>"HL" THEN 13260
11680 a$=o2$
11690 GOSUB 13730
11700 IF NOT rflag THEN 13260
11710 IF bef$="ADD " THEN code=&X1001:lp=1:GOTO 11750
11720 pw(1)=&ED:lp=2
11730 IF bef$="ADC " THEN code=&X1001010 :GOTO 11750
11740 IF bef$="SBC " THEN code=&X1000010 ELSE GOTO 13250
11750 pw(lp)=code OR (dd*16):GOTO 13100
11760 REM rotschie --------------------
11770 lp=2:pw(1)=&CB
11780 IF koflag THEN 13260
11790 a$=op$:GOSUB 13680
11800 IF NOT rflag THEN 13260
11810 pw(2)=(8*(zwi-24)) OR rrr
11820 GOTO 13100
11830 REM bitti ----------------------
11840 lp=2:pw(1)=&CB
11850 a$=o2$:GOSUB 13680
11860 IF NOT rflag THEN 13260
11870 bbb=ASC(o1$)-48
11880 IF (0>bbb) OR (7<bbb) OR (LEN(o1$)<>1) THEN 13260
```

```
11890 pw(2)=(64*(zwi-31))OR(bbb*8)OR rrr:GOTO 13100
11900 REM relative Spruenge -----------
11910 lp=1:pw(1)=&10:a$=op$:GOTO 11990
11920 lp=1
11930 IF NOT koflag THEN ccc=&X11:a$=op$:GOTO 11980
11940 a$=o1$:GOSUB 13760
11950 IF (NOT cflag) OR (ccc>3) THEN 13260
11960 ccc=ccc OR 4
11970 a$=o2$
11980 pw(1)=ccc*8
11990 IF LEFT$(a$,1)<>"$" THEN GOSUB 13860:lp=lp-2:IF i>ltp
THEN wert=mpc  :GOTO 12010:ELSE 12010
12000 wert=mpc+VAL(RIGHT$(a$,LEN(a$)-1))
12010 lp=lp+1:adr=mpc:ziel=wert
12020 GOSUB 14100
12030 pw(2)=of:GOTO 13100
12040 REM Spruenge --------------------
12050 zwi=1:lp=1
12060 IF bef$="RET " THEN code=0 ELSE code=&X100
12070 GOTO 12110
12080 IF op$="(HL)" THEN lp=1:pw(1)=&E9:GOTO 13100
12090 code=&X10
12100 zwi=0:lp=1
12110 IF bef$="RET " THEN IF op$="" THEN 12130 ELSE 12160 EL
SE
12120 IF koflag THEN 12160
12130 pw(1)=192 OR code OR 1 OR (zwi*8)
12140 a$=op$
12150 GOTO 12200
12160 a$=o1$:GOSUB 13760
12170 IF NOT cflag THEN 13260
12180 pw(1)=192 OR code OR(ccc*8)
```

```
12190 a$=o2$
12200 IF bef$="RET " THEN 13100
12210 GOSUB 13860:GOTO 13100
12220 REM Zaehlbefehle ---------------
12230 zwi=0:GOTO 12250
12240 zwi=1
12250 IF koflag THEN 13260
12260 lp=1:a$=op$:GOSUB 13680
12270 IF rflag THEN pw(1)=&X100 OR (rrr*8) OR zwi:GOTO 13100
12280 GOSUB 13730
12290 IF NOT rflag THEN 13260
12300 pw(1)=&X11 OR (dd*16) OR (zwi*8)
12310 GOTO 13100
12320 REM Stapelbefehle --------------
12330 code=&X11000001:GOTO 12350
12340 code=&X11000101
12350 a$=op$:dreg$(3)="AF":GOSUB 13730:dreg$(3)="SP"
12360 IF NOT rflag THEN 13260
12370 lp=1:pw(1)=code OR (dd*16):GOTO 13100
12380 REM restart --------------------
12390 a$=op$:GOSUB 14050
12400 zwi=wertl/8
12410 IF zwi<>INT(zwi) OR zwi>7 THEN 13260
12420 lp=1:pw(1)=&X11000111 OR (zwi*8):GOTO 13100
12430 REM I/O Befehle ----------------
12440 IF NOT(koflag AND klaflag) THEN 13260
12450 IF bef$="IN  " THEN zwi=0 ELSE zwi=1:zwi$=o2$:o2$=o1$:
o1$=zwi$
12460 IF klin$<>"C" THEN 12500
12470 a$=o1$:GOSUB 13680
12480 IF (NOT rflag) OR (klin$<>"C") THEN 13260
12490 lp=2:pw(1)=&ED:pw(2)=64 OR (rrr*8) OR zwi:GOTO 13100
12500 lp=1:a$=klin$:GOSUB 14050
12510 pw(1)=&X11011011 XOR (zwi*8):GOTO 13100
```

```
12520 REM interrupt modi --------------
12530 IF op$<>"0" AND op$<>"1" AND op$<>"2" THEN 13260
12540 lp=2:pw(1)=&ED
12550 pw(2)=&X1000110 OR ((VAL(op$)-(op$<>"0"))*8):GOTO 1310
0
12560 REM EX --------------------------
12570 lp=1
12580 IF op$="(SP),HL" THEN pw(1)=&E3:GOTO 13100
12590 IF op$="DE,HL" THEN pw(1)=&EB:GOTO 12620
12600 IF op$="AF,AF'" THEN pw(1)=&8:GOTO 13100
12610 GOTO 13260
12620 IF iflag THEN 13260 ELSE 13100
12630 REM ld---------------------------
12640 IF NOT koflag THEN 13260
12650 a$=o1$:GOSUB 13680
12660 IF rflag THEN 12860
12670 GOSUB 13730
12680 IF rflag THEN 12760
12690 a$=o2$:GOSUB 13730
12700 IF rflag THEN 12740
12710 zwi$=o2$:o2$=o1$:o1$=zwi$
12720 a=0:GOSUB 12940
12730 IF nflag THEN 13260 ELSE GOTO 13100
12740 IF NOT klaflag THEN 13260
12750 zwi$=o2$:zwiflag=1:GOTO 12800
12760 IF op$="SP,HL" THEN lp=1:pw(1)=&F9:GOTO 13100
12770 IF klaflag THEN zwi$=o1$:zwiflag=0:GOTO 12800
12780 a$=o2$
12790 lp=1:code=1:GOTO 12830
12800 a$=klin$
12810 IF zwi$="HL" THEN lp=1:code=&A:GOTO 12830
12820 lp=2:pw(1)=&ED:code=&X1001011
12830 code=code AND NOT (zwiflag*8)
12840 pw(lp)=code OR (dd*16)
```

```
12850 GOSUB 13860:GOTO 13100
12860 zzz=rrr:a$=o2$:GOSUB 13680
12870 IF NOT rflag THEN 12900
12880 lp=1:pw(1)=64 OR (zzz*8) OR rrr
12890 IF pw(1)=&76 THEN 13260 ELSE 13100
12900 a=1:GOSUB 12940
12910 IF NOT nflag THEN 13100
12920 lp=1:pw(1)=&X110 OR (rrr*8)
12930 a$=o2$ : GOSUB 14050:GOTO 13100
12940 REM 8-bit Lade Spezial ----------
12950 nflag=0
12960 IF o1$<>"A" THEN nflag=-1:RETURN
12970 IF klaflag THEN 13030
12980 IF o2$="I"THEN zwi=0:GOTO 13010
12990 IF o2$="R" THEN zwi=1:GOTO 13010
13000 nflag=-1:RETURN
13010 code =&X1000111:lp=2:pw(1)=&ED
13020 pp=(a*2) OR zwi:GOTO 13080
13030 IF klin$="BC" THEN zwi=0:GOTO 13070
13040 IF klin$="DE" THEN zwi=1:GOTO 13070
13050 Ip=1:pw(1)=&X110010 OR (a*8)
13060 a$=klin$:GOSUB 13860:RETURN
13070 code=&X10:lp=1:pp=(zwi*2)OR a
13080 pw(lp)=code OR (8*pp):RETURN
13090 REM
13100 REM Ausgabe **********
13110 IF iflag THEN 13310
13120 IF fe$<>"" THEN feza=feza+1
13130 IF NOT listflag THEN LOCATE 5,3:PRINT zenr:GOTO 13200
13140 IF fe$<>"" THEN PRINT CHR$(7);:PRINT#aus,fe$,TAB(30);z
enr;zeia$:GOTO 13210
13150 PRINT#aus,HEX$(mpc,4);" ";
13160 FOR i=1 TO lp:PRINT#aus,HEX$(pw(i),2);:POKE mpc+i-1,pw
(i):NEXT i
```

```
13170 PRINT#aus,TAB(14);USING"####";zenr;
13180 PRINT#aus,TAB(20);label$;TAB(27);bef$;TAB(32);ope$;"  "
;bemer$;
13190 PRINT#aus
13200 mpc=mpc+lp+ds
13210 lp=0:ds=0
13220 label$="":bef$="":ope$="":bemer$="":fe$=""
13230 GOTO 10130
13240 REM Fehlermeldungen -------------
13250 fe$="Syntax Error":GOTO 13100
13260 fe$="Syntax Error im Operanden":GOTO 13100
13270 fe$="Operand zuviel":GOTO 13100
13280 fe$="Label fehlt":GOTO 13100
13290 fe$="Operand fehlt":GOTO 13100
13300 fe$="illegal Quantity":GOTO 13100
13310 REM indiziert -------------------
13320 FOR j=lp TO 1 STEP -1
13330 pw(j+1)=pw(j):NEXT
13340 pw(1)=pwi:lp=lp+1
13350 IF NOT disflag THEN 13380
13360 IF lp=3 THEN pw(4)=pw(3)
13370 pw(3)=disw:lp=lp+1
13380 iflag=0:disflag=0
13390 GOTO 13120
13400 REM Ende programm **********
13410 PRINT#aus
13420 IF ultp=0 THEN 13470
13430 FOR i=0 TO ultp-1
13440 PRINT#aus,"undefiniertes Label ";ulata$(i);" in";udata
(i,0);" / Adresse &";HEX$(udata(i,1),4)
13450 feza=feza+1:NEXT i
13460 PRINT#aus
13470 PRINT#aus,"Programm :";name$
13480 PRINT#aus,"Start : &";HEX$(mpstart,4);"    Ende : &";H
EX$(mpc-1,4)
13490 PRINT#aus,"Laenge : ";HEX$(mpc-mpstart,4)
13500 PRINT#aus,feza;" Fehler"
13510 IF ltp=0 THEN 13560
13520 PRINT#aus,"Variablentabelle :"
```

```
13530 FOR i=0 TO ltp-1
13540 PRINT#aus,LEFT$(lata$(i)+"        ",7);HEX$(wlta(i),4)
,
13550 NEXT i
13560 PRINT#aus
13570 INPUT"Aufzeichnung (j/n):",t$
13580 IF t$<>"j" THEN 13600
13590 SAVE name$+".obj",B,mpstart,mpc-mpstart
13600 END
13610 REM Unterprogramme **********
13620 REM label test ------------------
13630 laas=ASC(UPPER$(LEFT$(a$,1)))
13640 IF laas<65 OR laas>90 THEN nolafl=-1:RETURN
13650 IF LEN(a$)>6 THEN PRINT"Label zu lang":a$=LEFT$(a$,6)
13660 lab$=a$:nolafl=0
13670 RETURN
13680 REM r test ----------------------
13690 FOR i=0 TO 7
13700 IF reg$(i)=a$ THEN rflag=-1:rrr=i:RETURN
13710 NEXT
13720 rflag=0:RETURN
13730 REM rps test --------------------
13740 FOR i=0 TO 3:IF dreg$(i)=a$ THEN rflag=-1:dd=i:RETURN
ELSE NEXT
13750 rflag=0:RETURN
13760 REM cond test -------------------
13770 FOR i=0 TO 7:IF a$=cond$(i) THEN cflag=-1:ccc=i:RETURN
 ELSE NEXT
13780 cflag=0:RETURN
13790 REM Zahltest --------------------
13800 wert=VAL(a$)
13810 laas=ASC(LEFT$(a$,1))
13820 IF wert=0 AND laas<>38 AND (laas>57 OR laas<48) THEN f
e$="illegal character":wert=0:RETURN
```

```
13830 IF wert>=0 THEN 13850
13840 IF LEFT$(a$,1)="&" THEN wert=wert+2^16 ELSE fe$="illeg
al Quantity":wert=0
13850 RETURN
13860 REM Werter ----------------------
13870 GOSUB 13620
13880 IF nolafl THEN GOSUB 13790:GOTO 13920
13890 FOR i=0 TO ltp:IF lata$(i)<>lab$ THEN NEXT
13900 IF i>ltp THEN 13980
13910 wert=wlta(i)
13920 werth=INT(wert/256)
13930 wertl=wert-256*werth
13940 lp=lp+2
13950 pw(lp-1)=wertl
13960 pw(lp)=werth
13970 RETURN
13980 ulata$(ultp)=a$
13990 udata(ultp,0)=zenr
14000 udata(ultp,1)=mpc+lp-iflag-disflag
14010 udata(ultp,2)=2+(bef$="DJNZ" OR bef$="JR  ")
14020 ultp=ultp+1
14030 wert=0
14040 GOTO 13920
14050 REM werter low ------------------
14060 GOSUB 13860
14070 lp=lp-1
14080 IF werth<>0 THEN fe$="illegal Quantity":wert=0
14090 RETURN
14100 REM offset berechnen ------------
14110 of =ziel-adr
14120 of=of-2
14130 IF of>129 OR of<-126 THEN fe$="illegal Quantity":of=0
14140 IF of<0 THEN of=256+of
14150 RETURN
```

```
14160 REM Initialisierung *********
14170 zei$="test"
14180 vapt=@zei$
14190 DEF FNdeek(x)=PEEK(x)+256*PEEK(x+1)
14200 MODE 2
14210 teadr$="LD  JR  DJNZCALLRET JP  INC DEC PUSHPOP RST IN
  OUT IM  EX  ADD ADC SUB SBC AND XOR OR  CP  RLC RRC RL  RR
  SLA SRA *** SRL BIT RES SET "
14220 teed$="CPD CPDRCPI CPIRIND INDRINI INIRLDD LDDRLDI LDI
RNEG OTDROTIROUTDOUTIRETIRETNRLD RRD "
14230 DATA A9,B9,A1,B1,AA,BA,A2,B2,A8,B8,A0,B0,44,BB,B3,AB,A
3,4D,45,6F,67
14240 teb1$="CCF CPL DAA DI  EI  EXX HALTNOP RLA RLCARRA RRC
ASCF "
14250 DATA 3F,2F,27,F3,FB,D9,76,00,17,07,1F,0F,37
14260 teps$="EQU ORG END DB  DW  DM  DS  "
14270 DIM lata$(70),wlta(70),ulata$(50),udata(50,2)
14280 DIM wb1(12),wed(20)
14290 FOR i=0 TO 20:READ a$:wed(i)=VAL("&"+a$):NEXT
14300 FOR i=0 TO 12:READ a$:wb1(i)=VAL("&"+a$):NEXT
14310 bpc=FNdeek(&170)+&170:mpc=40960:mpstart=mpc
14320 DIM reg$(7),cond$(7),dreg$(3)
14330 FOR i=0 TO 7:READ reg$(i):NEXT
14340 FOR i=0 TO 7:READ cond$(i):NEXT
14350 FOR i=0 TO 3:READ dreg$(i):NEXT
14360 DATA B,C,D,E,H,L,(HL),A
14370 DATA NZ,Z,NC,C,PO,PE,P,M
14380 DATA BC,DE,HL,SP
14390 GOTO 10020
```

Programmbeschreibung:

Zeile 1:

RAM Speicherplatz von &A000-&AB7F wird für das Maschinenprogramm reserviert, dann wird das Sourceprogramm zwischen Zeile 2 bis 9999 liegend, übersprungen.

Zeile 10010:

Verzweigung zum Programmteil Initialisierung, d.h. Aufbau der Befehlstabelle u.ä. (siehe Zeile 14160-).

Zeile 10020-10090:

Menue, listflag (V) und Ausgabekanal aus (V) werden bestimmt.

Zeile 10100-10190:

bpc zeigt auf die aktuelle Adresse im BASIC Sourceprogramm (bpc:BASIC Programm Counter). Am Anfang einer Zeile steht die Länge derselben als Low und High Byte. FNdeek(bpc) liest den 16-Bit Wert an Adresse bpc und bpc+1. Der Wert entspricht der Zeilenlänge laze (V). bpc wird um 2 erhöht, und die Zeilennummer zenr (V) wird gelesen. Ist sie größer als 9999, wird die Übersetzung beendet. In Zeile 10180 wird geprüft, ob das (') Zeichen am Anfang der Zeile steht. Wenn es dort nicht steht, wird die Fehlermeldung ausgegeben und die nächste Zeile gelesen.

Zeile 10200-10240:

Durch diesen Programmteil wird zei$ (V) mit der aktuellen Zeile gefüllt. Um die Geschwindigkeit des Assemblers möglichst hoch zu halten, geschieht das über eine Änderung der Stringzeiger von zei$ (V) in der internen Variablentabelle.

Zeile 10240-10420:

Zuerst wird ein evtl. vorhandener Kommentar in bemer$ (V) abgespeichert, dann wird die verbleibende Zeile bei jedem auftretenden Leerzeichen zerschnitten, und die zerschnittenen Teile werden in a$(j) (V) gespeichert. Ließ sich die Zeile in mehr als 3 Teile zerlegen (Label, Befehl, Operand), d.h. j>2, wird ein Syntax Error ausgegeben.

Zeile 10430-10540:

Hier wird geprüft, ob es sich bei a$(j) (V) um einen gültigen Befehl handelt. War der Befehl gültig, wird an die Stelle an der diese Befehle übersetzt werden verzweigt.

Zeile 10540-10550:

Wurde kein Befehl festgestellt, wird hier auf Pseudobefehle geprüft und verzweigt.

Zeile 10560-10800:

Handelt es sich um ein Label, so wird es in die Labeltabelle eingetragen und der mpc (Maschinenprogramm Counter) wird dem Label als Wert zugeordnet (Zeile 10610-10630). In den folgenden Zeilen bis 10800 wird geprüft, ob dieses Label schon einmal benutzt wurde, zu dem Zeitpunkt aber noch undefiniert war. Trifft das zu, wird nachträglich der entsprechende Wert gepoked und das Label aus der Tabelle der undefinierten ulata$(i) (V) gelöscht. Handelt es sich um kein gültiges Label (d.h. es fängt nicht mit einem Buchstaben an), so wird die Fehlermeldung "Label fehlt" ausgegeben (Zeile 10600).

Zeile 10810-10840:

Hier werden Befehle mit einem 1-Byte Opcode die keine Operanden besitzen ausgewertet. Der Code ergibt sich aus der Stellung im teb1$ (V) und dem dazugehörigen wb1(i) (V).

Zeile 10850-10880:

Hier werden die 2-Byte Befehle ohne Operand behandelt. Der erste Opcode ist immer &ED (pw(1)=&ED). Das zweite Byte des Opcodes ergibt sich aus der Stellung des Befehls im teed$ (V) und wed(i) (V).

Zeile 10890-11180:

Hier werden alle Pseudobefehle übersetzt.

Zeile 11190-13080:

Fällt der Befehl in keine der oben genannten Gruppen, wird er in diesem Programmteil ausgewertet. Zuerst wird geprüft, ob der Operand op$ (V) ein Komma enthält. Wenn ja, wird er in o1$ (V) (Vorkommateil) und o2$ (V) (Nachkommateil) zerlegt und koflag (V) wird auf -1 (=wahr) gesetzt. Weiterhin wird geprüft, ob Klammern auftreten. Wenn ja, wir der Klammerinhalt in klin$ (V) gespeichert und klflag (V) wird gesetzt (=-1).

Zeile 11280-11330:

Handelt es sich um einen indiziert adressierten Befehl, so wird zur Zeile 11450 verzweigt.

Zeile 11400-11440:

Hier wird aufgrund der Stellung in teadr$ (V) zur jeweiligen Befehlsbehandlungsroutine gesprungen.

Zeile 11450-11580:

Bei den indizierten Befehlen wird IX, IY bzw. (IX+d), (IY+d) durch HL ersetzt, iflag (V) und disflag (V) werden entsprechend gesetzt. Danach wird die normale Befehlsauswertung ab Zeile 11390 fortgesetzt. Nach der Interpretation wird die Veränderung wieder rückgängig gemacht, und der Code des indizierten Befehls (der dem von HL analog ist) wird ausgegeben.

Zeile 11590-11750:

Die arithmetischen Befehle (8- und 16-Bit) werden hier interpretiert.

Zeile 11760-11820:

Rotier- und Schiebebefehle

Zeile 11830-11890:

Bitmanipulationsbefehle

Zeile 11900-12030:

relative Sprünge (JR und DJNZ)

Zeile 12050-12210:

andere Sprünge (JP,RET,CALL)

Zeile 12220-12310:

Zählbefehle (INC,DEC)

Zeile 12320-12620

(siehe REM Zeilen)

Zeile 12630-13080:

Ladebefehle

Jede Routine zu erklären, würde den Rahmen dieses Buches sprengen. Greifen wir exemplarisch die Routine für die Bitmanipulationsbefehle heraus:

Zeile 11840:

lp (V) (Länge des Befehls, d.h. Anzahl der zu pokenden Werte) ist 2. Der erste Wert pw(1) (V) ist &CB. Das trifft für alle Bitbefehle zu.

Zeile 13620-13670:

Hier wird geprüft, ob in a$ (V) ein zulässiges Label steht.

Zeile 13680-13670:

Diese Zeilen prüfen, ob in a$ (V) ein Register steht (A, B, C, D, E, H, L, (HL)).

Zeile 13730-13750:

Hier wird geprüft, ob in a$ (V) eines der Registerpaare BC, DE, HL, SP steht.

Zeile 13760-13780:

Prüft, ob in a$ eine Bedingung C,NC,Z,NZ,So,PE,P,M steht.

Zeile 13790-13850:

Prüft, ob a$ (V) eine Zahl ist und gibt den Wert dieser Zahl zurück.

Zeile 13860-14040:

Diese Zeilen ermitteln den 2-Byte Wert von a$ (V). a$ kann sowohl eine Zahl, als auch eine Variable oder ein Label sein.

Zeile 14050-14090:

Ermittelt den 1-Byte Wert (Low Byte) von a$.

Zeile 14100-14150:

Berechnet den Offset für relative Sprünge.

Zeile 14160-14390:

Initialisierung: Die Datenfelder und Strings zum Vergleich werden erzeugt. vapt (V) zeigt auf die Adresse, an der die Stringlänge von zei$ (V) gespeichert ist.
FNdeek(X) gibt den 16-Bit Wert zweier aufeinanderfolgender Speicherstellen an.
bpc wird auf den Anfang der auf Zeile 1 folgenden Zeile gesetzt.
mpc wird auf &A000 gesetzt.

Variablenliste

(SUB bedeutet: Unterprogramm)

a-	Übergabe an SUB "8-Bit Lade Spezial"
a$-	Übergabe an verschiedene Unterprogramme
adr-	Übergabe an SUB"Offset berechnen": Absprung-adresse
aus-	Kanal des Ausgabegerätes (0 oder 8)
bbb-	Bitnummer Code bei Bitmanipulationsbefehlen
bef$-	Assemblerbefehlswort
bemer$-	Bemerkung der Assemblerzeile
bepo-	Position des Anfangs der Bemerkung in der jeweiligen Zeile
bpc-	BASIC Programmzeiger
ccc-	Bedingungscode bei Sprüngen
cflag-	gesetzt (d.h. =-1), wenn Bedingung gefunden rückgesetzt (d.h. =0), wenn nicht, Rückgabe von SUB "cond test"
code-	Zur Erzeugung der Opcodes des jeweiligen Befehls benutzt
disflag-	gesetzt bei indiziertem Befehl mit Distanzangabe sonst rückgesetzt
disw-	Wert der angegebenen Distanz (2er Komplement)
ds-	Enthält die Anzahl der durch einen DS-Befehl reservierten Speicherplätze
fe$-	Fehlermeldung
feza-	Fehlerzahl
i,j,r-	Zähler für Schleifen
iflag-	gesetzt, bei indizierten Befehlen, sonst rückgesetzt
ipo-	Position vom Indexregister (IX oder IY) im Operanden
ireg$-	Wenn indizierte Adressierung vorliegt, enthält ireg$ entweder IX oder IY
klaflag-	gesetzt, falls Klammern im Operanden, sonst rückgesetzt
klin$-	enthält Klammerinhalt vom Operanden (falls vorhanden)
koflag-	gesetzt, falls Komma im Operanden, sonst rück-

gesetzt

laas- ASCII Code des ersten Zeichen eines zu prüfenden Labels (SUB"Label Test")

lab$- Rückgabe des Labelnamens vom SUB Label Test

label$- aktueller Labelname

laze- Länge der aktuellen Sourceprogrammzeile die übersetzt wird

listflag- gesetzt, wenn Listing gewünscht, sonst rückgesetzt

lp- Länge des jeweiligen Befehls (Objektcodelänge)

ltp- Zeiger auf freien Platz in Labeltabelle (lata$) (Label Tabellenpointer)

mpc- Maschinenprogrammcounter: zeigt auf die Speicherstelle, an der der nächste Maschinencode gespeichert wird

mpstart- Anfangsadresse des Maschinenprogrammes

name$- Programmname

nflag- gesetzt, wenn bei einem Ladebefehl unmittelbare Adressierung vorliegt, sonst rückgesetzt (SUB"8-Bit Lade Spezial)

nolafl- no Label Flag: gesetzt, wenn der Label Test erfolglos war, sonst rückgesetzt; Rückgabe vom SUB Label Test

o1$- Vorkommateil des Operanden

o2$- Nachkommateil des Operanden

of- berechneter Offset von SUB Offset

op$- Operand zur Bearbeitung

ope$- Operand, original zur Ausgabe

po- Position des Befehlswortes in den Teststrings

pokla- Position der "Klammer auf" im Operanden

poklz- Position der "Klammer zu " im Operanden

poko- Position des Kommas im Operanden

pwi- erstes Opcode Byte bei indizierter Adressierung, also &FF oder &DF

rflag- gesetzt, wenn SUB"rtest"eines der Register A,B, C,D,E,H,L,(HL) festgestellt hat, sonst rückgesetzt

rrr- Code des Registers; Rückgabe von SUB"rtest"

sppo- Spaceposition, Stellung des Leerzeichens in der Zeile

t$- Eingabestring (Menue)
teadr$- Test Adressierte: Enthält alle Befehlswörter, die mit einem Operanden vorkommen
teb1$- Test 1 Byter: Enthält alle Befehlswörter, die nur ohne Operand vorkommen und einen 1-Byte Opcode haben
teed$- Test &ED: Enthält alle Befehlswörter, die nur ohne Operand vorkommen, einen 2-Byte Opcode haben und dessen erstes Byte &ED ist
teps$- Test pseudo: Enthält alle Pseudo-Befehle
ultp- Undefinierte Label Tabellen Pointer: zeigt auf nächsten freien Platz in der Tabelle ulata$ bzw. udata
vapt- Variablen Pointer: zeigt auf die Adresse von zei$ in der internen Variablentabelle
wert- Wert eines Ausdrucks, Rückgabe von SUB"Werter" bzw. SUB"Zahltest"
werth- High Byte von Wert
wertl- Low Byte von Wert
zei$- enthält die aktuelle Zeile zur Verarbeitung
zeia$- enthält die aktuelle Zeile (original)
zenr- aktuelle Zeilennummer
Ziel- Übergabe von SUB"Offset berechnen"an Zieladresse
zwi- diverse Zwischenspeicheraufgaben
zwi$- diverse Zwischenspeicheraufgaben

Tabellen

lata$(50)- Labeltabelle
wlta(50)- Werte der Label der Wertetabelle
ulata$(50)- Tabelle der undefinierten Label
udata$(50,2)- Daten zu undefinierten Label
(i,0) : Zeilennummer des Auftretens
(i,1) : Adresse des nachträglich zu pokenden Wertes
(i,2) : Typ, d.h. 16-Bit (=2) oder Offset (=1)
wb1(12)- Opcodes der 1-Byte Befehle (teb1$)

```
 wed(20)- Opcode der 2-Byte Befehle (teed$)
 reg$(7)- Registertabelle: B,C,D,E,H,L,(HL),A
cond$(7)- Bedingungstabelle: NZ,Z,NC,C,PO,PE,a,M
dreg$(3)- Doppelregistertabelle: BC,DE,HL,SP
```

5.2 Programmierung

Als erstes größeres Programmprojekt wollen wir uns noch einmal mit dem Bildschirm befassen.

Wahrscheinlich ist es Ihnen bei der Programmierung der Beispielaufgaben auch passiert, daß Sie nicht >MODE 2< vor dem Programmstart eingegeben haben. Die Ergebnisse sehen dann etwas merkwürdig aus. Dieses Phänomen wollen wir jetzt erklären:

Nach der Eingabe von >MODE 2< entspricht das erste Bildschirmbyte links oben, der Adresse &C000:

Durch >POKE &C000,255< erhalten wir an dieser Stelle einen Strich.

Gehen Sie nun mit dem Cursor an den unteren Bildschirmrand und scrollen einmal den Bildschirm, indem Sie den Cursor einen Schritt nach unten bewegen. Dann lassen Sie den Cursor an den Anfang der mittleren Bildschirmzeile laufen. Nach nochmaliger Eingabe von >POKE &C000,255< erscheint der Strich im unteren Bildschirmbereich. Geben Sie dagegen >Poke &C050,255< ein, so erscheint der Strich wieder an der alten Stelle. Die Differenz zwischen &C000 und &C050 ist &50 also dezimal 80. Diese Differenz entspricht den 80 Zeichen der Zeile, die beim Scrollen oben aus dem Bildschirm "gelaufen" ist. Scrollen Sie nun wiederum den Bildschirm, so erhalten Sie den Strich nur durch >POKE &C0A0,255< wieder (&C050 + &50 = &C0A0).

Die Differenz von &C000 zu der wirklichen Adresse des linken oberen Bildschirmbytes wird intern an den Adressen &B1C9 (Low)(664:&B7C4 / 6128:&B7C6) und &B1CA (High)(664:&B7C5 / 6128:&B7C7) gespeichert. Lesen wir den 16-Bit Wert dieser Speicherstelle aus. Wenn nicht zwischendurch wieder "gescrollt" wurde, ergibt

>PRINT HEX$(PEEK(&B1C9)+PEEK(&BACA)*256)< den Wert A0.
bzw.
>PRINT HEX$(PEEK(&B7C4)+PEEK(&B7C5)*256)< den Wert A0
bzw.
>PRINT HEX$(PEEK(&B7C6)+PEEK(&B7C7)*256)< den Wert A0

Das ist genau die Differenz von &C000 zu &C0A0. Durch ein Ändern der Inhalte von &B1C9 und &B1CA entstehen u.U. interessante Effekte auf dem Bildschirm.

Zum Beispiel:

```
FOR I=0 TO 255:POKE &B1C9,I:PRINT"*":NEXT  :CPC 464
FOR I=0 TO 255:POKE &B7C4,I:PRINT"*":NEXT  :CPC 664
FOR I=0 TO 255:POKE &B7C4,I:PRINT"*":NEXT  :CPC 6128
```

Bei allen den Bildschirm betreffenden Operationen müssen wir diese Differenz berücksichtigen.

Unter Berücksichtigung der Scrolldifferenz wollen wir jetzt das Programm zum Invertieren des oberen linken Zeichens ändern.

Zunächst wird HL wieder mit &C000 geladen. Da wir mit dem Assembler arbeiten, speichern wir &C000 in einer Variablen. Das Programm starten wir wie üblich ab Adresse &A000. Die ersten Zeilen sehen folgendermaßen aus:

```
 5 'Scrdif  EQU &B1C9 ; für 6128:&B7C4 für 664:&B7C4
10 'Bildad EQU &C000 ; Bildschirmbasisadresse
20 'ORG    &A000
30 'LD  HL,Bildad
```

Nun muß die jeweilige Differenz zur Basisadresse addiert werden.

```
40 'LD  DE,(&B1C9) ; "Scrolldifferenz"
50 'ADD HL,DE ; Startadresse errechnen
```

Der 16-Bit Ladebefehl LD DE,(&B1C9) in Zeile 50, lädt Low und High Byte in das Registerpaar DE. Nun verfahren wir auf die gleiche Weise, wie im Programm in Kapitel 4.10.

```
 60 'LD  DE,&800 ; Differenz
 70 'LD  B,8 ; Zaehler für Schleife
 80 'wieder LD A,(HL) ; aktuelle Bitmatrix
 90 'CPL ; Invertieren
100 'LD (HL),A ; wieder speichern
110 'ADD HL,DE ; Differenz addieren
```

Nun kann jedoch ADD HL,DE bewirken, daß HL größer als &FFFF wird.

Beispiel:

```
HL=&F9A0           DE=&800
```

Nach ADD HL,DE :

```
HL=&01A0           Carry=1
```

Das wäre sicherlich nicht die richtige Adresse im Bildschirmspeicher, da an dieser Adresse unser BASIC Programm steht. Die auf die in Adresse &FFFF gespeicherten folgenden Punkte stehen an Adresse &C000.
Ist also ein Übertrag (CF=1) aufgetreten, müssen wir &C000 zu HL addieren.
Versuchen Sie dieses Programm in Maschinensprachen zu vollenden.

Lösung:

```
120 'CALL C,DIFADD ; Unterprogramm zur Korrektur
130 'DJNZ wieder ; 8* wiederholen
140 'RET ; Rücksprung Unterprogramm
150 'DIFADD PUSH DE ; Unterprogramm start,DE retten
160 'LD DE,Bildad ; Bildad=&C000
170 'ADD HL,DE ; zu HL addieren
180 'POP DE ; DE holen
190 'RET ; Rücksprung Unterprogramm
200 'END
```

Erläuterung:

Zeile 120:

Ist ein Übertrag aufgetreten, wird zur Korrekturroutine gesprungen.

Zeile 150 und 190:

Alle Registerpaare sind bereits benutzt. Die 16-Bit Addition ist jedoch nur implizit adressiert möglich. Daher wird der Inhalt von DE durch PUSH DE kurzzeitig auf dem Stack zwischengespeichert und nach der Addition mit POP DE wieder vom Stapel geholt.

Assemblieren Sie dieses Programm. Betrachten wir das Assemblerlisting:

```
A000            5   Scrdif EQU &b1c9 ; 6128:&B7C4,664:&....
A000            10  BILDAD EQU &c000 ;Bildschirmadresse
A000            20         ORG &a000
A000 2100C0     30         LD  hl,bildad
A003 ED5BC9B1   40         LD  de,(&b1c9) ;"Scrolldifferenz
A007 19         50         ADD hl,de ;neue Startadresse
A008 110008     60         LD  de,&800 ;Differenz
A00B 0608       70         LD  b,8 ;Zaehler fuer Schleife
A00D 7E         80  WIEDER LD  a,(hl) ;aktuelle Bitmatrix
A00E 2F         90         CPL  ;Invertieren
```

```
A00F 77       100          LD   (hl),a ;wieder Speichern
A010 19       110          ADD  hl,de ;Differenz addieren
A011 DC0000   120          CALL c,DIFADD ;Unterprogramm zur Korrektur
A014 10F7     130          DJNZ wieder
A016 C9       140          RET
**** Zeile  120  : DIFADD=A017
A017 D5       150   DIFADD PUSH de ;DE retten
A018 1100C0   160          LD   de,bildad ;=&C000
A01B 19       170          ADD  hl,de ;zu hl addieren
A01C D1       180          POP  de
A01D C9       190          RET

Programm :Invers
Start : &A000    Ende : &A01D
Laenge : 001E
 0  Fehler
Variablentabelle :
SCRDIF B1C9 BILDAD C000   WIEDER A00D   DIFADD A017
```

In Zeile 130 erfolgt ein Sprung zum Label DIFADD. DIFADD taucht aber erst in Zeile 150 wieder auf. Daher wird zunächst DC0000 als Code gespeichert. Bei der Übersetzung von Zeile 150 findet der Assembler das Label DIFADD und gibt an, daß dieses Label in Zeile 120 vorkam. Der Code DC0000 wird dabei automatisch richtig gestellt. Das ist genauso bei JR und JP möglich. Dieses Problem tritt auf, wenn im Programm ein Vorwärtssprung stattfindet.

Die Verarbeitung von Vorwärtssprüngen auf diese Weise ist notwendig, da es sich bei dem Assembler um einen 1-Pass Assembler handelt. Das bedeutet, daß der Assembler das Source-programm nur ein einziges Mal durchsucht.

Ein 2-Pass Assembler dagegen sucht beim ersten Durchlauf nur alle Variablen und Labels heraus und ordnet ihnen Werte zu. Erst beim zweiten Pass wird die Übersetzung vorgenommen. Große professionelle Assembler führen mehrere Durchläufe (PASSES) aus. Der 1-Pass Assembler ist für unsere Zwecke insofern sinnvoller, da er ca. doppelt so schnell wie ein 2-Pass Assembler ist.

Doch zurück zum Programm:

Natürlich gibt es noch einige andere Lösungen dieser Programmaufgabe. Zunächst ist nur entscheidend, ob das Programm die gestellte Aufgabe löst. Sinnvoll ist es jedoch nach der kürzesten und schnellsten Version zu suchen.

Bei den folgenden Programmen werden wir weniger auf Geschwindigkeit und Speicherplatzbedarf achten, als vielmehr auf die Verständlichkeit dieser Programme.

HL darf niemals kleiner als &C000 sein. Für H sind also die Werte &C0 bis &FF möglich. Bei allen Werten dieser Form, sind die obersten beiden Bits (Nummer 7 und 6) gesetzt. Zur Vorbeuge gegen Fehler, können wir bei jedem Schleifendurchlauf diese Bits auf 1 setzen. Das Unterprogramm ab Zeile 160 entfällt dann , und für Zeile 130 schreiben wir:

```
130 'SET 6,H
135 'SET 7,H
```

Mit der OR Verknüpfung kann diese Aufgabe noch schneller gelöst werden (mit OR können Bits gesetzt werden!).

```
 85 'LD C,&X11000000
130 'LD A,H
133 'OR C
135 'LD H,A
```

Auch die anderen bisher geschriebenen Programme zur Manipulation des Bildschirms, können durch das Berücksichtigen der Scrolldifferenz universell einsetzbar gemacht werden. Diese Änderungen bleiben Ihnen überlassen.

Monitorroutine BASIC

Wir haben die Arbeitsweise des Assemblers kennengelernt. Es gibt noch einige andere Hilfsmittel zur Verbesserung der Arbeit

mit der Maschinensprache. Zu denen zählt u.A. der sogenannte Monitor.

Hier handelt es sich nicht um den Monitor (Bildschirm) ihres Computers, vielmehr um ein Programm, mit dem Sie z.B. den Speicherinhalt anschauen (engl. to monitor: anschauen) oder ändern können. Ein Monitor bietet auch die Möglichkeit Maschinenprogramme zu speichern, zu laden oder zu starten. Im Folgenden wollen wir einige Funktionen eines solchen Monitors in Maschinensprache programmieren.

Dadurch "schlagen wir zwei Fliegen mit einer Klappe:"
Sie lernen grundsätzliche Programmiertechniken kennen und erhalten als Ergebnis ein einfaches Monitorprogramm.

Wie schon erwähnt, ist die grundsätzliche Aufgabe eines Monitorprogramms, den Speicherinhalt anzuzeigen. Das läßt sich im BASIC mit >PEEK< verwirklichen.

Schreiben Sie ein Programm, daß bei Eingabe von der Startadresse (V) und der Endadresse (V) die dazwischenliegenden Speicherinhalte ausgibt. Verwenden Sie bei der Ausgabe das übliche Format für einen Hex DUMP (Ausgabe der Speicherinhalte in hexadezimaler Form), und zwar:

Hex Dump von Adresse &10 bis &27 beim 464:

```
0010 C3 16 BA C3 10 BA D5 C9 C.:C.:UI
0018 C3 BF B9 C3 B1 B9 E9 00 C?9C19i.
0020 C3 CB BA C3 B9 B9 00 00 CK:C99..
```

Beim CPC 664 und CPC 6128 treten leicht andere Werte bei dem obigen Hex Dump auf.

Beachten Sie, daß rechts neben dem eigentlichen Hex Dump die ASCII Darstellung der Codes erfolgt. Codes, die größer als 127 sind, werden vorher um 128 erniedrigt. Nicht darstellbare Codes (0-31), werden als Punkt ausgegeben.

Lösung:

```
 10 REM Monitorroutine BASIC
 20 MODE 1
 30 INPUT start
 40 INPUT ende
 50 FOR i=start TO ende STEP 8
 60 ascii$=""
 70 PRINT HEX$(i,4);" ";
 80 FOR j=0 TO 7
 90 w=PEEK(i+j)
100 PRINT HEX$(w,2);" ";
110 IF w>127 THEN w=w-128
120 IF w<32 THEN w=46
130 ascii$=ascii$+CHR$(w)
140 NEXT j
150 PRINT" ";ascii$;
160 NEXT i
170 END
```

Mit diesem Programm können Sie sich nun das gesamte RAM des Rechner ansehen. Geben Sie in ihrem Monitorprogramm folgende Zeile ein:

1 REM Dies ist die erste Zeile

Sehen wir uns den Speicherinhalt von &170 bis &200 an. In der ASCII Darstellung der Speicherinhalte erkennen wir die erste Zeile, d.h. den Kommentar "Dies ist die erste Zeile" wieder. Ab &170 werden die BASIC Programme im Speicher abgelegt. Direkt auf das BASIC Programm folgt eine intern verwaltete Seite aller benutzten Variablen im Programm, wobei für die numerischen Variablen der Zahlenwert direkt abgespeichert ist, und für Stringvariablen die Adresse und die Länge der Zeichenkette gespeichert ist. Die Variablen sind dort in der Reihenfolge ihres Auftretens im Programm abgelegt.

Professionelle Monitorprogramme bieten die Möglichkeit direkt in der Anzeige die Speicherinhalt zu ändern. Soviel zum M (Monitor) Befehl des Programms.

Fillroutine

Nun beschäftigen wir uns mit der Routine "Fill". Sie wird benutzt, um einen beliebigen Speicherbereich mit einem beliebigen, festen Wert zu füllen. So kann zum Beispiel der gesamte Bildschirmspeicher gelöscht, d.h. mit Null gefüllt werden. Der F(-ill) Befehl wird z.B. benutzt, um vor der Programmausführung bestimmte Vorraussetzungen an Speicherinhalten zu realisieren. Es stellt sich folgendes Programmproblem:

Vom BASIC Programm wird die Eingabe der Start- und Endadresse des zu füllenden Bereiches und der Wert, mit dem dieser Bereich gefüllt werden soll, abgefragt. Im BASIC Programm soll geprüft werden, ob die Startadresse (V) kleiner als die Endadresse (V) ist und ob es sich um 2-Byte Zahlen, d.h. Zahlen zwischen 0 und 2^16-1 handelt. Weiterhin muß der Wert (V) auf den Bereich von 0-255 (1 Byte) getestet werden. Diese drei Werte (entsprechen 5 Bytes) werden dann an fest definierte Speicherplätze "gepoked", so daß sie dann nach dem Maschinenspracheaufruf der Fillroutine zur Verfügung stehen. Das Maschinenprogramm soll das eigentliche "Füllen" erledigen, danach erfolgt ein Rücksprung zum BASIC.

Hier folgt nun ein BASIC Programm, das eine Eingabe dieser Form verarbeitet und auf die oben aufgezeigten Kriterien überprüft.

```
 10 MEMORY &9FFF
 90 MODE 2
100 LOCATE 10,5:PRINT"MONITORPROGRAMM"
110 LOCATE  5,8:PRINT"BEDIEN
120 LOCATE 7,10:INPUT"STARTADRESSE:",START
130 IF START <0 OR START>=2^16 then 120
140 IF START <>INT (START) THEN 120
150 LOCATE 7,11:INPUT"ENDADRESSE:",ENDE
160 IF ENDE<= START OR ENDE >=2^16 THEN 150
170 IF ENDE <> INT (ENDE) THEN 150
180 LOCATE 7,12:INPUT"WERT:",WERT
190 IF WERT <0 OR WERT >255 OR (WERT <> INT(WERT))
THEN 180
200 POKE &A000,WERT
210 POKE &A002, INT(START/256): POKE &A001, START-INT (START/256)*256
220 POKE &A004, INT(ENDE/256): POKE &A003, ENDE-INT (ENDE/256)*256
230 CALL &A005
240 END
```

Für das Maschinenprogramm steht also der Wert (V) an Adresse &A000, die Startadresse steht ab &A001 (Low/High) und die Endadresse steht ab &A003 (Low/High) zur Verfügung.

Da die ersten Speicherplätze ab &A000 besetzt sind, starten wir das Maschinenprogramm ab Adresse &A005.
Der erste Teil des Sourceprogramms:

```
10 'ORG          &A005
20 'Start    EQU &A001
30 'Ende     EQU &A003
40 'Wert     EQU &A000
50 'LD   A,(Wert)
60 'LD  DE,(Start) ;Blockzeiger
```

Programmbeschreibung:

Zeile 10:

Start-Programm auf &A005

Zeile 20-40:

Der Übersichtlichkeit halber werden die Adressen der übergebenen Daten (Übergabeadressen) als Variablen definiert. Es braucht dann bei einer Änderung der Übergabeadressen nur der Wert in der Variablendefinition geändert zu werden.

Zeile 50-60:

Der Wert wird in den Akku (1Byte), die Endadresse in das HL Registerpaar (2 Byte) und die Startadresse in das DE Registerpaar geladen.

Damit kommen wir zur eigentlichen Fillroutine.
Zunächst die naheliegendste Lösung:

```
 70 'Schlei LD (DE),A ; Wert schreiben
 80 'INC DE           ; Zeiger erhöhen
 90 'LD HL,(Ende) ;berechnen,
100 'SBC HL,DE   ;ob schon
110 'JR NZ,Schlei    ; Ende erreicht ?
120 'LD (DE),A       ; letztes Element füllen
130 'RET
140 'END
```

Programmbeschreibung

Zeile 50:

HL mit Enadresse (V) laden

Zeile 70:

Anfang der Schleife. An Adresse HL wird der Wert (A) gespeichert.

Zeile 80:

Der Adresszeiger (DE) wird erhöht.

Zeile 100:

16-Bit Subtraktion der aktuellen Adresse von der Endadresse (HL-DE)

Zeile 110:

Ist der Adresszeiger DE kleiner als die Endadresse in HL, so ist das Z Flag nicht gesetzt, da HL-DE ungleich 0 ist. In diesem Fall (NZ) wird zum Schleifenanfang (Schlei) gesprungen. Ist HL jedoch gleich DE, so ist Z=1 und der nächstfolgende Befehle (Zeile 120) wird aufgeführt.

Zeile 120:

Hier wird der Wert A (=Akkuinhalt) auch noch an die Endadresse des zu füllenden Bereichs geschrieben. Das war noch nicht geschehen !! (Warum??)

Zeile 130:

Zurück zum BASIC

Wenn Sie dieses Programm vom Assembler übersetzen lassen, erhalten Sie folgendes Assemblerlisting:

```
A000           10  START  EQU  &a001
A000           20  ENDE   EQU  &a003
A000           30  WERT   EQU  &a000
A005           40         ORG  &A005
A005 3A00A0    50         LD   a,(Wert)
A008 ED5B01A0  60         LD   de,(Start) ;Blockzeiger
A00C 12        70  SCHLEI LD   (de),a ;Wert cchreiben
A00D 13        80         INC  de ;Zeiger erhoehen
```

```
A00E 2A03A0    90       LD   hl,(Ende)
A011 ED52     100       SBC  hl,de ;Ende erreicht?
A013 20F7     110       JR   nz,Schlei ;nein, weiter
A015 12       120       LD   (de),a ;letztes Element
fuellen
A016 C9       130       RET

Programm :Fill
Start : &A005    Ende : &A016
Laenge : 0012
 0  Fehler
Variablentabelle :
START  A001   ENDE   A003   WERT   A000   SCHLEI A00C
```

Schreiben Sie einen BASIC Lader für dieses Programm, und integrieren Sie ihn in das BASIC Fillprogramm.

```
20 FOR I=&A000 TO &A016:READ a$:a=VAL("&"+a$):POKE i,a:NEXT
25 DATA ff,00,c0,ff,ff
30 DATA 3a,00,a0,ed,5b,01,a0,12
40 DATA 13,2a,03,a0,ed,52,20,f7
50 DATA 12,c9
```

Wie schon erwähnt, ist dieses Programm die wohl einfachste Möglichkeit die Fillroutine zu realisieren. Es ist jedoch zu lang und zu langsam.
Die schnellste Lösung erhalten wir unter Benutzung der Blockladebefehle. Zum Füllen eines Bereiches müssen wir sie absichtlich falsch benutzen. (Siehe Kapitel 4.3)

Source Programm: Zeile 10 bis 70 wie oben

```
 80 'SBC HL,DE     ; Länge des Blockes
 90 'LD B,H        ; Byte-Counter mit
100 'LD C,L       ; Blocklänge laden
110 'LD H,D       ; Quellblock Anfang (HL)
```

```
120 'LD L,E        ; Mit Startadresse laden
130 'INC DE        ; Zieladresse=Startadresse+1
140 'LD (HL),A     ; erstes Quellbyte mit Wert laden
150 'LDIR
160 'RET
170 'END
```

Übersetzen Sie auch dieses Maschinenprogramm für einen BASIC Lader. Starten Sie das BASIC Programm, wählen Sie die Startadresse &C000, Endadresse &CFFF und Wert &FF.
Der Block liegt im Bildschirmbereich. Wert=&FF=&X1111 1111 entspricht 8 gesetzten Punkten. Als Ergebnis sollten auf dem Bildschirm waagerecht, ein Punkt breite Streifen entstehen.

Transferroutine

Als nächstes wollen wir die Blockladebefehle "richtig" einsetzen, um eine Transferroutine zu schreiben. Dieses Programm soll einen Speicherbereich an eine andere Stelle übertragen. Mit Hilfe eines BASIC Programms, soll die Anfangs- und Endadresse des Quellblockes, sowie die Anfangsadresse des Zielblocks eingegeben und auf Richtigkeit überprüft werden. Für die Übergabe benutzen wir folgende Adressen:

```
Quellblock   Anfang: &A020/&A021
Quellblock   Ende  : &A022/&A023
Zielblock    Anfang: &A024/&A025
```

Die Anfangsadresse des Maschinenprogramms ist dann &A026.

Für den Fall, daß der Quell- und der Zielblock sich nicht überlappen, soll der Zielblock die richtigen Daten enthalten, auch wenn dadurch der alte Inhalt des Quellblocks überschrieben wird.

Source Programm:

```
  5 'BLOCKVERSCHIEBEROUTINE
 10 'QANF  EQU  &A020  ;QUELLBLOCK ANFANGSADRESSE
 20 'QEND  EQU  &A022  ;QUELLBLOCK ENDADRESSE
 30 'ZANF  EQU  &A024  ;ZIELBLOCK ANFANGSADRESSE
 40 'ORG   EQU  &A026  ;PROGRAMMSTART
 45 ' ;PROGRAMMSTART, BLOCKLAENGE ERMITTELN
 50 'LD HL,(QEND)
 60 'LD DE,(QANF)
 70 'OR A; CARRY FUER SBC LOESCHEN
 80 'SBC HL,DE ;=BLOCKLAENGE-1
 90 'INC HL    ;+1=BLOCKLAENGE
100 'LD B,H   ;BLOCKLAENGE NACH
110 'LD C,L   ;BC SPEICHERN
115 ' ;ENTSCHEIDUNG AUF INC-ODER DECREMENTIEREN
120 'LD HL,(QANF)
130 'SBC HL,DE ;ZANF KLEINER ALS
140 'JR C,LADINC ;QANF,DANN LADINC
150 'SBC HL,BC   ;DIFFERENZ GROESSER ALS
160 'JR NC,LADINC ;BLOCKLAENGE,DANN LADINC
170 'LD HL,(ZANF)
180 'ADD HL,BC ;ZANF+LAENGE
190 'DEC HL    ;-1=ZIELBLOCKENDE
200 'EX DE,HL  ;VON HL NACH DE LADEN
210 'LD HL,(QEND) ; QUELLBLOCKENDE
220 'LDDR
230 'RET
240 ' ;BLOCKLADEN INCREMENTIEREN
250 'LADINC EX DE,HL ; QUELLANFANG VON DE NACH HL
260 'LD DE,(ZANF)
270 'LDIR
280 'RET
290 'END
```

Anfang und Ende des Programms bedürfen keiner weiteren Erklärung. Schwieriger ist der Mittelteil, wo entschieden wird, ob der Befehl LDDR oder LDIR angewendet werden soll. (Zeile 115-160). Vergegenwärtigen Sie sich die Notwendigkeit dieser Unterscheidung (Kapitel 2.3). Im Normalfall, d.h. bei keiner Überlappung der Blöcke, verwenden wir den LDIR Befehl. Ist die Anfangszieladresse kleiner als die Quellblockanfangsadresse, kann auch LDIR verwendet werden. Durch die Subtraktion in Zeile 130 und den Sprung in Zeile 140, wird für den Fall Zanf<Qanf, zum Blockladen Incrementieren verzweigt. Ist Zanf>=Qanf, muß entschieden werden, ob Zanf<=Qend ist.

```
           Zanf <=  Qend
           Zanf <=  Qanf+Länge-1
Zanf-Qanf-Länge <=-1
          HL-BC <= -1
```

Ist nach HL-BC das Carry gesetzt (Ergebnis kleiner oder gleich -1), so muß LDDR verwendet werden. Ist das Carry=0, so war HL-BC >=0, also lag Zanf nicht im Quellblock, und es wird zu LDIR verzweigt.
Um dieses Programm wieder in den Monitor einzubinden, mußte es in DATA Zeilen abgelegt werden. Bei einem Programm dieser Länge entstehen dabei oft Fehler. Um diesem Problem zu begegnen, gibt es zwei Möglichkeiten. Während des Lesens der Data Zeilen werden alle gelesenen Werte addiert, und die Endsumme wird mit einer Prüfsumme verglichen. Stimmt die Endsumme nicht mit der Prüfsumme überein, so liegt ein Fehler vor. Für unser Programm sieht das folgendermaßen aus:

```
10 FOR I=&A020 TO &A051
20 READ a$:a=VAL("&"+a$):POKE i,a:s=s+a:NEXT
30 DATA 00,80,ff,bf,00,c0
40 DATA 2A,22,A0,ED,5B,20,A0,B7
50 DATA ED,52,23,44,4D,2A,24,A0
```

```
 60 DATA ED,52,38,10,ED,42,30,0C
 70 DATA 2A,24,A0,09,2B,EB,2A,22
 80 DATA A0,ED,B8,C9,EB,ED,5B,24
 90 DATA A0,ED,B0,C9
100 IF s<> 5186 THEN PRINT"Fehler in Datas" ELSE PRINT "ok!"
```

Für uns ist die zweite Möglichkeit einfacher:
Nach dem Assemblerlisting des Programms, haben Sie sicherlich den Objektcode auf Cassette (Diskette) gespeichert. Mit >LOAD "Programmname"< können Sie dieses Programm auch von einem BASIC Programm aus laden.
Ein Monitorprogramm sollte auch die Möglichkeit bieten Maschinenprogramme zu laden und zu speichern.

```
Mit Hilfe von >LOAD "Name",Adresse< und
              >SAVE "Name",B,Startadresse,Länge<
```

läßt sich das leicht realisieren.
Verbinden Sie alle Funktionen, so "kann" ihr Monitor folgende Befehle:

```
M-(engl.Monitor)  - Speicherbereich anzeigen
F-(engl.Fill)     - Speicherbereich mit Wert (V) füllen
T-(engl.Transfer)- Speicherbereiche verschieben
L-(engl.Load)     - Maschinenprogramme laden
S-(engl.Save)     - Maschinenprogramme speichern
```

Compareroutine

Nun beschäftigen wir uns mit der Compareroutine. Sie dient zum Vergleichen zweier Speicherbereiche. Ihr Befehlskürzel ist C. Als Eingaben von BASIC Programmen aus, benötigt die Routine die Anfangs- und Endadresse des Ausgangsblockes und die Anfangsadresse des zu vergleichenden Blockes. Alle Adressen des zu vergleichenden Blockes, an denen die gespeicherten Werte nicht mit dem entsprechenden des Ausgangsblockes übereinstimmen, sollen ausgegeben werden.

Source Programm:

```
 10 ' ORG &A060
 20 ' Flag DB 1
 30 ' Anf DS 2
 40 ' Ende DS 2
 50 ' Anfver DS 2 ;Vergleichsblockanfang
 60 ' LD DE,(Anf)
 70 ' LD HL,(Ende)
 80 ' OR A
 90 ' SBC HL,DE ;Blocklaenge
100 ' INC HL ;+1
110 ' LD B,H
120 ' LD C,L ;nach BC laden
130 ' EX DE,HL ;Anf nach HL
140 ' LD DE,(Anfver) ;Blockzeiger
150 ' Weiter LD A,(DE) ;Vergleichselement
160 ' INC DE
170 ' CPI ;Vergleich (HL) mit A
180 ' JR NZ,Ausga ;ungleich dann Ausgabe
190 ' JP PE,Weiter ;naechstes Element
200 ' LD A,B
210 ' LD (Flag),A ;Ende, Flag=0
220 ' RET
230 ' Ausga LD (Anf),HL
240 ' LD (Anfver),DE
250 ' RET PE ;Ungleiche Adresse
260 ' DEC B ;B=255
270 ' LD A,B
280 ' LD (Flag),A ; Flag=255
290 ' RET ;nach Flag
300 ' END
```

In den Zeilen 20-50 wird Speicherplatz für die zu übergebenden Daten reserviert. Dazu werden die Pseudobefehle benutzt. Der Befehl DB (Define Byte) legt an der aktuellen Adresse den als Operand angegebenen Wert ab.
In unserem Fall wird dadurch der Wert 1 an Adresse &A060 gespeichert. Diese Speicherstelle dient als Flag für die Kommuni-

kation mit dem BASIC Programm. Folgende Werte für Flag sind möglich:

```
  1- Beim Vergleich wurde Ungleichheit festgestellt, der
     ist jedoch noch nicht zuende verglichen
  0- Der Block ist bis zum Ende verglichen
255- Der Block ist bis zum Ende verglichen und beim
     letzten Blockelement wurde Ungleichheit festgestellt.
```

In den Zeilen 30 und 50 steht der DS (Define Storage: Definiere Speicherplatz) verwendet. Der Pseudobefehl DS weist den Assembler an den mpc, um die angegebene Zahl von Speicherstellen zu erhöhen. Dadurch wird dieser Platz freigehalten, und wir können dort die Übergabevariablen speichern. In unserem Fall benötigen wir für das Abspeichern von Anf, Ende und Anver je zwei Bytes (Low und High Byte der Adresse), folglich haben wir DS 2 benutzt.

In den Zeilen 60-120 wird der Bytecounter BC mit der Länge des Ausgangsblockes geladen. Der INC HL Befehl in Zeile 100 ist notwendig, da sonst das letzte Element nicht mehr verglichen würde.

In Zeile 130 wird HL mit der Anfangsadresse des Ausgangsblockes, und in Zeile 140 DE mit der Anfangsadresse des Vergleichsblockes geladen.

Ab Zeile 150 beginnt die Hauptschleife des Programms. Zunächst wird der Akku mit dem jeweiligen Wert des Vergleichsblockes geladen (150), und der Zeiger im Vergleichsblock wird erhöht (160). CPI hat mehrere Funktionen. Es vergleicht den Akkuinhalt (=Wert aus Vergleichsblock) mit dem Wert an der Adresse HL (=Wert im Ausgangsblock). Je nach Ausgang des Vergleiches wird das Z Flag beeinflußt. Weiterhin wird HL erhöht und BC erniedrigt. Ist BC danach gleich Null, so wird das P/V rückgesetzt (PO), ansonsten gesetzt (PE).

In Zeile 180 wird zur Ausgabe gesprungen, wenn die verglichenen Werte ungleich waren. Lag Gleichheit vor, wird durch Zeile 190 die Schleife wiederholt, wenn P/V=0, d.h. PE war. Ist

P/V dagegen 1, so wird in Zeile 200 das Flag auf 0 gesetzt und es erfolgt ein Rücksprung ins BASIC.

Ab Zeile 220 beginnt der Programmteil zur Ausgabe.
Zuerst werden die aktuellen Blockzeiger abgespeichert. DE enthält dabei die um 1 erhöhte Adresse der Speicherstelle, die nicht gleich ist. Nach der Ausgabe dieser Adresse durch das BASIC Programm wird die Routine erneut aufgerufen und an der richtigen Stelle fortgesetzt, da Anf und Anfver vor dem Sprung ins BASIC durch die Zeilen 230 une 240 auf den aktuellen Stand gesetzt wurden. Ist der Block noch nicht zuende verglichen, d.h. BC<>0 und P/V=1 also PE, wird der RET Befehl ausgeführt. Ist dagegen das letzte Element verglichen worden (Gleichheit liegt nicht vor), so wird durch die Zeilen 260/280 Flag (V) auf 255 gesetzt, um von dem Fall, daß das Blockende erreicht war und Gleichheit (!) vorlag (d.h. Flag=0) zu unterscheiden.

```
A060           10          ORG  &A060
A060 01        20   FLAG   DB   1
A061           30   ANF    DS   2
A063           40   ENDE   DS   2
A065           50   ANFVER DS   2 ;Vergleichsblockanfang
A067 ED5B61A0  60          LD   DE,(Anf)
A06B 2A63A0    70          LD   HL,(Ende)
A06E B7        80          OR   A
A06F ED52      90          SBC  HL,DE ;Blocklaenge
A071 23        100         INC  HL ; +1
A072 44        110         LD   B,H
A073 4D        120         LD   C,L ;nach BC laden
A074 EB        130         EX   DE,HL ;Anf nach HL
A075 ED5B65A0  140         LD   DE,(Anfver) ;Blockzeiger
A079 1A        150  WEITER LD   A,(DE) ;Vergleichselement
A07A 13        160         INC  DE
A07B EDA1      170         CPI   ;Vergleich (HL) mit A
A07D 20FE      180         JR   NZ,Ausga ;ungleich dann Ausg
abe
A07F EA79A0    190         JP   PE,Weiter ;naechstes Element

A082 78        200         LD   a,b
A083 2260A0    210         LD   (Flag),A ;Ende, Flag=0
```

```
A086 C9       220         RET
**** Zeile  180  : AUSGA=A087
A087 2261A0   230  AUSGA  LD   (Anf),HL
A08A ED5365A0 240         LD   (Anfver),DE
A08E E8       250         RET  PE ;Ungleiche Adresse
A08F 05       260         DEC  B ;B=255
A090 78       270         LD   a,b
A091 3260A0   280         LD   (Flag),a
A094 C9       290         RET   ;nach Flag

Programm :Compare
Start : &A060    Ende : &A094
Laenge : 0035
 0  Fehler
Variablentabelle :
FLAG   A060   ANF    A061   ENDE   A063   ANFVER A065   WEI
TER A079   AUSGA  A087
```

Das BASIC Programm zum Aufruf der Routine sieht so aus:

```
 10 REM COMPARE
 20 MEMORY &9FFF
 30 MODE 2
 40 POKE &A060,1
 50 INPUT "Blockanfang :&",a$
 60 adr=&A061: GOSUB 170
 70 INPUT "Blockende   :&",a$:
 80 adr=&A063 : GOSUB 170
 90 INPUT "Vergleichsblockanfang :&",a$
100 adr=&A065 : GOSUB 170
110 CALL &A067
120 w=PEEK (&A060)
130 IF w=0 THEN END
140 PRINT HEX$(PEEK(&A061)+256*PEEK(&A062)-1,4)
150 IF w=1 THEN 110
160 END
170 a=VAL ("&"+a$)
```

```
180 IF a<0 THEN a=a+2^16
190 ah=INT (a/256)
200 POKE adr,a-ah*256
210 POKE adr+1,ah
220 RETURN
```

Den Befehl GO (G), also den Aufruf eines Maschinenprogramms vom Monitorprogramm aus (z.B. zu Testzwecken), können Sie leicht mit dem BASIC Befehl >CALL Adresse<, und einer entsprechenden Eingaberoutine für die Adresse (V), selbst programmieren.

Bei dem Compareprogramm ist das Hin- und Herspringen zwischen BASIC und Maschinensprache recht umständlich und unübersichtlich. Die Verbindung zwischen Maschinensprache und BASIC war notwendig, da wir in Maschinensprache noch keine Ein- und Ausgabe (d.h. >INPUT< bzw. >PRINT<) programmieren können. Diese Routinen sind relativ kompliziert. Zum Ausgeben z.B. eines Buchstabens auf dem Bildschirm, müßte die richtige Position des Buchstaben unter Berücksichtigung der Scrolldifferenz berechnet werden. Danach müssen die 8-Bytes, die zur Darstellung des Zeichens dienen, aus dem Zeichenspeicher (ROM &3800 bis &3FFF) gelesen und in den Bildschirmspeicher geschrieben werden. Da die Ausgabe von Zeichen auf dem Bildschirm schon nach dem Einschalten des Rechners funktioniert, muß die Routine dafür schon im ROM existieren. Wenn wir diese Routine oder wenigstens ihre Startadresse kennen würden, könnten wir sie direkt von unserem Maschinenspracheprogramm aus aufrufen. Diese Möglichkeit mit Hilfe der Maschinensprache sogenannte Systemroutinen aufzurufen ist sehr nützlich und interessant.

KAPITEL VI: BENUTZUNG VON SYSTEMROUTINEN

6.1 Disassembler und Einzelschrittsimulator

Die CPC's besitzten 32K ROM plus 16K Disk ROM (falls ein Diskettenlaufwerk vorhanden ist). Diese 32 Kilobyte sind mit Systemroutinen beschrieben. Die oberen 16K ROM (&C000 bis &FFFF) enthalten das BASIC, die unteren 16K (&0 bis &3FFF) das Betriebssystem des Rechners. Im Betriebssystem und auch im BASIC sind viele Routinen enthalten, die für den Maschinenprogrammierer von Interesse sind.

Zum Analysieren dieser Routinen benötigen wir ein weiteres "Werkzeug", den Disassembler.

Ein Disassembler interpretiert die Bytes eines eingegebenen Bereiches als Maschinencode und übersetzt die Zahlen in die dazugehörigen Assemblerbefehle. Damit bildet der Disassembler das Gegenstück zum Assembler. Mit dem Disassembler können wir fremde Maschinenprogramme, die als BASIC Lader (DATA Zeilen) gegeben sind, nach dem Laden in Assemblerbefehle rückübersetzen. Auch rechnerinterne Routinen lassen sich übersetzen. Aus diesen, von "Profis" erstellten Programmen, läßt sich viel abgucken. Außerdem können wir die Routinen noch in unseren eigenen Programmen verwenden.

Das im folgenden abgedruckte Programm "Disassi+Simula" beinhaltet unter anderem einen Disassembler. Starten Sie das Programm nach dem Eintippen mit >RUN<. Der erscheinende Stern am linken Rand zeigt an, daß eine Eingabe erwartet wird. In der ersten Bildschirmzeile steht außerdem "Eingabemodus". Wählen Sie also zunächst durch Eingabe von "d" den Disassembler.

Das dann erscheinende "D" zeigt an, daß Ihre Eingabe verstanden wurde. Geben Sie nach dem & Zeichen nun die Adresse an der die Disassemblierung beginnen soll ein. Probieren Sie es mit 20 und drücken dann >RETURN< bzw. >ENTER<. Ein zweites

& Zeichen zeigt an, das Sie auch noch die Endadresse eingeben sollen. Geben Sie für diesen ersten "Testlauf" 22 ein. Nun beginnt der Disassembler:

0020 C3C6BA JP &BACB
(beim 664 steht JP &BAC6/ beim 6128 steht JP &BAC6)

Setzen Sie die Disassemblierung an der angegebenen Sprungadresse fort. Dazu müssen Sie zunächst wieder den Disassembler mit "D4 aufrufen. Es gibt allerdings noch eine weitere komfortablere Möglichkeit:

Sobald Sie sich im Eingabemodus befinden, können Sie durch Drücken von >RETURN< bzw. >ENTER< die zuletzt ausgeführte Funktion nochmals aufrufen. Geben Sie jetzt also die Sprungadresse an die Sie eben übersetzt haben und schließen mit >RETURN/ENTER< ab.

Die Eingabe der Endadresse können Sie auch erledigen, indem Sie einfach >RETURN< bzw. >ENTER< drücken. Daraufhin wird zu der eingegebenen Startadresse &18 addiert. Das hat zur Folge, daß in den meisten Fällen "ein Bildschirm voll" disassembliert wird.

Sie erhalten folgendes Bild.

```
BACB  F3        DI
BACC  D9        EXX
BACD  59        LD    E,C
BACE  CB D3     SET   2,E
BAD0  CB DB     SET   3,E
BAD2  ED 59     OUT   (C),E
BAD4  D9        EXX
BAD5  7E        LD    A,(HL)
BAD6  D9        EXX
BAD7  ED 49     OUT   (C),C
BAD9  D9        EXX
BADA  FB        EI
BADB  C9        RET
BADC  D9        EXX
```

```
BADD  79          LD    A,C
BADE  F6 0C       OR    &0C
BAE0  ED 79       OUT   (C),A
BAE2  DD 7E 00    LD    A,(IX+0)
```

Die am Anfang jeder Zeile stehenden Adressen sind bei Ihnen anders, wenn Sie nicht den 464 benutzen. Die übersetzten Befehle sind jedoch bei allen Schneider Rechnern gleich.

Diese eben übersetzte Systemroutine dient zum Lesen des RAM's. Der an Adresse HL im RAM stehende Wert wird unabhängig vom jeweiligen ROM/RAM Zustand in den Akku geladen. Die Routine wird über den RST &20 Befehl aufgerufen.

Wenn Sie jetzt noch weiter disassemblieren wollen, geben Sie >RETURN< ein (für Disassembler), >RETURN< für die Anfangsadresse, was bedeutet, daß ab der letzten übersetzten Adresse fortgefahren wird, und >RETURN< für die Endadresse, was die schon erklärte Bedeutung hat.

Auf diese Weise können Sie fortlaufende Programme einfach disassemblieren.

Betrachten wir noch einige andere Funktionen des Programms:

Durch das Eingeben von "H" (Help) erhalten Sie die Liste aller Befehle, die dieses Programm zur Verfügung stellt.

Mit "P" können Sie den Printer (Drucker) an- bzw. ausschalten. Der aktuelle Zustand ist oben rechts angezeigt.

Durch "B" verabschiedet sich das Programm von Ihnen.

Eine wichtige Funktion im Zusammenhang mit dem Disassembler ist "R", was für den ROM/RAM Status steht. Geben Sie "R" ein:

Das "RAM" hinter "lo:" erscheint invers. "lo:" bedeutet, daß der Status des lower ROM, also die Adressen &0 bis &3FFF gemeint sind. Durch Drücken der Leertaste können Sie von RAM zum ROM Status gelangen und umgekehrt.
Haben Sie Ihre Wahl für das low ROM getroffen, so gehen Sie mit >RETURN< zum hi RAM (&C000 bis &FFFF). Die Leertaste hat dieselbe Funktion wie bereits erwähnt.

Es gibt aber hier auch die Möglichkeit einen von 252 Expansionsroms einzuschalten. Zu den Expansionsroms zählt z.B. das Disketten ROM, das die Nummer 7 hat. Wenn Sie einen Expansionsrom wählen wollen, geben Sie >CTRL+E< ein.

Danach können Sie hinter "hi:" die gewünschte Nummer eingeben z.B. 7 für das Disk ROM. Anschließend ist nur für die Adressen &C000 bis &FFFF der gewählte ROM selektiert. Alle anderen Adressen sprechen dann das RAM an. Die RSX Erweiterung RDEEK, die auch ROM's "PEEKen" kann, wird im Kapitel RSX genau erklärt.

Eine weitere Funktion, die direkt mit dem Disassembler zusammenhängt, ist der Sourceerzeuger.

Der Sourceerzeuger erzeugt nach Aufruf durch "S" ein Sourcelisting der angegebenen Adressen, das ab Zeile 10000 abgelegt wird. Dieses Sourceprogramm kann direkt vom Assembler verändert werden. Damit haben Sie eine einfache Methode z.B. Systemroutinen für eigene Programme zu verwenden, ohne sie abtippen zu müssen. Die erzeugten Sourceprogramme können natürlich später verändert und den individuellen Bedürfnissen angepaßt werden.

Machen Sie einen Testlauf:

Selektieren Sie mit "R" usw. das Lower ROM. Die Frage: "Labelnamen definieren?" überspringen Sie zunächst durch Eingabe von Komma und >RETURN<. Dann geben Sie 1D2 als

Start- und 1E1 als Endadresse ein. Verabschieden Sie sich (B) und listen Sie die Zeilen ab 10000. Das so erzeugte Sourceprogramm können Sie nun getrennt abspeichern und weiterverarbeiten.

Kommen in dem zu disassemblierenden Stück Adressen vor, die bekannt sind, also welche, denen man ein Label zuordnen kann, so ist das nach der Frage "Labelnamen definieren:" möglich.

Die Routine &BCCB wird oft als "ROMWALK" bezeichnet, da sie die vorhandene Speicheraufteilung feststellt. Wir wollen also z.B. der Adresse &BCCB das Label ROMWALK zuordnen.

Selektieren Sie hi-ROM und rufen den Sourceerzeuger auf. Dann geben Sie

Romwalk,&BCCB

ein. Fahren Sie genauso fort, um noch mehr Labels zu definieren. Beenden Sie die Eingabe mit "," und >RETURN<.

Nun geben Sie als Adresse &C006 bis &C00C ein und schauen sich das Ergebnis an.
Die Routine, die vom BASIC aus BASIC Zeilen erzeugt, wie das bei dem Sourceerzeuger der Fall ist, werden wir später genau besprechen.

Die übrigen Funktionen des Programms betreffen alle den Einzelschrittsimulator.

Erklärung zum Einzelschrittsimulator

Eine der Schwierigkeiten beim Programmieren in Assembler besteht darin, daß für die erstellten Programme keine einfache Testmöglichkeit besteht. Ein Fehler, der im BASIC unter Umständen nur zu einem "Syntax Error" o.ä. führen würde, hat bei Maschinenprogrammen meist einen Systemabsturz zur Folge. Dadurch wird die Entwicklungszeit für ein Programm sehr verlängert und eine Fehlersuche wird schwierig. Um den Fehler

im Programm zu finden, wäre es hilfreich, das Maschinenprogramm Schritt für Schritt auszuführen und jeweils die Registerinhalte auf Korrektheit zu überprüfen. Ohne den Simulator muß diese langwierige Arbeit per Hand ausgeführt werden.

Die Simulatorfunktion des Programms wird diese Arbeit für Sie erledigen. Wählen Sie ROM's aus, rufen Sie den Simulator mit "Z" auf und geben Sie 1D7 ein:

```
SZ H PNC  A  B C  D E  H L   IX   IY  SZ H PNC' A' B'C' D'E' H'L'  SP
00000000  00 0000 0000 0000 0000 0000 00000000  00 0000 0000 0000 9FF0
01D7  23            INC   HL
```

An dieser Stelle zeigt der "*" an, daß eine Eingabe erwartet wird. Folgende Eingaben sind vorgesehen:

Leertaste
A
E

Durch das Drücken der Leer- oder Returntaste wird die Simulation fortgesetzt. Es erscheinen die evtl. veränderten Registerinhalte und der nächste Befehl:

```
SZ H PNC  A  B C  D E  H L   IX   IY  SZ H PNC' A' B'C' D'E' H'L'  SP
00000000  00 0000 0000 0000 0000 0000 00000000  00 0000 0000 0000 9FF0
01D8  70            LD    (HL),B
```

Die A Taste hat eine wichtige Funktion:

Durch das Drücken von "A" geht das Programm in den Editormodus. Damit haben Sie die Möglichkeit die Registerinhalte und auch die Programmadresse beliebig zu ändern. Gehen Sie dabei unbedingt in folgender Reihenfolge vor:

Nach dem Betätigen der A Taste steht der Cursor direkt unter dem S (=Sign Flag) der letzten Registerausgabe. Die in dieser Zeile angezeigten Inhalte, entsprechen den angegebenen Registern. Innerhalb dieser Zeile können Sie nun die Änderungen vornehmen, indem Sie die von Ihnen gewünschten Registerinhalte an den vorgegebenen Stellen in hexadezimaler Form (Groß- und Kleinschrift ist erlaubt) eingeben. Die Eingabe für das Flagregister muß natürlich in binärer Form, d.h. mit 0 und 1, erfolgen. Vorhandene Inhalte können mit Copy übernommen werden. Sollte Copy nicht bewegbar sein, geben Sie vorher "Shift+Cursor rechts" und "Shift+Cursor links" ein. Noch einen Satz zur Anzeige:

Am Anfang der Zeile steht das Flagregister als Binärzahl. Über den jeweiligen Bits steht ihre Bedeutung:

```
S- Sign Flag
Z- Zero Flag
H- Halbübertrag Flag
P- P/V als Parität oder Überlauf Flag
N- BCD Subtraktion Flag
C- Carry Flag
```

Darauf folgen der Akkumulator A (8-Bit) und die Universalregister B bis L, jeweils zu 16-Bit Registern zusammengefaßt, gefolgt vom IX und IY Register.
Darauf folgend wiederholt sich analog die Registerreihenfolge für den Zweitregistersatz. Am Ende der Zeile steht der Stapelzeiger.

Der Stapelzeiger (SP) darf niemals Adressen kleiner als &.... enthalten. Sein Maximalwert ist die Startadresse eines evtl. vorhandenen Maschinenprogrammes, also meist &A000. Bei Werten kleiner als &.... würde das Maschinenprogramm, welches die Simulation durchführt, überschrieben werden, was zum Absturz des Rechners führen würde . Ggf. muß SP geändert werden. Das sollte jedoch nur am Anfang einer Simulation geschehen, da sonst die Reihenfolge der Rücksprungadressen durcheinandergerät.

Sie müssen mindestens mit Copy die gesamte Zeile durchlaufen bzw. direkt eingegeben haben. Wenn Sie also fertig sind, gehen Sie mit >RETURN< in die nächste Zeile. Dafür benutzen Sie bitte nicht die Cursor- oder andere Tasten. Die Folge einer Fehlbedienung wäre, daß die Register nicht mit den gewünschten Werten geladen werden würden.

Nachdem Sie mit >RETURN< die Änderungen in der oberen Zeile abgeschlossen haben, wird der Cursor am Anfang der Zeile, die den übersetzten Befehl enthält, stehen. In dieser Zeile können Sie nun den PC ändern, d.h. die ersten 4 Hexziffern in der Zeile. Die von Ihnen hier eingegebene Adresse wird wiederum nach Eingabe von >RETURN< als Fortsetzungsadresse für die Simulation betrachtet. Auch wenn Sie diese Adresse nicht ändern wollen, müssen Sie sie, wie oben, mindestens mit Copy übernehmen.

Damit ist der, durch das Betätigen der A Taste, eingeschaltete Editormodus beendet und die Simulation wird fortgesetzt.

Die letzte Bedienungsmöglichkeit ist die E Taste.

Sie schaltet zwischen Echtsimulation oder Pseudosimulation hin und her. Echtsimulation bedeutet, daß alle Befehle wirklich ausgeführt werden. Das kann allerdings bei einigen Befehlen fatale Folgen haben. Zu diesen "gefährlichen" Befehlen zählen alle, die die Konfiguration des Systems real verändern. Dazu gehören Befehle, die die Speicherinhalte verändern, z.B. LD (&B1DB),A. Die veränderten Speicherinhalte könnten wichtige Informationen, die der BASIC Interpreter oder das Betriebssystem benutzen, verändern und dadurch Fehler verursachen. Auch die I/O Befehle können, da sie unter anderem für die Auswahl von ROM und RAM zuständig sind, sofort zum Absturz des Rechners führen, falls sie "echtsimuliert" werden. Vorsicht ist auch bei Befehlen, die das SP direkt betreffen, empfohlen.

Aus diesen Gründen können Sie bevor der Befehl ausgeführt wird, jedesmal mit der E Taste entscheiden, ob Sie den Befehl wirklich ausführen lassen wollen oder nicht.

Die Anzeige in der rechten oberen Ecke zeigt Ihnen dabei, ob zur Zeit echtsimuliert wird oder nicht.

Betrachten wir als Beispiel die Simulation der Test HL-DE Routine ab ROM Adresse &FF. Selektieren Sie HI ROM und schalten Sie die Echtsimulation ein. Rufen Sie mit 7 den Simulator auf und geben &D232 (664:&D282 / 6128:&D282) als Startadresse ein. Vergleichen Sie Ihre Anzeige mit der hier abgedruckten. Bis auf die Sprungadressen und die Adressen der Befehle sollte Ihre Anzeige mit der hier abgedruckten übereinstimmen (beim 464 vollständige Übereinstimmung!).

```
SZ H PNC  A  B C  D E  H L   IX   IY  SZ H PNC' A' B'C' D'E' H'L'  SP
00000000  00 0000 0000 0000 0000 0000 00000000  00 0000 0000 0000 9FF0
D232  11 4F 00       LD    DE,&004F
SZ H PNC  A  B C  D E  H L   IX   IY  SZ H PNC' A' B'C' D'E' H'L'  SP
00000000  00 0000 004F 0000 0000 0000 00000000  00 0000 0000 0000 9FF0
D235  CD B8 FF       CALL  &FFB8
SZ H PNC  A  B C  D E  H L   IX   IY  SZ H PNC' A' B'C' D'E' H'L'  SP
00000000  00 0000 004F 0000 0000 0000 00000000  00 0000 0000 0000 9FEE
FFB8  7C             LD    A,H
SZ H PNC  A  B C  D E  H L   IX   IY  SZ H PNC' A' B'C' D'E' H'L'  SP
00000000  00 0000 004F 0000 0000 0000 00000000  00 0000 0000 0000 9FEE
FFB9  92             SUB   D
SZ H PNC  A  B C  D E  H L   IX   IY  SZ H PNC' A' B'C' D'E' H'L'  SP
01000010  00 0000 004F 0000 0000 0000 00000000  00 0000 0000 0000 9FEE
FFBA  C0             RET   NZ
SZ H PNC  A  B C  D E  H L   IX   IY  SZ H PNC' A' B'C' D'E' H'L'  SP
01000010  00 0000 004F 0000 0000 0000 00000000  00 0000 0000 0000 9FEE
FFBB  7D             LD    A,L
SZ H PNC  A  B C  D E  H L   IX   IY  SZ H PNC' A' B'C' D'E' H'L'  SP
01000010  00 0000 004F 0000 0000 0000 00000000  00 0000 0000 0000 9FEE
FFBC  93             SUB   E
SZ H PNC  A  B C  D E  H L   IX   IY  SZ H PNC' A' B'C' D'E' H'L'  SP
10110011  B1 0000 004F 0000 0000 0000 00000000  00 0000 0000 0000 9FEE
FFBD  C9             RET
SZ H PNC  A  B C  D E  H L   IX   IY  SZ H PNC' A' B'C' D'E' H'L'  SP
10110011  B1 0000 004F 0000 0000 0000 00000000  00 0000 0000 0000 9FF0
D238  D2 AB CE       JP    NC,&CEAB
```

Zunächst wird DE mit &004F geladen. Beachten Sie die Änderung in der Anzeige unter DE. Dann erfolgt der Aufruf der Test HL-DE Routine. Nach diesem Befehl hat sich SP erniedrigt, da die Rücksprungadresse auf den Stapel gelegt wurde. Die Routine führt die Subtraktion HL-DE durch. Wie zu erwarten war, ist beim Rücksprung das Carry Flag gesetzt (=1), um anzuzeigen, das DE größer ist als HL.

Probieren Sie bitte dieselbe Simulation ab der Anfangsadresse aus. Laden Sie jedoch HL und DE zuvor mit verschiedenen Werten (&H101, &H1000, &H80) und beobachten Sie die Wirkung auf die Flags. Der RET bzw. RET NZ Befehl beendet schließlich die Routine. Die Rücksprungadresse wird vom Stapel geholt, und die Programmausführung wird mit dem, auf den RST &H20 folgenden Befehl, fortgesetzt.

Sind Ihnen Programme oder Befehle unklar, so benutzen Sie den Simulator, um ihre Wirkungsweise zu testen.

Programmbeschreibung

Vor der systematischen Erklärung des Programmes, wollen wir noch einiges über die Funktionsweise des Simulators sagen.

Das Kernstück des Simulators besteht aus einem Maschinenprogramm, das am Ende des Programms geladen wird. Sozusagen in die Mitte dieses Programms wird der auszuführende Befehl gepoked. Zuvor werden die Register mit den jeweiligen Werten geladen. Nach der Ausführung des Befehls, werden die evtl. geänderten Registerinhalte zur Übergabe ins BASIC wieder in den Speicher geschrieben. Der Simulator ruft dieses Programm durch >CALL &9700< in Zeile 1140 auf. Zum besseren Verständnis ist hier das Assemblerlisting des Maschinenprogramms aufgeführt.

Überlegen Sie sich anhand des Listings die Funktionsweise des Programms. Ein wenig "Stapelakrobatik" ist dazu schon notwendig. Wollen Sie es simulieren, so schalten Sie die Echtsimulation

von den SP beeinflussenden Befehlen aus, und führen Sie die nötige Änderung mit "A" durch. Um das Programm möglichst kurz zu halten, werden die Registerinhalte nicht mit LD Befehlen in den Speicher geschrieben, sondern über PUSH und POP Befehle gelesen und geschrieben. Dazu wird zuvor des SP auf den Anfang, des für die Übergabe reservierten Bereiches (SPUELO-SP Übergabe Low Adresse) gesetzt.

Mit Hilfe dieser Methode können alle Befehle bis auf die Sprungbefehle, ausgeführt werden. Bei einem Sprungbefehl würde eine Verzweigung aus dem Maschinenprogramm hinaus stattfinden und der Rücksprung ins BASIC wäre unmöglich.

Aus diesem Grund werden die Sprungbefehle sämtlich im BASIC behandelt. Die notwendigen Manipulationen am Stapel und am PC werden vom BASIC aus vorgenommen.

Da der Zweitregistersatz des Z80-A beim Schneider für Systemroutinen reserviert ist, wird außerdem der wirkliche Zweitregistersatz vor dem eigentlichen Programm "gepushed" und am Ende mit "POP" wieder eingelesen.

Die Veränderung des wirklichen Zweitregistersatzes in selbstgeschriebenen Programmen kann einen Systemabsturz zur Folge haben. Das folgende Listing ist mit dem CPC 6128 entstanden. Änderungen für den 464 und den 664 sind im Programm angegeben.

```
A000              10                ; Simulator
9F70              20          ORG   &9f70
9F70 F3           30          DI    ; Interrupt ausschalten, da
9F71 D9           40          EXX
; Zweitregistersatz benutzt wird
9F72 08           50          EX    af,af'
9F73 F5           60          PUSH af ; Zweitregistersatz
9F74 C5           70          PUSH bc ; vor Benutzung
9F75 D5           80          PUSH de ; auf Stapel retten
9F76 E5           90          PUSH hl
9F77 08          100          EX    af,af'
9F78 D9          110          EXX
9F79 ED730000    120          LD    (spreal),sp
; echten SP retten
9F7D 310000      130          LD    sp,spuelo
; SP fuer Registerubergabe
9F80 F1          140          POP   af
9F81 C1          150          POP   bc ; Register lesen
9F82 D1          160          POP   de
9F83 E1          170          POP   hl
9F84 D9          180          EXX
9F85 08          190          EX    af,af'
9F86 DDE1        200          POP   ix
9F88 FDE1        210          POP   iy
9F8A F1          220          POP   af
9F8B C1          230          POP   bc
9F8C D1          240          POP   de
9F8D E1          250          POP   hl
9F8E ED7B0000    260          LD    sp,(spsimu)
```

```
; Simulations SP laden
9F92 0000     270  BEFEHL DW   0 ; Platz fuer Befehl
9F94 0000     280         DW   0 ; reservieren
9F96 ED730000 290         LD   (spsimu),sp
; Simulations SP speichern
9F9A 310000   300         LD   sp,spsimu
; fuer Registeruebergabe
9F9D E5       310         PUSH hl
9F9E D5       320         PUSH de
; Register zurueckschreiben
9F9F C5       330         PUSH bc
9FA0 F5       340         PUSH af
9FA1 FDE5     350         PUSH iy
9FA3 DDE5     360         PUSH ix
9FA5 08       370         EX   af,af'
9FA6 D9       380         EXX
9FA7 E5       390         PUSH hl
9FA8 D5       400         PUSH de
9FA9 C5       410         PUSH bc
9FAA F5       420         PUSH af
9FAB ED7B0000 430         LD   sp,(spreal)
; wirklichen SP wiederholen
9FAF D9       440         EXX
9FB0 08       450         EX   af,af' ; Zweitregistersatz
9FB1 E1       460         POP  hl ; mit alten Werten
9FB2 D1       470         POP  de ; laden
9FB3 C1       480         POP  bc
9FB4 F1       490         POP  af
9FB5 08       500         EX   af,af'
9FB6 D9       510         EXX
9FB7 FB       520         EI    ; danach Interrupt freigeben
9FB8 C9       530         RET
```

```
**** Zeile  120  : SPREAL=&9FB9
**** Zeile  430  : SPREAL=&9FB9
9FB9             540  SPREAL DS   2
**** Zeile  130  : SPUELO=&9FBB
9FBB             550  SPUELO DS   20
; Platz fuer Registeruebergabe
**** Zeile  260  : SPSIMU=&9FCF
**** Zeile  290  : SPSIMU=&9FCF
**** Zeile  300  : SPSIMU=&9FCF
9FCF 0000        560  SPSIMU DW   stapel
; Platz fuer Simulations SP
9FD1             570         DS   4
; Platz fuer Stapelueberlauf
**** Zeile  560  : STAPEL=&9FD5
9FD5             580  STAPEL DS   50
; Platz fuer Simulationsstapel
Programm :simula
Start : &9F70    Ende : &A006
Laenge : 0097
 0  Fehler
Variablentabelle :
BEFEHL 9F92  SPREAL 9FB9  SPUELO 9FBB  SPSIMU 9FCF
STAPEL 9FD5
```

```
10 ' Z80 - Disassembler, Simulator und Sourceerzeuger
20 ' von H.D.  02/10/1985
30 MEMORY &9EFF
40 MODE 2
50 GOSUB 2760:REM Init
60 REM Eingabe
70 PRINT pri$:LOCATE #3,1,VPOS(#0):PRINT#3,"*";
80 CLS#1:PRINT#1,"Eingabemodus"
90 a$=INKEY$:IF a$="" THEN 90
100 IF a$=CHR$(13) THEN 120
110 FOR me=0 TO bean:IF(ASC(a$)OR &20)<>(ASC(menu$(me))OR &2
0) THEN NEXT:GOTO 70
120 CLS#1:PRINT#1,menu$(me);:CLS#3
130 PRINT LEFT$(menu$(me),1);" ";
140 ON me+1 GOSUB 380,1600,1510,510,630,580,1260,210,250,190
150 IF NOT zflag THEN zflag=0 ELSE zflag=1
160 GOTO 70
170 ' ****************************************
180 REM end
190 MODE 2:END
200 ' **************************************
210 REM Drucker an/aus
220 IF aus=0 THEN PEN #2,0:PAPER #2,1:aus=8:PRINT#2,"an "; E
LSE PEN #2,1:PAPER #2,0:aus=0:PRINT#2,"aus";
230 GOSUB 460:RETURN
240 ' **************************************
250 REM rom/ram Status
260 GOSUB 460:PEN #4,0:PAPER #4,1
270 LOCATE #4,1,1:PRINT #4,romstat$(lostat);
280 a$=INKEY$:IF a$="" THEN 280
290 IF a$=CHR$(13) THEN PEN #4,1:PAPER #4,0:LOCATE #4,1,1:PR
INT #4,romstat$(lostat); ELSE lostat=1-lostat:GOTO 270
300 PEN #5,0:PAPER #5,1:IF histat=2 THEN histat=1
310 LOCATE #5,1,1:PRINT #5,romstat$(histat);
```

```
320 a$=INKEY$:IF a$="" THEN 320
330 IF a$=CHR$(13) THEN PEN #5,1:PAPER #5,0:LOCATE #4,1,1:PR
INT #5,romstat$(histat);:GOTO 350
340 IF a$<>CHR$(5) THEN histat=1-histat:GOTO 310 ELSE PEN #5
,1:PAPER #5,0:CLS #5:INPUT #5,;"",exrom:histat=2
350 IF histat=2 THEN status=exrom ELSE status=252+lostat+his
tat*2
360 RETURN
370 ' ***********************************
380 REM Help
390 CLS
400 LOCATE 10,2:PRINT"Z 8 0  -  D I S A S S E M B L E R  +
S I M U L A T O R"
410 LOCATE 1,5:FOR i=0 TO bean:LOCATE 15,VPOS(#0)
420 PRINT LEFT$(menu$(i),1);" / ";CHR$(ASC(menu$(i)) OR &20)
;" - ";menu$(i)
430 NEXT
440 RETURN
450 ' ++++++++++++++
460 REM sub zflag
470 LOCATE 1,VPOS(#0):PRINT " ";CHR$(8);
480 IF zflag=1 THEN zflag=-1
490 RETURN
500 ' *********************************
510 REM Simulator
520 GOSUB 1790:IF noflag THEN pc=pc ELSE pc=a
530 zflag=-1:PRINT:GOTO 1140
540 FOR i=adrbef TO adrbef+3:POKE i,0:NEXT:qq=0
550 GOSUB 1880:REM disassemblieren
560 RETURN
570 ' *********************************
580 REM echtsimulator ein/aus
590 echtsim=-1-echtsim
600 IF echtsim THEN PEN#6,0:PAPER #6,1:PRINT#6,"an "; ELSE P
```

```
EN #6,1:PAPER #6,0:PRINT#6,"aus";
610 GOSUB 460:RETURN
620 ' ***********************************
630 REM Leertaste = Simulieren
640 IF ABS(zflag)<>1 THEN PRINT"Zuerst den Simulator mit Z a
ufrufen":RETURN
650 GOSUB 460
660 IF NOT echtsim THEN 1140
670 IF po=0 THEN bef$=pr$:GOTO 690
680 bef$=LEFT$(LEFT$(pr$,po-1)+"    ",4)
690 REM Sprungbefehle gesondert behandeln
700 IF bef$="JP  " OR bef$="JR  " THEN GOSUB 970:GOTO 1140
710 IF bef$<>"RST " THEN 730
720 spradr=pc-dwflag:pc=VAL(MID$(pr$,5,4)):GOTO 790
730 IF bef$<>"DJNZ" THEN 770
740 b=PEEK(adrreg+15):b=b-1-256*(b=0):POKE adrreg+15,b
750 IF b<>0 THEN GOSUB 1050
760 GOTO 1140
770 IF bef$<>"CALL" THEN 860
780 spradr=pc:GOSUB 970
790 REM ruecksprungadrsse auf Stapel legen
800 sp=PEEK(stapel)+256*PEEK(stapel+1)
810 sp=sp-1:POKE sp,INT(spradr/256)-256*(spradr<0)
820 sp=sp-1:POKE sp,spradr-256*INT(spradr/256)
830 POKE stapel,sp-INT(sp/256)*256
840 POKE stapel+1,INT(sp/256)-256*(sp<0)
850 GOTO 1140
860 IF LEFT$(bef$,3)<>"RET" THEN 1130
870 GOSUB 1060
880 IF bedflg=0 THEN 1140
890 REM Ruecksprungadresse vom Stapel holen
900 sp=PEEK(stapel)+256*PEEK(stapel+1)
910 pc=PEEK(sp+1)*256+PEEK(sp)
920 sp=sp+2
```

```
930 POKE stapel,sp-INT(sp/256)*256
940 POKE stapel+1,INT(sp/256)-256*(sp<0)
950 GOTO 1140
960   +++++++++++++++
970 REM sub pc setzen bei Spruengen
980 GOSUB 1060
990 IF bedflg THEN 1050
1000 IF bedflg=0 THEN RETURN
1010 a$=MID$(pr$,po+2,2)
1020 IF a$="HL" THEN pc=PEEK(adrreg+18)+256*PEEK(adrreg+19):
RETURN
1030 IF a$="IX" THEN pc=PEEK(adrreg+8)+256*PEEK(adrreg+9):RE
TURN
1040 IF a$="IY" THEN pc=PEEK(adrreg+10)+256*PEEK(adrreg+11):
RETURN
1050 pc=VAL(RIGHT$(pr$,5)):RETURN
1060 REM sub test bedingung
1070 a$=MID$(pr$,po+1,2)
1080 FOR i=0 TO 7:IF a$<>cond$(i) THEN NEXT
1090 IF i=8 THEN bedflg=1:RETURN
1100 bit=((i\2)*2-(i>3)-((i=4)OR(i=5)))
1110 IF (PEEK(adrreg+12)AND 2^bit)/2^bit=-(i MOD 2 =1) THEN
bedflg=-1 ELSE bedflg=0
1120 RETURN
1130 IF echtsim THEN CALL &9F70 : REM Befehl ausfuehren
1140 REM Register ausgeben
1150 PRINT#aus,"SZ H PNC  A  B C  D E  H L   IX   IY  SZ H P
NC' A' B'C' D'E' H'L'  SP "
1160 FOR q=1 TO 0 STEP -1
1170 PRINT#aus,BIN$(PEEK(adrreg+q*12),8);"  ";HEX$(PEEK(adrr
eg+1+q*12),2);" ";
1180 FOR k=2 TO 6 STEP 2:pore=q*12+k:GOSUB 1220:NEXT
1190 IF q=1 THEN pore=8:GOSUB 1220:pore=10:GOSUB 1220 ELSE p
ore=20:GOSUB 1220
```

```
1200 NEXT:PRINT#aus
1210 GOTO 540
1220 REM Ausgabe 16 Bit Register
1230 PRINT#aus,HEX$(256*PEEK(adrreg+pore+1)+PEEK(adrreg+pore
),4);" ";
1240 RETURN
1250 ' ************************************
1260 REM Register aendern
1270 IF ABS(zflag)<>1 THEN PRINT"Zuerst den Simulator mit Z
aufrufen":RETURN
1280 GOSUB 460
1290 LOCATE 1,VPOS(#0)-2
1300 LINE INPUT "",z$
1310 q=12:GOSUB 1400 : REM erster Registersatz
1320 q=0.1:po=29:GOSUB 1450 : REM IX
1330 po=34:GOSUB 1450 : REM IY
1340 q=0:GOSUB 1400 : REM zweiter Registersatz
1350 q=-4:po=67:GOSUB 1450
1360 LINE INPUT "",z$
1370 pc=VAL("&"+LEFT$(z$,4))
1380 RETURN
1390 ' ++++++++++++++
1400 REM sub geaenderte Register an Mapro uebergeben
1410 POKE adrreg+1+q,VAL("&"+MID$(z$,11-(q=0)*38,2))
1420 POKE adrreg+q,VAL("&X"+MID$(z$,1-(q=0)*38,8))
1430 FOR po=14 TO 24 STEP 5:GOSUB 1450:NEXT
1440 RETURN
1450 REM sub Registerwerte bestimmen
1460 wert=VAL("&"+MID$(z$,po-(q=0)*38,4))
1470 POKE adrreg+q+(po\5)*2-1,INT(wert/256)-256*(wert<0)
1480 POKE adrreg+q+(po\5)*2-2,wert-INT(wert/256)*256
1490 RETURN
1500 ' ************************************
```

```
1510 REM Z80 Sourcerzeuger
1520 souflag=-1
1530 PRINT
1540 PRINT"Labelnamen definieren:"
1550 FOR i=0 TO 50:INPUT la$(i),wa$(i):IF la$(i)="" THEN PRI
NT:GOTO 1580
1560 z$=STR$(zeino)+CHR$(32)+CHR$(39)+CHR$(32)+la$(i)+" EQU
"+wa$(i)+CHR$(0)+CHR$(0)
1570 CALL mst,@z$:zeino=zeino+10:NEXT
1580 maxlab=i-1
1590 ' ***********************************
1600 REM Disassembler
1610 IF me= 1 THEN souflag=0:maxlab=0
1620 GOSUB 1790:IF noflag THEN pc=pc ELSE pc=a
1630 GOSUB 1790:ende=a:IF noflag THEN ende=pc+&18
1640 PRINT
1650 IF NOT souflag THEN 1720
1660 z$=STR$(zeino)+CHR$(32)+CHR$(39)+CHR$(32)+";"+CHR$(0)+C
HR$(0)
1670 CALL mst,@z$
1680 zeino=zeino+10
1690 z$=""
1700 z$=STR$(zeino)+CHR$(32)+CHR$(39)+CHR$(32)
1710 FOR i=0 TO maxlab:IF pc=VAL(wa$(i)) THEN z$=z$+la$(i)+"
 " ELSE NEXT
1720 GOSUB 1880
1730 IF NOT souflag THEN 1760
1740 z$=z$+pr$+CHR$(0)+CHR$(0)
1750 CALL mst,@z$
1760 IF pc>ende THEN RETURN
1770 IF souflag THEN 1680 ELSE 1720
1780 ' +++++++++++++++
1790 REM sub Input Hexadresse
1800 x=POS(#0)+1:y=VPOS(#0)
```

```
1810 LOCATE x,y:INPUT"&",a$
1820 IF a$="" THEN noflag=-1:GOTO 1850
1830 noflag=0
1840 a=VAL("&"+a$)
1850 LOCATE x+5,y
1860 RETURN
1870 '·+++++++++++++++++++++++++++++++++++++++++
1880 REM sub disassi
1890 adr=pc
1900 PRINT#aus,HEX$(adr,4);"  ";
1910 iflag=0
1920 GOSUB 2690
1930 GOSUB 2040
1940 IF iflag THEN 2350
1950 IF w=&CF OR w=&D7 OR w=&DF OR w=&EF THEN pr$=pr$+" /DW:
nn":dwflag=2 ELSE dwflag=0
1960 IF INSTR(pr$,"n")<>0 THEN 2460
1970 IF INSTR(pr$,"e")<>0 THEN 2600
1980 po=INSTR(pr$," ")
1990 IF PR$="" THEN PR$="???"
2000 IF po=0 THEN PRINT#aus,TAB(21);pr$;:GOTO 2020
2010 PRINT#aus,TAB(21);LEFT$(pr$,po-1);TAB(27);RIGHT$(pr$,LE
N(pr$)-po);
2020 PRINT#aus
2030 RETURN
2040 REM Interpretieren
2050 IF (w=&DD OR w=&FD) AND NOT iflag THEN 2240
2060 IF w=&ED THEN 2210
2070 IF w=&CB THEN 2150
2080 GOSUB 2290
2090 ON col GOTO 2110,2130,2100
```

```
2100 pr$=bef$(w):RETURN
2110 IF w=&76 THEN pr$="HALT":RETURN
2120 pr$="LD "+regtab$(co2)+","+reg$:RETURN
2130 IF co2=0 OR co2=1 OR co2=3 THEN a$=" A," ELSE a$=" "
2140 pr$=arilog$(co2)+a$+reg$:RETURN
2150 REM cb
2160 GOSUB 2690
2170 IF iflag THEN dis=w:GOSUB 2690
2180 GOSUB 2290
2190 IF co1=0 THEN pr$=rotschi$(co2)+" "+reg$ ELSE pr$=bitti
$(co1)+STR$(co2)+","+reg$
2200 RETURN
2210 REM ed
2220 GOSUB 2690
2230 IF w<&40 OR w>&BF THEN pr$="???":RETURN ELSE GOTO 2100
2240 REM xy
2250 iflag=-1
2260 IF w=&DD THEN i$="IX" ELSE i$="IY"
2270 GOSUB 2690
2280 GOTO 2040
2290 REM code zerlegen
2300 co1=(w AND &X11000000)/64
2310 co2=(w AND &X111000)/8
2320 co3=w AND &X111
2330 reg$=regtab$(co3)
2340 RETURN
2350 REM indiziert
2360 po=INSTR(pr$,"HL")
2370 IF po=0 THEN pr$="???":GOTO 1980
2380 IF INSTR(pr$,"(HL)")<>0 THEN 2420
2390 IF pr$="EX DE,HL" THEN pr$="???":GOTO 1980
2400 IF pr$="ADD HL,HL" THEN pr$="ADD "+i$+","+i$:GOTO 1980
2410 pr$=LEFT$(pr$,po-1)+i$+RIGHT$(pr$,LEN(pr$)-po-1):GOTO 1
950
```

```
2420 IF LEFT$(pr$,2)="JP" THEN 2410
2430 IF pc-adr<3 THEN GOSUB 2690:dis=w
2440 IF dis>127 THEN dis$=STR$(dis-256) ELSE dis$="+"+RIGHT$
(STR$(dis),LEN(STR$(dis))-1)
2450 i$=i$+dis$:GOTO 2410
2460 REM n ersetzen
2470 po=INSTR(pr$,"nn")
2480 IF po<>0 THEN 2530
2490 po=INSTR(pr$,"n")
2500 GOSUB 2690
2510 pr$=LEFT$(pr$,po-1)+"&"+HEX$(w,2)+RIGHT$(pr$,LEN(pr$)-p
o)
2520 GOTO 1980
2530 GOSUB 2690:lb=w
2540 GOSUB 2690
2550 wert=w*256+lb
2560 FOR i=0 TO maxlab:IF UNT(wert)=VAL(wa$(i)) THEN pr$=LEF
T$(pr$,po-1)+la$(i)+RIGHT$(pr$,LEN(pr$)-po-1):GOTO 1980
2570 NEXT
2580 pr$=LEFT$(pr$,po-1)+"&"+HEX$(wert,4)+RIGHT$(pr$,LEN(pr$
)-po-1)
2590 GOTO 1980
2600 REM e ersetzen
2610 po=INSTR(pr$,"e")
2620 GOSUB 2690
2630 IF w>127 THEN w=w-256:REM 2er-Komp.
2640 w=w+2
2650 a$="$"+STR$(w)+"  >"+"&"+HEX$(pc+w-2,4)
2660 FOR i=0 TO maxlab:IF UNT(pc+w-2)=VAL(wa$(i)) THEN a$=la
$(i) ELSE NEXT
2670 pr$=LEFT$(pr$,po-1)+a$+RIGHT$(pr$,LEN(pr$)-po)
2680 GOTO 1980
2690 REM Byte lesen
2700 |RPEEK,@w%,pc,status:w=w%
```

```
2710 pc=pc+1
2720 IF ABS(zflag)=1 THEN POKE adrbef+qq,w:qq=qq+1
2730 PRINT#aus,HEX$(w,2);" ";
2740 RETURN
2750 ' ************************************
2760 REM init
2770 romstat$(1)="RAM":romstat$(0)="ROM"
2780 status=255:w%=0
2790 adrreg=&9FBB:adrbef=&9F92:stapel=&9FCF:' s2=PRINT PRINT
 PRINT
2800 POKE stapel,&F0:POKE stapel+1,&9F
2810 DIM la$(50),wa$(50)
2820 zeino=10000
2830 LOCATE 1,1:PRINT"lo : RAM   hi : RAM";:lostat=1:histat=
1
2840 LOCATE 68,1:PRINT"Printer aus";
2850 LOCATE 61,2:PRINT"Echtsimulation aus";
2860 WINDOW #0,3,80,3,22
2870 WINDOW #1,30,66,1,1
2880 WINDOW #2,76,78,1,1
2890 WINDOW #3,1,2,3,22
2900 WINDOW #4,6,9,1,1:WINDOW #5,17,20,1,1
2910 WINDOW #6,76,78,2,2
2920 pri$=CHR$(10)+CHR$(11)+CHR$(11)
2930 aus=0
2940 READ bean:FOR i=0 TO bean:READ menu$(i):NEXT
2950 GOSUB 380
2960 DIM cond$(7),regtab$(7),rotschi$(8),bitti$(3),arilog$(7
),bef$(255)
2970 FOR i=0 TO 7:READ cond$(i):NEXT
2980 FOR i=0 TO 7:READ regtab$(i):NEXT
2990 FOR i=0 TO 7:READ rotschi$(i):NEXT
3000 FOR i=1 TO 3:READ bitti$(i):NEXT
```

```
3010 FOR i=0 TO 7:READ arilog$(i):NEXT
3020 FOR i=0 TO &7F:READ bef$(i):NEXT
3030 FOR i=&80 TO &9F:bef$(i)="":NEXT
3040 FOR i=&A0 TO &FF:READ bef$(i):NEXT
3050 mst=&9F00
3060 FOR i=mst TO mst+&19:READ a$:w=VAL("&"+a$)
3070 s=s+w:POKE i,w:NEXT
3080 IF s<> 4273 THEN PRINT"Fehler in Datas 1":END ELSE s=0
3081 ' 464: if s<> 4121 then ....
3082 ' 664: if s<> 4288 then ....
3090 FOR i=&9F20 TO &9F6C
3100 READ a$:w=VAL("&H"+a$)
3110 s=s+w:POKE i,w:NEXT
3120 IF s<> 9278 THEN PRINT"Fehler in Datas 2":END ELSE s=0
3121 ' 464: if s<> 9405 then ....
3122 ' 664: if s<> 9341 then ....
3130 CALL &9F20:REM RSX einbinden
3140 FOR i=&9F70 TO &9FBB
3150 READ a$:w=VAL("&H"+a$)
3160 s=s+w:POKE i,w:NEXT
3170 IF s<> 12811 THEN PRINT"Fehler in Datas 3":END
3180 RETURN
3190 REM DATAS ++++++++++++++++++++
3200 DATA 9
3210 DATA Help,Disassembler,Sourcerzeuger,Z80-Simulator
3220 DATA " (Leertaste)=Befehl simulieren",Echtsimulation,Ae
ndern der Register
3230 DATA Printer an/aus,ROM/RAM Status,Bis bald !!
3240 DATA NC,"C,",PO,PE,NZ,"Z,","P,","M,"
3250 DATA B,C,D,E,H,L,(HL),A
3260 DATA RLC,RRC,RL,RR,SLA,SRA,???,SRL
3270 DATA BIT,RES,SET
3280 DATA ADD,ADC,SUB,SBC,AND,XOR,OR,CP
3290 DATA NOP,"LD BC,nn","LD (BC),A",INC BC,INC B,DEC B,"LD
B,n",RLCA
```

```
3300 DATA "EX AF,AF'","ADD HL,BC","LD A,(BC)",DEC BC,INC C,D
EC C,"LD C,n",RRCA
3310 DATA DJNZ e,"LD DE,nn","LD (DE),A",INC DE,INC D,DEC D,"
LD D,n",RLA
3320 DATA JR e,"ADD HL,DE","LD A,(DE)",DEC DE,INC E,DEC E,"L
D E,n",RRA
3330 DATA "JR NZ,e","LD HL,nn","LD (nn),HL",INC HL,INC H,DEC
 H,"LD H,n",DAA
3340 DATA "JR Z,e","ADD HL,HL","LD HL,(nn)",DEC HL,INC H,DEC
 H,"LD L,n",CPL
3350 DATA "JR NC,e","LD SP,nn","LD (nn),A",INC SP,INC (HL),D
EC (HL),"LD (HL),n",SCF
3360 DATA "JR C,e","ADD HL,SP","LD A,(nn)",DEC SP,INC A,DEC
A,"LD A,n",CCF
3370 DATA "IN B,(C)","OUT (C),B","SBC HL,BC","LD (nn),BC",NE
G,RETN,IM 0,"LD I,A"
3380 DATA "IN C,(C)","OUT (C),C","ADC HL,BC","LD BC,(nn)",,R
ETI,,"LD R,A"
3390 DATA "IN D,(C)","OUT (C),D","SBC HL,DE","LD (nn),DE",,,
IM 1,"LD A,I"
3400 DATA "IN E,(C)","OUT (C),E","ADC HL,DE","LD DE,(nn)",,,
IM 2,"LD A,R"
3410 DATA "IN H,(C)","OUT (C),H","SBC HL,HL","LD (nn),HL",,,
,RRD
3420 DATA "IN L,(C)","OUT (C),L","ADC HL,HL","LD HL,(nn)",,,
,RLD
3430 DATA ,,"SBC HL,SP","LD (nn),SP",,,,
3440 DATA "IN A,(C)","OUT (C),A","ADC HL,SP","LD SP,(nn)",,,
,
3450 DATA LDI,CPI,INI,OUTI,,,,,LDD,CPD,IND,OUTD,,,,
3460 DATA LDIR,CPIR,INIR,OTIR,,,,,LDDR,CPDR,INDR,OTDR,,,,
3470 DATA RET NZ,POP BC,"JP NZ,nn",JP nn,"CALL NZ,nn",PUSH B
C,"ADD A,n",RST &00
```

```
3480 DATA RET Z,RET,"JP Z,nn",->,"CALL Z,nn",CALL nn,"ADC A,
n",RST &08
3490 DATA RET NC,POP DE,"JP NC,nn","OUT (n),A","CALL NC,nn",
PUSH DE,"SUB n",RST &10
3500 DATA RET C,EXX,"JP C,nn","IN A,(n)","CALL C,nn",->,"SBC
 A;n",RST &18
3510 DATA RET PO,POP HL,"JP PO,nn","EX (SP),HL","CALL PO,nn"
,PUSH HL,"AND n",RST &20
3520 DATA RET PE,JP (HL),"JP PE,nn","EX DE,HL","CALL PE,nn",
->,"XOR n",RST &28
3530 DATA RET P,POP AF,"JP P,nn",DI,"CALL P,nn",PUSH AF,"OR
n",RST &30
3540 DATA RET M,"LD SP,HL","JP M,nn",EI,"CALL M,nn",->,"CP n
",RST &38
3550 DATA DF,04,9f,C9,07,9f,FD,EB
3560 DATA 23,5E,23,56,EB,CD,4d,de
3561 '464:..,..,..,..,..,..,61,dd
3562 '664:..,..,..,..,..,..,52,de
3570 DATA B7,C8,CD,cf,EE,D0,CD,a5
3571 '464:..,..,..,04,ee,..,..,c6
3572 '664:..,..,..,d4,ee,..,..,aa
3580 DATA E7,C9
3581 '464:e6,..
3582 '664:e7,.
3590 DATA 01,2F,9F,21,3A,9F,CD,D1
3600 DATA BC,3E,C9,32,20,9F,C9,34
3610 DATA 9F,C3,3E,9F,52,50,45,45
3620 DATA CB,00,00,00,00,00,DF,42
3630 DATA 9F,C9,45,9F,FD,FE,03,C2
3640 DATA 55,D0,7A,FE,00,C2,1D,C2
3641 '464:ed,cf,..,..,..,..,1d,c2
3642 '664:58,d0,..,..,..,..,50,cb
3650 DATA 7B,32,6A,9F,DD,6E,02,DD
3660 DATA 66,03,DF,68,9F,DD,6E,04
3670 DATA DD,66,05,77,97,23,77,C9
3680 DATA 6B,9F,FD,7E,C9
```

```
3690 DATA F3,D9,08,F5,C5,D5,E5,08
3700 DATA D9,ED,73,B9,9F,31,BB,9F
3710 DATA F1,C1,D1,E1,D9,08,DD,E1
3720 DATA FD,E1,F1,C1,D1,E1,ED,7B
3730 DATA CF,9F,00,00,00,00,ED,73
3740 DATA CF,9F,31,CF,9F,E5,D5,C5
3750 DATA F5,FD,E5,DD,E5,08,D9,E5
3760 DATA D5,C5,F5,ED,7B,B9,9F,D9
3770 DATA 08,E1,D1,C1,F1,08,D9,FB
3780 DATA C9
ARILOG $ 2140 2960 3010
ADR ! 1890 1900 2430
ADRREG ! 740 740 1020 1020 1030 1030 1040 1040 1110 1170 117
0 1230 1230 1410 1420 1470 1480 2790
ADRBEF ! 540 540 2720 2790
A ! 520 1620 1630 1840
AUS ! 220 220 220 1150 1170 1200 1230 1900 2000 2010 2020 27
30 2930
A $ 90 90 100 110 280 280 290 320 320 330 340 1010 1020 1030
 1040 1070 1080 1810 1820 1840 2130 2130 2140 2650 2660 2670
 3060 3060 3100 3100 3150 3150
BITTI $ 2190 2960 3000
BIT ! 1100 1110 1110
BEDFLG ! 880 990 1000 1090 1110 1110
B ! 740 740 740 740 740 750
BEF $ 670 680 700 700 710 730 770 860 2100 2960 3020 3030 30
40
BEAN ! 110 410 2940 2940
CO3 ! 2320 2330
CO2 ! 2120 2130 2130 2130 2140 2190 2190 2310
CO1 ! 2090 2190 2190 2300
COND $ 1080 2960 2970
DIS $ 2440 2440 2450
DIS ! 2170 2430 2440 2440 2440 2440
```

```
DWFLAG ! 720 1950 1950
ENDE ! 1630 1630 1760
ECHTSIM ! 590 590 600 660 1130
EXROM ! 340 350
HISTAT ! 300 300 310 330 340 340 340 350 350 2830
I $ 2260 2260 2400 2400 2410 2450 2450
IFLAG ! 1910 1940 2050 2170 2250
I ! 410 420 420 420 540 540 1080 1080 1090 1100 1100 1100 11
00 1110 1550 1550 1550 1550 1560 1560 1580 1710 1710 1710 25
60 2560 2560 2660 2660 2660 2940 2940 2970 2970 2980 2980 29
90 2990 3000 3000 3010 3010 3020 3020 3030 3030 3040 3040 30
60 3070 3090 3110 3140 3160
K ! 1180 1180
LB ! 2530 2550
LA $ 1550 1550 1560 1710 2560 2660 2810
LOSTAT ! 270 290 290 290 350 2830
MAXLAB ! 1580 1610 1710 2560 2660
MST ! 1570 1670 1750 3050 3060 3060
MENU $ 110 120 130 420 420 420 2940
ME ! 110 110 120 130 140 1610
NOFLAG ! 520 1620 1630 1820 1830
PORE ! 1180 1190 1190 1190 1230 1230
PR $ 670 680 720 1010 1050 1070 1740 1950 1950 1960 1970 198
0 1990 1990 2000 2010 2010 2010 2100 2110 2120 2140 2190 219
0 2230 2360 2370 2380 2390 2390 2400 2400 2410 2410 2410 241
0 2420 2470 2490 2510 2510 2510 2510 2560 2560 2560 2560 258
0 2580 2580 2580 2610 2670 2670 2670 2670
PO ! 670 680 1010 1070 1320 1330 1350 1430 1460 1470 1480 19
80 2000 2010 2010 2360 2370 2410 2410 2470 2480 2490 2510 25
10 2560 2560 2580 2580 2610 2670 2670
PC ! 520 520 520 720 720 780 910 1020 1030 1040 1050 1370 16
20 1620 1620 1630 1710 1760 1890 2430 2650 2660 2700 2710 27
10
PRI $ 70 2920
```

```
Q ! 1160 1170 1170 1180 1190 1310 1320 1340 1350 1410 1410 1
420 1420 1460 1470 1480
QQ ! 540 2720 2720 2720
ROTSCHI $ 2190 2960 2990
REG $ 2120 2140 2190 2190 2330
REGTAB $ 2120 2330 2960 2980
ROMSTAT $ 270 290 310 330 2770 2770
S ! 3070 3070 3080 3080 3110 3110 3120 3120 3160 3160 3170
SOUFLAG ! 1520 1610 1650 1730 1770
STAPEL ! 800 800 830 840 900 900 930 940 2790 2800 2800
SP ! 800 810 810 810 820 820 820 830 830 840 840 900 910 910
 920 920 930 930 940 940
SPRADR ! 720 780 810 810 820 820
STATUS ! 350 350 2700 2780
W % 2700 2700 2780
W ! 1950 1950 1950 1950 2050 2050 2060 2070 2100 2110 2170 2
230 2230 2260 2300 2310 2320 2430 2510 2530 2550 2630 2630 2
630 2640 2640 2650 2650 2660 2700 2720 2730 3060 3070 3070 3
100 3110 3110 3150 3160 3160
WA $ 1550 1560 1710 2560 2660 2810
WERT ! 1460 1470 1470 1480 1480 2550 2560 2580
X ! 1800 1810 1850
Y ! 1800 1810 1850
ZEINO ! 1560 1570 1570 1660 1680 1680 1700 2820
Z $ 1300 1360 1370 1410 1420 1460 1560 1570 1660 1670 1690 1
700 1710 1710 1740 1740 1750
ZFLAG ! 150 150 150 480 480 530 640 1270 2720
```

Programmbeschreibung

Zeile 10-60:

Initialisierung

Zeile 70-160:

Hauptschleife

Zeile 70:

Wenn Bildschirmende erreicht scrollen (pri$) "*" ausgeben

Zeile 100:

Bei Eingabe von >RETURN< letzte Eingabe wiederholen

Zeile 110:

Anhand der Anfangsbuchstaben der Befehle die Eingabe prüfen

Zeile 120-140:

Erkannten Buchstaben und Befehl ausgeben, zur jeweiligen Routine springen

Zeile 150:

zflag entscheidet darüber, ob mit der Leertaste simuliert werden darf

Zeile 180/190:

Menuepunkt END bzw. "Bis bald!"

Zeile 210-230:

Menuepunkt Printer an/aus

Zeile 230:

Unterprogramm, das Z Flag falls gesetzt wieder -setzt

Zeile 250-360:

Menuepunkt ROM/RAM Status wählen

Zeile 260-290:

Eingabe für low ROM

Zeile 300-360:

Eingabe für high ROM

Zeile 340:

Behandlung der Expansionsromnummerneingabe

Zeile 380-440:

Die Funktionen P,E und R sollen keinen Einfluß auf zflag und auf die Cursorposition haben. Das wird durch dieses Unterprogramm erreicht.

Zeile 510-560:

Menuepunkt Z80 Simulator

Zeile 520:

Anfangsadresse holen

Zeile 530:

Register ausgeben

Zeile 540:

Befehlsübergabebereich löschen

Zeile 550:

Befehl disassemblieren

Zeile 580-610:

Menuepunkt Echtsimulation an/aus

Zeile 630-1130:

Menuepunkt Leertaste=simulieren

Zeile 640:

Nur weitermachen, wenn zflag gesetzt ist

Zeile 650:

zflag Status beibehalten

Zeile 660:

Nur wenn Echtsim an ist, den Befehl ausführen, sonst Register unverändert ausgeben

Zeile 670-680:

Befehlswort nach bef$

Zeile 690-1120:

Sprungbefehle vom BASIC aus simulieren

Zeile 700:

JP/JR Befehl: PC auf angegebene Zieladresse setzen

Zeile 710-720:

RST Befehle: PC auf Zieladresse; Rücksprungadresse auf Stapel

Zeile 730-760:

DJNZ Befehl mit B incrementieren

Zeile 770-780:

CALL Befehl und Routine, die die Rücksprungadresse auf den Stapel legt

Zeile 790-850:

Hier wird die Korrektur von Stapel, SP und PC für Unterprogrammsprünge (CALL und RST) ausgeführt

Zeile 860-880:

RET Befehl bedflg=0 bedeutet RET ohne Bedingung

Zeile 890-950:

Korrektur von Stapel, SP und PC bei Rücksprüngen (RET)

Zeile 970-1050:

Hier wird in Abhängigkeit von FL der PC gesetzt. Die indirekten Sprünge werden in den Zeilen 830-860 behandelt

Zeile 1050:

Setzt PC auf angegebene Adresse

Zeile 1060-1120:

SUB (Unterprogramm) setzt FL auf:
 1 - ,wenn gar keine Bedingung vohanden ist
 0 - ,wenn Bedingung nicht erfüllt ist
 -1 - ,wenn Bedingung erfüllt ist

Zeile 1100:

Berechnet die Bitnummer des jeweiligen Flags im Flagregister

Zeile 1130:

Eigentliche Simulation durch ein Maschinenprogramm, falls Echtsimulation angeschaltet ist und falls kein Sprungbefehl simuliert wird

Zeile 1140-1210:

Formatierte Ausgabe der Registerinhalte

Zeile 1220-1240:

Gibt ein 16-Bit Register aus. Die Stellung der Register ergibt sich aus der Reihenfolge der PUSH bzw. POP Befehle des Maschinenprogramms

Zeile 1260-1380:

Eingaberoutine, wenn Register geändert werden.

Zeile 1400-1440:

Hier werden die Standardregister nach Änderung an die entsprechenden Stellen im Speicher geschrieben.

Zeile 1450-1490:

Nach einer Änderung kann mit dieser Routine ein beliebiges Register neu gesetzt werden.

Zeile 1500-1580:

Menuepunkt Sourceerzeuger

Zeile 1540-1580:

Definiert eventuell eingegebene Labels und erzeugt im Sourcelisting die entsprechenden "EQU Zeilen".

Zeile 1600-1770:

Menupunkt Disassembler (auch Sourceerzeuger)

Zeile 1610:

Ist der Disassembler gewählt, werden souflag und maxlab zurückgesetzt

Zeile 1620:

Eingabe Startadresse (mit >RETURN< Option)

Zeile 1630:

Eingabe Endadresse (mit >RETURN< Option)

Zeile 1650:

übersprint Sourceerzeuger bei Disassemblerfunktion

Zeile 1660:

Erzeugt eine Leerzeile

Zeile 1670:

Ruft Maschinenprogramm auf, das z$ in Zeile umwandelt

Zeile 1680:

Zeiso für Nächstesmal erhöhen

Zeile 1700:

erzeugt Anfang der nächsten Zeile

Zeile 1710:

Prüft, ob eingegebenes Label übereinstimmt

Zeile 1720:

Ruft Disassembler auf

Zeile 1740/1750:

erzeugt eben disassemblierte Zeile

Zeile 1790-1860:

Unterprogramm zur Eingabe von HEX Adressen mit der Erkennung, wenn nur >RETURN< eingegeben wird

Zeile 1880-1900:

Hier wird die aktuelle Adresse ausgegeben

Zeile 1920:

Nächstes Byte lesen und ausgeben

Zeile 1930:

Sprung ins Unterprogramm, daß die Interpretation ausführt

Zeile 1940:

Zur Behandlung indizierter Befehle springen, wenn iflag gesetzt ist (=-1)

Zeile 1950:

Behandlung der RST-Befehle, die das folgende Dataword benutzen

Zeile 1960:

Verzweigung, wenn Befehl Zahlen enthält

Zeile 1970:

Verzweigung, wenn Befehl relative Distanzen enthält

Zeile 1980-2020:

Formatierte Ausgabe

In den Zeilen 2040-2280:

stehen die Unterprogramme zur Interpretation

Zeile 2050:

Verzweigung zur Behandlung Indizierter Befehle

Zeile 2060:

Verzweigung zur Behandlung der Befehle, die mit &ED beginnen

Zeile 2070:

Verzweigung zur Behandlung der Befehle, die mit &CB beginnen

Zeile 2080:

Sprung ins Unterprogramm, das w in co1(Bit 6,7),co2(Bit 5-3) und co3 (Bit 2-0) zerlegt.

Zeile 2090:

Bei co1=0 und co1=3 zu Zeile 360, d.h. Befehle aus Tabelle lesen, sonst Zeile 370 Befehle der Form LD reg,reg- bzw. Zeile 390 8-Bit Arithmetisch Logische Befehle

Zeile 2100:

pr$ aus Tabelle bestimmen

Zeile 2130/2120:

Befehle der Form LD r,r- und HALT

Zeile 2130/2140:

Arilog-Befehle

In den Zeilen 2150-2200:

findet die Behandlung der Befehle, die mit dem Code &CB anfangen statt

Zeile 2160:

nächstes Byte lesen

Zeile 2180:

in co1,co2,co3 zerlegen

Zeile 2190:

pr$ für Rotier-bzw. Schiebebefehle (co1=0) und Bit-Manipulationsbefehle erzeugen

In den Zeilen 2210-2230:

findet die Behandlung der Befehle, die mit dem Code &ED anfangen statt

Zeile 2220:

nächstes Byte lesen

Zeile 2230:

wenn gültiger Code selbigen aus Tabelle (Zeile 2100) ermitteln

In den Zeilen 2240-2280:

findet die erste Behandlung der Indizierten Befehle statt

Zeile 2250:

Flag setzen

Zeile 2260:

in i$ das Register speichern (entweder IX oder IY)

Zeile 2270:

nächstes Byte lesen

Zeile 2280:

Interpretation nochmals beginnen (Zeile 300)

Zeile 2290-2330:

SUB Code zerlegen, w wird in co1 (Bit7,6), co2 (Bit 5-3) und co3 (Bit 2-0) zerlegt. reg$ enthält das zu co3 gehörende Register

Zeile 2350-2450:

Indizierte Befehl "zweite Behandlung". Prüfen, ob indizierter Befehl zulässig; wenn ja, dann HL durch i$ ersetzen. Falls Distanz vorhanden erzeugen Zeilen 2430/2440 die Distanzangabe

Zeile 2460-2590:

Enthält pr$ ein "n", so wird hier eine Zahl für n eingesetzt

Zeile 2490-2520:

1 Byte Zahlen (n)

Zeile 2530-2590:

2 Byte Zahlen (nn) ersetzen. Wenn der Sourceerzeuger läuft,wird geprüft, ob Adresse=Label ist.

Zeile 2600-2680:

Offset (e) ersetzen

Zeile 2630/2640:

Offset berechnen

Zeile 2650-2670:

Offset einsetzen und prüfen, ob Zieladresse=Label ist

Zeile 2690-2740:

SUB nächstes Byte lesen und ausgeben.
Wenn der Simulator ausgeschaltet ist, Befehlscodes ins Maschinenprogramm poken.

Zeile 2750-3180:

Initialisierung:Aufbau der Tabellen

Zeile 3190-3780:

DATA Zeilen

Zeile 2750-3180:

Initialisierung:Aufbau der Tabellen

Zeile 3190-3780:

DATA Zeilen

Variablenliste

A- Rückgabe von der "Hex.-Dez. Umwandlung" Wert von A$ als Hex.-
zahl interpretiert
A$- Übergaben von SUBs verschiedener Aufgaben
ADR- Adresse des ersten Codes des aktuellen Befehls
ADRREG- Registerübergabeadresse
ADRBEF- Befehlsübergabeadresse
AUS- Kanal für Ausgabe
B- Wert B Register
BEAN- Befehl-Anzahl
BEDFLG- Bedingungsflag -1,0,1
BEF$- Befehlswort
BIT- Bitnummer
C01- Bit 7 und 6
C02- Bit 5 bis 3
C03- Bit 2-0
DIS- Distanz bei indizierten Befehlen
DIS$- Distanz String für Ausgabe
ENDE- Letzte Adresse für Disassemblierung
ECHTSIM- Echtsimulationsflag
EXROM- Nummer des Expansions ROM's
HISTAT- Upper ROM Status 0,1 ROM/RAM
IFLAG- gesetzt, wenn indizierte Adressierung, sonst rückgesetzt
I- Zähler
K- Zähler
LB- Zwischenspeicherung des Low Bytes bei 2-Byte Zahlen
LOSTAT- Lower ROM Status
MST- Startadresse Maschinenprogramm, Lineerzeuger
MAXLAB- Anzahl eingegebener Labels
ME- Menuepunktnummer
NOFLAG- "Wenn >RETURN< bei Adresseneingabe" Flag
PC- Programmzeiger zeigt auf die Adressen der aktuellen Position
PO- Position von n, nn, e, HL... in PR$
PORE- Position Register

PRI$- String für Scrolling
PR$- PR$ enthält übersetzten Assemblerbefehl
Q- Zähler
QQ- Zähler für Befehlsübergabe an Simulator
REG$- Register für Befehlsübergabe an Maschinenprogramm
ROMSTAT- Enthält ROM/RAM an/aus
S- Prüfsumme
SOUFLAG- Sourceerzeuger Flag
STAPEL- Adresse von SP
SP- SP
SPRADR- "Ab"sprungadresse
STATUS- ROM/RAM Status für Far Call
W%- Wert von rdeek gelesen
WERT- 2-Byte Wert
X- X-Position Eingabe
Y- Y-Position Eingabe
ZEINO- Zeilennummer für Sourci
Z$- Zeile für Sourci+ Eingabe Zeile Registerändern
ZFLAG- Simulationsflag

Tabellen

regtab$(7) - Register
rotschie$(7) - Rotier- und Schiebebefehle
bitti$(3) - Bit-Manipulationsbefehle
arilog$(7) - Arithmetisch bzw. Logische Befehle
bef$(255)- 0 bis &3F: Befehl, die mit &ED beginnen und als Byte eins die Nummer haben
&40-&BF: Befehle, die mit &ED bginnen und als Byte zwei die Nummer haben
&BF-&FF: Befehle, die als Byte eins die Nummer haben
menu$ - ausgeschriebene Befehlswörter (Bean)
LA$ - Label$
WA$ - zu Labels gehörige Adressen
COND$ - Bedingungen (7)

6.2 Systemroutinen

Einführung:

Die effektive Benutzung von Systemroutinen ist das A und O der Maschinenprogrammierung. Bei den Schneider Computern stehen dem zunächst jedoch einige Probleme entgegen.

Beim Einschalten des Rchners sind die gesamten 64K des Rechners mit RAM belegt, der nur einen kleinen Teil der Systemroutinen enthält.

An den Adressen &0 bis &3FFF liegt, wie erwähnt, das Betriebssystem "unter" dem RAM. Die mittleren Adresse von &4000 bis &BFFF sind grundsätzlich auf RAM geschaltet. Im obersten Adressenbereich von &C000 bis &FFFF überlagern sich der Video RAM, der BASIC ROM, das Diskettenbetriebssystem ROM und evtl. weitere 252 vorhandene Expansions ROM's. Beim CPC 6128 kommt hinzu, daß der gesamte Adressenbereich nocheinmal von den 2ten 64K RAM überlagert wird.

Um all diese ROM's und RAM's "unter einen Hut" zu bekommen, ist ein Verfahren notwendig, daß man als Bank Switching bezeichnet. Das bedeutet, daß je nach Bedarf für den aktuellen Adressbereich der gewünschte Speicher eingeschaltet werden kann. Um das zu erreichen, werden die Restart Befehle verwendet. Diese Befehle sprechen die Adressen &0 bis &38 an. An diesen Stellen ist der ROM und RAM Inhalt gleich. Sie können das leicht mit dem Disassembler nachprüfen. Damit ist gewährleistet, daß die Restart Befehle unabhängig von der aktuellen Speicherkonfiguration immer in gleicher Weise benutzt werden können.

Die Restart Befehle verzweigen fast alle in den Adressbereich von &AB80 bis &BFFF. Dort ist, wie schon erwähnt, grundsätzlich RAM selektiert, d.h. daß die dort angesprungenen Routinen immer vorhanden sind. Diese Routinen erledigen dann, je nach Bedarf das Hin- und Herschalten zwischen den verschiedenen ROM's bzw. RAM's (beim 6128).

In dem erwähnten RAM Bereich befinden sich außerdem Sprungvektoren zu allen wichtigen Routinen im Betriebssystem und zu den Arithmetikroutinen. Über diese Sprungvektoren läuft jegliche Kommunikation zwischen BASIC und Betriebssystem ab.

Ein Sprungvektor ist eine sehr praktischen Einrichtung. Anstatt direkt zu einer bestimmten Routine zu springen, wird zu einer vorher festgelegten (!) Adresse gesprungen, an der dann der Sprungbefehl auf die eigentliche Routinen steht. Dieses Verfahren bringt mehrere Vorteile mit sich.

1. Es ist gerade beim Schneider, wo inzwischen drei verschiedene ROM Versionen existieren, für den Maschinenspracheprogrammierer möglich mit Hilfe der festgelegten Vektoren Programme zu schreiben, die problemlos auf allen drei Rechnern laufen.

2. Es ist auch gut möglich die vorhandenen Vektoren zu "verbiegen" und auf eigene Routinen zu richten (patchen). Damit ist dann sichergestellt, daß die eigene Routine automatisch vom System benutzt wird. Dieses Verfahren wird auch angewandt, wenn Sie ein Diskettenlaufwerk an den Schneider anschließen.

Neben den Vektoren liegen im 3ten RAM (&8000 bis &BFFF) noch viele Speicherstellen, in denen für das System wichtige Informationen gespeichert werden, die sogenannten PEEK's und POKE's.

Fassen wir noch einmal zusammen:

An den Adressen &0 bis &40 liegen im ROM und RAM die RST Befehlsroutinen, mit denen die Kommunikation zwischen verschiedenen Speicherbausteinen möglich wird. Da für jeden Restart nur 8 Bytes als Programmlänge zur Verfügung stehen, verzweigen die Restart Routinen ihrerseits zu den eigentlich ausführenden Routinen, die im dritten RAM, das ständig eingeschaltet ist, verzweigen.

Damit sind einige Restart Befehle bei den Schneider Rechnern sozusagen erweiterte Jump bzw. Call Befehle, bei denen es zusätzlich möglich ist, den ROM/RAM Status für die Sprungzieladresse anzugeben. Die genaue Funktion der einzelnen RST Befehle werden wir noch im folgenden besprechen.

Weiterhin befinden sich Sprungvektoren, die hauptsächlich RST Befehle sind, im dritten RAM, sowie wichtige PEEK's und POKE's.

Eine der wohl wichtigsten Systemroutinen ist die zur Ausgabe eines Zeichens auf dem Bildschirm. Mit CALL &BB5A kann sie aufgerufen werden. Diese Routine gibt das Zeichen aus, daß dem im Akku enthaltenen Wert entspricht.
Schreiben Sie ein Programm zur Ausgabe des Zeichensatzes (Codes 32 bis 255) des Schneider CPC.

Lösung:

```
10 ' ORG &A000
20 ' PRINT EQU &BB5A
30 ' LD B,223 ; ZAEHLER=255-32
40 ' LD A,32
50 ' SCHLEI CALL PRINT ; CHR$(A) AUSGEBEN
60 ' INC A
70 ' DJNZ SCHLEI
80 ' RET
90 ' END
```

Im Zusammenhang mit der Ausgabe von Zeichen besitzt der Assembler den Pseudobefehl DM. Auf den DM Befehl folgt als Operand ein Wort in Anführungszeichen. Die ASCII Codes der Buchstaben des Wortes werden durch DM ab der aktuellen Adresse abgelegt. Sehen Sie sich folgendes Programm an:

```
 10 ' ORG &A000
 20 ' PRINT EQU &BB5A
 30 ' LD HL,wort ; Adresse des auszugebenden Wortes
 40 ' schlei LD A,(HL) ; Akku mit ASCII Code des jeweiligen Buchstaben
laden
 50 ' INC HL ; Zeiger auf naechsten Buchstaben setzen
 60 ' CALL PRINT
 70 ' OR A ; Flags setzen
 80 ' JR NZ,schlei ; noch nicht 0,dann naechsten Buchstaben
 90 ' RET
100 ' wort DM "Schneider"
110 ' DM 0
120 ' END
```

Das durch DB 0 erzeugte Nullbyte am Ende des Wortes (Zeile 110), dient zur Erkennung des Endes des auszugebenden Wortes.

Disassemblieren Sie ab der Einsprungadresse der Routine (&BB5A). Sie erhalten folgende Ausgabe:

BB5A CF 00 94 RST &08/DW: &9400

An Adresse &BB5A steht ein Restart Befehl nach Adresse &0008. Übersetzen wir dort weiter:

0008 C3 82 B9 JP &B982

Diese Routine wird als "RST &08 Routine" bezeichnet. Sie bewirkt, daß die beiden hinter dem RST &08 Befehle stehenden Bytes (Low/High) besonders behandelt werden. Aus diesem Grund gibt der Disassembler die auf den RST &08 folgenden Bytes mit dem Kennzeichen DW (DATA Word, d.h. Low und High Byte) aus. DW stellt auch eine Pseudobefehl dar, der bewirkt, daß das auf den Befehl folgende Data word, also eine 2-Byte Zahl, an der jeweiligen Stelle im Speicher abgelegt wird. die Bits 0 bis 13 werden als Sprungzeilenadresse interpretiert (mit 14 Bits sind Adressen im Bereich von &0 bis &3FFF darstellbar). Bit 14 und 15 dienen zur Selektion von ROM bzw. RAM.

Bit 14 bestimmt den Status des Bereiches von &0 bis &3FFF. Bit 15 bestimmt den Status der Bereiches von &C000 bis &FFFF (Bildschirm RAM oder BASIC ROM). Ein gesetztes Bit selektiert das ROM, ein rückgesetztes wählt das ROM aus.

Welchen Status un welches Sprungziel hat folgender Befehl?

```
RST &08
DW  &9400
```

Zerlegen wir:
&9400=&8000+&1400=&X10 01 0100 0000 0000

```
Bit 15=1 => Bildschirm RAM
Bit 14=0 => Betriebssystem ROM
Adresse  :  &1400
```

RST &08/DW &9400 bewirkt einen Sprung zur Betriebssystemroutine an Adresse &1400. Der Bildschirm (RAM) ist selektiert.

Obwohl die RST Befehle prinzipiell Unterprogrammsprünge sind, d.h. die Rücksprungadresse wird auf den Stack gelegt, ist der RST &08 Befehl kein Unterprogramm, sondern ein normaler Sprung. Das wird durch Stapelmanipulation in der Routine ab &B982 erreicht.

Auch die anderen RST Befehl haben besondere Aufgaben. Wir werden sie im Laufe des Kapitels besprechen.

Mit Hilfe der Print Routine wollen wir jetzt ein Monitorprogramm schreiben.

Der Monitor

Da wir die Speicherinhalte als Hexzahlen ausgeben, brauchen wir zunächst ein Unterprogramm, daß ein Byte als Hexzahl ausgibt. Das auszugebende Byte wird im Akku übergeben.

Beispiel: A= 63 = &3F = &X 0011 1111

F entspricht den unteren 4 Bit (High Nibble).
3 entspricht den oberen 4 Bit (Low Nibble).

Zuerst wird das High Nibble ausgegeben. Dazu verschieben wir den Akkuinhalt 4 mal nach rechts (8-Bit Rotation). Das Ergebnis dieser Verschiebung ist &X1111 0011. Dann werden mit AND die obersten 4 Bit gelöscht. Anschließend enthält der Akku den Wert &X0000 0011=3. Dieser Wert (3) soll ausgegeben werden.

Der ASCII Code von 3 ist 51. Um den Wert 51 im Akku zu erhalten, müssen wir 48 zum Akkuinhalt (=3) addieren. Danach wird die >PRINT< Routine aufgerufen. Zum Ausgeben des Low Nibbles löschen wir die obersten 4 Bit vom alten Akkuinhalt. Nach der Addition von 48 erhalten wir 63. Als Ausgabe wollen wir F (&F=15) erhalten. Der ASCII Wert von F ist 70. Das bedeutet, daß wenn die auszugebende Hexziffer größer als 9 ist (also als Buchstabe dargestellt wird) muß zusätzlich 7 addiert werden, bevor die Routine aufgerufen wird.

Versuchen Sie die Routine für die Ausgabe eines Bytes in hexadezimaler Form zu schreiben.

```
A000          10        ORG  &a000
A000          20 PRINT  EQU  &bb5a
A000          30 ; ausgahex
A000          40 ; gibt Akku hexadezimal aus
A000          50 ; E-Register wird veraendert
A000 5F       60 AUSHEX LD   e,a ; Akku zwischenspeichern
A001 0F       70        RRCA  ; Akku um
A002 0F       80        RRCA  ; 4 Bits nach
A003 0F       90        RRCA  ; rechts
A004 0F      100        RRCA  ;rotieren
```

```
A005 E60F     110          AND  &X1111 ; Bit 4-7 loeschen
A007 CD0000   120          CALL conv ; High-Nibble ausgeben
A00A 7B       130          LD   a,e ; alter Akkuinhalt
A00B E60F     140          AND  &X1111 ; Bit 4-7 loeschen
A00D CD0000   150          CALL conv ; Low-Nibble ausgeben
A010 C9       160          RET   ; ende aushex
A011          170 ; Routine conv
A011          180 ; gibt dem Akkuinhalt entsprechende Hexzif
fer aus
**** Zeile  120  : OONV=&A011
**** Zeile  150  : CONV=&A011
A011 FE0A     190  CONV    CP   &a ; Wert der Ziffer < 10
A013 38FE     200          JR   c,zahl ; ja dann nach Zahl
A015 C607     210          ADD  a,7 ; 7 fuer Buchstaben addi
eren
**** Zeile  200  : ZAHL=&A017
A017 C630     220  ZAHL    ADD  a,48 ; =ASCI Code der Hex-Zi
ffer
A019 CD5ABB   230          CALL print
A01C C9       240          RET   ; ende conv

Programm :ausgahex
Start : &A000    Ende : &A01C
Laenge : 001D
 0  Fehler
Variablentabelle :
PRINT  BB5A  AUSHEX A000  CONV   A011  ZAHL   A017
```

Mit diesem Programm können wir nun die Ausgabe für das Compareprogramm aus dem letzten Kapitel in Maschinensprache schreiben. Verbinden Sie beide Programme so, daß die Ausgabe der Adressen von obiger Routine erledigt wird.

```
A000           10                ;compare
A000           20          ORG  &A000
A000           30   PRINT  EQU  &bb5a
A000           40   ANF    DS   2
A002           50   ENDE   DS   2
A044           60   ANFVER DS   2 ;Vergleichsblockanfang
A006 ED5B00A0  70          LD   DE,(Anf)
A00A 2A02A0    80          LD   HL,(Ende)
A00D B7        90          OR   A
A00E ED52      100         SBC  HL,DE ;Blocklaenge
A010 23        110         INC  HL ;+1
A011 44        120         LD   B,H
A012 4D        130         LD   C,L ;nach BC laden
A013 EB        140         EX   de,hl ;Anf nach HL
A014 ED5B04A0  150         LD   DE,(Anfver) ;Blockzeiger
A018 1A        160  WEITER LD   A,(DE) ;Vergleichselement
A019 13        170         INC  DE
A01A EDA1      180         CPI   ;Vergleich (HL) mit A
A01C C40000    190         CALL NZ,Ausga ;ungleich dann Ausg
abe
A01F EA18A0    200         JP   PE,Weiter ;naechstes Element
A022 C9        210         RET   ; ende compare
A023           220               ;
**** Zeile  190  : AUSGA=&A023
A023 D5        230  AUSGA  PUSH de ; Blockzeiger retten
A024 F5        240         PUSH af ; Flags retten
A025 2B        250         DEC  hl ; hl fuer ausgabe erniedr
igen
A026 7C        260         LD   a,h
A027 CD0000    270         CALL aushex ; Low Byte ausgeben
A02A 7D        280         LD   a,l
A02B CD0000    290         CALL aushex ; High Byte ausgeben
A02E 23        300         INC  hl ; hl wieder richtigstelle
n
A02F 3E20      310         LD   a,&20 ; Leerzeichen
A031 CD5ABB    320         CALL print
A034 F1        330         POP  af
A035 D1        340         POP  de
A036 C9        350         RET   ;ende ausga
A037           360               ;
```

```
A037          370              ; ausgahex
A037          380              ; gibt Akku hexadezimal aus
A037          390              ; E-Register wird veraender
t
**** Zeile  270  : AUSHEX=&A037
**** Zeile  290  : AUSHEX=&A037
A037 5F       400  AUSHEX LD   e,a ; Akku zwischenspeichern
A038 0F       410         RRCA  ; Akku um
A039 0F       420         RRCA  ; 4 Bits nach
A03A 0F       430         RRCA  ; rechts
A03B 0F       440         RRCA  ;rotieren
A03C E60F     450         AND  &x1111 ; Bit 4-7 loeschen
A03E CD0000   460         CALL conv ; High-Nibble ausgeben
A041 7B       470         LD   a,e ; alter Akkuinhalt
A042 E60F     480         AND  &x1111 ; Bit 4-7 loeschen
A044 CD0000   490         CALL conv ; Low-Nibble ausgeben
A047 C9       500         RET   ; ende aushex
A048          510              ; Routine conv
A048          520              ; gibt dem Akkuinhalt entsp
rechende Hexziffer aus
**** Zeile  460  : CONV=&A048**** Zeile  490  : CONV=&A048
A048 FE0A     530  CONV   CP   &a ; Wert der Ziffer < 10
A04A 38FE     540         JR   c,zahl ; ja dann nach Zahl
A04C 38FE     550         JR   c,zahl
A04E C607     560         ADD  a,7 ; 7 fuer Buchstaben addi
eren
**** Zeile  540    ZAHL=&A050
**** Zeile  550  : ZAHL=&A050
A050 C630     570  ZAHL   ADD  a,48 ; =ASCI Code der Hex-Zi
ffer
A052 CD5ABB   580         CALL print
A055 C9       590         RET   ; ende conv

Programm :comheaus
Start : &A000    Ende : &A055
Laenge : 0056
 0  Fehler
Variablentabelle :
PRINT  BB5A  ANF    A000  ENDE   A002  ANFVER A004  WEITER
A018  AUSGA  A023  AUSHEX A037  CONV   A048  ZAHL   A050
```

Jetzt wollen wir mit der Monitorroutine fortfahren.
Vom BASIC aus wird die Start- und Endadresse übergeben.

```
10 ' ORG &A000
20 ' PRINT EQU &BB5A
30 ' START DS 2
40 ' ENDE DS 2
```

Laden wir zunächst HL mit der Startadresse und geben diese aus:

```
50 ' LD HL,(START) ; Zeiger
60 ' WEI16 LD A,H ; High Byte ausgeben
70 ' CALL AUSHEX
80 ' LD A,L ; Low Byte ausgeben
90 ' CALL AUSHEX
```

Danach soll ein Leerzeichen ausgegeben werden:

100 ' LD A,&20 ; ASCII von Leerzeichen
110 ' CALL PRINT

Nun werden die Werte der 16 (8 bei Mode 1) folgenden Speicherstellen ausgegeben:

```
120 ' LD B,16 ; Zaehler
130 ' WEI LD A,(HL) ; Byte in Akku laden
140 ' CALL AUSHEX ; und ausgeben
150 ' LD A,&20 ; Leerzeichen
160 ' CALL PRINT ; ausgeben
170 ' INC HL
180 ' DJNZ WEI
```

Als Nächstes wird ein Leerzeichen ausgegeben und die letzten 16 (8) Bytes werden als ASCII Zeichen ausgegeben. Von Codes, die größer als 127 sind, wird 128 abgezogen (Bit 7 wird rückgesetzt). Für Codes, die kleiner als 32 sind (Steuerzeichen), wird ein Punkt (ASCII=46) ausgegeben.

```
190 ' LD A,&20
200 ' CALL PRINT ; Leerzeichen
210 ' LD DE,16
220 ' OR A ; Carry = 0
230 ' SBC HL,DE ; Zeiger um 16 erniedrigen
240 ' LD B,16
250 ' WEIAS LD A,(HL) ; Akku mit Byte laden
260 ' INC HL ; Zeiger erhoehen
270 ' RES 7,A ; Grafikzeichen in ASCII umwandeln
280 ' CP &20 ; groesser gleich 32 ??
290 ' JR NC,PR ; ja, dann Ausgabe
300 ' LD A,46 ; ASCII fuer Punkt
310 ' PR CALL PRINT ; Zeichen ausgeben
320 ' DJNZ WEIAS
```

Um auf den Anfang der nächsten Zeile zu gelangen, wird der Code 13 und Code 10 gesendet:

(CHR$(13) = Carriage Return=Enter)
(CHR$(10) = Line Feed-Zeilenvorschub)

```
330 ' LD A,13 ; Carriage Return
340 ' CALL PRINT ; ausgeben
350 ' LD A,10 ; Zeilenvorschub
360 ' CALL PRINT ; ausgeben
```

Nun wird geprüft, ob bereits das Ende erreicht ist:

```
370 ' PUSH HL ; Zeiger retten
380 ' LD DE,(ENDE)
390 ' OR A ; Carry = 0
400 ' SBC HL,DE ; Zeiger-Ende<=0
410 ' POP HL ; Zeiger holen (keine Flagbeeinflussung !!)
420 ' JR C,WEI16 ; HL-DE<0, dann weiter
430 ' JR Z,WEI16 ; HL-DE=0, dann weiter
440 ' RET ; HL-DE>0, dann Ende
450 ' ;Ende Monitor
```

Jetzt muß noch die Routine Aushex angehängt oder vorangestellt werden, und unser Programm ist lauffähig:

```
460 ' ;Ausgahex
470 ' ; gibt Akku Hexadezimal aus
480 ' ; E Register wird veraendert
490 ' AUSHEX LD E,A ; Akku zwischenspeichern
500 ' RRCA ; Akku um
510 ' RRCA ; 4 Bits nach
520 ' RRCA ; rechts
530 ' RRCA ; rotieren
540 ' AND &X1111 ; Bit 4-7 loeschen
550 ' CALL CONV ; High Nibble ausgeben
560 ' LD A,E ; alter Akkuinhalt
570 ' AND &X1111 ; Bit 4-7 loeschen
580 ' CALL CONV ; Low Nibble ausgeben
590 ' RET ; Ende Aushex
600 ' ; Routine Conv
610 ' ; gibt dem Akkuinhalt entsprechende Hexziffer aus
620 ' CONV CP &A ; Wert der Ziffer <10
630 ' JR C,ZAHL ; ja, dann nach Zahl
640 ' ADD A,7 ; 7 fuer Buchstaben addieren
650 ' ZAHL ADD A,48 ; =ASCII Code der Hexziffer
660 ' CALL PRINT
670 ' RET ; Ende Conv
680 ' END
```

Mit dieser Routine können wir allerdings nur den RAM des Rechners auslesen. Um auch auf das ROM zuzugreifen, benutzen wir den RST &18 Befehl. Dieser Befehl bewirkt bei den CPC's einen sogenannten Far Call. Die beiden auf den RST &18 folgenden Bytes stellen einen Zeiger auf die Adresse eines Sprungvektors dar. An der angegebenen Vektoradresse stehen 3 Bytes. Die ersten beiden zeigen auf die eigentliche Sprungadresse, und das 3te Byte bestimmt dabei den ROM/RAM Status.

```
Beispiel:       .
                .
                .
  &A000         RST &18
  &A001         DW  Vekadr
  &A003         RET
                .
                .
       VekadrDW Zielad
                DB Status
```

Durch den RST &18 Befehl in &A000 wird ein Unterprogrammsprung nach Zielad (V) ausgeführt, wobei der Status (V) bestimmt, ob ROM bzw. RAM selektiert ist.
Für Status (V) gelten folgende Werte:

Status	&0-&3FFF (Betriebssystem)	&C000-&FFFF(BASIC)
&FC =252:	ROM	ROM
&FD =253:	RAM	ROM
&FE =254:	ROM	RAM
&FA =255:	RAM	RAM

Alle anderen Werte für den Status selektieren einen Expansions ROM.
Der Bereich von &4000 bis &BFFF ist grundsätzlich RAM Adressenbereich.

Der Name "Far Call" (soviel wie: "weiter Ruf"), drückt aus, daß über den RST &18 Befehl Sprünge in alle RAM's und ROM's des Rechners möglich sind. Der Far Call wirkt wie ein CALL, d.h. die Programmausführung nach dem RET Befehl wird hinter dem aufrufenden RST &18 Befehl fortgesetzt.

Wollen wir also mit der Monitorroutine das ROM auslesen, so rufen wir sie über den RST &18 Befehl auf. Die Adresse, die der Sprungvektor angibt, ist dann die Startadresse der Monitorroutine. Zum Selektieren beider ROM's muß der Status 252 sein.

Die Erweiterung des Programmes sieht dann folgendermaßen aus:

```
10 ' ORG &A000
20 ' RST &18
30 ' DW  vektor
40 ' RET ; zurueck zum BASIC
50 ' VECTOR DW MONITO ; Adresse Sprungvektor
60 ' STATUS DB 252 ; ROM/RAM Status
70 ' PRINT EQU &BB5A
80 ' START DS 2
90 ' ENDE  DS 2
100 ' MONITO LD HL,(START) ; Zeiger
        .
        .
```

Das komplette Assemblerlisting:

```
A000             10          ORG  &a000
A000 DF          20          RST  &18
A001 0000        30          DW   vektor
A003 C9          40          RET   ; zurueck zu BASIC
**** Zeile  30   : VEKTOR=&A004
A004 0000        50   VEKTOR DW   monito ;adresse Sprungvektor
A006 FC          60   STATUS DB   252 ; ROM/RAM Status
A007             70   PRINT  EQU  &bb5a
A007             80   START  DS   2
A009             90   ENDE   DS   2
**** Zeile  50   : MONITO=&A00B
A00B 2A07A0     100   MONITO LD   hl,(start) ; Zeiger
A00E 7C         110   WEI16  LD   a,h ; High Byte ausgeben
A00F CD0000     120          CALL aushex
A012 7D         130          LD   a,l ; Low Byte ausgeben
A013 CD0000     140          CALL aushex
A016 3E20       150          LD   a,&20 ;ASCII von Leerzeichen
A018 CD5ABB     160          CALL print
A01B 0610       170          LD   b,16 ; Zaehler
A01D 7E         180   WEI    LD   a,(hl) ; Byte in Akku laden
A01E CD0000     190          CALL aushex ; und ausgeben
A021 3E20       200          LD   a,&20 ; Leerzeichen
A023 CD5ABB     210          CALL print ; ausgeben
```

```
A026 23        220         INC  hl
A027 10F4      230         DJNZ wei
A029 3E20      240         LD   a,&20
A02B CD5ABB    250         CALL print ; Leerzeichen
A02E 111000    260         LD   de,16
A031 B7        270         OR   a ; Carry = 0
A032 ED52      280         SBC  hl,de ; Zeiger um 16 erniedr
igen
A034 0610      290         LD   b,16
A036 7E        300   WEIAS LD   a,(hl) ; Akku mit Byte laden
A037 23        310         INC  hl ; Zeiger erhoehen
A038 CBBF      320         RES  7,a ;Grafikzeichen in ASCII
umwandeln
A03A FE20      330         CP   &20 ; groesser gleich 32 ??
A03C 30FE      340         JR   nc,pr ; ja, dann ausgabe
A03E 3E2E      350         LD   a,46 ; ASCII fuer Punkt
**** Zeile  340   : PR=&A040
A040 CD5ABB    360   PR    CALL print ; zeichen ausgeben
A043 10F1      370         DJNZ weias
A045 3E0D      380         LD   a,13 ; Carriage Return
A047 CD5ABB    390         CALL print ; ausgeben
A04A 3E0A      400         LD   a,10 ; Zeilenvorschub
A04C CD5ABB    410         CALL print ; ausgeben
A04F E5        420         PUSH hl ; Zeiger retten
A050 ED5B09A0 430          LD   de,(ende)
A054 B7        440         OR   a ; Carry = 0
A055 ED52      450         SBC  hl,de ; Zeiger-Ende<=0
A057 E1        460         POP  hl ;Zeiger holen (keine Flag
beeinflussung!!)
A058 38B4      470         JR   c,wei16 ;hl-de<0,dann weiter
A05A 28B2      480         JR   z,wei16 ;hl-de=0,dann weiter
A05C C9        490         RET   ; hl-de>0,dann ende
A05D           500               ; ende monitor
A05D           510               ; ausgahex
A05D           520               ; gib  Aku hexadezimal aus
A05D           530               ; E Register wird veraendert
**** Zeile  120   : AUSHEX=&A05D
**** Zeile  140   : AUSHEX=&A05D
**** Zeile  190   : AUSHEX=&A05D
A05D 5F        540   AUSHEX LD   e,a ;Akku zwischenspeichern
```

```
A05E 0F        550          RRCA  ; Akku um
A05F 0F        560          RRCA  ; 4 Bits nach
A060 0F        570          RRCA  ; rechts
A061 0F        580          RRCA  ;rotieren
A062 E60F      590          AND   &x1111 ; Bit 4-7 loeschen
A064 CD0000    600          CALL  conv ; High-Nibble ausgeben
A067 7B        610          LD    a,e ; alter Akkuinhalt
A068 E60F      620          AND   &x1111 ; Bit 4-7 loeschen
A06A CD0000    630          CALL  conv ; Low-Nibble ausgeben
A06D C9        640          RET   ; ende aushex
A06E           650                ; Routine conv
A06E           660                ;gibt dem Akkuinhalt entspr
echende Hexziffer aus
**** Zeile  600  : CONV=&A06E
**** Zeile  630  : CONV=&A06E
A06E FE0A      670  CONV    CP    &a ; Wert der Ziffer <10
A070 38FE      680          JR    c,zahl ; ja dann nach Zahl
A072 C607      690          ADD   a,7 ;7 fuer Buchstaben addie
ren
**** Zeile  680  : ZAHL=&A074
A074 C630      700  ZAHL    ADD   a,48 ;=ASCII Code der Hex-Zif
fer
A076 CD5ABB    710          CALL  print
A079 C9        720          RET   ; ende conv

Programm :monitor
Start : &A000    Ende : &A079
Laenge : 007A
 0  Fehler
Variablentabelle :
VEKTOR A004  STATUS A006  PRINT  BB5A  START  A007
ENDE   A009  MONITO A00B  WEI16  A00E  WEI    A01D
WEIAS  A036  PR     A040  AUSHEX A05D  CONV   A06E
ZAHL   A074
```

Das BASIC Bedienprogramm:

```
 10 REM Monitor Bedienprogramm
 20 MEMORY &9FFF
 30 LOAD"monitor.obj"
 40 r$(0)="ROM":r$(1)="RAM"
 50 MODE 2
 60 PRINT"M O N I T O R   R O M / R A M   L E S E N
 70 INPUT"Startadresse : &",a$
 80 adr=&A007:GOSUB 220
 90 INPUT"Endadresse   : &",a$
100 adr=&A009:GOSUB 220
110 p=VPOS(§0)
120 PRINT"Betriebssystem :";r$(betrsta);CHR$(13);
130 a$=INKEY$:IF a$="" THEN 130
140 IF a$<>CHR$(13) THEN betrsta=betrsta XOR 1:GOTO 120
ELSE PRINT
150 PRINT"BASIC          :";r$(basista);CHR$(13);
160 a$=INKEY$:IF a$="" THEN 160
170 IF a$<>CHR$(13) THEN basista=basista XOR 1:GOTO 150
ELSE PRINT
180 status=&X11111100 OR basista*2 OR betrsta
190 POKE &A006,status
200 CALL &A000
210 GOTO 70
220 a=VAL("&"+a$)
230 IF a<0 THEN a=a+2^16
240 ah=INT(a/256):POKE adr+1,ah
250 POKE adr,a-ah*256
260 RETURN
```

Beim Auswählen des Status wird durch >ENTER< der angezeigte Status beibehalten. Durch das Drücken einer bliebigen an deren Taste wird der Status verändert. Nun können wir uns auf die "Reise in die Firmware" begeben.

Starten Sie das Programm und geben folgendes ein:

```
Startadresse      : &CC00        >ENTER<
Endadresse        : &CE00        >ENTER<
Betriebssystem    : ROM          >ENTER<
BASIC             : ROM          >ENTER<
```

Geben Sie dieselben Adressen ein, ändern jedoch den Status des BASIC Bereiches zu RAM, erhalten Sie anstatt der Fehlermeldungen des Rechners (die im ROM liegen), nur ein Hexdump des Bildschirmbereichs (RAM).

Ab &660 (ROM) steht die Einschaltmeldung des Computers. Im BASIC ROM steht ab &E380 die Liste der BASIC Befehlsworte, die der Interpreter benutzt.

Sehen Sie sich die ROM und RAM Inhalte einmal an, damit Sie sich ein Bild von der Aufteilung machen können.

Der Großteil des ROM Inhaltes sind Programme. Schreiben Sie eine Routine mit dem RST &18 Befehl, die den Inhalt einer ROM Speicherstelle an ein BASIC Programm übergibt, und bauen Sie diese im Disassembler in das Unterprogramm ab Zeile 940 ein. Dann haben Sie die Möglichkeit die Programme im ROM zu übersetzen. Zum Verständnis der internen Routinen ist ein kommentiertes ROM Listing sehr sinnvoll.

Der Breakpoint

Als nächstes wollen wir zeigen, wie man prinzipiell ein Testprogramm für Maschinenprogramme schreibt. Sicherlich ist Ihnen auch schon manchmal der Rechner abgestürzt, und Sie hatten keine Ahnung, woran das lag. Ein Maschinenprogramm gibt keine Fehlermeldung aus, es stürzt in den meisten Fällen wenn ein Fehler vorliegt einfach ab. Eine Rekonstruktion des Weges zum Fehler hin ist nicht möglich.

Sinnvoll wäre es, wenn man an beliebiger Stelle das Programm unterbrechen könnte, um sich die Registerinhalte anzuschauen. Anhand dieser Informationen kann dann ein Fehler aufgespürt werden. Um das zu erreichen benutzen wir den RST &30 Befehl. Dieser Restart ist nicht vom Betriebssystem benutzt, und steht zur freien Verfügung. Die anderen Restartbefehle dürfen auf keinen Fall benutzt werden, da sie Aufgaben für das Betriebssystem erfüllen.

Der RST &30 Befehl hat den Code &F7. An der Stelle, wo das Programm unterbrochen werden soll, wird mit >POKE< der Code &F7 geschrieben. Durch den Befehlscode &F7 wird das Programm unterbrochen, daher der Name Breakpoint. Dann wird das zu testende Programm gestartet. Trifft es auf den RST &30 Befehl, wird ein Unterprogrammsprung nach Adresse &30 ausgeführt. An Adresse &30 schreiben wir einen JP Befehl, der zur eigentlichen Registerausgaberoutine verzweigt. Das folgende Assemblerlisting dokumentiert sich selbst:

Assemblerlisting:

```
A000            10       ORG  &a500
A500            20             ; Breakpoint aktivieren
A500 3EC3       30       LD   a,&c3 ; Code fuer JP
A502 323000     40       LD   (&0030),a ; nach &30 (RST) l
aden
A505 210000     50       LD   hl,redump ;startadresse Regis
ter-Dump
A508 223100     60       LD   (&0031),hl ; nach &31/&32
A50B C9         70       RET   ;aktivieren ende
A50C            80             ; Register-Dump
**** Zeile  50  : REDUMP=&A50C
A50C F5         90  REDUMP PUSH af ; Register auf
A50D C5        100       PUSH bc ; Stack legen
A50E D5        110       PUSH de
A50F E5        120       PUSH hl
A510 210000    130       LD   hl,0 ; urspruenglichen
A513 39        140       ADD  hl,sp ; SP Inhalt
A514 110A00    150       LD   de,10 ; berechnen
A517 19        160       ADD  hl,de ; und auf den
```

```
A518 E5        170          PUSH hl ; Stack legen
A519 060C      180          LD   b,12 ; Anzahl auszugebender
Bytes
A51B           190                ;Ausgabe in der Reihenfolge
A51B           200                ; PC AF BC DE HP SP
A51B 2B        210   PRNT   DEC  hl
A51C 7E        220          LD   a,(hl) ; Byte vom Stack holen
A51D CD0000    230          CALL aushex ; und ausgeben
A520 10F9      240          DJNZ prnt
A522 E1        250          POP  hl ;alten SP holen und nicht
beachten
A523 E1        260          POP  hl ; restliche
A524 D1        270          POP  de ; Register
A525 C1        280          POP  bc ; vom Stapel
A526 F1        290          POP  af ; holen
A527 DDE1      300          POP  ix ;Ruecksprungadresse holen
A529 C9        310          RET   ; ins BASIC springen

........: Routine ausgahex

Programm :breakpoi
Start : &A500    Ende : &A529
Laenge : 002A
 0  Fehler
Variablentabelle :
REDUMP A50C  PRNT   A51B
```

Auch hier müssen Sie natürlich die Ausgahexroutine wieder "anhängen".

Durch Call &A500 wird die Startadresse der Register Dump Routine mit dem JP Befehl ab Adresse &30 gespeichert. Damit steht der RST &30 Befehl zur Verfügung um Programme zu testen. Probieren Sie nach Call &A500 folgendes Programm:

```
10 ' ORG &A000
20 ' LD A,1
30 ' LD BC,&0203
40 ' LD DE,&0405
50 ' LD HL,&0607
60 ' RET
70 ' END
```

Nach der Übersetzung starten Sie das Programm mit Call &A000. Die Register sollten mit den Werten 1 bis 7 geladen sein. Um dies zu prüfen, setzen wir anstelle des RET Befehls den RST &F7 Befehl:

>POKE &A00B,&F7<

Geben Sie nun CALL &A000 ein so erhalten Sie die Ausgabe:

A00C0168020304050607BFF8

Die ersten beiden Bytes sind der PC Inhalt nach der Unterbrechung (die Unterbrechung fand an Adresse &A00B statt). Dann folgt der Akku (=1). Als nächstes das Flagregister:

```
&68=&X0 1 1 0 1 0  0 0
        S Z   H  P/V N C
```

Darauf folgen die Register B, C, D, E, H, L.
Die letzten 4 Ziffern stellen den Inhalt von SP vor der Unterbrechung dar.

Beachten Sie, daß der Breakpoint (der Code &F7, also RST &30) mit dieser Routine nur in der obersten Programmebene stehen darf. Wird er in einem Unterprogramm angetroffen, findet der korrekte Rücksprung ins BASIC nicht statt, da nur eine Rücksprungadresse durch POP IX vom Stapel geholt wird.

Auf das Prinzip dieser Routine aufbauend, ist es möglich, komfortable Programmierhilfen wie z.B. einen Einzelschrittsimulator zu schreiben. Gute Programmpakete enthalten solche Testprogramme.

Suchroutine

Um Ihre Sammlung von Monitorroutinen vollständig zu machen, folgt hier noch die Routine, die den Speicher nach einer Zeichenfolge durchsucht. Wollen Sie auch das ROM durchsuchen, müssen Sie, wie bei der Monitorroutine den Aufruf über einen Far Call &18 Befehl bewerkstelligen. Die Routine "ausgahex" muß mit integriert werden.

```
A000           10                 ; Sucher
A000           20           ORG  &a000
A000           30   PRINT   EQU  &bb5a
A000           40   START   DS   2
A002           50   ENDE    DS   2
A004           60   LAENGE  DS   1 ;laenge der Zeichenfolge
A005           70   TAB1    DS   1 ; Anfang Zeichenfolge
A006           80   TAB2    DS   19 ;max.20 Zeichen reserviert
A019           90                 ; Anfang Sucher
A019 3A05A0    100          LD   a,(tab1) ; erste Element
A01C ED5B00A0  110          LD   de,(start) ;Blockanfang
A020 2A02A0    120          LD   hl,(ende) ;Blockende
A023 B7        130          OR   a ;Carry=0
A024 ED52      140          SBC  hl,de ; Blocklaenge
A026 23        150          INC  hl ; nach BC
A027 44        160          LD   b,h ; laden
A028 4D        170          LD   c,l
A029 EB        180          EX   de,hl ; start nach HL
A02A EDB1      190  COMP    CPIR  ; suchen bis
A02C CC0000    200          CALL z,found ;Gleichheit dann nach
found
A02F E0        210          RET  po ;RET wenn Block durchsucht
A030 18F8      220          JR   comp
**** Zeile  200  : FOUND=&A032
A032 F5        230  FOUND   PUSH af
A033 C5        240          PUSH bc
A034 E5        250          PUSH hl
A035 3A04A0    260          LD   a,(laenge)
```

```
A038 4F       270          LD   c,a ; laenge nach
A039 0600     280          LD   b,0 ; BC speichern
A03B 0D       290          DEC  c ;da ab 2.Element verglichen
wird
A03C 28FE     300          JR   z,ok
A03E 1106A0   310          LD   de,tab2 ; Adresse 2.Element
A041 1A       320    COMP1 LD   a,(de) ; naechstes Element
A042 EDA1     330          CPI   ; vergleichen
A044 13       340          INC  de ; Zeiger erhoehen
A045 20FE     350          JR   nz,rueck ;ungleich,zum CPIR-
Befehl
A047 EA42A0   360          JP   pe,comp1 ; noch nicht BC=0,
dann weiter vergleichen
**** Zeile  300  : OK=&A04B
A04A E1       370    OK    POP  hl ; adresse der gefundenen
Folge+1
A04B 2B       380          DEC  hl
A04C 7C       390          LD   a,h ; High Byte
A04D CD0000   400          CALL aushex ; ausgeben
A05E 7D       410          LD   a,l ; Low Byte
A051 CD0000   420          CALL aushex ; ausgeben
A054 3E20     430          LD   a,32 ; Leerzeichen
A056 CD5ABB   440          CALL print ; ausgeben
A059 23       450          INC  hl ;alten Wert wiederherstel-
len
A05A C1       460          POP  bc
A05B F1       470          POP  af
A05C C9       480          RET   ; weitersuchen
**** Zeile  350  : RUECK=&A05E
A05D E1       490    RUECK POP  hl ; nicht gleich
A05E C1       500          POP  bc
A06F F1       510          POP  af
A060 C9       520          RET   ; weitersuchen
                     .
                     .
                     .
          Routine Ausgahex
```

```
Programm :sucher
Start : &A000    Ende : &A061
Laenge : 0062
 0  Fehler
Variablentabelle :
PRINT  BB5A  START  A000  ENDE   A002  LAENGE A004  TAB1
A005  TAB2   A006  COMP   A02A  FOUND  A032  COMP1  A042
OK    A04B   RUECK  A05E
```

Die zu suchende Folge muß vor dem Aufruf der Routine mit Call &A019 ab Adresse &A005 gespeichert werden; dieses sowie das Poken der Länge der Start- und Endadresse wird von einem BASIC Programm erledigt.

Eingabe von Daten

Bisher haben wir Systemroutinen kennengelernt, die eine Ausgabe von Maschinensprache aus ermöglichen. Jetzt wollen wir uns mit der Eingabe von Daten beschäftigen. Variable Daten, wie Anfangs- und Endadresse mußten bisher relativ umständlich vom BASIC aus mit >POKE< Befehlen an die Maschinenprogramme übergeben werden.

Das Schneider BASIC bietet uns aber die Möglichkeit mit dem CALL Befehl Daten zu übergeben. Mit diesem Befehl ist eine Übergabe von bis zu 32 2-Byte Zahlen möglich. Der erweiterte CALL Befehl hat folgende Form:

CALL Adresse,Ausdruck,Ausdruck,...

Ausdruck (V) kann dabei eine 16-Bit Zahl, eine Funktion oder eine Variable sein, deren Wert eine 16-Bit Zahl ist. Da bis zu 32 Zahlen übergeben werden können, ist es nicht möglich alle in den Registern zu speichern. Die übergebenen Zahlen werden auf den Stapel gelegt. Der Akku enthält die Anzahl der übergebenen Ausdrücke. Das DE Register enthält den letzten angegebenen Wert. Die Stapeladresse, an der die letzte Eintragung der übergebenen Register steht, wird im IX Register übergeben. Das

C Register enthält den ROM/RAM Status (siehe Far Call RST &18), dieser ist beim Standardaufruf immer &FF (also RAM's ausgewählt). HL zeigt immer auf die Adresse, an der der jeweilige Call Befhehl endet. Fassen wir zusammen:

Register	keine Übergabe von Zahlen	Übergabe von von Zahlen
A	0	n (Anzahl)
F	F=&68 (Z=1)	F=&28 (Z=0 !)
B	&20	&20-n
C	&FF (Status)	&FF
DE	Anzuspringende Adresse	letzte übergebene Zahl
HL	Adresse des Call Befehls Endes	
IX	Stapeladresse &BFFE	Stapeladresse des letzten Elements =&BFFE-2*n

Benutzen wir diese Art der Eingabe um das Monitorprogramm mit den entsprechenden Werten zu versorgen. Übergeben werden soll:

Die Startadresse
Die Endadresse
Der ROM/RAM Status (Far Call)

Der Aufruf hat dann folgendes Format:

Call &A000,Startadresse,Endadresse,Status

Die Änderungen im Programm sehen folgendermaßen aus:

```
10 ' ORG &A000
15 ' CP 3 ; 3 Parameter
20 ' RET NZ ; nein,dann Ende
```

Zunächst wird geprüft, ob 3 Werte eingegeben wurden (A=3). Falls das nicht zutrifft, erfolgt ein Rücksprung ins BASIC.

```
25 ' LD A,D
30 ' OR A
40 ' LD A,E
45 ' LD (Status),A
```

In den Zeilen 25, 30 und 35 wird geprüft, ob D=0 ist. Falls D nicht gleich Null ist, wird das Programm beendet. Der Status ist eine 1-Byte Zahl. Es können aber auch 2-Byte große Zahlen eingegeben werden. Aus diesem Grund muß geprüft werden, ob das zweite Byte, also das High Byte gleich Null ist. In den Zeilen 40 und 45 wird der eingegebene Status an die für den RST &18 Befehl richtige Stelle geschrieben.

```
50 ' LD E,(IX+2)
55 ' LD D,(IX+3)
60 ' LD L,(IX+4)
65 ' LD H,(IX+5)
```

In den Zeilen 50 und 55 wird die übergebene Endadresse nach DE geladen. In Zeile 60 und 65 wird die Startadresse in HL gespeichert.

```
 70 ' RST &18
 75 ' DW vektor
 80 ' RET ;zurueck zum BASIC
 85 ' vektor DW monito ; Adresse Sprungvektor
 90 ' status DS 1 ;ROM/RAM Status
 95 ' ende DS 2
100 ' monito LD (ende),de
```

.......Fortsetzung wie bekannt
Nach der Übersetzung des gesamten Programms, erreichen Sie zum Beispiel mit

CALL &A000,&CC50,&CE60,252

die Ausgabe der Fehlermeldung des BASIC ROM's.

Eine weitere wichtige Systemroutine ist die, zur Eingabe einer Taste. Nach Aufruf von &BB06 wartet der Rechner, bis eine Taste gedrückt ist. Der Wert der gedrückten Taste wird dann im Akku zurückgegeben.

Mit folgender einfachen Routine können wir eine einfache Eingabe über die Tastatur realisieren.

```
A000          10        ORG  &A000
A000          20  GET   EQU  &BB06
A000          30  PRINT EQU  &bb5a
A000 CD06BB   40  EIN   CALL get
A003 CD5ABB   50        CALL print
A006 FE0D     60        CP   13 ; Enter ???
A008 20F6     70        JR   nz,ein
A00A 3E0A     80        LD   a,10
A00C CD5ABB   90        CALL print ; Zeilenvorschub
A00F C9      100        RET

Programm :eingabe
Start : &A000    Ende : &A00F
Laenge : 0010
 0 Fehler
Variablentabelle :
GET    BB06  PRINT  BB5A  EIN    A000
```

Anmerkung: Bei dieser Eingabe funktionieren alle CTRL Steuerzeichen ab CTRL L für Bildschirm löschen oder CTRL G für Ton klingen lassen.

Befehlserweiterung mit RSX

Ein erweitern des Schneider BASIC Befehlsvorrates ist auf verschiedene Weisen möglich. Nur (!) beim 464 kann man über die "Patchbereiche" im RAM die vorhandenen BASIC ROM Routinen erweitern. Doch diese Möglichkeit ist relativ eingeschränkt, da nur 9 solcher Patches möglich sind. Beim CPC

6128/664 sind sie außerdem aus Platzgründen ganz weggelassen worden. Es gibt jedoch bei allen CPC's die standardmäßig vorgesehene Methode, die die Erweiterung mit eigenen Befehlen ermöglicht. Sie alle kennen diese Methode. Sie wird für alle AMSDOS Diskettenbefehle verwendet.

Sicherlich haben Sie schon einmal mit >|CPM< den CPM Modus aufgerufen, jedenfalls wenn Sie Diskettenlaufwerksbesitzer sind. Der CPM Befehl beginnt mit einem Strich ("|"). Bei Eingabe von "CPM" (ohne Strich) erhalten Sie nur einen "Syntax Error". Betreiben Sie den Rechner ohne Floppy (nur bei 464 möglich), erhalten Sie nach >|CPM< ein "Unknown Command Error".

Es handelt sich also um einen Befehl, der erst vorhanden ist, wenn eine Floppy angeschlossen ist. Damit ist er eine Erweiterung.

Der "|" teilt dem System mit, daß es sich im folgenden um einen Erweiterungsbefehl handelt. Die Methode, von vornherein evtl. Erweiterungen einheitlich durch ein Sonderzeichen zu kennzeichnen, um sie dann zu erkennen und auszuführen, ist bei den Schneider Rechnern fest eingebaut. Diese Art der Einbindung von Befehlen bezeichnet man als RSX. RSX steht für "Resident System Extention", was soviel bedeutet wie "feste Systemerweiterung". D.h., das RSX ursprünglich für die Einbindung von neuen Befehlen, die in zusätzlichen ROM's (Expansions ROM's) vorhanden sind, gedacht war. Für den Floppybetrieb gibt es einen solchen Expansions ROM, in dem auch die Befehle |CPM, |ERA usw. enthalten sind.

Wir können die RSX Methode aber auch benutzen, um unsere selbstgeschriebenen Routinen wie wirkliche Befehle einzubinden und somit komfortabel zu benutzen.

Die Einbindung mit der RSX Methode wollen wir am Beispiel des DOKE Befehls verdeutlichen.

DOKE ist quasi ein "Doppelter >POKE<" Befehl. Mit >POKE< kann eine Speicherstelle mit einem Wert zwischen 0 und 255 beschrieben werden. DOKE beschreibt zwei aufeinanderfolgende

Speicherstellen mit einem Wert zwischen 0 und 65500. Dazu wird der Wert in Low Byte und High Byte zerlegt. Das Low Byte wird an der unteren Speicherstelle abgelegt, das High Byte an der Speicherstelle mit der höheren Adresse. Mit DOKE wird das Ändern vieler Systemparameter die ins RAM gespeichert sind (üblicherweise als PEEK's und POKE's) vereinfacht.

Die Parameterübergabe bei den RSX ist analog der des CALL Befehls.

Der DOKE Befehl benötigt zwei Parameter:
Die Adresse, ab der der Wert gespeichert werden soll und der Wert, der ab dieser Adresse abgelegt werden soll. Damit hat der Befehl folgendes Format:

|DOKE, Adresse, Wert

Die Parameter werden folgendermaßen übergeben:

A: Enthält Anzahl der übergebenen Parameter

Flags: Zero Flag=1 wenn keine Parameter übergeben wurden, sonst 0

B: Enthält 32-Anzahl der Parameter

DE: Enthält letzten übergebenen Parameter

IX: Adresse des letzten Elementes. Der vorletzte Parameter steht dann an Adresse IX+2, der nächste an Adresse IX+4 usw.

Die DOKE Routine sieht dann so aus:

```
100 ' LD L,(IX+2) ; übergebene Adresse
110 ' LD H,(IX+3) ; ins HL Register laden
120 ' LD (HL),E ; Low Byte speichern
130 ' INC HL ; Adresse auf High Byte setzen
140 ' LD (HL),D ; High Byte speichern
150 ' RET ; fertig!
```

Mit CALL &A000,Adresse,Wert könnten wir nach der Assemblierung der Routine den DOKE Befehl aufrufen, vorausgesetzt, er wurde ab &A000 abgespeichert. Probieren Sie z.B. einmal:

CALL &A000,..,..

Damit haben wir eine funktionierende DOKE Routine. Allerdings ist der Aufruf mit CALL etwas lästig. DOKE soll jetzt auch mit DOKE aufgerufen werden. Dazu muß eine kleine Einbindungsroutine geschrieben werden. Die Hauptarbeit nimmt uns dabei die Routine "Logext" des Betriebssystems ab, die die eigentliche Einbindung vornimmt. Wir müssen lediglich einige Werte an Logext übergeben.
Um DOKE einzubinden muß das System folgendes "wissen":

1) Name der Erweiterung
2) Adresse der Routine
3) Adresse von 4 unbenutzten Bytes die intern zur Verwal tung der Routine benutzt werden

Die Adresse der vier Systembytes wird vor dem Aufruf von Logext ins HL Register geladen. Außerdem wird BC mit der Startadresse einer Tabelle geladen, in der die weiteren Informationen enthalten sind.

Diese Tabelle enthält als erstes die Adresse einer zweiten Tabelle, nämlich der, in welcher der Name oder auch mehrere Namen verschiedener einzubindender Routinen, enthalten ist. Auf die Adresse der 2ten Tabelle in der 1ten Tabelle folgt ein Sprungbefehl auf die einzubindende Routine bzw. Routinen. Werden mehrere Routinen eingebunden, müssen Sprungbefehle und Routinennamen einander entsprechen.

Die 2te Tabelle enthält, wie gesagt, die Namen der Routinen. Dabei muß, um die einzelnen Namen voneinander zu trennen, das Bit 7 des letzten Buchstaben eines Namens immer gesetzt sein.

Doch betrachten wir unser Beispiel:

```
 10 ' LD BC,RSXTAB ; Adresse der 1ten Tabelle
 20 ' LD HL,SYSBYT ; Adresse des Systembytes
 30 ' CALL &BCD1 ; Routine Logext aufrufen
 40 ' RET ; Einbindung fertig
 50 ' RSXTAB DW NAMTAB ; Tabelle 1 enthält als erste Adresse die
      Anfangsadressen der 2ten Tabelle (Namenstabelle)
 60 ' JP START ; und dann Sprungbefehle auf die Routine
 70 ' NAMTAB DM "DOK" ; Namenstabelle
 80 ' DB &C5 : =ASC("E")+128 damit Bit 7 gesetzt ist
 90 ' DB 0 ; Nullbyte kennzeichnet das Ende der Namenstabelle
 95 ' SYSBYT DS 4 ; 4 Bytes für Kernal reservieren
100 ' START  LD.... hier beginnt die Routine
                  :                    :
```

Nach dem Assemblieren des gesamten Programms (bzw. BASIC Lader), wird einmal mit CALL &A000 der DOKE Befehl eingebunden. Von diesem Zeitpunkt an ist DOKE eine Befehlserweiterung (RSX), d.h. der Befehl kann wie jeder andere im Direktmodus oder im Programm benutzt werden.
Der neue Befehl hat folgendes Format:

>DOKE,Adresse,Wert

Achten Sie darauf die Einbindungsroutine nur einmal aufzurufen. Die 4 Systembytes werden u.a. dazu benutzt, auf evtl. weitere RSX Tabellen zu zeigen. Wird zum Zweitenmal dieselbe Routine initialisiert, so wird der Zeiger, der auf die nächste RSX Tabelle zeigen soll, logischer Weise wieder auf dieselbe Tabelle gerichtet.

Dadurch kann es passieren, daß die Befehlserkennung in einer Endlosschleife hängen bleibt, was natürlich recht ungünstig ist. Folgendes kleines Programm verhindert zusätzlich, daß ein Wiederaufruf stattfinden kann, indem es an den Anfang der Init Routine ein RET Befehl schreibt.

```
A000 010000     10  INIT   LD   bc,rsxtab
A003 210000     20         LD   hl,sysbyt
A006 CDD1BC     30         CALL &bcd1
A009 3EC9       40         LD   a,&c9
A00B 3200A0     50         LD   (init),a
A00E C9         60         RET
**** Zeile  10  : RSXTAB=&A00F
A00F 0000       70  RSXTAB DW   namtab
A011 C30000     80         JP   start
**** Zeile  70  : NAMTAB=&A014
A014 444F4B     90  NAMTAB DM   "DOK"
A017 C5        100         DB   &c5
A018 00        110         DB   0
**** Zeile  20  : SYSBYT=&A019
A019           120  SYSBYT DS   4
**** Zeile  80  : START=&A01D
A01D DD6E02    130  START  LD   l,(ix+2)
A020 DD6603    140         LD   h,(ix+3)
A023 73        150         LD   (hl),e
A024 23        160         INC  hl
A025 72        170         LD   (hl),d
A026 C9        180         RET

Programm :doke
Start : &A000    Ende : &A026
Laenge : 0027
 0  Fehler
Variablentabelle :
INIT   A000  RSXTAB A00F  NAMTAB A014  SYSBYT A019
START  A01D
```

Ein weiteres Beispiel für eine RSX eingebundene Routine haben Sie bereits im Zusammenhang mit dem Disassembler kennengelernt. Das folgende Assemblerlisting gibt genau den Befehl, der im erwähnten Programm eingebunden ist, wieder.

Neu an diesem Programm ist die Möglichkeit, auch Werte vom Maschinenprogramm aus zurück ans BASIC zu übergeben. Dazu wird die Variablenpointerfunktion "@" benutzt.

Der "@" (Klammeraffe) ist eine Funktion des Schneider BASIC, die bisher in keinem Handbuch aufgetaucht ist. Um diese Funktion zu verstehen und anzuwenden, müssen wir uns kurz mit der internen Speicherung von BASIC Variablen beschäftigen.

Es gibt drei verschiedene Arten der Darstellung von Daten im Computer:

```
1. Integer        INT - Ganzzahl
2. Real           REL - Kommazahlen
3. Strings        STR - alphanumerisch
```

Die erste Art, die der Ganzzahldarstellung, haben Sie bereits kennengelernt. Eine Integerzahl wird in Low Byte und High Byte zerlegt. Bit Nummer 7 des High Bytes wird als Vorzeichen benutzt. Null bedeutet positive und 1 negative Zahlen. Die negativen Zahlen werden im Zweierkomplement dargestellt, d.h. der Betrag der Zahl wird komplementiert und 1 addiert (siehe Kapitel 4.6 Arithmetische Befehle). Damit können INTEGER Konstanten oder Variablen ganzzahlige Werte von -32768 bis +32767 erhalten.

dezimal	binär	hex
-32768	1 000 0000 0000 0000	80 00
-32767	1 000 0000 0000 0001	80 01
-32766	1 000 0000 0000 0010	80 02
-32765	1 000 0000 0000 0011	80 03
	...	
-2	1 111 1111 1111 1110	FF FE
-1	1 111 1111 1111 1111	FF FF
0	0 000 0000 0000 0000	00 00
1	0 000 0000 0000 0001	00 01
2	0 000 0000 0000 0010	00 02
	...	
32766	0 111 1111 1111 1110	7F FE
32767	0 111 1111 1111 1111	7F FF

Die @ Funktion gibt die Adresse, an der der Wert einer Variablen gespeichert ist. Bei Integervariablen ist das die Adresse des Low Bytes. Probieren Sie:

>W%=&1234:POKE @W%,&56:PRINT HEX$(W%)< ergibt &1256

Auf diese Weise können Werte aus dem Maschinenprogramm ans BASIC übergeben werden, indem beim Aufruf der Routine der Variablenpointer der Variablen übergeben wird. Das Maschinenprogramm schreibt an diese Adresse den zu übergebenen Wert. Nach dem Rücksprung im BASIC steht der Wert dann in der jeweiligen Variablen zur Verfügung. Hier folgt das Assemblerlisting des RPEEK Befehls, der eine Wertrückgabe auf diese Weise bewerkstelligt.

```
A000             10                   ; rpeek
A000             20                   ; RSX Befehl
A000             30
; Format :  "|rpeek,Uebergabevariable(INT),Adresse,ROM/RAM-S
tatus"
9F20             40            ORG   &9f20
9F20             50
;  CPC      6128 ;   464   ,   664
9F20             60   OPMIS    EQU   &d055 ; &cfed , &d058
9F20             70   IMPARG   EQU   &c21d ; &c205 , &cb50
9F20 010000      80   INIT     LD    bc,rsxtab
9F23 210000      90            LD    hl,sysbyt
9F26 CDD1BC     100            CALL  &bcd1 ; Log Ext
9F29 3EC9       110            LD    a,&c9
9F2B 32209F     120            LD    (init),a
9F2E C9         130            RET
**** Zeile  80  : RSXTAB=&9F2F
9F2F 0000       140   RSXTAB   DW    namtab
9F31 C30000     150            JP    start
**** Zeile  140  : NAMTAB=&9F34
9F34 52504545 160   NAMTAB   DM    "RPEE"
9F38 CB         170            DB    &cb
9F39 00         180            DB    0
**** Zeile  90  : SYSBYT=&9F3A
9F3A            190   SYSBYT   DS    4
**** Zeile  150  : START=&9F3E
9F3E DF         200   START    RST   &18 ; Wegen Fehlerroutinen
9F3F 0000       210            DW    vekto1
; Ansprung ueber farr call
9F41 C9         220            RET    ; mit upper ROM on
**** Zeile  210  : VEKTO1=&9F42
9F42 0000       230   VEKTO1   DW    anfang
9F44 FD         240            DB    253
**** Zeile  230  : ANFANG=&9F45
9F45 FE03       250   ANFANG   CP    3 ; zwei Parameter ?
9F47 C255D0     260            JP    nz,opmis
; nein, dann Operand missing
9F4A 7A         270            LD    a,d
```

```
9F4B FE00      280          CP    0
9F4D C21DC2    290          JP    nz,imparg
; Status>255, dann Fehler
9F50 7B        300          LD    a,e
9F51 320000    310          LD    (status),a
; Status fuer farr call
9F54 DD6E02    320          LD    l,(ix+2) ; Adresse low
9F57 DD6603    330          LD    h,(ix+3) ; Adresse high
9F5A DF        340          RST   &18 ; Routine Byte lesen
9F5B 0000      350          DW    vekto2 ; anspringen
9F5D DD6E04    360          LD    l,(ix+4)
; Variablenzeiger der
9F60 DD6605    370          LD    h,(ix+5)
; INT-Variablen lesen
9F63 77        380          LD    (hl),a
; gelesenen Wert an Var. uebergeben
9F64 97        390          SUB   a
9F65 23        400          INC   hl ; High Byte von Var. = 0
9F66 77        410          LD    (hl),a
9F67 C9        420          RET
**** Zeile  350  : VEKTO2=&9F68
9F68 0000      430   VEKTO2 DW    redbyt
**** Zeile  310  : STATUS=&9F6A
9F6A FD        440   STATUS DB    253
**** Zeile  430  : REDBYT=&9F6B
9F6B 7E        450   REDBYT LD    a,(hl)
; Routine zum Byte lesen
9F6C C9        460          RET
End Assumed

Programm :rpeek
Start : &9F20    Ende : &9F6C
Laenge : 004D
 0  Fehler
Variablentabelle :
OPMIS  D055   IMPARG C21D   INIT   9F20   RSXTAB 9F2F
NAMTAB 9F34   SYSBYT 9F3A   START  9F3E   VEKTO1 9F42
ANFANG 9F45   VEKTO2 9F68   STATUS 9F6A   REDBYT 9F6B
```

Nachdem wir jetzt die Übergabe von Zahlen besprochen haben, schauen wir uns einen weiteren, vollkommen anderen Variablentyp an, die Stringvariable:

In einem String werden alphanumerische Daten, d.h. Zeichen, wie Buchstaben oder Ziffern gespeichert. Jedem Zeichen ist dabei ein Code zugeordnet. Die Zeichencodes entsprechen, für die Codes von 0 bis 127, weitgehend dem sog. American Standard Code for Information Interchange (ASCII).

Zum Speichern eines Zeichens ist also genau ein Byte notwendig. Ein "String" (engl.string: Schnur, Kette), ist nun eine Kette von aufeinanderfolgenden Zeichencodes. Da ein Zeichen einem Byte entspricht, wird ein String in einer Reihe aufeinanderfolgender Speicherstellen im RAM gespeichert. Um einer bestimmten Variablen einen bestimmten String zuzuordnen, sind zwei Informationen notwendig, die den sog. String Descriptor bilden:

1. Die Adresse der ersten Speicherstelle, die den ersten Zeichencode des Strings enthält.

2. Die Länge des Strings, also die Anzahl der Bytes, die die Zeichenkette bilden.

Diese beiden Daten werden zusammen mit dem Variablennamen im BASIC Variablenbereich abgespeichert. Die eigentliche Zeichenkette steht an anderer Stelle. Entweder im Programm selbst oder in dem speziell dafür reservierten Stringbereich.

Im BASIC können wir wieder die "@" Funktion benutzen, um die Adresse des Stringdescriptors einer Variablen zu lesen. Der Stringdescriptor besteht aus drei Bytes:

```
1. Byte     : Länge des Strings
2.+3. Byte : Startadresse des Strings
```

Die "@" Funktion gibt die Adresse des ersten Bytes des Stringdescriptors aus.
Probieren Sie folgendes Programm aus:

```
10 X$="ZZZeichenkeTTTTe"
20 AD=@(X$)
30 LA=PEEK(AD)
40 ST=PEEK(AD+1)+256*PEEK(AD+2)
50 FOR I=ST TO ST+LA-1
60 PRINT CHR$(PEEK(I));
70 NEXT
```

Der "@" im Zusammenhang mit der Stringbehandlung ist auch bereits im Sourceerzeugerprogramm benutzt worden. Das dort verwendete Maschinenprogramm konnte eine als String gegebene BASIC Zeile in eine wirkliche BASIC Zeile umwandeln. Dazu wird einfach so getan, als sei der String im Direktmodus per Hand eingegeben worden. Dann wird er auf die selbe Weise in eine Zeile übersetzt, wie im Direktmodus.

Einzige Einschränkung für folgendes Programm ist, daß die zu erzeugende Zeile eine höhere Zeilennummer haben muß, als die aktuelle. Ansonsten würde das Programm nach RET nicht mehr an die richtige Stelle zurückfinden.

```
A000            10                     ; liner
A000            20                     ; erzeugt BASIC Zeilen
A000            30                     ; 1. im Direktmodus
A000            40                     ; 2. im Programm, wenn
A000            50
;    Zeilennummer > als aktuelle
A000            60
; a$ enthalte die Zeile im ASCII-
A000            70
; Code, durch Nullbyte beendet
A000            80                     ;
A000            90                     ; Format :  call &a000,@a$
A000           100                     ;
9F00           110          ORG   &9f00
9F00           120
;   CPC      6128 ;  464  ,  664
9F00           130  CHRSKP EQU  &de4d ; &dd61 , &de52
9F00           140  TESTER EQU  &eecf ; &ee04 , &eed4
9F00           150  ASSEMB EQU  &e7a5 ; &e6c6 , &e7aa
9F00 DF        160         RST  &18 ; Far Call
9F01 0000      170         DW   vektor ; da das BASIC ROM
9F03 C9        180         RET   ; eingeschaltet sein muss
**** Zeile  170  : VEKTOR=&9F04
9F04 0000      190  VEKTOR DW   start
9F06 FD        200         DB   253 ; low ROM off, upp ROM on
**** Zeile  190  : START=&9F07
9F07 EB        210  START  EX   de,hl
; uebergebene Adresse nach HL
9F08 23        220         INC  hl ; Stringlaenge ueberlesen
9F09 5E        230         LD   e,(hl)
; Lo Byte Stringadresse
9F0A 23        240         INC  hl
9F0B 56        250         LD   d,(hl)
; Hi Byte Stringadresse
9F0C EB        260         EX   de,hl ; Stringadresse nach HL
9F0D CD4DDE    270         CALL chrskp
9F10 B7        280         OR   a
9F11 C8        290         RET  z
```

```
9F12 CDCFEE   300          CALL tester
9F15 D0       310          RET  nc
9F16 CDA5E7   320          CALL assemb
; Zeile uebersetzen und einfuegen; HL muss auf erstes Byte d
er ASCII Zeile zeigen
9F19 C9       330          RET   ; Fertig !!!

Programm :liner
Start : &9F00    Ende : &9F19
Laenge : 001A
 0  Fehler
Variablentabelle :
CHRSKP DE4D  TESTER EECF  ASSEMB E7A5  VEKTOR 9F04
START  9F07
```

Im Folgenden finden Sie das Maschinenprogramm in ein Programm eingebunden, das aus Maschinenprogrammen, die als Objekt Code im Speicher stehen, BASIC Lader erzeugen kann. Das ist sinnvoll, um Maschinenprogramme auf einfache Weise weiterzugeben, auch an Nichtbesitzer eines Assemblers.

```
10 MEMORY &9EFF
20 lin=&9F00
30 FOR i=lin TO lin+&19:READ a$:w=VAL("&"+a$)
40 s=s+w:POKE i,w:NEXT
50 IF s<> 4273 THEN PRINT"Fehler in Datas":END
51 '464: if s<> 4121 then ....
52 '664: if s<> ???? then ....
60 PRINT"ok!":s=0
70 DATA DF,04,9F,C9,07,9F,FD,EB
80 DATA 23,5E,23,56,EB,CD,4D,DE
81 '464:..,..,..,..,..,..,61,DD
82 '664:..,..,..,..,..,..,52,DE
90 DATA B7,C8,CD,CF,EE,D0,CD,A5
91 '464:..,..,..,04,..,..,..,C6
92 '664:..,..,..,9B,EF,..,..,AA
100 DATA E7,C9
101 '464:E6,..
102 '664:E7,..
110 INPUT"Startadresse ",st
120 INPUT"Endadresse   ",en
130 az$=CHR$(34):REM Anfuehrungszeichen
140 nu$=CHR$(0)+CHR$(0):REM Endekennzeichen
150 a$="1000 for i=&"+HEX$(st,4)+" to &"+HEX$(en,4)+nu$
160 CALL lin,@a$
170 a$="1010 read a$:w=val("+az$+"&H"+az$+"+a$)"+nu$
180 CALL lin,@a$
190 a$="1020 s=s+w:poke i,w:next"+nu$
200 CALL lin,@a$
210 z=1050
220 i=st
230 a$=""
240 j=0
250 w=PEEK(i+j):s=s+w
260 a$=a$+HEX$(w,2)+","
270 IF i+j<en THEN j=j+1:IF j<8 THEN 250 ELSE ELSE ef=-1
280 a$=LEFT$(a$,LEN(a$)-1)
290 a$=STR$(z)+" data "+a$+nu$
300 CALL lin,@a$
310 z=z+10
```

```
320 IF NOT ef THEN i=i+8:GOTO 230
330 a$="1030 if s<>"+STR$(s)+"then print"+az$+"Fehler in Dat
as"+az$+":end"+nu$
340 CALL lin,@a$
350 a$="1040 print"+az$+"ok!"+az$+":end"+nu$
360 CALL lin,@a$
370 DELETE 10-380
380 END
```

Obiges Programm war das letzte in diesem Buch. Natürlich gäbe es noch viel bezüglich der praktischen Programmierung zu erklären, besonders wichtig ist jedoch das selbständige Üben. Um Ihnen dabei weitere Anregungen zu geben, empfehlen wir das Buch "CPC Tips & Tricks Band II" von DATA BECKER, in dem viele Anwendungen von Maschinensprache, wie zum Beispiel das XREF Programm, mit dem die Variablenübersicht des Disassemblers/Simulators erstellt wurde, enthalten sind.

Bei der weiteren "Hackerei" wünschen wir:

Viel Spaß!

KAPITEL VII: PERSPEKTIVEN

Sie haben die grundsätzlichen Programmiertechniken und Hilfsprogramme zur Erstellung von Maschinenprogrammen kennengelernt.

Programmierung in Assembler ist für größere Programmprobleme unerläßlich. Die Entwicklungszeiten für Software sind jedoch viel länger, als die, für Programme in höheren Sprachen. Aus diesem Grund sind gute Entwicklungsprogramme für die effektive Programmierung notwendig.

Die Eigenschaften solcher Programme werden wir kurz besprechen. Zu einem Programmpaket zur Entwicklung von Maschinenprogrammen, gehört mindestens ein Assemblerprogramm und ein umfangreiches Monitorprogramm.

Der Assembler ist die Voraussetzung, zur Entwicklung größerer Programme. Zusätzlich zu den Ihnen bekannten Pseudobefehlen, bieten viele Assembler Möglichkeiten, die die Programmentwicklung noch weiter vereinfachen, z.B gehört dazu die Definition von Macros, bedingt zu assemblieren und auf externe Programme bzw. Variablen zuzugreifen.

Macros:

Oft ist es der Fall, daß eine bestimmte Folge von Befehlen mehrmals in einem Programm vorkommt. Durch die Benutzung von Macros wird vermieden, daß Sie in solchen Fällen ein und dieselbe Befehlsfolge immer wieder eingeben. Mit Hilfe einer Macrodefinition kann einer Befehlsfolge ein Name gegeben werden. Dann kann im Sourceprogramm anstelle der Befehlsfolge einfach der Macroname gesetzt werden.

Der Assembler ersetzt den Macronamen automatisch durch die zugeordnete Befehlsfolge. Auch Sourceprogramme werden durch die Benutzung von Macros übersichtlicher und kürzer.

Bedingte Assemblierung:

Bei der bedingten Assemblierung ist es möglich bestimmte Teile des Programms in Abhängigkeit von einer Bedingung zu übersetzen. Die bedingte Assemblierung macht es möglich, daß ein allgemeines Sourceprogramm wie eine Dateiverwaltung geschrieben und dann auf die jeweilige Anwendung zugeschnitten werden.

Externe Programme und Variablen

Bei der Programmierung in Assembler ist es sehr sinnvoll strukturiert zu programmieren. Das bedeutet, daß größere Probleme in viele kleine unterteilt werden und jeder Programmabschnitt für sich erstellt wird. Oft tauchen ständig dieselben Unterprogramme auf, z.B. benutzten wir die Routine zur hexadezimalen Ausgabe eines Zeichens in verschiedenen Programmen. Solche oft benötigten Routinen und häufig verwendeten Variablen bilden bei einem komfortablen Assembler eine Programm/Variablen Bibliothek. Die Routinen werden durch ihren Namen im Sourceprogramm gekennzeichnet und dann von Cassette/Diskette automatisch nachgeladen und in das Objektprogramm eingefügt.

Das Programm, das die Verbindung von verschiedenen Assemblerprogrammen durchführt, bezeichnet man auch als "Linker" (link engl.: verbinde). Damit verbunden ist meist noch ein sogenannter Relocator (Verschieber), der die Adressen, die sich durch das Einfügen und Verschieben der Programme ändern, wieder korrigiert. Programme, die diese Fähigkeit auch enthalten sind allerdings sehr umfangreich und ralativ teuer. Die Programmierung wird dafür aber auch um ein Vielfaches komfortabler und schneller. Zudem besitzen viele Assembler einen eigenen Editor, d.h. die Eingabe der Assemblerbefehle ist nicht mehr an eine Zeilennummer gebunden.

Es gibt noch einige andere zusätzliche Hilfsprogramme zum Assembler. Die meisten werden zu einem Monitor zusammengefaßt. Die Standardroutinen eines Monitors haben Sie kennengelernt. Der Disassembler ist meist im Monitorprogramm integriert. Ein

wichtiges Merkmal eines Monitors sind seine Möglichkeiten zum Testen von Programmen.

Die Möglichkeit einen Breakpoint zu setzen ist die einfachste der Testmöglichkeiten. Umfangreichere Testroutinen werden oft zu einem sogenannten Debugger (Fehlerbeseitiger) zusammengefaßt. Das wichtigste Programm in diesem Zusammenhang ist der Einzelschrittsimulator, der der TRON Funktion des Schneider BASIC auf Maschinensprache übertragen, entspricht.

Mit dem Besitz guter Hilfsprogramme zur Softwareentwicklung ist es jedoch nicht getan. Viel wichtiger ist der Schritt in die Praxis des Programmierens. Dieses Buch hat Ihnen grundlegende Techniken vermittelt, die zur Programmierung des Z80A notwendig sind. Erst durch das Praktizieren werden Sie die Maschinensprache richtig lernen. Bei der Erstellung Ihrer eigenen Maschinenprogramme wünschen wir nochmals:

Viel S p a ß !!!

ANHANG

NÜTZLICHE SYSTEMROUTINEN

Adresse &0000: RST 0 : RESET

Aufruf mit CALL 0. Wirkt wie ein An-/Ausschalten

Adresse &0008: RST &08- Low Jump

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

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

Bit 14 und 15 von HL üben dieselbe Funktion wie bei RST &08 aus.

Adresse &0010: RST &10 : Side Call

Dient zum Aufruf einer Routine in einem Expansionsrom

Adresse &0018: RST &18- Far Call

Dient zum Aufruf einer Routine irgendwo im ROM oder RAM. (Siehe Kapitel 6.2)

Adresse &0020: RST &20- RAM Lam

Der Akku wird mit dem Wert der Adresse auf die HL zeigt geladen. Dabei ist immer RAM selektiert.

Adresse &0028: RST &28- Firm Jump

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

Adresse &0030: RST &30- User Restart

Dieser steht für eigene Programme zur Verfügung

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

ASCII Code der gedrückten Taste wird im Akku zurückgegeben

Adresse &BB24: KM - Get Joystick

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

Adresse &BB5A: TXT Output

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

Adresse &BB6C: TXT Clear Window

Löscht das aktuelle Bildschirmfenster

Adresse &BB75: TXT Set Cursor

H/L entspricht Zeile/Spalte

Adresse &BB78: TXT Get Cursor

Adresse &BB81: TXT Cursor On

Adresse &BB84: TXT Cursor Off

Adresse &BBC0: GRA Move Absolut

DE Register ist X Koordinate, HL Register ist Y Koordinate

Adresse &BBC6: GRA Ask Cursor

Registerbelegung wie bei Move Absolut

Adresse &BBEA: GRA Plot Absolut

Adresse &BC9B: CAS (bzw. Disk) Cas Catalog

Adresse &BCD1: KL (Kernel) Log Ext

Bindet RSX Erweiterungen ein (siehe Kapitel 6.2).

Adresse &BD0D: KL Time please

Gibt den Wert des Timers als 4 Byte Wert in DE und HL zurück

Adresse &BD10: KL Time Set

Adresse &BD1C: MC (Machine) Set Screen Mode

Setzt den Bildschirmmodus auf den Wert des Akkus

Adresse &BD2B: MC Print Char

Gibt den Wert des Akkus zum Drucker aus

Adresse &BD37: Jump Restore : Notbremse

Setzt alle eventuell verbogenen Sprungvektoren auf ihre Ausgangswerte zurück

UMRECHNUNGSTABELLE

dezimal	hex	binär	dezimal	hex	binär
0	&00	&X00000000	26	&1A	&X00011010
1	&01	&X00000001	27	&1B	&X00011011
2	&02	&X00000010	28	&1C	&X00011100
3	&03	&X00000011	29	&1D	&X00011101
4	&04	&X00000100	30	&1E	&X00011110
5	&05	&X00000101	31	&1F	&X00011111
6	&06	&X00000110	32	&20	&X00100000
7	&07	&X00000111	33	&21	&X00100001
8	&08	&X00001000	34	&22	&X00100010
9	&09	&X00001001	35	&23	&X00100011
10	&0A	&X00001010	36	&24	&X00100100
11	&0B	&X00001011	37	&25	&X00100101
12	&0C	&X00001100	38	&26	&X00100110
13	&0D	&X00001101	39	&27	&X00100111
14	&0E	&X00001110	40	&28	&X00101000
15	&0F	&X00001111	41	&29	&X00101001
16	&10	&X00010000	42	&2A	&X00101010
17	&11	&X00010001	43	&2B	&X00101011
18	&12	&X00010010	44	&2C	&X00101100
19	&13	&X00010011	45	&2D	&X00101101
20	&14	&X00010100	46	&2E	&X00101110
21	&15	&X00010101	47	&2F	&X00101111
22	&16	&X00010110	48	&30	&X00110000
23	&17	&X00010111	49	&31	&X00110001
24	&18	&X00011000	50	&32	&X00110010
25	&19	&X00011001	51	&33	&X00110011

UMRECHNUNGSTABELLE DEZIMAL - HEXADEZIMAL - BINÄR

dezimal	hex	binär	dezimal	hex	binär
52	&34	&X00110100	78	&4E	&X01001110
53	&35	&X00110101	79	&4F	&X01001111
54	&36	&X00110110	80	&50	&X01010000
55	&37	&X00110111	81	&51	&X01010001
56	&38	&X00111000	82	&52	&X01010010
57	&39	&X00111001	83	&53	&X01010011
58	&3A	&X00111010	84	&54	&X01010100
59	&3B	&X00111011	85	&55	&X01010101
60	&3C	&X00111100	86	&56	&X01010110
61	&3D	&X00111101	87	&57	&X01010111
62	&3E	&X00111110	88	&58	&X01011000
63	&3F	&X00111111	89	&59	&X01011001
64	&40	&X01000000	90	&5A	&X01011010
65	&41	&X01000001	91	&5B	&X01011011
66	&42	&X01000010	92	&5C	&X01011100
67	&43	&X01000011	93	&5D	&X01011101
68	&44	&X01000100	94	&5E	&X01011110
69	&45	&X01000101	95	&5F	&X01011111
70	&46	&X01000110	96	&60	&X01100000
71	&47	&X01000111	97	&61	&X01100001
72	&48	&X01001000	98	&62	&X01100010
73	&49	&X01001001	99	&63	&X01100011
74	&4A	&X01001010	100	&64	&X01100100
75	&4B	&X01001011	101	&65	&X01100101
76	&4C	&X01001100	102	&66	&X01100110
77	&4D	&X01001101	103	&67	&X01100111

UMRECHNUNGSTABELLE DEZIMAL - HEXADEZIMAL - BINÄR

dezimal	hex	binär	dezimal	hex	binär
104	&68	&X01101000	130	&82	&X10000010
105	&69	&X01101001	131	&83	&X10000011
106	&6A	&X01101010	132	&84	&X10000100
107	&6B	&X01101011	133	&85	&X10000101
108	&6C	&X01101100	134	&86	&X10000110
109	&6D	&X01101101	135	&87	&X10000111
110	&6E	&X01101110	136	&88	&X10001000
111	&6F	&X01101111	137	&89	&X10001001
112	&70	&X01110000	138	&8A	&X10001010
113	&71	&X01110001	139	&8B	&X10001011
114	&72	&X01110010	140	&8C	&X10001100
115	&73	&X01110011	141	&8D	&X10001101
116	&74	&X01110100	142	&8E	&X10001110
117	&75	&X01110101	143	&8F	&X10001111
118	&76	&X01110110	144	&90	&X10010000
119	&77	&X01110111	145	&91	&X10010001
120	&78	&X01111000	146	&92	&X10010010
121	&79	&X01111001	147	&93	&X10010011
122	&7A	&X01111010	148	&94	&X10010100
123	&7B	&X01111011	149	&95	&X10010101
124	&7C	&X01111100	150	&96	&X10010110
125	&7D	&X01111101	151	&97	&X10010111
126	&7E	&X01111110	152	&98	&X10011000
127	&7F	&X01111111	153	&99	&X10011001
128	&80	&X10000000	154	&9A	&X10011010
129	&81	&X10000001	155	&9B	&X10011011

UMRECHNUNGSTABELLE DEZIMAL - HEXADEZIMAL - BINÄR

dezimal	hex	binär	dezimal	hex	binär
156	&9C	&X10011100	182	&B6	&X10110110
157	&9D	&X10011101	183	&B7	&X10110111
158	&9E	&X10011110	184	&B8	&X10111000
159	&9F	&X10011111	185	&B9	&X10111001
160	&A0	&X10100000	186	&BA	&X10111010
161	&A1	&X10100001	187	&BB	&X10111011
162	&A2	&X10100010	188	&BC	&X10111100
163	&A3	&X10100011	189	&BD	&X10111101
164	&A4	&X10100100	190	&BE	&X10111110
165	&A5	&X10100101	191	&BF	&X10111111
166	&A6	&X10100110	192	&C0	&X11000000
167	&A7	&X10100111	193	&C1	&X11000001
168	&A8	&X10101000	194	&C2	&X11000010
169	&A9	&X10101001	195	&C3	&X11000011
170	&AA	&X10101010	196	&C4	&X11000100
171	&AB	&X10101011	197	&C5	&X11000101
172	&AC	&X10101100	198	&C6	&X11000110
173	&AD	&X10101101	199	&C7	&X11000111
174	&AE	&X10101110	200	&C8	&X11001000
175	&AF	&X10101111	201	&C9	&X11001001
176	&B0	&X10110000	202	&CA	&X11001010
177	&B1	&X10110001	203	&CB	&X11001011
178	&B2	&X10110010	204	&CC	&X11001100
179	&B3	&X10110011	205	&CD	&X11001101
180	&B4	&X10110100	206	&CE	&X11001110
181	&B5	&X10110101	207	&CF	&X11001111

UMRECHNUNGSTABELLE DEZIMAL - HEXADEZIMAL - BINÄR

dezimal	hex	binär	dezimal	hex	binär
208	&D0	&X11010000	234	&EA	&X11101010
209	&D1	&X11010001	235	&EB	&X11101011
210	&D2	&X11010010	236	&EC	&X11101100
211	&D3	&X11010011	237	&ED	&X11101101
212	&D4	&X11010100	238	&EE	&X11101110
213	&D5	&X11010101	239	&EF	&X11101111
214	&D6	&X11010110	240	&F0	&X11110000
215	&D7	&X11010111	241	&F1	&X11110001
216	&D8	&X11011000	242	&F2	&X11110010
217	&D9	&X11011001	243	&F3	&X11110011
218	&DA	&X11011010	244	&F4	&X11110100
219	&DB	&X11011011	245	&F5	&X11110101
220	&DC	&X11011100	246	&F6	&X11110110
221	&DD	&X11011101	247	&F7	&X11110111
222	&DE	&X11011110	248	&F8	&X11111000
223	&DF	&X11011111	249	&F9	&X11111001
224	&E0	&X11100000	250	&FA	&X11111010
225	&E1	&X11100001	251	&FB	&X11111011
226	&E2	&X11100010	252	&FC	&X11111100
227	&E3	&X11100011	253	&FD	&X11111101
228	&E4	&X11100100	254	&FE	&X11111110
229	&E5	&X11100101	255	&FF	&X11111111
230	&E6	&X11100110			
231	&E7	&X11100111			
232	&E8	&X11101000			
233	&E9	&X11101001			

	0	1	2	3	4	5	6	7
0	NOP	LD BC,nn	LD (BC),A	INC BC	INC B	DEC B	LD B,n	RLCA
1	DJNZ of	LD DE,nn	LD (DE),A	INC DE	INC D	DEC D	LD D,n	RLA
2	JR NZ,of	LD HL,nn	LD (nn),HL	INC HL	INC H	DEC H	LD H,n	DAA
3	JR NC,of	LD SP,nn	LD (nn),A	INC SP	INC (HL)	DEC (HL)	LD (HL),n	SCF
4	LD B,B	LD B,C	LD B,D	LD B,E	LD B,H	LD B,L	LD B,(HL)	LD B,A
5	LD D,B	LD D,C	LD D,D	LD D,E	LD D,H	LD D,L	LD D,(HL)	LD D,A
6	LD H,B	LD H,C	LD H,D	LD H,E	LD H,H	LD H,L	LD H,(HL)	LD H,A
7	LD (HL),B	LD (HL),C	LD (HL),D	LD (HL),E	LD (HL),H	LD (HL),L	HALT	LD (HL),A
8	ADD A,B	ADD A,C	ADD A,D	ADD A,E	ADD A,H	ADD A,L	ADD A,(HL)	ADD A,A
9	SUB B	SUB C	SUB D	SUB E	SUB H	SUB L	SUB (HL)	SUB A
A	AND B	AND C	AND D	AND E	AND H	AND L	AND (HL)	AND A
B	OR B	OR C	OR D	OR E	OR H	OR L	OR (HL)	OR A
C	RET NZ	POP BC	JP NZ,nn	JP nn	CALL NZ,nn	PUSH BC	ADD A,n	RST & 00
D	RET NC	POP DE	JP NC,nn	OUT (n),A	CALL NC,nn	PUSH DE	SUB n	RST & 10
E	RET PO	POP HL	JP PO,nn	EX (SP),HL	CALL PO,nn	PUSH HL	AND n	RST & 20
F	RET P	POP AF	JP P,nn	DI	CALL P,nn	PUSH AF	OR n	RST & 30

	8	9	A	B	C	D	E	F
0	EX AF,AF	ADD HL,BC	LD A,(BC)	DEC BC	INC C	DEC C	LD C,n	RRCA
1	JR of	ADD HL,DE	LD A,(DE)	DEC DE	INC E	DEC E	LD E,n	RRA
2	JR Z,of	ADD HL,HL	LD HL,(nn)	DEC HL	INC L	DEC L	LD L,n	CPL
3	JR C,of	ADD HL,SP	LD A,(nn)	DEC SP	INC A	DEC A	LD A,n	CCF
4	LD C,B	LD C,C	LD C,D	LD C,E	LD C,H	LD C,L	LD C,(HL)	LD C,A
5	LD E,B	LD E,C	LD E,D	LD E,E	LD E,H	LD E,L	LD E,(HL)	LD E,A
6	LD L,B	LD L,C	LD L,D	LD L,E	LD L,H	LD L,L	LD L,(HL)	LD L,A
7	LD A,B	LD A,C	LD A,D	LD A,E	LD A,H	LD A,L	LD A,(HL)	LD A,A
8	ADC A,B	ADC A,C	ADC A,D	ADC A,E	ADC A,H	ADC A,L	ADC A,(HL)	ADC A,A
9	SBC A,B	SBC A,C	SBC A,D	SBC A,E	SBC A,H	SBC A,L	SBC A,(HL)	SBC A,A
A	XOR B	XOR C	XOR D	XOR E	XOR H	XOR L	XOR (HL)	XOR A
B	CP B	CP C	CP D	CP E	CP H	CP L	CP (HL)	CP A
C	RET Z	RET	JP Z,nn	⟶	CALL Z,nn	CALL nn	ADC A,n	RST & 08
D	RET C	EXX	JP C,nn	IN A,(n)	CALL C,nn	⟶	SBC A,n	RST & 18
E	RET PE	JP (HL)	JP PE,nn	EX DE,HL	CALL PE,nn	⟶	XOR n	RST & 28
F	RET M	LD SP,HL	JP M,nn	EI	CALL M,nn	⟶	CP n	RST & 38

	0	1	2	3	4	5	6	7
0	RLC B	RLC C	RLC D	RLC E	RLC H	RLC L	RLC (HL)	RLC A
1	RL B	RL C	RL D	RL E	RL H	RL L	RL (HL)	RL A
2	SLA B	SLA C	SLA D	SLA E	SLA H	SLA L	SLA (HL)	SLA A
3								
4	BIT 0,B	BIT 0,C	BIT 0,D	BIT 0,E	BIT 0,H	BIT 0,L	BIT 0,(HL)	BIT 0,A
5	BIT 2,B	BIT 2,C	BIT 2,D	BIT 2,E	BIT 2,H	BIT 2,L	BIT 2,(HL)	BIT 2,A
6	BIT 4,B	BIT 4,C	BIT 4,D	BIT 4,E	BIT 4,H	BIT 4,L	BIT 4,(HL)	BIT 4,A
7	BIT 6,B	BIT 6,C	BIT 6,D	BIT 6,E	BIT 6,H	BIT 6,L	BIT 6,(HL)	BIT 6,A
8	RES 0,B	RES 0,C	RES 0,D	RES 0,E	RES 0,H	RES 0,L	RES 0,(HL)	RES 0,A
9	RES 2,B	RES 2,C	RES 2,D	RES 2,E	RES 2,H	RES 2,L	RES 2,(HL)	RES 2,A
A	RES 4,B	RES 4,C	RES 4,D	RES 4,E	RES 4,H	RES 4,L	RES 4,(HL)	RES 4,A
B	RES 6,B	RES 6,C	RES 6,D	RES 6,E	RES 6,H	RES 6,L	RES 6,(HL)	RES 6,A
C	SET 0,B	SET 0,C	SET 0,D	SET 0,E	SET 0,H	SET 0,L	SET 0,(HL)	SET 0,A
D	SET 2,B	SET 2,C	SET 2,D	SET 2,E	SET 2,H	SET 2,L	SET 2,(HL)	SET 2,A
E	SET 4,B	SET 4,C	SET 4,D	SET 4,E	SET 4,H	SET 4,L	SET 4,(HL)	SET 4,A
F	SET 6,B	SET 6,C	SET 6,D	SET 6,E	SET 6,H	SET 6,L	SET 6,(HL)	SET 6,A

	8	9	A	B	C	D	E	F
0	RRC B	RRC C	RRC D	RRC E	RRC H	RRC L	RRC (HL)	RRC A
1	RR B	RR C	RR D	RR E	RR H	RR L	RR (HL)	RR A
2	SRA B	SRA C	SRA D	SRA E	SRA H	SRA L	SRA (HL)	SRA A
3	SRL B	SRL C	SRL D	SRL E	SRL H	SRL L	SRL (HL)	SRL A
4	BIT 1,B	BIT 1,C	BIT 1,D	BIT 1,E	BIT 1,H	BIT 1,L	BIT 1,(HL)	BIT 1,A
5	BIT 3,B	BIT 3,C	BIT 3,D	BIT 3,E	BIT 3,H	BIT 3,L	BIT 3,(HL)	BIT 3,A
6	BIT 5,B	BIT 5,C	BIT 5,D	BIT 5,E	BIT 5,H	BIT 5,L	BIT 5,(HL)	BIT 5,A
7	BIT 7,B	BIT 7,C	BIT 7,D	BIT 7,E	BIT 7,H	BIT 7,L	BIT 7,(HL)	BIT 7,A
8	RES 1,B	RES 1,C	RES 1,D	RES 1,E	RES 1,H	RES 1,L	RES 1,(HL)	RES 1,A
9	RES 3,B	RES 3,C	RES 3,D	RES 3,E	RES 3,H	RES 3,L	RES 3,(HL)	RES 3,A
A	RES 5,B	RES 5,C	RES 5,D	RES 5,E	RES 5,H	RES 5,L	RES 5,(HL)	RES 5,A
B	RES 7,B	RES 7,C	RES 7,D	RES 7,E	RES 7,H	RES 7,L	RES 7,(HL)	RES 7,A
C	SET 1,B	SET 1,C	SET 1,D	SET 1,E	SET 1,H	SET 1,L	SET 1,(HL)	SET 1,A
D	SET 3,B	SET 3,C	SET 3,D	SET 3,E	SET 3,H	SET 3,L	SET 3,(HL)	SET 3,A
E	SET 5,B	SET 5,C	SET 5,D	SET 5,E	SET 5,H	SET 5,L	SET 5,(HL)	SET 5,A
F	SET 7,B	SET 7,C	SET 7,D	SET 7,E	SET 7,H	SET 7,L	SET 7,(HL)	SET 7,A

	0	1	2	3	4	5	6	7
4	IN B,(C)	OUT (C),B	SBC HL,BC	LD (nn),BC	NEG	RETN	IM 0	LD I,A
5	IN D,(C)	OUT (C),P	SBC HL,DE	LD (nn),DE			IM 1	LD A,I
6	IN H,(C)	OUT (C),H	SBC HL,HL	LD (nn),HL				RRD
7			SBC HL,SP	LD (nn),SP				
8								
9								
A	LDI	CPI	INI	OUTI				
	LDIR	CPIR	INIR	OTIR				

	8	9	A	B	C	D	E	F
4	IN C,(C)	OUT (C),C	ADC HL,BC	LD BC,(nn)		RETI		LD R,A
5	IN E,(C)	OUT (C),E	ADC HL,DE	LD DE,(nn)			IM 2	LD A,R
6	IN L,(C)	OUT (C),L	ADC HL,HL	LD HL,(nn)				RLD
7	IN A,(C)	OUT (C),A	ADC HL,SP	LD SP,(nn)				
8								
9								
A	LDD	CPD	IND	OUTD				
B	LDDR	CPDR	INDR	OTDR				

Erklärung zu den folgenden Tabellen:

In der ersten Tabelle stehen für die Codes &CB, &ED, &DD und &FD Pfeile. Das hat folgende Bedeutung:

&CB: Ist der erste zu übersetzende Code &CB, so muß der zweite Code in der zweiten Tabelle nachgeschlagen werden. Diese Befehle sind die Rotier- und Schiebebefehle.

&ED: Ist der erste zu übersetzende Code &ED, so muß der zweite Code in der dritten Tabelle nachgeschlagen werden.

&DD und &FD: Ist der erste Code &ED oder &FD, so handelt es sich um indiziert adressierte Befehle. Bei &DD ist das IX Register betroffen, und bei &FD ist das IY Register betroffen.
Die indiziert adressierten Befehle sind nicht in einer weiteren Tabelle aufgeführt. Sie können aus den vorhandenen Tabellen in folgender Weise ermittelt werden:
Der zweite Code wird wie üblich in den Tabellen nachgeschaut. Der erhaltene Befehl muß das HL Register enthalten. Kommt das HL Register nicht im Operanden vor, oder wurde der EX DE,HL Befehl ermittelt, handelt es sich um einen ungültigen Befehl (wird vom Disassembler als ??? ausgegeben). Handelt es sich um einen gültigen Befehl, muß das HL Register durch IX bzw. IY ersetzt werden.

Aus HL wird IX bzw. IY
Aus (HL) wird (IX+d) bzw. (IY+d), wobei das durch den dritten Code gegeben ist.

Diese Regeln gelten, mit Ausnahme des JP (HL) Befehls, für alle Befehle die HL enthalten. Aus JP (HL) wird, obwohl HL in Klammern steht, nach dem Einsetzen der Indexregister JP (IX) bzw. JP (IY).

		SOURCE															
		IMPLIED		REGISTER							REG INDIRECT			INDEXED		EXT. ADDR.	IMME.
		I	R	A	B	C	D	E	H	L	(HL)	(BC)	(DE)	(IX + d)	(IY + d)	(nn)	n
DESTINATION: REGISTER	A	ED 57	ED 5F	7F	78	79	7A	7B	7C	7D	7E	0A	1A	DD 7E d	FD 7E d	3A n n	3E n
	B			47	40	41	42	43	44	45	46			DD 46 d	FD 46 d		06 n
	C			4F	48	49	4A	4B	4C	4D	4E			DD 4E d	FD 4E d		0E n
	D			57	50	51	52	53	54	55	56			DD 56 d	FD 56 d		16 n
	E			5F	58	59	5A	5B	5C	5D	5E			DD 5E d	FD 5E d		1E n
	H			67	60	61	62	63	64	65	66			DD 66 d	FD 66 d		26 n
	L			6F	68	69	6A	6B	6C	6D	6E			DD 6E d	FD 6E d		2E n
REG INDIRECT	(HL)			77	70	71	72	73	74	75							36 n
	(BC)			02													
	(DE)			12													
INDEXED	(IX+d)			DD 77 d	DD 70 d	DD 71 d	DD 72 d	DD 73 d	DD 74 d	DD 75 d							DD 36 d n
	(IY+d)			FD 77 d	FD 70 d	FD 71 d	FD 72 d	FD 73 d	FD 74 d	FD 75 d							FD 36 d n
EXT. ADDR	(nn)			32 n n													
IMPLIED	I			ED 47													
	R			ED 4F													

		SOURCE									
		REGISTER							IMM. EXT.	EXT. ADDR.	REG. INDIR.
		AF	BC	DE	HL	SP	IX	IY	nn	(nn)	(SP)
DESTINATION REGISTER	AF										F1
	BC								01 n n	ED 4B n n	C1
	DE								11 n n	ED 5B n n	D1
	HL								21 n n	2A n n	E1
	SP				F9		DD F9	FD F9	31 n n	ED 7B n n	
	IX								DD 21 n n	DD 2A n n	DD E1
	IY								FD 21 n n	FD 2A n n	FD E1
EXT. ADDR.	(nn)		ED 43 n n	ED 53 n n	22 n n	ED 73 n n	DD 22 n n	FD 22 n n			
PUSH INSTRUCTIONS → REG. IND.	(SP)	F5	C5	D5	E5		DD E5	FD E5			

↑ POP INSTRUCTIONS

NOTE: The Push & Pop Instructions adjust the SP after every execution

		IMPLIED ADDRESSING				
		AF′	BC′, DE′ & HL′	HL	IX	IY
IMPLIED	AF	08				
	BC, DE & HL		D9			
	DE			EB		
REG. INDIR.	(SP)			E3	DD E3	FD E3

			SOURCE	
			REG. INDIR.	
			(HL)	
DESTINATION	REG. INDIR.	(DE)	ED A0	'LDI' – Load (DE)←(HL) Inc HL & DE, Dec BC
			ED B0	'LDIR,' – Load (DE)←(HL) Inc HL & DE, Dec BC, Repeat until BC = 0
			ED A8	'LDD' – Load (DE)←(HL) Dec HL & DE, Dec BC
			ED B8	'LDDR' – Load (DE)←(HL) Dec HL & DE, Dec BC, Repeat until BC = 0

Reg HL points to source
Reg DE points to destination
Reg BC is byte counter

SEARCH LOCATION	
REG. INDIR.	
(HL)	
ED A1	'CPI' Inc HL, Dec BC
ED B1	'CPIR', Inc HL, Dec BC repeat until BC = 0 or find match
ED A9	'CPD' Dec HL & BC
ED B9	'CPDR' Dec HL & BC Repeat until BC = 0 or find match

HL points to location in memory
to be compared with accumulator
contents
BC is byte counter

SOURCE

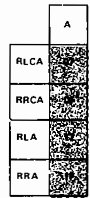
	REGISTER ADDRESSING							REG. INDIR.	INDEXED		IMMED.
	A	B	C	D	E	H	L	(HL)	(IX+d)	(IY+d)	n
'ADD'	87	80	81	82	83	84	85	86	DD 86 d	FD 86 d	C6 n
ADD w CARRY 'ADC'	8F	88	89	8A	8B	8C	8D	8E	DD 8E d	FD 8E d	CE n
SUBTRACT 'SUB'	97	90	91	92	93	94	95	96	DD 96 d	FD 96 d	D6 n
SUB w CARRY 'SBC'	9F	98	99	9A	9B	9C	9D	9E	DD 9E d	FD 9E d	DE n
'AND'	A7	A0	A1	A2	A3	A4	A5	A6	DD A6 d	FD A6 d	E6 n
'XOR'	AF	A8	A9	AA	AB	AC	AD	AE	DD AE d	FD AE d	EE n
'OR'	B7	B0	B1	B2	B3	B4	B5	B6	DD B6 d	FD B6 d	F6 n
COMPARE 'CP'	BF	B8	B9	BA	BB	BC	BD	BE	DD BE d	FD BE d	FE n
INCREMENT 'INC'	3C	04	0C	14	1C	24	2C	34	DD 34 d	FD 34 d	
DECREMENT 'DEC'	3D	05	0D	15	1D	25	2D	35	DD 35 d	FD 35 d	

Decimal Adjust Acc, 'DAA'	27
Complement Acc, 'CPL'	2F
Negate Acc, 'NEG' (2's complement)	ED 44
Complement Carry Flag, 'CCF'	3F
Set Carry Flag, 'SCF'	37

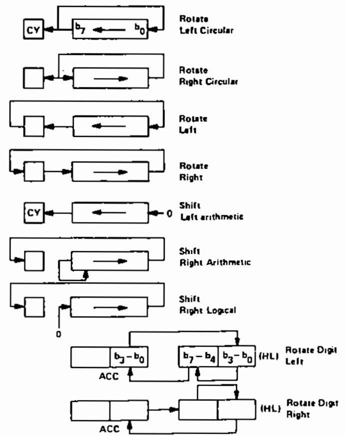
		SOURCE					
DESTINATION		BC	DE	HL	SP	IX	IY
'ADD'	HL	09	19	29	39		
	IX	DD 09	DD 19		DD 39	DD 29	
	IY	FD 09	FD 19		FD 39		FD 29
ADD WITH CARRY AND SET FLAGS 'ADC'	HL	ED 4A	ED 5A	ED 6A	ED 7A		
SUB WITH CARRY AND SET FLAGS 'SBC'	HL	ED 42	ED 52	ED 62	ED 72		
INCREMENT 'INC'		03	13	23	33	DD 23	FD 23
DECREMENT 'DEC'		0B	1B	2B	3B	DD 2B	FD 2B

Source and Destination

TYPE OF ROTATE OR SHIFT	A	B	C	D	E	H	L	(HL)	(IX + d)	(IY + d)
'RLC'	CB 07	CB 00	CB 01	CB 02	CB 03	CB 04	CB 05	CB 06	DD CB d 06	FD CB d 06
'RRC'	CB 0F	CB 08	CB 09	CB 0A	CB 0B	CB 0C	CB 0D	CB 0E	DD CB d 0E	FD CB d 0E
'RL'	CB 17	CB 10	CB 11	CB 12	CB 13	CB 14	CB 15	CB 16	DD CB d 16	FD CB d 16
'RR'	CB 1F	CB 18	CB 19	CB 1A	CB 1B	CB 1C	CB 1D	CB 1E	DD CB d 1E	FD CB d 1E
'SLA'	CB 27	CB 20	CB 21	CB 22	CB 23	CB 24	CB 25	CB 26	DD CB d 26	FD CB d 26
'SRA'	CB 2F	CB 28	CB 29	CB 2A	CB 2B	CB 2C	CB 2D	CB 2E	DD CB d 2E	FD CB d 2E
'SRL'	CB 3F	CB 38	CB 39	CB 3A	CB 3B	CB 3C	CB 3D	CB 3E	DD CB d 3E	FD CB d 3E
'RLD'								ED 6F		
'RRD'								ED 67		



	A
RLCA	[illegible]
RRCA	[illegible]
RLA	[illegible]
RRA	[illegible]



	BIT	REGISTER ADDRESSING A	B	C	D	E	H	L	REG. INDIR. (HL)	INDEXED (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 51	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	GB 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 80	CB B1	CB 82	CB 83	CB 84	CB B5	CB 86	DD CB d 86	FD CB d 86
	1	CB BF	CB 88	CB 89	CB BA	CB BB	CB 8C	CB BD	CB BE	DD CB d 8E	FD CB d 8E
	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 BB.	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 D5	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

CONDITION

			UN-COND.	CARRY	NON CARRY	ZERO	NON ZERO	PARITY EVEN	PARITY ODD	SIGN NEG	SIGN POS	REG B≠0
JUMP 'JP'	IMMED. EXT.	nn	C3 n n	DA n n	D2 n n	CA n n	C2 n n	EA n n	E2 n n	FA n n	F2 n n	
JUMP 'JR'	RELATIVE	PC+e	18 e-2	38 e-2	30 e-2	28 e-2	20 e-2					
JUMP 'JP'	REG. INDIR.	(HL)	E9									
JUMP 'JP'		(IX)	DD E9									
JUMP 'JP'		(IY)	FD E9									
'CALL'	IMMED. EXT.	nn	CD n n	DC n n	D4 n n	CC n n	C4 n n	EC n n	E4 n n	FC n n	F4 n n	
DECREMENT B, JUMP IF NON ZERO 'DJNZ'	RELATIVE	PC+e										10 e-2
RETURN 'RET'	REGISTER INDIR.	(SP) (SP+1)	C9	D8	D0	C8	C0	E8	E0	F8	F0	
RETURN FROM INT 'RETI'	REG. INDIR.	(SP) (SP+1)	ED 4D									
RETURN FROM NON MASKABLE INT 'RETN'	REG. INDIR.	(SP) (SP+1)	ED 45									

NOTE—CERTAIN FLAGS HAVE MORE THAN ONE PURPOSE. REFER TO SECTION 6.0 FOR DETAILS

CALL ADDRESS		OP CODE	
	0000_H	[illegible]	'RST 0'
	0008_H	[illegible]	'RST 8'
	0010_H	[illegible]	'RST 16'
	0018_H	[illegible]	'RST 24'
	0020_H	[illegible]	'RST 32'
	0028_H	[illegible]	'RST 40'
	0030_H	[illegible]	'RST 48'
	0038_H	[illegible]	'RST 56'

SOURCE PORT ADDRESS

INPUT DESTINATION				IMMED. (n)	REG. INDIR. (C)
	INPUT 'IN'	REG ADDRESSING	A		ED 78
			B		ED 40
			C		ED 48
			D		ED 50
			E		ED 58
			H		ED 60
			L		ED 68
BLOCK INPUT COMMANDS	'INI' – INPUT & Inc HL, Dec B	REG. INDIR	(HL)		ED A2
	'INIR' – INP, Inc HL, Dec B, REPEAT IF B≠0				ED B2
	'IND' – INPUT & Dec HL, Dec B				ED AA
	'INDR' – INPUT, Dec HL, Dec B, REPEAT IF B≠0				ED BA

SOURCE

			REGISTER							REG. IND.
			A	B	C	D	E	H	L	(HL)
'OUT'	IMMED.	(n)	[illegible]							
	REG. IND.	(C)	ED 79	ED 41	ED 49	ED 51	ED 59	ED 61	ED 69	
'OUTI' – OUTPUT Inc HL, Dec b	REG. IND.	(C)								ED A3
'OTIR' – OUTPUT, Inc HL, Dec B, REPEAT IF B≠0	REG. IND.	(C)								ED B3
'OUTD' – OUTPUT Dec HL & B	REG. IND.	(C)								ED AB
'OTDR' – OUTPUT, Dec HL & B, REPEAT IF B≠0	REG. IND.	(C)								ED BB

BLOCK OUTPUT COMMANDS (OUTI, OTIR, OUTD, OTDR)

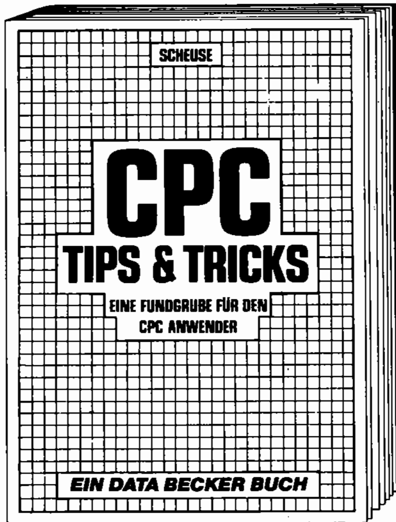
PORT DESTINATION ADDRESS (IMMED./REG. IND. columns)

'NOP'	00	
'HALT'	76	
DISABLE INT '(DI)'	F3	
ENABLE INT '(EI)'	FB	
SET INT MODE 0 'IM0'	ED 46	8080A MODE
SET INT MODE 1 'IM1'	ED 56	CALL TO LOCATION 0038_H.
SET INT MODE 2 'IM2'	ED 5E	INDIRECT CALL USING REGISTER I AND 8 BITS FROM INTERRUPTING DEVICE AS A POINTER.

Instruction	C	Z	P/V	S	N	H	Comments
ADD A, s; ADC A,s	↕	↕	V	↕	0	↕	8-bit add or add with carry
SUB s; SBC A, s, CP s, NEG	↕	↕	V	↕	1	↕	8-bit subtract, subtract with carry, compare and negate accumulator
AND s	0	↕	P	↕	0	1	Logical operations
OR s; XOR s	0	↕	P	↕	0	0	And set's different flags
INC s	•	↕	V	↕	0	↕	8-bit increment
DEC m	•	↕	V	↕	1	↕	8-bit decrement
ADD DD, ss	↕	•	•	•	0	X	16-bit add
ADC HL, ss	↕	↕	V	↕	0	X	16-bit add with carry
SBC HL, ss	↕	↕	V	↕	1	X	16-bit subtract with carry
RLA; RLCA, RRA, RRCA	↕	•	•	•	0	0	Rotate accumulator
RL m; RLC m; RR m; RRC m SLA m; SRA m; SRL m	↕	↕	P	↕	0	0	Rotate and shift location m
RLD, RRD	•	↕	P	↕	0	0	Rotate digit left and right
DAA	↕	↕	P	↕	•	↕	Decimal adjust accumulator
CPL	•	•	•	•	1	1	Complement accumulator
SCF	1	•	•	•	0	0	Set carry
CCF	↕	•	•	•	0	X	Complement carry
IN r, (C)	•	↕	P	↕	0	0	Input register indirect
INI; IND; OUTI; OUTD	•	↕	X	X	1	X	Block input and output
INIR; INDR; OTIR; OTDR	•	1	X	X	1	X	Z = 0 if B ≠ 0 otherwise Z = 1
LDI, LDD	•	X	↕	X	0	0	Block transfer instructions
LDIR, LDDR	•	X	0	X	0	0	P/V = 1 if BC ≠ 0, otherwise P/V = 0
CPI, CPIR, CPD, CPDR	•	↕	↕	↕	1	X	Block search instructions Z = 1 if A = (HL), otherwise Z = 0 P/V = 1 if BC ≠ 0, otherwise P/V = 0
LD A, I; LD A, R	•	↕	IFF	↕	0	0	The content of the interrupt enable flip-flop (IFF) is copied into the P/V flag
BIT b, s	•	↕	X	X	0	1	The state of bit b of location s is copied into the Z flag
NEG	↕	↕	V	↕	1	↕	Negate accumulator

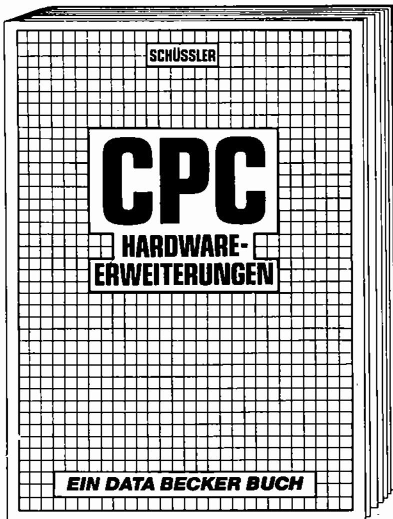
The following notation is used in this table:

Symbol	Operation
C	Carry/link flag. C=1 if the operation produced a carry from the MSB of the operand or result.
Z	Zero flag. Z=1 if the result of the operation is zero.
S	Sign flag. S=1 if the MSB of the result is one.
P/V	Parity or overflow flag. Parity (P) and overflow (V) share the same flag. Logical operations affect this flag with the parity of the result while arithmetic operations affect this flag with the overflow of the result. If P/V holds parity, P/V=1 if the result of the operation is even, P/V=0 if result is odd. If P/V holds overflow, P/V=1 if the result of the operation produced an overflow.
H	Half-carry flag. H=1 if the add or subtract operation produced a carry into or borrow from into bit 4 of the accumulator.
N	Add/Subtract flag. N=1 if the previous operation was a subtract.
	H and N flags are used in conjunction with the decimal adjust instruction (DAA) to properly correct the result into packed BCD format following addition or subtraction using operands with packed BCD format.
↕	The flag is affected according to the result of the operation.
•	The flag is unchanged by the operation.
0	The flag is reset by the operation.
1	The flag is set by the operation.
X	The flag is a "don't care."
V	P/V flag affected according to the overflow result of the operation.
P	P/V flag affected according to the parity result of the operation.
r	Any one of the CPU registers A, B, C, D, E, H, L.
s	Any 8-bit location for all the addressing modes allowed for the particular instruction. .
ss	Any 16-bit location for all the addressing modes allowed for that instruction.
ii	Any one of the two index registers IX or IY.
R	Refresh counter.
n	8-bit value in range <0, 255>
nn	16-bit value in range <0, 65535>
m	Any 8-bit location for all the addressing modes allowed for the particular instruction.



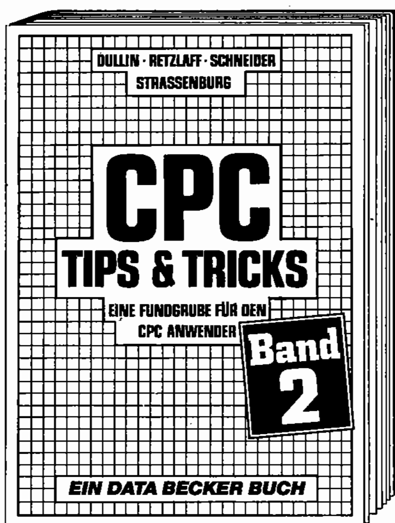
Rund um den CPC 464 viele Anregungen und wichtige Hilfen! Von Hardwareaufbau, Betriebssystem, BASIC-Tokens, Anwendungen der Windowtechnik und sehr vielen interessanten Programmen bis zu einer umfangreichen Dateiverwaltung, Soundeditor, komfortablem Zeichengenerator und kompletten Listings spannender Spiele bietet dieses Buch eine Fülle von Möglichkeiten. Diese Tips kommen von den DATA BECKER Spezialisten!

Englisch/Germer/Scheuse/ Thrun
CPC 464 Tips & Tricks
271 Seiten, DM 39,–
ISBN 3-89011-039-8



Speziell für den Hobbyelektroniker, der mehr aus seinem CPC machen möchte! Von nützlichen Tips zur Platinenherstellung über Adreßdecodierung, Adapterkarten und Interfaces bis zu EPROM-Programmierboard und -Programmiernetzteil oder Motorsteuerung für Gleich- und Schrittschaltmotoren werden machbare Erweiterungen ausführlich und praxisnahh beschrieben. Am besten gleich anfangen!

Schüssler
CPC Hardware-Erweiterungen
445 Seiten, DM 49,–
ISBN 3-89011-083-5



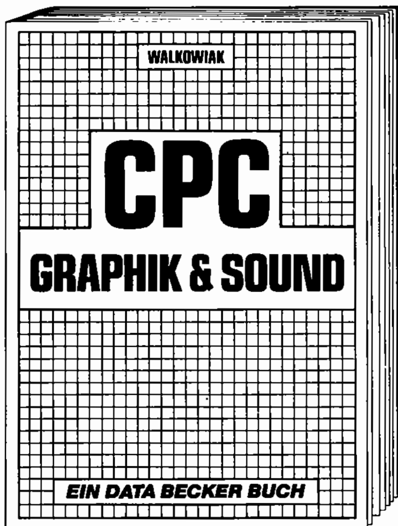
Der 2. Band CPC Tips & Tricks ist für alle CPC Besitzer interessant. Ob sie nun einen 464, 664 oder 6128 besitzen! Aus dem Inhalt: Menuegenerator, Maskengenerator, BASIC-Befehlserweiterungen, Programmierhilfen wie Dump, BASIC-Zeile von BASIC aus erzeugen, wichtige Systemroutinen und deren Nutzung, Beschleunigung von Programmen u.v.m. Wer noch mehr über seinen CPC wissen will, der kommt an diesem Buch nicht vorbei!

Dullin/Straßenburg/Retzlaff
CPC Tips & Tricks Band II
über 250 Seiten, DM 39,–

ISBN 3-89011-131-9

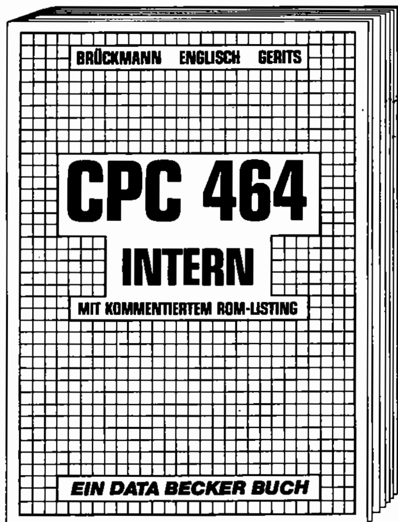
DFÜ für Jedermann mit dem CPC bietet eine ausführliche und verständliche Einführung in das Gebiet der Datenfernübertragung: was ist DFÜ, BTX, DATEX, Mailbox, alles über Modems und Koppler. Begriffserklärung: Originate, Answer, Half-Duplex usw. eine serielle Schnittstelle am CPC, RS-232/V.24 simuliert, Mailboxsoftware – selbstgestrickt, Postbestimmungen u.v.m. Steigen Sie mit diesem Buch in die Welt der Datennetze und Datenfernübertragung ein!

Serverin
DFÜ für Jedermann zum CPC
über 250 Seiten, DM 39,–
ISBN 3-89011-141-6



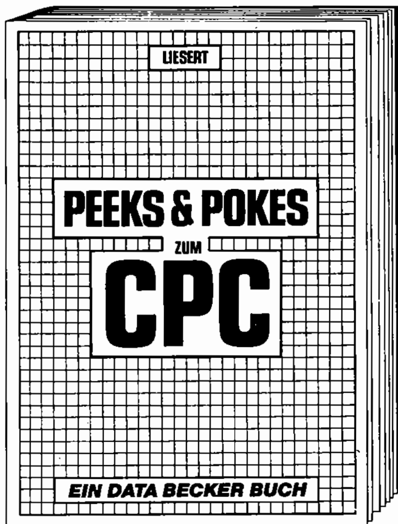
Nutzen Sie die außergewöhnlichen Grafik- und Soundmöglichkeiten des CPC 464! Natürlich mit vielen interessanten Beispielen und Programmen: Grafikgrundlagen, Sprites, Shapes, Strings, mehrfarbige Darstellungen, Koordinationstransformation, Verschiebungen, Drehungen, Rotation, 3-D-Funktionsplotter, CAD, Synthesizer, Miniorgel, Hüllkurven u.v.m. Dieses Buch wird Sie begeistern!

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



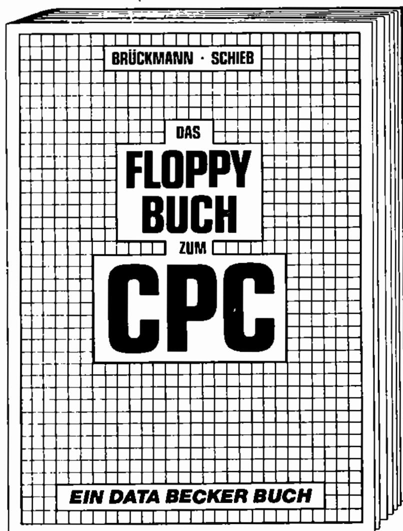
Wirklich alle Geheimnisse des CPC 464 lüftet dieses Standardwerk: Neben dem kommentierten BASIC-ROM-Listing enthält es Kapitel zu Speicheraufteilung, Prozessor, Besonderheiten des Z80, Gate Array, Video-Controller und Video-Ram, Soundchip, Schnittstellen, Betriebssystem, Routinenutzung, Character-Generator, u.v.m. Für den fortgeschrittenen BASIC-Programmierer unentbehrlich, für den Assembler-Programmierer ein absolutes Muß!

Brückmann/Englisch/Gerits
CPC 464 Intern mit kommentiertem ROM-Listing
548 Seiten, DM 69,–
ISBN 3-89011-080-0



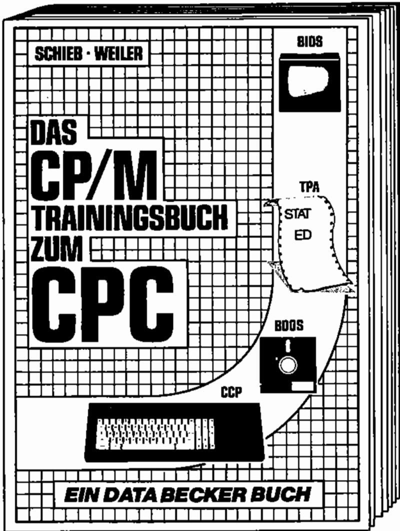
Wer PEEKS und POKES zum CPC 464 kennen und anwenden will, der findet hier umfassende Information! Sie reicht vom Adreßbereich des Prozessors über Betriebssystem und Interpreter bis hin zur Einführung in die Maschinensprache. Dazu Programmierhilfen, Routinen sowie reichlich Material zu den Themen Grafikfunktionen, Massenspeicherung und Peripherie, Tricks und Formeln in BASIC und RAM-Pages!

Liesert
Peeks & Pokes zum CPC
180 Seiten, DM 29,–
ISBN 3-89011-092-4



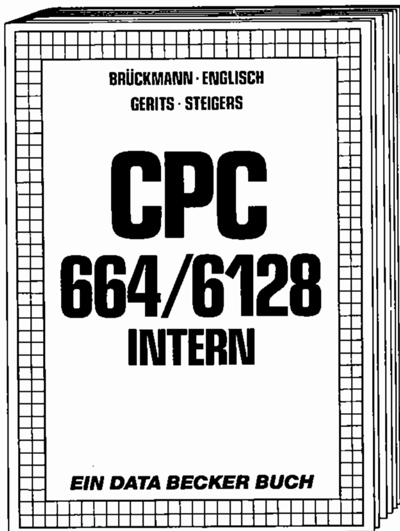
Alles über Floppyprogrammierung vom Einsteiger bis zum Profi. Natürlich mit ausführlichem ROM-Listing, einer äußerst komfortablen Dateiverwaltung, einem hilfreichen Disk-Monitor und einem ausgesprochen nützlichen Disk-Manager. Dazu eine Fundgrube verschiedener Programme und Hilfsroutinen, die das Buch für jeden Floppy-Anwender zur Pflichtlektüre machen!

Brückmann/Schieb
Das Floppy-Buch zum CPC
250 Seiten, DM 49,–
ISBN 3-89011-093-2



Endlich CP/M beherrschen! Von grundsätzlichen Erklärungen zu Speicherung von Zahlen, Schreibschutz oder ASCII, Schnittstellen und Anwendung von CP/M-Hilfsprogrammen. Für Fortgeschrittene: Fremde Diskettenformate lesen, Erstellen von Submit-Dateien u. v. m. Dieses Buch berücksichtigt die Versionen CP/M 2.2 und 3.0 für Schneider 464, 664 und 6128.

Schieb/Weiler
Das CP/M-Trainingsbuch zum CPC
260 Seiten, DM 49,–
ISBN 3-89011-089-4



Ein Muß für jeden, der sich professionell mit dem CPC 6128 oder dem CPC 664 beschäftigt. Einführung in das System, den Prozessor, das Gate Array, den Video-Controller, den Schnittstellenbaustein 8255, den Soundchip, die Schnittstellen. Mit Disassembler und ausführlichen Kommentaren zu den Routinen von Interpreter und Betriebssystem. Ein Superbuch, wie alle Titel der INTERN-Reihe!

Gerits/Englisch/Brückmann/Steigers
CPC 6128/664 Intern
ca. 400 Seiten, DM 69,–
ISBN 3-89011-135-1

DAS STEHT DRIN:

So wird der Einstieg in die Maschinensprache leichtgemacht! Das Maschinensprachebuch zum CPC 464, 664 & 6128 ist wichtig für jeden, der wirklich Insider werden will.

Aus dem Inhalt:

- Was ist Maschinensprache
- Die Zahlensysteme
- Rechneraufbau
- Der Z80-Prozessor
- Aufbau der CPU
- Der Befehlssatz des Z80 mit ausführlichen Erklärungen
- Programmierung des Z80-Prozessors
- Kompletter Assembler zum Abtippen mit Programmbeschreibung
- Disassembler und Einzelschrittsimulator
- Wichtige Systemroutinen für eigene Programme
- Maschinensprachemonitor als Listing

UND GESCHRIEBEN HABEN DIESES BUCH:

Holger Dullin und Hardy Straßenburg sind Studenten der Biologie und Physik und engagierte Programmierer, die sich einen der ersten CPC's holten und sich sofort mit der Programmierung des Z80-Prozessors auseinandersetzten.
Außerdem sind sie Autoren folgender DATA BECKER BÜCHER: Das Maschinensprachebuch und Tips & Tricks zu MSX und das große EPSON-Drucker-Buch.

ISBN 3-89011-070-3