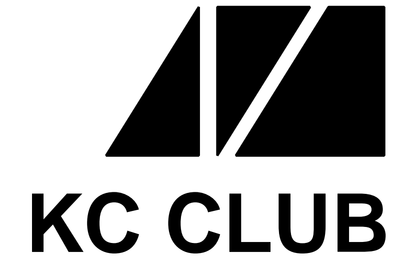
Assembler-Programmierkurs



Inhalt

Teil 1 – Aufbau eines Mikrorechners . . . ***von Frank Dachselt*** **1**

1.1 CPU . . . 1
1.2 Bussystem . . . 1
1.3 Hauptspeicher . . . 1
1.4 Ein-/Ausgabebausteine . . . 2

Teil 2 – Grundlagen der binären Informationsverarbeitung . . . ***von Frank Dachselt*** **2**

2.1 Vom Bitmuster zur Dualzahl . . . 2
2.2 Zahlensysteme . . . 2
2.3 Umrechnungen zwischen Zahlensystemen . . . 4
2.4 Rechnen mit Dualzahlen (ganze Zahlen) . . . 5
2.5 Darstellung von gebrochenen Zahlen . . . 13
2.6 Codes und nichtnumerische Informationen . . . 15
2.7 Übungsaufgaben für Teil 2 . . . 18

Teil 3 – Der Z80 . . . ***von Jörg Linder*** **19**

3.1 Prozessorarchitektur . . . 19

3.2 Befehlssatz . . . 21

3.3 Befehlssatz – Tabellarische Übersichten . . . 25

Teil 4 – Assembler . . . *von Ralf Kästner* **32**

4.1 Einleitung . . . 32

4.2 Begriffsdefinitionen . . . 33

4.3 Assembler-Entwicklungssysteme . . . 34

4.4 EDAS – Assemblerprogrammierung unter CAOS . . . 35

4.5 Assembleranweisungen (Statements) . . . 42

4.6 Syntax der Assemblersprache . . . 43

4.7 Das erste Programm „Hallo KC-Club !“ . . . 45

4.8 Hilfsmittel für die Assemblerprogrammierung . . . 48

4.9 Literatur . . . 48

Teil 5 – Einfache Datenstrukturen . . . *von Jörg Linder* **50**

5.1 Bytes . . . 50

5.2 Zeichen . . . 52

5.3 Worte . . . 54

5.4 Übungsaufgaben für Teil 5 . . . 57

Teil 6 – Einfache Programmstrukturen . . . *von Frank Dachselt* **58**

6.1 Befehlsabarbeitung . . . 58

6.2 Sequenzen . . . 59

6.3 Verzweigungen . . . 60

6.4 Schleifen . . . 63

6.5 Unterprogramme . . . 65

6.6 Beispielprogramme und Übungsaufgaben . . . 68

Teil 7 – Komplexe Datenstrukturen . . . *von Jörg Linder* **71**

7.1 Adressen und Zeiger . . . 71
7.2 Bit-Manipulationen . . . 73
7.3 Felder . . . 79
7.4 Zeichenketten . . . 100
7.5 Übungsaufgaben (I) . . . 112
7.6 Mengen . . . 115
7.7 Verbunde . . . 119
7.8 Tabellen . . . 123
7.9 Übungsaufgaben (II) . . . 128

Teil 8 – Komplexe Programmstrukturen . . . *von Frank Dachselt* **129**

8.1 Der Stack . . . 129
8.2 Unterprogramme . . . 137
8.3 Entscheidungsbäume . . . 137
8.4 Sprungverteiler . . . 137
8.5 Übungsaufgaben . . . 137

Teil 9 – Assembler unter CP/M . . . *von Mario Leubner* **146**

9.1 Einleitung . . . 147
9.2 Der Editor . . . 148
9.3 Der Assembler ASM.COM . . . 150
9.4 Der Linker LINK.COM . . . 160
9.5 Softwarebibliotheken . . . 162

9.6 BDOS & BIOS . . . 166
9.7 Weitere Hilfsmittel . . . 169
9.8 Übungsaufgaben . . . 172

Anhang A – Lösungen zu den Übungsaufgaben **A-1**
A.1 Lösungen für Teil 2 . . . A-1
A.2 Lösungen für Teil 5 . . . A-3
A.3 Lösungen für Teil 6 . . . A-6

Anhang B – Tabellen und Übersichten **B-1**
B.1 ASCII-Code . . . B-1
B.2 Flagbeeinflussung der Z80-Befehle . . . B-2

Teil 1 – Aufbau eines Mikrorechners

von Frank Dachselt

Wer mit dem Assembler programmiert, arbeitet sehr hardwarenah. Deshalb ist es unumgänglich, sich zunächst mit einigen Grundlagen der Mikrorechentechnik vertraut zu machen. Die folgenden Abschnitte sollen die wichtigsten Sachverhalte kurz darstellen. Details, die die konkrete Hardware des KC 85 betreffen sowie Programmbeispiele werden an geeigneten Stellen in den nachfolgenden Teilen des Programmierkurses ausführlich erläutert.

Zu den Besonderheiten des KC 85 in seiner Ausbaustufe mit dem D004 gehört, daß er aus zwei Teilsystemen, die sich im Grundgerät D001 und im Floppyaufsatz D004 befinden, besteht. Wenn im folgenden von einem Mikrorechner die Rede ist, dann treffen diese Aussagen im Prinzip auf jedes dieser beiden Teilsysteme zu.

Bild 1 zeigt das Blockschaltbild und damit die Grundstruktur eines Mikrorechners wie sie auch beim KC 85 zu finden ist.

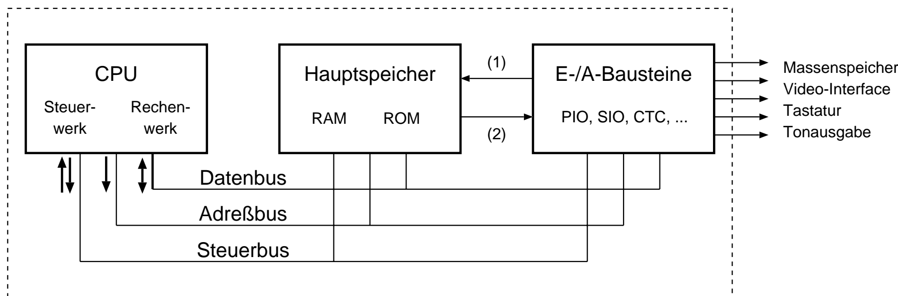


Bild 1: Blockschaltbild eines Mikrorechners

1.1 CPU

Das Herz eines Mikrorechners ist die CPU (Central Processing Unit, Zentrale Verarbeitungseinheit), die beim KC 85 bekanntermaßen aus einem integrierten Schaltkreis mit dem Namen U880 bzw. Z80 besteht. Die CPU steuert zum einen alle Vorgänge im Mikrorechner, zum anderen werden in ihr auch alle Operationen zur Verarbeitung (numerische Operationen wie Addition und Subtraktion sowie logische Operationen wie UND- und ODER-Verknüpfungen) von Daten durchgeführt. Deshalb wird die CPU mitunter auch noch einmal in die Teile Steuerwerk und Rechenwerk untergliedert.

1.2 Bussystem

Über ein Bussystem, bestehend aus Daten-, Adreß- und Steuerbus, ist die CPU mit den anderen Komponenten des Mikrorechners verbunden. Auf dem Datenbus werden Befehle und Daten zwischen der CPU und den übrigen Komponenten transportiert. Der Datenbus arbeitet bidirektional (in zwei Richtungen), d.h. es können Daten von den übrigen Komponenten zur CPU (Lesen) wie auch von der CPU zu den übrigen Komponenten (Schreiben) transportiert werden. Beim KC 85 besteht der Datenbus aus 8 einzelnen Leitungen, man sagt auch, er hat eine Breite von 8 Bit. Auf dem Adreßbus zeigt die CPU an, welche der 8 Bit breiten Informationseinheiten in den übrigen Komponenten gelesen oder beschrieben werden sollen. Der Adreßbus arbeitet unidirektional (in einer Richtung), d.h. nur die CPU kann eine Adresse auf dem Adreßbus ausgeben, die von den anderen Komponenten gelesen wird. Der KC 85 besitzt einen 16 Bit breiten Adreßbus. Der Steuerbus schließlich besteht aus einer Vielzahl einzelner Leitungen, auf denen Signale zur Steuerung der einzelnen Abläufe im Mikrorechner gebildet werden. Sie werden haupsächlich von der CPU generiert, können aber auch von anderen Komponenten ausgehen. So zeigt die CPU auf dem Steuerbus zum Beispiel an, ob Daten gelesen oder geschrieben werden sollen und ob ein Zugriff auf den Speicher oder auf einen E/A-Baustein erfolgt. Aber auch die Interruptsteuerleitungen sind Bestandteil des Steuerbusses.

1.3 Hauptspeicher

Die schon mehrfach erwähnten übrigen Komponenten des Mikrorechners sind der Hauptspeicher und die E/A-Bausteine. Im Hauptspeicher, kurz Speicher, befinden sich alle Informationen, die zur Abarbeitung eines Programmes benötigt werden. Das sind zum einen die Befehle, die von der CPU ausgeführt werden sollen, und zum anderen die Daten, die im Laufe der Programmausführung verarbeitet werden sowie natürlich die Ergebnisse dieser Verarbeitung. Der Speicher im KC 85 ist in Einheiten zu 8 Bit organisiert, die auch Speicherzellen genannt werden. Jede dieser Speicherzellen kann einzeln über eine Adresse angesprochen, d.h. gelesen oder beschrieben werden. Der Speicher eines Mikrorechners kann in RAM- und ROM-Blöcke

aufgeteilt sein. ROM (Read-only Memory, Nur-Lese-Speicher) bezeichnet dabei einen Speicherbereich, aus dem die Informationen nur gelesen und durch Schreibzugriffe nicht verändert werden können. Solche Speicher behalten auch nach dem Abschalten der Betriebsspannung ihren Inhalt. Beim KC 85 befinden sich im Grundgerät ROM-Blöcke, in denen sich das Betriebssystem und auch Anwenderprogramme befinden. Der überwiegende Teil des Speichers ist beim KC 85 jedoch als RAM (Random Access Memory, Schreib-Lese-Speicher) vorhanden. Ein solcher Speicher kann durch CPU nicht nur gelesen, sondern auch beschrieben werden.

1.4 Ein-/Ausgabebausteine

Den Verbindung des Mikrorechners zur Außenwelt erfolgt über die E/A-Bausteine. Sie bilden das Interface zu den externen Geräten zur Datenein- und -ausgabe sowie zu den Massenspeichern. Am Grundgerät D001 gehöhren dazu u.a. die Tastatur, das Kassetteninterface, die Tonausgabe, der Drucker und das Videointerface zum Anschluß des Fernsehers bzw. des Monitors. Im D004 sind das die Diskettenlaufwerke und die Festplatte. Die E/A-Bausteine stellen dabei sicher, daß sich die verschiedenen Interfaces CPU-seitig einheitlich verhalten und so den Datenaustausch über das Bussystem ermöglichen. Dabei ist wie bei den Speicherzugriffen das Lesen und Schreiben von 8 Bit breiten Einheiten über den Datenbus möglich, wobei der Adreßbus jetzt zur Adressierung, also zur Auswahl, der entsprechenden E/A-Bausteine verwendet wird. Diese adreßierbaren Einheiten werden Ports genannt. Als E/A-Bausteine werden z.B. im Grundgerät D001 spezielle Systemschaltkreise der Z80-Familie verwendet: PIO (parallele Ein-/Ausgabe), SIO (serielle Ein-/Ausgabe) und CTC (Zähler und Zeitgeber).

Eine Besonderheit im E/A-Bereich des KC 85 ist die Modulsteuerung. Die bekannte Verwaltung der Module mittels Moduladressen und Steuerbytes erfolgt ebenfalls über E/A-Zugriffe. Werden dabei Speichermodule geschaltet, beeinflußt das natürlich auch die Zugriffe der CPU auf den Speicher (Verbindung (1) in Bild 1).

Eine weitere Besonderheit stellt das Videointerface dar. Es besitzt einen eigenen Zugriff auf einen bestimmten Teil des Hauptspeichers, den IRM (Bildwiederholspeicher), aus dessen Inhalt es das Bildschirmbild generiert. Diese Zugriffe geschehen aus Sicht der CPU und damit auch der des Programmierers unter Umgehung des Bussystems und beeinflussen die Funktionen der CPU scheinbar nicht (Verbindung (2) in Bild 1).

Teil 2 – Grundlagen der binären Informationsverarbeitung

von Frank Dachselt

2.1 Vom Bitmuster zur Dualzahl

Alle Informationen, die der Rechner verarbeitet, werden auf sogenannte Bitmuster zurückgeführt, die aus einer Folge von Nullen und Einsen bestehen. Ein solches Bitmuster könnte wie folgt aussehen:

1 0 0 1 1 1 0 1

Die Anzahl solcher Nullen und Einsen zusammen, die in einem Rechner als Grundlage für die Darstellung von Informationen verwendet wird, nennt man Wortlänge. Eine besonders charakteristische Wortlänge ist 8 Bit. Ein solches Muster von 8 Bit nennt man ein Byte.

Im vorhergehenden Abschnitt war beim Zugriff auf Speicher und E/A-Bausteine bereits von „8 Bit langen Einheiten“ die Rede. Man kann also auch sagen, Speicher und E/A-Bereich sind beim KC 85 byteweise organisiert.

Grundlage für die Darstellung von Zahlen im Rechner ist das Dualsystem. Dabei handelt es sich um ein Zahlensystem, das auf nur zwei Ziffern basiert, nämlich Null (0) und Eins (1). Um nun vom Bitmuster zu einer Dualzahl zu gelangen, müssen den einzelnen Bits, die im Bitmuster bisher alle gleichberechtigt und ohne gegenseitige Beziehungen nebeneinanderstehen, noch Bedeutungen – genauer gesagt Wertigkeiten – zugeordnet werden. Auf diese Weise gelangt man zu den Positionssystemen zur Zahlendarstellung, die auch polyadische Zahlensysteme genannt werden. Zunächst jedoch ein kurzer Blick in den Alltag, wo jeder bereits ein solches Positionssystem – bewußt oder unbewußt – anwendet.

2.2 Zahlensysteme

Aus dem täglichen Leben ist der Umgang mit Dezimalzahlen bekannt: Dezimalzahlen werden aus 10 Ziffern (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) aufgebaut; 10 ist die Basis des Dezimalsystems. Den einzelnen Stellen einer Zahl sind von rechts nach links die aufsteigenden Zehnerpotenzen 10^0, 10^1, 10^2, ... zugeordnet. Bei einer Dezimalzahl, z.B.

309, ergibt sich der Wert durch eine Summierung der mit den Ziffern der jeweiligen Stellen gewichteten Zehnerpotenzen (es gilt $10^0 = 1$):

$$\begin{aligned} 309 &= 3 \cdot 10^2 + 0 \cdot 10^1 + 9 \cdot 10^0 \\ &= 3 \cdot 100 + 0 \cdot 10 + 9 \cdot 1 \\ &= 309 \end{aligned}$$

Man kommt nun zum Dualsystem, indem man die Zahlen zur Basis 2 aufbaut. Der Wert einer Dualzahl ergibt sich durch eine Summierung von Zweierpotenzen:

$$\begin{aligned} 1101\,\mathrm{B} &= 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \\ &= 1 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 \\ &= 13 \end{aligned}$$

Das nachgestellte „B“ dient zur Kennzeichnung einer Dualzahl. Dezimalzahlen – im letzten Beispiel also alle Zahlen rechts der Gleichheitszeichen – werden dagegen wie gewohnt ohne zusätzliche Kennzeichnung verwendet. Diese Schreibweise wird auch von vielen Assemblern benutzt. Man erkennt, daß es bei der gleichzeitigen Verwendung von verschiedenen Zahlensystemen auf den Unterschied zwischen dem *Wert* einer Zahl und ihrer *Schreibweise* ankommt.

Da man schon für kleinere Werte sehr lange, unübersichtliche Dualzahlen erhält, verwendet man häufig die bequemere Schreibweise als Hexadezimalzahl. Das sind Zahlen zur Basis 16, die aus 16 Ziffern (0, 1, 2, ..., 9, A, B, C, D, E, F) aufgebaut werden. Der Wert einer Hexadezimalzahl ergibt sich dann durch eine Summierung von 16er-Potenzen:

$$\begin{aligned} 1\mathrm{A2F}\,\mathrm{H} &= 1 \cdot 16^3 + 10 \cdot 16^2 + 2 \cdot 16^1 + 15 \cdot 16^0 \\ &= 1 \cdot 4096 + 10 \cdot 256 + 2 \cdot 16 + 15 \cdot 1 \\ &= 6703 \end{aligned}$$

Das nachgestellte „H“ besagt, daß es sich um eine Zahl zur Basis 16, eben eine Hexadezimalzahl, handelt. Man kann sich Hexadezimalzahlen als eine Abkürzung von Dualzahlen vorstellen, wobei vier Dualstellen jeweils eine Hexadezimalstelle ergeben. (der tiefere Grund: Zweierpotenzen sind immer in 16er-Potenzen enthalten).

In ähnlicher Weise können Dualzahlen auch als Oktalzahlen (Basis 8) dargestellt werden, die aus 8 Ziffern (0, 1, ..., 7) aufgebaut werden. Drei Dualstellen werden auf eine Oktalstelle abgebildet. Die Oktalschreibweise wird zwar nur selten verwendet, hat aber den Vorteil, daß man keine zusätzlichen Ziffern wie bei den Hexadezimalzahlen einführen muß, um zu einer kompakteren Schreibweise zu gelangen. Die Kennzeichnung von Oktalzahlen erfolgt mit einem nachgestellten „O“:

$$\begin{aligned} 1725\,\mathrm{O} &= 1 \cdot 8^3 + 7 \cdot 8^2 + 2 \cdot 8^1 + 5 \cdot 8^0 \\ &= 1 \cdot 512 + 7 \cdot 64 + 2 \cdot 8 + 5 \cdot 1 \\ &= 981 \end{aligned}$$

Bei der Verwendung von Dualzahlen und deren alternativen Schreibweisen ist es günstig, wenn man ein paar Zweierpotenzen kennt bzw. weiß, wo man nachschauen kann. Deshalb an dieser Stelle eine kleine Aufstellung zusammen mit der Gegenüberstellung der ersten ganzen Zahlen in den vier genannten Zahlensystemen:

n	2^n		
0	2^0	=	1
1	2^1	=	2
2	2^2	=	4
3	2^3	=	8
4	2^4	=	16
5	2^5	=	32
6	2^6	=	64
7	2^7	=	128
8	2^8	=	256
9	2^9	=	512
10	2^{10}	=	1024
11	2^{11}	=	2048
12	2^{12}	=	4096
13	2^{13}	=	8192
14	2^{14}	=	16384
15	2^{15}	=	32768
16	2^{16}	=	65536

dezimal	dual	hexadezimal	oktal
0	0000 0000	0000	000
1	0000 0001	0001	001
2	0000 0010	0002	002
3	0000 0011	0003	003
4	0000 0100	0004	004
5	0000 0101	0005	005
6	0000 0110	0006	006
7	0000 0111	0007	007
8	0000 1000	0008	010
9	0000 1001	0009	011
10	0000 1010	000A	012
11	0000 1011	000B	013
12	0000 1100	000C	014
13	0000 1101	000D	015
14	0000 1110	000E	016
15	0000 1111	000F	017
16	0001 0000	0010	020
17	0001 0001	0011	021
18	0001 0010	0012	022

2.3 Umrechnungen zwischen Zahlensystemen

Da eine Zahl in unterschiedlichen Zahlensystemen dargestellt werden kann, besteht die Notwendigkeit, Umwandlungen (Konvertierungen) zwischen diesen Systemen durchzuführen. Bei kleinen Zahlen hilft ein Blick in die oben angegebene Tabelle, bei größeren Zahlen wird es aber schnell zweckmäßig sein, einen geeigneten Algorithmus zur Umwandlung zur Verfügung zu haben. An dieser Stelle soll die Darstellung auf die Konvertierung „von Hand“ beschränkt bleiben, wie sie bei der täglichen Programmierpraxis ständig notwendig ist. Die Umsetzung von Konvertierungsalgorithmen in Assemblerprogramme wird später in diesem Kurs noch ausführlich behandelt.

Insbesondere zwei Methoden für diese Zahlenkonvertierungen können direkt aus den zugrundeliegenden Bildungsgesetzen für Zahlen in polyadischen Zahlensystemen abgeleitet werden:

- Potenz-Methode
- Methode der wiederholten Division (Quotient-Methode)

Bei den folgenden Beschreibungen wird angenommen, daß die zu konvertierende Zahl in einer beliebigen Basis vorliegt und in die Darstellung mit der Basis B umgewandelt werden soll. Diese Darstellung sei symbolisch gegeben durch

$$b_n \; b_{n-1} \; \cdots \; b_1 \; b_0$$

mit den $n+1$ zu bestimmenden Ziffern b_n bis b_0.

2.3.1 Potenz-Methode

Bei dieser Methode wird in einfachster Weise die Bildung einer Zahl als Summe von gewichteten Potenzen der Basis umgesetzt:

$$Z \;=\; b_0 \cdot B^0 \;+\; b_1 \cdot B^1 \;+\; \cdots \;+\; b_N \cdot B^N$$

Dabei bezeichnen Z den Wert der Zahl selbst, B die Basis der Zahlendarstellung und b_i Ziffer an der i-ten Stelle, wobei die Zählung von rechts beginnt. Für die folgenden Überlegungen wird weiterhin vorausgesetzt, daß sich die Zahl auch tatsächlich mit $n+1$ Ziffern darstellen läßt.

Beginnend mit der höchsten Potenz B^i $(i = n)$ wird die größte Ziffer b_i bestimmt, für die gilt:

$$b_i \cdot B^i \leq Z \tag{2.1}$$

Diese erste Ziffer – und auch weitere – können natürlich auch Null sein, wenn die Zahl kleiner als B^n ist. Das ergibt bei der Konvertierung sogenannte „führende Nullen“, die aber zunächst einmal nicht stören. Der Wert $b_i \cdot B^i$ wird von der zu wandelnden Zahl subtrahiert

$$Z = Z - b_i \cdot B^i, \tag{2.2}$$

um daraufhin die Ziffer der nächsten Stelle mit

$$i = i - 1 \tag{2.3}$$

zu ermitteln. Dieser Prozeß wird wiederholt, bis schließlich der Index $i = 0$ erreicht ist und alle Ziffern b_i bestimmt wurden.

In dieser Konvertierungsmethode wird also die höchstwertige Ziffer zuerst bestimmt, die niederwertigen Ziffern folgen („von Groß nach Klein“). Die Schritte (2.1), (2.2) und (2.3) sind hier bereits in Form eines Algorithmus notiert. Das folgende Beispiel soll dieses Verfahren demonstrieren.

Beispiel: Umwandlung der Zahl 39 in das Dualsystem nach der Potenz-Methode

i	Rest von Z	Vergleichswert 2^i	$2^i \leq Z$?	Binärziffer b_i
5	39	32	ja	1
4	7	16	nein	0
3	7	8	nein	0
2	7	4	ja	1
1	3	2	ja	1
0	1	1	ja	1
				1 0 0 1 1 1

Ergebnis: 39 = 100111 B

2.3.2 Methode der wiederholten Division

Bei dieser algorithmisch sehr einfach aufgebauten Umwandlungsmethode werden die Ziffern b_0 bis b_n durch fortlaufende Division der Zahl Z durch die Basis B bestimmt. Der jeweils entstehende Rest der Division, der stets kleiner als B ist, ergibt die nächste Ziffer, wobei b_0 zuerst bestimmt wird („von Klein nach Groß“). Beginnend mit $i = 0$ läßt sich der Algorithmus wie folgt angeben:

$$Z = Z/B \quad \text{Rest } r \quad \longrightarrow \quad b_i = r \tag{2.4}$$

$$i = i + 1 \tag{2.5}$$

Nach $Z/B = 0$ wird der Algorithmus abgebrochen. Ein Vorteil dieser Konvertierung besteht darin, daß hier keine „führenden Nullen“ entstehen, also nur genau soviel Schritte durchgeführt werden, wie Ziffern zur Darstellung der Zahl mit der Basis B notwendig sind. Die folgenden Tabelle zeigt die Wirkung des Algorithmus an einem Beispiel.

Beispiel: Umwandlung der Zahl 39 in das Dualsystem nach der Quotient-Methode

i	Division $Z/2$	Quotient	Rest $\longrightarrow$ Binärziffer b_i
0	39/2	19	1
1	19/2	9	1
2	9/2	4	1
3	4/2	2	0
4	2/2	1	0
5	1/2	0	1
			1 0 0 1 1 1

Ergebnis: 39 = 100111 B

2.3.3 Konvertieren durch Rechnen in der Zielbasis

Die beiden bisher vorgestellten Verfahren habe eines gemeinsam: Da die zu konvertierende Zahl zunächst nur in der alten Basis vorliegt, müssen auch alle Konvertierungsschritte in dieser alten Basis durchgeführt werden.

Für eine Rechnung „von Hand“ schränkt das die Anwendung ein, und zwar auf die Konvertierung von Dezimalzahlen in die Darstellung mit einer anderen Basis, also z.B. in die Dual-, Oktal- oder Hexadezimaldarstellung. Dann nämlich können die Rechnungen im gewohnten Dezimalsystem stattfinden.

Für die Konvertierung von Dual-, Oktal- und Hexadezimalzahlen in das Dezimalsystem bietet sich dagegen ein sehr einfaches Verfahren an, bei dem alle Rechnungen in der Zielbasis – also im Dezimalsystem – durchgeführt werden. Dabei bildet man die Potenzen der Ausgangsbasis im Dezimalsystem und addiert diese mit den Ziffern der zu konvertierenden Zahl gewichtet auf. Beispiele dafür sind die bereits im Abschnitt 2.2 gezeigten Umrechnungen.

2.3.4 Konvertierung bei „verwandten“ Basen

Für die Konvertierung von Dual-, Oktal- und Hexadezimalzahlen in das Dezimalsystem und umgekehrt stehen nun also Verfahren zur Verfügung. Wie geht man nun aber bei Konvertierungen vom und zum Dual-, Oktal- und Hexadezimalsystem vor?

Da es sich hierbei um „verwandte“ Zahlensysteme handelt, kann die Konvertierung beliebig großer Zahlen auf die Konvertierung der einzelnen Ziffern zurückgeführt werden, wobei mitunter der Umweg über das Dualsystem notwendig ist. Bei der Umwandlung von und zur Hexadezimalzahl wird die Dualzahl in Gruppen zu je vier Dualziffern, bei der Umwandlung von und zur Oktalzahl in Gruppen zu je drei Dualziffern aufgeteilt. Die einzelnen Ziffern werden z.B. unter Zuhilfenahme einer Tabelle (siehe Abschnitt 2.2) umgewandelt. Das folgende Beispiel verdeutlicht die Vorgehensweise.

Beispiel: Umwandlung der Zahl 1A7C H in die Binär- und Oktaldarstellung

Hexadezimalzahl		1	A	7	C
$\updownarrow$		$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$
Dualzahl		0001	1010	0111	1100
$\updownarrow$					
Dualzahl	001	101	001	111	100
$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$	$\updownarrow$
Oktalzahl	1	5	1	7	4

Ergebnis: 1A7C H = 0001 1010 0111 1100 B = 15 174 O

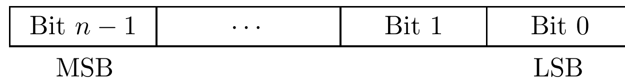
2.4 Rechnen mit Dualzahlen (ganze Zahlen)

Um im Rechner arithmetische Operationen mit Dualzahlen ausführen zu können, müssen diese einer bestimmten, dem Rechenwerk bekannten, einheitlichen Form dargestellt werden.

Die arithmetische Operationen folgen in allen polyadischen Zahlensystemen den gleichen Regeln. Deshalb ist es möglich, die vom Dezimalsystem her bekannten Vorgehensweisen auf das Dualsystem anzuwenden.

2.4.1 Zahlenbereich im Rechner (ganze Zahlen)

Bei allen arithmetischen Operationen muß davon ausgegangen werden, daß zur Darstellung von Zahlenwerten nur eine begrenzte Anzahl von Bitstellen zur Verfügung steht. Dabei stellt der 8 Bit breite Datenbus beim Z80 keineswegs die obere Grenze dar, sondern es können auch mehrere Bytes für die Darstellung großer Zahlen zusammengefaßt werden. In einer n Bit breiten Darstellung werden die Bits von rechts nach links durchnummeriert:



Daraus ergibt sich die etwas eigenartig anmutende, aber in der Rechentechnik übliche Zählweise:

- das erste Bit ist *Bit 0*,
- das zweite Bit ist *Bit 1*,

  ⋮
- das n-te Bit ist *Bit* $n-1$.

Zwei Bitpositionen werden dabei mit besonderen Namen belegt:

- LSB: least significant bit (niedrigstwertiges Bit), kennzeichnet das Bit mit der geringsten Wertigkeit (2^0);
- MSB: most significant bit (höchstwertiges Bit), kennzeichnet das Bit mit der höchsten Wertigkeit (2^{n-1}).

Da die Darstellungsbreite begrenzt ist, können Zahlenwerte nur bis zu einem genau vorgegebenen Maximalwert dargestellt werden. Sollen arithmetische Operationen durchgeführt werden, muß das Bestehen dieser Darstellungsgrenze beachtet werden. Dies führt zu Situationen, die als „Übertrag“ (engl.: carry) bzw. „Überlauf“ (engl.: overflow) bezeichnet werden.

Eine weitere Unterscheidung, die bei Durchführung arithmetischer Operationen beachtet werden muß, ist die Verwendung des Vorzeichens von Zahlen. Man unterscheidet daher

- vorzeichenlose Zahlen (engl.: unsigned numbers) und
- vorzeichenbehaftete Zahlen (engl.: signed numbers).

2.4.2 Vorzeichenlose Zahlen

Bei der Verwendung vorzeichenloser Zahlen zur Darstellung positiver Zahlen werden alle n Bitstellen mit Wertigkeiten des Dualsystems belegt. Der kleinste darstellbare Zahlenwert ist Null (alle Bitstellen sind Null), der größte Zahlenwert ist $2^n - 1$ (alle Bitstellen sind Eins).

Beispiele gebräuchlicher Darstellungsbreiten:

Darstellungsbreite	darstellbarer Zahlenbereich
8 Bit	0 ... 255
16 Bit	0 ... 65 535
24 Bit	0 ... 16 777 215
32 Bit	0 ... 4 294 967 295

2.4.3 Vorzeichenbehaftete Zahlen

Um negative Zahlen darzustellen, ist die Reservierung einer Bitstelle notwendig, die die Information über das Vorzeichen (plus oder minus) enthält. Gebräuchlich ist die Benutzung des MSB zu diesem Zweck, wobei folgende Zuordnung gilt (s = Vorzeichenbit (engl.: sign bit)):

Typ	Vorzeichenbit
Positive Zahlen	s = 0
Negative Zahlen	s = 1

Positive Zahlen werden damit genauso dargestellt wie vorzeichenlose Dualzahlen, mit der Einschränkung, daß das MSB nicht 1 werden darf.

Für die Realisierung negativer Dualzahlen (s = 1) gibt es unterschiedliche Möglichkeiten, von denen drei im folgenden behandelt werden sollen:

- Vorzeichen und Absolutbetrag
- Einerkomplement
- Zweierkomplement

2.4.3.1 Vorzeichen und Absolutbetrag

Kennzeichnend für diese Realisierungsform ist, daß im MSB das Vorzeichen gespeichert wird, während die restlichen $n-1$ Bits den Absolutbetrag der Zahl als vorzeichenlose Dualzahl enthalten:

s	Absolutbetrag

Die folgende Tabelle demonstriert das Format am Beispiel einer 8 Bit breiten Darstellung:

Zahl	Binärdarstellung
+0	0000 0000
+1	0000 0001
+2	0000 0010
+3	0000 0011
⋮	
−0	1000 0000
−1	1000 0001
−2	1000 0010
−3	1000 0011
⋮	

Der darstellbare Zahlenbereich liegt symmetrisch zur Null, der kleinste darstellbare Zahlenwert ist $-2^{n-1}-1$, der größte Zahlenwert ist $2^{n-1}-1$.

Beispiele gebräuchlicher Darstellungsbreiten:

Darstellungsbreite	darstellbarer Zahlenbereich		
8 Bit	−127	...	127
16 Bit	−32 767	...	32 767
32 Bit	−8 388 607	...	8 388 607
64 Bit	−2 147 483 647	...	2 147 483 647

Diese Zahlendarstellung hat den Vorteil eines sehr einfachen Bildungsgesetzes; sie entspricht vollkommen der Form, in der das Dezimalsystem „manuell" eingesetzt wird. Diese einfache Definition negativer Zahlen ist insbesondere bei der Multiplikation und der Division von Vorteil.

Nachteilig ist, daß bei dieser Repräsentation zwei Darstellungen der Zahl Null möglich sind; es existiert sowohl eine positive als auch eine negative Null. Ein weiterer Nachteil besteht darin, daß zur Durchführung der Addition und der Subtraktion zwei getrennte, unabhängige Recheneinheiten notwendig werden.

2.4.3.2 Komplement-Darstellung negativer Zahlen

Die Komplement-Darstellung der Zahlen hat den Vorteil, daß im Rechenwerk eines Prozessors auch Subtraktionen mit einem Addierwerk ausgeführt werden können; eine spezielle Subtrahiereinheit ist nicht erforderlich.

Zu diesem Zweck wird die durchzuführende Operation $a-b$ zunächst umgeformt:

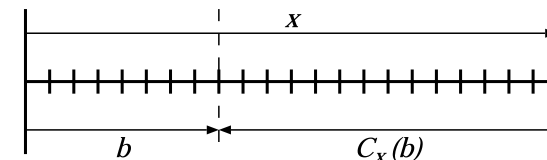
$$a-b=a+(x-b)-x \tag{2.6}$$

Auf der rechten Seite der Gleichung wurde $x-x$ addiert. Das ändert den Wert des Ausdruckes nicht, macht aber eine symbolische Umformung möglich. Der eingeklammerte Ausdruck

$$C_x(b)=x-b \tag{2.7}$$

wird als Komplement von b bezüglich x bezeichnet.

Anschaulich bedeutet die Operation „Komplement" die Ergänzung auf ein vorgegebenes Ganzes. In grafischer Form sieht das wie folgt aus:



Technisch sinnvoll wird die Gleichung (2.6) erst, wenn für x ein Wert gewählt wird, der einerseits die Komplementbildung $x-b$ einfach durchführbar macht und der andererseits auch die Korrektur $-x$ erlaubt, ohne letztendlich doch noch eine Subtraktion durchführen zu müssen.

Insbesondere zwei Werte für x ermöglichen dies, was zu den beiden wichtigen Komplement-Begriffen im Dualsystem führt:

- Einerkomplement (engl: one's complement): $x = 2^n - 1$
- Zweierkomplement (engl: two's complement): $x = 2^n$

Die Größe n gibt wiederum die Anzahl der zur Verfügung stehenden Bitpositionen an.

2.4.3.3 Das Einerkomplement

Um das Einerkomplement $C_1(b)$ von b zu bilden, wird $x = 2^n - 1$ gesetzt, d.h. x entspricht der größten vorzeichenlosen Zahl, die mit n Bits darstellbar ist.

Beispiel für $n = 8$:

x = | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Mit dieser Wahl von x wird die Bildung des Komplements $x - b$ besonders einfach. Da x in allen Bitpositionen eine Eins enthält, kann es bei der Differenzbildung niemals zu einem „Borgen" kommen (siehe Abschnitt 2.4.4). Dies bedeutet:

> Das Einerkomplement von b wird durch stellenweises Invertieren aller Bits in b gebildet.

Beispiel für $n = 8$:

$$\begin{aligned} b &= 1011\,0001 \\ C_1(b) &= 0100\,1110 \end{aligned}$$

2.4.3.4 Das Zweierkomplement

Im Fall des Zweierkomplements $C_2(b)$ wird $x = 2^n$ gesetzt, d.h. x ist um 1 größer als beim Einerkomplement und damit auch um 1 größer als die mit n Bits darstellbare Zahl.

Beispiel für $n = 8$:

x = | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Damit führt Weg zur Ermittlung des Zweierkomplements einer Dualzahl am einfachsten über das Einerkomplement:

> Zur Ermittlung des Zweierkomplements von b wird zunächst das Einerkomplement von b gebildet und anschließend der Wert 1 addiert.

Beispiel für $n = 8$:

$$\begin{array}{rcr} b & = & 1011\,0001 \\ C_1(b) & = & 0100\,1110 \\ & + & 1 \\ \hline C_2(b) & = & 0100\,1111 \end{array}$$

2.4.3.5 Verwendung von Einer- und Zweierkomplement

Hat man sich für die Verwendung einer der beiden Komplementdarstellungen entschieden, gilt für die Darstellung negativer Zahlen:

> Bei der Komplementdarstellung wird bei negativen Zahlen anstelle des Zahlenwertes $-x$ (x sei hier der (positive) Betrag der Zahl) das Komplement von x gespeichert.

Die folgende Tabelle stellt Einer- und Zweierkomplement für einige 8 Bit breite Dualzahlen gegenüber:

Dezimalzahl	Dualzahl	Einerkomplement	Zweierkomplement
0	0000 0000	1111 1111	0000 0000
1	0000 0001	1111 1110	1111 1111
2	0000 0010	1111 1101	1111 1110
3	0000 0011	1111 1100	1111 1101
⋮			
126	0111 1110	1000 0001	1000 0010
127	0111 1111	1000 0000	1000 0001
128	–	–	1000 0000

Wird das Einerkomplement verwendet, liegt der darstellbare Zahlenbereich symmetrisch zur Null; es sind Zahlenwerte von $-2^{n-1} + 1$ bis $2^{n-1} - 1$ darstellbar, wobei für die Null zwei Darstellungen existieren.

Demgegenüber liegt der Zahlenbereich beim Zweierkomplement unsymmetrisch zur Null (-2^{n-1} bis $2^{n-1}-1$), für die kleinste (negative) Zahl gibt es keine positive Entsprechung. Dafür ist die Darstellung der Null eindeutig.

Beispiele gebräuchlicher Darstellungsbreiten für Zahlen im Zweierkomplement:

Darstellungsbreite	darstellbarer Zahlenbereich		
8 Bit	−128	...	127
16 Bit	−32 768	...	32 767
32 Bit	−8 388 608	...	8 388 607
64 Bit	−2 147 483 648	...	2 147 483 647

Die Operationen „Einerkomplement“ und „Zweierkomplement“ sind sogenannte Involutionen, d.h. ihre zweimalige Anwendung hintereinander erzeugt wieder die ursprüngliche Zahl. Diese Eigenschaft entspricht der bekannten Tatsache $-(-x) = x$ für den gewohnten Umgang mit negativen Zahlen. Damit ist es auf einfache Weise möglich, den Betrag einer negativen Zahl in Komplementdarstellung zu bestimmen. Das ist zum Beispiel notwendig, wenn die Zahl in einem anderen Zahlensystem dargestellt werden soll (zum Beispiel bei der Ausgabe als Dezimalzahl).

Beispiele für $n = 8$:

Einerkomplement		Zweierkomplement	
0010 1011	43	0111 1001	121
‖		‖	
Einerkomplement		Zweierkomplement	
⇓		⇓	
1101 0100	−43	1000 0111	−121
‖		‖	
Einerkomplement		Zweierkomplement	
⇓		⇓	
0010 1011	43	0111 1001	121

2.4.4 Addition und Subtraktion von Dualzahlen

Die Addition zweier Dualzahlen läuft wie im Dezimalsystem positionsweise von rechts nach links ab („schriftliche Addition“). Zur Vereinfachung werden jeweils nur zwei Dualzahlen addiert. Sollen mehr Dualzahlen zusammenaddiert werden, wird zunächst eine Zwischensumme aus den ersten beiden gebildet und zu dieser dann schrittweise die weiteren Summanden addiert.

Die beiden zu addierenden Summanden und die resultierende Summe seien mit folgendem Schema gegeben:

1. Summand	a		a_{n-1}	$\cdots$	a_1	a_0
2. Summand	b	$+$	b_{n-1}	$\cdots$	b_1	b_0
Summe	s	$=$	s_{n-1}	$\cdots$	s_1	s_0

Die Addition zweier Bits a_i und b_i an einer beliebigen Stelle i läuft zunächst nach folgenden Regeln ab:

a_i	$+$	b_i	$=$	s_i	(Übertrag c_i)
0	+	0	=	0	(Übertrag 0)
0	+	1	=	1	(Übertrag 0)
1	+	0	=	1	(Übertrag 0)
1	+	1	=	0	(Übertrag 1)

Neben dem Summenbit s_i der jeweiligen Stelle kann also auch noch ein Übertrag (carry) in die nächste Stelle entstehen. Zur vollständigen Addition $a+b$ ist es deshalb notwendig, auch den Übertrag c_{i-1} (carry-in) aus der Addition in der vorhergehenden Stelle $i-1$ zu berücksichtigen. Dazu wird die Addition an der Stelle i erweitert auf

$$c_{i-1} + a_i + b_i = s_i \quad \text{(Übertrag } c_i\text{)}, \tag{2.8}$$

wobei die folgende Wertetabelle gilt:

c_{i-1}	a_i	b_i	s_i	c_i
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Das folgende Beispiel demonstriert die Addition zweier Dualzahlen und stellt sie der Addition mit Dezimalzahlen gegenüber:

	0011 1011	⇔		59
+	0000 1110		+	14
Überträge	0111 1100		Überträge	10
Ergebnis	0100 1001		Ergebnis	73

Auch die „normale“ Subtraktion von Dualzahlen läuft nach den von der „schriftlichen Subtraktion“ von Dezimalzahlen her bekannten Regeln ab. Anstelle des Übertrages tritt hier das „Borgen“ von der jeweils nächsthöheren Stelle:

	0110 1101	⇔		109
−	0101 0110		−	86
Borgen	0010 1100		Borgen	100
Ergebnis	0001 0111		Ergebnis	23

In dieser Form wird die Subtraktion in einem Prozessor nicht ausgeführt, da ein extra Rechenwerk für diese Operation notwendig wäre. Stattdessen führt man die Subtraktion als Addition des Komplements des Minuenden durch und kann so das Rechenwerk für die Addition benutzen. Das obige Subtraktionsbeispiel $109 - 86$ verändert sich bei der Verwendung des Zweierkomplements in $109 + (-86)$ und sieht dann wie folgt aus:

	0110 1101	(109)
+	1010 1010	(Zweierkomplement von 86)
Ergebnis	0001 0111	(23)

Die vorhergehenden Beispiele erfolgten ohne Berücksichtigung der Darstellungsbreite des Ergebnisses. In den folgenden Beispielen soll die Zahlendarstellung dagegen auf 6 Bits beschränkt werden und das Zweierkomplement Verwendung finden:

	1 0 0 1 1 1	(−25)		1 1 0 1 1 1	(−9)
+	0 0 1 1 0 0	(12)	+	1 0 0 1 1 0	(−26)
=	1 1 0 0 1 1	(−13)	=	1 0 1 1 1 0 1	(−35) → (29)

Im Beispiel auf der linken Seite ist das Ergebnis richtig und offensichtlich problemlos darstellbar.

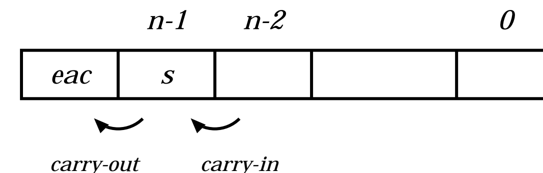
Auf der rechten Seite würde sich der falsche Ergebniswert 001011 B = 29 ergeben, da das führende Bit mit dem Wert 1 nicht darstellbar ist. Es entsteht ein sogenannter Überlauf (overflow), da die Länge der gewählten Zahlendarstellung nicht ausreicht, um das Additionsergebnis zu speichern (der darstellbare Zahlenbereich mit 6 Bit im Zweierkomplement ist $-32 \ldots 31$).

Dieses Problem der begrenzten Darstellungsbreite tritt nicht nur bei der Addition auf. Auch bei der Subtraktion muß auf diese Einschränkung geachtet werden.

Da allerdings nicht nach jeder numerischen Rechnung mit Dualzahlen eine Vergleichsrechnung z.B. im Dezimalsystem durchgeführt werden kann, soll ein Weg aufgezeigt werden, mit dem die Richtigkeit des Ergebnisses bereits während der Rechnung im Dualsystem überprüft werden kann.

Dieser Algorithmus ist in Form zweier Regeln gegeben, die die Addition vorzeichenbehafteter Dualzahlen in Komplementdarstellung vollständig beschreiben und gleichzeitig die gewünschte Überprüfung der Rechnung erlauben.

Die Darstellung einer vorzeichenbehafteten Dualzahl wird zu diesem Zweck um das Hilfsbit *eac* (engl: „end-around-carry“) erweitert:



Dieses Bit existiert nicht als realer Bestandteil der Zahlendarstellung, sondern wird lediglich vom Additionsalgorithmus benutzt.

Additionsregel

Diese Regel zeigt, wie das *End-around-carry*-Bit behandelt wird, wobei zwischen Einer- und Zweierkomplement unterschieden werden muß:

Einerkomplement:
Existiert ein „end-around-carry“ (*eac* = 1), wird ”1ßur Zwischensumme addiert.

Zweierkomplement:
Ein evtl. auftretendes „end-around-carry“ ($eac = 1$), wird vernachlässigt.

Ergebniskontrolle

Diese Regel zeigt, wie während der Addition das Ergebnis auf Richtigkeit überprüft werden kann; sie gilt gleichermaßen für Einer- und Zweierkomplementdarstellung:

Einerkomplement und Zweierkomplement:
Entspricht das an der Stelle $n - 2$ erzeugte Übertragsbit (carry bit) nicht dem Übertragsbit, das an der Position $n - 1$ (Position des Vorzeichenbits) gebildet wird, liegt eine Fehlersituation vor; es existiert in diesem Fall ein Überlauf (overflow), d.h. das Rechenergebnis kann nicht mit der vorgegebenen Informationsbreite dargestellt werden.
Anschaulich bedeutet diese Regel, daß der Übertragswert, der in die Vorzeichenposition hineintransportiert wird, auch wieder heraustransportiert werden muß (carry-in und carry-out an der Position des Vorzeichenbits müssen gleich sein!).

Die hier aufgestellten Regeln für die Addition sollen an einfachen Beispielen erläutert werden. Es wird n = 4 gewählt:

	Dezimal	Einerkompl.	Zweierkompl.	Fehler
Beispiel 1:	6	0 1 1 0	0 1 1 0	
	- 7	1 0 0 0	1 0 0 1	
carry bits:		0 0 0 0	0 0 0 0	
Zwischensumme:		0 1 1 1 0	0 1 1 1 1	
Ergebnis:	- 1	1 1 1 0	1 1 1 1	Nein
Beispiel 2:	6	0 1 1 0	0 1 1 0	
	- 4	1 0 1 1	1 1 0 0	
carry bits:		1 1 1 0	1 1 0 0	
Zwischensumme:		1 0 0 0 1	1 0 0 1 0	
Korrektur (+1):		1		
Ergebnis:	+ 2	0 0 1 0	0 0 1 0	Nein

	Dezimal	Einerkompl.	Zweierkompl.	Fehler
Beispiel 3:	6	0 1 1 0	0 1 1 0	
	+ 6	0 1 1 0	0 1 1 0	
carry bits:		0 1 1 0	0 1 1 0	
Zwischensumme:		0 1 1 0 0	0 1 1 0 0	
Ergebnis:	(+12)	1 1 0 0	1 1 0 0	Ja
Beispiel 4:	- 7	1 0 0 0	1 0 0 1	
	- 7	1 0 0 0	1 0 0 1	
carry bits:		1 0 0 0	1 0 0 1	
Zwischensumme:		1 0 0 0 0	1 0 0 1 0	
Korrektur (+1):		1		
Ergebnis:	(-14)	0 0 0 1	0 0 1 0	Ja

An den Beispielen 3 und 4 ist ersichtlich, daß die in der zweiten Regel geforderte Bedingung nicht erfüllt wird (carry-in und carry-out im Vorzeichenbit unterscheiden sich). Die Ergebnisse sind also falsch. Offensichtlich sind die zu erwartenden Ergebniswerte (+12 bzw. -14) nicht mehr in einer 4-Bit-Darstellung realisierbar.

Anmerkungen:

Der Z80-Prozessor unterstützt mit seinem Rechenwerk das Rechnen sowohl mit vorzeichenlosen als auch mit vorzeichenbehafteten Dualzahlen in Zweierkomplementdarstellung. Dazu werden bei jeder Addition und Subtraktion zwei Flags gesetzt, die eine Kontrolle des Ergebnisses nach den oben angegeben Regeln ermöglichen (siehe auch Abschnitt 3.2.3).

Das „End-around-carry“ wird in das C-Flag übernommen, während im P/V-Flag ein Überlauf nach der Regel für die Ergebniskontrolle bei Komplementdarstellung signalisiert wird. Die Beeinflussung dieser Flags ist natürlich unabhängig davon, ob die verarbeiteten Zahlen als vorzeichenlos oder als solche in Komplementdarstellung angesehen werden. Die richtige Interpretation der Flags muß in Abhängigkeit vom verwendeten Zahlenformat vorgenommen werden.

Prinzipiell läßt sich auch das Einerkomplement für die Darstellung negativer Zahlen anwenden. Da hier aber mitunter eine zusätzlich Korrektur des Ergebnisses vorgenommen werden muß, bleibt diese Variante nur speziellen Anwendungsfällen vorbehalten.

2.4.5 Multiplikation und Division von Dualzahlen

Die Multiplikation und Division von Dualzahlen geschieht wie im Dezimalsystem schrittweise und wird auf Additionen bzw. Subtraktionen zurückgeführt. Da es sich dabei bereits um relativ komplexe Verfahren handelt, existiert eine Vielzahl unterschiedlicher Algorithmen, die Multiplikationen und Divisionen unter verschiedenen Randbedingungen optimal durchführen. In den beiden folgenden Abschnitten soll nur jeweils die Grundform vorgestellt und einige Eigenschaften erläutert werden.

2.4.5.1 Multiplikation von Dualzahlen

Zum Multiplizieren werden die Dualzahlen positionsweise multipliziert und die stellenverschobenen Zwischenwerte addiert. Allerdings ist die Multiplikation im Dualsystem wesentlich einfacher als im Dezimalsystem, da nur Multiplikationen mit den Werten 0 oder 1 auftreten. Der stellenverschobene Multiplikand muß also nur wiederholt addiert werden.

Der dualen Multiplikation liegt die folgende Multiplikationstabelle zu Grunde (das „Kleine Einmaleins“ des Dualsystems):

a	b	$a \cdot b$
0	0	0
0	1	0
1	0	0
1	1	1

Bei der Multiplikation wird für das Ergebnis ein erhöhter Speicherbedarf benötigt. Sind beide zu multiplizierenden Faktoren n-stellig, müssen für die Speicherung des Produkts $2 \cdot n$ Bits vorgesehen werden, doppelt soviel wie für die einzelnen Faktoren.

Das „Standardverfahren“ zur Multiplikation von Dualzahlen läuft ähnlich der „schriftlichen Multiplikation“ von Dezimalzahlen ab.

Beispiel: $20 \cdot 11 = 220$

$10100 \cdot 1011$		
10100	$10100 \cdot 1$	3 Stellen nach links verschoben
+ 00000	$10100 \cdot 0$	2 Stellen nach links verschoben
+ 10100	$10100 \cdot 1$	1 Stelle nach links verschoben
+ 10100	$10100 \cdot 1$	
11011100	Ergebnis: 11011100 B = 220	

Bei diesem Verfahren kann natürlich in gleicher Form von rechts nach links vorgegangen werden. Beide Faktoren müssen hier als vorzeichenlose Dualzahlen vorliegen. Bei vorzeichenbehafteten Zahlen ist zunächst eine Überführung in das Format „Betrag und Vorzeichen“ notwendig. Danach werden die Beträge multipliziert und das Vorzeichen des Produkts nach den bekannten Regeln aus den Vorzeichen der einzelnen Faktoren bestimmt. Dieses Ergebnis kann nun wieder in ein anderes Darstellungsformat konvertiert werden.

Das gezeigte Verfahren kann aber auch einfach für die Multiplikation von vorzeichenbehafteten Zahlen (z.B. in Zweierkomplement-Darstellung) erweitert werden.

Das „Standardverfahren“ ist auch zur Rechneranwendung geeignet, hat aber den Nachteil, daß $2n$-stellige Additionen notwendig sind, obwohl in den einzelnen Schritten jeweils nur n Stellen addiert werden.

Im Dezimalsystem ist die Multiplikation mit Potenzen von 10 besonders einfach. Der Exponent einer solchen Potenz gibt an, um wieviele Stellen alle Ziffern nach links verschoben bzw. wieviele Nullen angehangen werden müssen. Das gilt in gleicher Weise auch für Dualzahlen und die Potenzen von 2. Auch hier sind lediglich „Verschiebungen“ notwendig, um diese Multiplikationen auszuführen:

00000101	5
00001010	$2 \cdot 5 = 10$
00010100	$4 \cdot 5 = 20$
00101000	$8 \cdot 5 = 40$
01010000	$16 \cdot 5 = 80$

2.4.5.2 Division von Dualzahlen

Die Division von Dualzahlen erfolgt ähnlich der „schriftlichen Division“ von Dezimalzahlen durch schrittweises Subtrahieren des Divisors. Diese Subtraktion kann auch durch eine Addition im Einer- oder Zweierkomplement ersetzt werden.

Auch hier gilt, daß auf Grund der dualen Teilmultiplikationen die Rechnung einfacher ausführbar wird:

```
    11011100  :  10100  =  1011
  − 10100
  ---------
     11110
  −  10100
  ---------
      10100
  −   10100
  ---------
      00000
```

Ergebnis: 1101 1100 : 10100 = 1011

Da bei der dualen Division eine Quotientenstelle höchstens 1 sein kann, genügt zur Ermittlung einer Stelle eine einzige Subtraktion. Ist dabei der stellenrichtig verschobene Divisor größer als der Dividend, dann entsteht in der dazugehörigen Quotientenstelle eine 0 und die nächste Stelle wird an den Dividenden angehängt. Ist der Divisor kleiner als der Dividend, dann wird der Divisor vom Dividenden subtrahiert, die Quotientenstelle erhält eine 1 und die nächste Stelle wird wiederum an den Dividenden angehängt.

Wie bei der Multiplikation werden auch hier zunächst nur vorzeichenlose Zahlen verarbeitet, was eine extra Behandlung des Vorzeichens notwendig macht. Allerdings läßt sich das gezeigte Verfahren auch zur Division von Zahlen in Komplementdarstellung erweitern.

Auch bei der binären Division gibt es eine Gruppe von Rechnungen, die besonders einfach durchzuführen sind, nämlich die Division durch Potenzen von 2. Ähnlich der Division durch Zehnerpotenzen im Dezimalsystem, werden auch hier alle Ziffern eine bestimmte Anzahl von Stellen nach rechts geschoben. Das führende Bit (MSB) wird dabei stets Null, eine eventuell im LSB stehende Eins geht als Rest bei dieser Division verloren:

11010000	208	
01101000	208 : 2 = 104	
00110100	208 : 4 = 52	
00011010	208 : 8 = 26	
00001101	208 : 16 = 13	
00000110	208 : 32 = 6	Rest 1

2.5 Darstellung von gebrochenen Zahlen

Nachdem nun die Darstellung von ganzen Zahlen und die dazugehörigen arithmetischen Grundoperationen bekannt sind, besteht auch irgendwann einmal der Wunsch, mit gebrochenen Zahlen zu arbeiten. Die folgenden Abschnitte sollen das Thema jedoch nur kurz anreißen und die wichtigsten Eigenschaften zweier gebräuchlicher Darstellungen für gebrochenen Zahlen aufzeigen.

2.5.1 Festkommazahlen

Festkommazahlen sind *Zahlen mit gedachtem Komma*. Das Komma dient als Trennsymbol zwischen dem ganzzahligen und dem gebrochenen Teil von Dualzahlen. In einer n Bit breiten Zahlendarstellung werden die höherwertigen k Bit für den ganzzahligen Anteil (den Vorkommateil) verwendet, die niederwertigen l Bit für den gebrochenen Anteil (den Nachkommateil). Dabei gilt natürlich $k + l = n$.

Die Position des Kommas braucht dabei nicht mit abgespeichert zu werden, da sie unveränderlich ist. Lediglich die numerischen Operationen müssen auf die Lage des Kommas abgestimmt sein.

Während im Dezimalsystem die Nachkommastellen die *Zehntel*, die *Hundertstel*, die *Tausendstel* usw. eines gegebenen Zahlenwertes zählen, haben die Nachkommastellen in der dualen Festkommadarstellung die Wertigkeiten ein *Halbes*, ein *Viertel*, ein *Achtel* usw. Sie entsprechen damit den negativen Potenzen der Basis.

Als Beispiel soll eine 7 Bit breite Darstellung betrachtet werden, bei der 4 Bit für den Vor- und 3 Bit für den Nachkommateil verwendet werden ($n = 7, k = 4, l = 3$):

$$\begin{aligned} 1101{,}101\,\mathrm{B} &= 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} \\ &= 1 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 + 1 \cdot 0,5 + 0 \cdot 0,25 + 1 \cdot 0,125 \\ &= 13{,}625 \end{aligned}$$

Zur Konvertierung von Festkommazahlen in andere Zahlensysteme können die bereits im Abschnitt 2.3 vorgestellten Verfahren mit kleinen Modifikationen eingesetzt werden. Die Potenzmethode braucht lediglich um l Schritte mit negativen Zweierpotenzen, beginnend mit $2^{-1} = \frac{1}{2}$, erweitert werden. Die Quotient-Methode wird zur Quotient-Produkt-Methode erweitert, bei der die Zahlenwerte getrennt nach Vor- und Nachkommateil konvertiert. Der Vorkommateil wird dabei weiter wie bisher behandelt. Der Nachkommateil wird dagegen schrittweise mit 2 multipliziert und der dabei entstehende Vorkommawert – Null oder Eins – ergibt jeweils die nächste Dualstelle nach dem Komma (die Nachkommastellen werden hier entgegen den Vorkommastellen mit fallenden Wertigkeiten bestimmt).

Beispiel: Umwandlung der Zahl 13,625 in die binäre Festkommadarstellung nach der Potenz-Methode

i	Rest von Z	Vergleichswert 2^i	$2^i \leq Z$?	Binärziffer b_i
3	13,625	8	ja	1
2	5,625	4	ja	1
1	1,625	2	nein	0
0	1,625	1	ja	1
-1	0,625	0,5	ja	1
-2	0,125	0,25	nein	0
-3	0,125	0,125	ja	1
				1 1 0 1 1 0 1

Ergebnis: 13,625 = 1101,101 B

Beispiel: Umwandlung der Zahl 13,625 in die binäre Festkommadarstellung nach der Quotient-Produkt-Methode

i	Division $Z/2$	Quotient	Rest $\longrightarrow$ Binärziffer b_i
3	$13/2$	6	1
2	$6/2$	3	0
1	$3/2$	1	1
0	$1/2$	0	1
i	Multiplikation $Z \cdot 2$	Nachkommateil	Vorkommateil $\longrightarrow$ Binärziffer b_i
-1	$0,625 \cdot 2$	0,250	1
-2	$0,250 \cdot 2$	0,500	0
-3	$0,500 \cdot 2$	0,000	1
			1 1 0 1 1 0 1

Ergebnis: 13,625 = 1101,101 B

Die begrenzte Anzahl der zur Verfügung stehenden Bits führt bei Festkommazahlen zu zwei Effekten. Während die Länge des Vorkommateils wie bisher die Grenzen des darstellbaren Zahlenbereichs bestimmt, wird durch die Länge des Nachkommateils die darstellbare Genauigkeit der Zahlen begrenzt. Mit l Nachkommastellen beträgt diese Genauigkeit $2^{-l} = \frac{1}{2^l}$.

Bei der Addition und Subtraktion von Festkommazahlen müssen die Zahlen so verschoben werden, daß die Kommas übereinander stehen. Dann kann unter Vernachlässigung des Kommas wie mit ganzen Zahlen addiert bzw. subtrahiert werden. Die Kommaposition des Ergebnisses ist die gleiche wie bei den Ausgangszahlen.

Beispiel: Addition der beiden Festkommazahlen 1011,11 B und 110,011 B ($11,75 + 6,375 = 18,125$)

$$\begin{array}{rr} & 1011{,}110 \\ + & 0110{,}011 \\ \hline = & 1\ 0010{,}001 \end{array}$$

Die Darstellung vorzeichenbehafteter Festkommazahlen kann wie bei den ganzen Zahlen in den Formaten *Vorzeichen und Betrag, Einerkomplement und Zweierkomplement* erfolgen. Die Komplementbildung wird dabei unter Vernachlässigung des Kommas durchgeführt (Interpretation der Festkommazahl als ganze Zahl).

Multiplikation und Division können zunächst ohne Berücksichtigung der Kommaposition der Ausgangszahlen durchgeführt werden. Die Kommaposition im Ergebnis ergibt sich danach durch eine gesonderte Betrachtung. Die Länge des Nachkommateils im Produkt ist gleich der Summe der Längen der Nachkommateile der Faktoren.

2.5.2 Gleitkommazahlen

In Gegensatz zu Festkommazahlen ist die Kommaposition bei Gleitkommazahlen variabel und muß daher auch abgespeichert werden. Die Darstellung von Gleitkommazahlen entspricht der auch in Dezimalsystem bekannten Exponentialschreibweise bzw. der „wissenschaftlichen Schreibweise“ bei Taschenrechnern. Dieses Format besteht aus zwei Teilen: der Mantisse und dem Exponenten. Die Mantisse enthält dabei die Ziffernfolge des Zahlenwertes und der Exponent bestimmt die Position des Kommas. Im Dezimalsystem sehen Zahlen in dieser Darstellung zum Beispiel wie folgt aus:

$$275\,400 = 0,2754 \cdot 10^6 \qquad 0,0003856 = 0,3856 \cdot 10^{-3}$$

Dabei wird die Basis 10 fest vereinbart und braucht nicht mit abgespeichert zu werden. Das führt zu der von Taschenrechnern bekannten, kompakteren Darstellung mit dem „E“ als Trennzeichen zwischen Mantisse und Exponent (Beispiel: 0,2754 E 6 bzw. 0,3856 E -3).

Werden Zahlen im Rechner auf diese Weise dargestellt, dann verwendet man z.B. die Basis 2. Die zur Speicherung vorgesehene Bitbreite wird in zwei Bereiche aufgeteilt, in denen jeweils Mantisse und Exponent eingetragen werden. Die Mantisse ist dabei eine vorzeichenbehaftete Festkommazahl, der Exponent (auch Charakteristik genannt) eine vorzeichenbehaftete ganze Dualzahl. Für die Darstellung der Vorzeichen sind verschiedene Realisierungen anzutreffen, eine gebräuchliche ist das Format „Betrag und Vorzeichen“. Damit hat die Gleitkommazahl folgenden Aufbau:

S_M	Betrag der Mantisse	S_E	Betrag des Exponenten

Dabei bedeuten: S_M ... Vorzeichen der Mantisse (1 Bit), S_E ... Vorzeichen des Exponenten (1 Bit). Die Anzahl der Bitstellen für die Beträge von Mantisse und Exponent kann unabhängig voneinander gewählt werden und hängt von der gewünschten Genauigkeit bzw. von der Größe des darzustellenden Zahlenbereichs ab.

Beispiel: Umwandlung der Zahl 13,625 in die binäre Gleitkommadarstellung (16 Bit breite Darstellung mit 10 Bit für die Mantisse und 6 Bit für den Exponenten)

1. Schritt: Umwandlung des Betrages in die binäre Festkommadarstellung

$$13{,}625 = 1101{,}101\,\mathrm{B}$$

2. Schritt: Normalisierung der Mantisse und Bestimmung des Exponenten (Rechtsverschiebung um 4 Stellen → Exponent = 4)

$$1101{,}101\,\mathrm{B} = 0{,}1101\,101\,\mathrm{B} \cdot 2^4 = 0{,}1101\,101\,\mathrm{B} \cdot 2^{100\,\mathrm{B}}$$

3. Schritt: Verkettung von Mantisse (Nachkommateil) und Exponent einschließlich der Vorzeichen zur endgültigen Darstellung

0	1	1	0	1	1	0	1	0	0	0	0	0	1	0	0

Die Normalisierung der Mantisse stellt sicher, daß von rechts beginnend stets die größtmögliche Anzahl von Null verschiedener Ziffern abgespeichert und damit die größtmögliche Genauigkeit erreicht wird. Im gezeigten Beispiel wird die Mantisse jeweils so normalisiert, daß der Vorkommateil gleich Null und die dem Komma folgende Ziffer gleich Eins ist. Gespeichert werden dann nur die Ziffern des Nachkommateils. Zahlen mit Beträgen kleiner als $\frac{1}{2}$ werden durch Linksverschiebungen normalisiert und erhalten einen Exponenten mit negativem Vorzeichen.

Die Normalisierung von Gleitkommazahlen ist auch nach jeder numerischen Operation erforderlich. Die Operationen werden mitunter mit größerer Genauigkeit (größere Darstellungsbreite der Mantisse) durchgeführt, das Ergebnis muß aber wieder auf die ursprüngliche Darstellungsbreite reduziert werden.

Vor einer Addition oder Subtraktion werden Gleitkommazahlen entnormalisiert, damit beide Summanden den gleichen Exponenten erhalten. Dann werden die Mantissen addiert bzw. subtrahiert und anschließend das Ergebnis – ausgehend vom entnormalisierten Exponenten – wieder normalisiert.

Bei der Multiplikation oder Division werden zunächst die Mantissen ohne Berücksichtigung der Exponenten multipliziert bzw. dividiert und der neue Exponent als Summe bzw. Differenz der alten Exponenten bestimmt. Die anschließende Normalisierung liefert das endgültige Ergebnis.

2.6 Codes und nichtnumerische Informationen

Die in den vorangegangenen Abschnitten betrachteten Informationen waren alle numerischer Natur – es waren also in irgendeiner Form Zahlen. Bereits im Abschnitt über gebrochenen Zahlen wurde deutlich, daß nicht alle Zahlen Dualzahlen sind, trotzdem aber als Bitmuster darstellbar und somit auch als Dualzahl gelesen werden können (die Zahl 13,625 als Gleitkommazahl im Abschnitt 2.5.2 ergab das Bitmuster 0110 1101 0000 0100, das auch als 6D04 H = 27 908 gelesen werden kann).

Solche Mehrdeutigkeiten sind in der Rechentechnik nicht nur unumgänglich, sondern sie werden gezielt zur Informationsdarstellung genutzt. Der Vorgang, der sich dahinter verbirgt, ist die *Kodierung*. Die folgenden Abschnitte sollen ein paar wichtige Vertreter aus der nahezu unbegrenzten Menge möglicher Kodierungen vorstellen.

Zunächst jedoch ein Beispiel, das auf den ersten Blick nicht viel mit digitaler Rechentechnik zu tun hat.

2.6.1 Codes – ein einführendes Beispiel

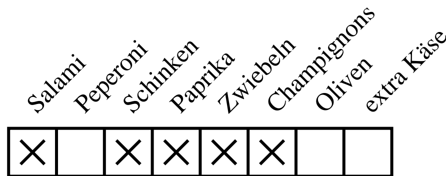
In einem Pizza-Restaurant kann man sich eine Pizza ganz nach seinem Geschmack zusammenstellen. Dazu steht eine Reihe von Pizzaauflagen zur Auswahl, aus denen man sich eine beliebige Kombination zusammenstellen kann. Folgende acht Auflagen werden angeboten:

- Salami
- Pepperoni
- Schinken
- Paprika
- Zwiebeln
- Champignons
- Oliven
- extra Käse

Es sei an dieser Stelle einmal dahingestellt, ob jede Kombination auch wirklich eine gute Pizza ergibt, aber über Geschmack läßt sich ja bekanntlich streiten.

Um nun in diesem Restaurant eine Pizza mit den Auflagen seiner Wahl zu bestellen, hat man sich folgende Vorgehensweise ausgedacht: Auf einem Streifen mit acht Feldern kreuzt man genau die Auflagen an, die auf die Pizza kommen sollen, die anderen

Felder bleiben leer. Dabei ist jedem der acht Felder genau eine der acht Auflagen fest zugeordnet. Entscheidet man sich zum Beispiel für eine Pizza mit Salami, Schinken, Paprika, Zwiebeln und Champignons, dann sieht der Streifen nach dem Ankreuzen wie folgt aus:



Damit ist der Bestellvorgang aber noch nicht beendet. Um eine möglichst kurze Beschreibung der Pizza zu bekommen, wird im folgenden der ausgefüllte Bestellstreifen als Bitmuster einer (vorzeichenlosen) Dualzahl interpretiert. Ein Kreuz entspricht dabei einem gesetzten Bit – also einer Eins, ein freies Feld einem nicht gesetzten Bit – also einer Null. Mit den Kenntnissen aus Abschnitt 2.3 erhält man schnell die gesuchte Zahl in einer beliebigen Basis:

$$1011\,1100\,\mathrm{B} = \mathrm{BC\,H} = 188$$

Mit dieser Zahl bestellt man nun seine Pizza und die Bedienung teilt sie dem Pizzabäcker mit. Dieser macht die Kodierung wieder rückgängig, indem er die Zahl in ein Bitmuster überführt, den Bitstellen die einzelnen Auflagen in der gleichen Weise zuordnet, wie es der Gast bei der Bestellung getan hat, und für die Zubereitung der Pizza genau die Auflagen auswählt, deren Bitstellen auf Eins gesetzt sind.

Die Umwandlung des Pizzawunsches in eine Zahl ist also eine Kodierung, die entstehenden Zahlen nennt man Codewörter oder auch kurz Codes. Das Wesen einer Kodierung besteht darin, eine Zuordnung zwischen Objekten der einen Menge (mögliche Pizzakombinationen) und denen einer anderen Menge (Zahlen von 0 bis 255) zu schaffen. Eine solche Zuordnung kann in Form eines Algorithmus (wie im obigen Beispiel) oder, falls das nicht möglich ist, auch in Form einer Tabelle angegeben werden. Der Grund, warum man überhaupt Kodierungen verwendet, ist leicht einzusehen: Die Codewörter sind oft kürzer als die Beschreibung des Originalobjekts (wie im obigen Beispiel), lassen sich einfacher verarbeiten oder machen Speicherung bzw. Verarbeitung überhaupt erst möglich und haben oft noch andere vorteilhafte Eigenschaften.

Eine Kodierung des Pizzawunsches im obigen Beispiel ist eindeutig umkehrbar, d.h. jeder Kombination von Auflagen wird genau eine Zahl zugeordnet (angefangen mit der „leeren“ Pizza ohne jegliche Auflagen, kodiert durch die Zahl 0, bis hin zur Pizza mit allen möglichen Auflagen, kodiert durch die Zahl 255) und jede Zahl zwischen 0 und 255 entspricht einer anderen Kombination von Auflagen.

Wer keine Pizza mag, dem seien die folgenden Abschnitte empfohlen. An dieser Stelle soll der kulinarische Teil des Themas Kodierung verlassen und die rechentechnisch bedeutsameren Kodierungen in den Mittelpunkt des Interesses gerückt werden, auch wenn die obige Pizzakodierung natürlich sehr gut für eine Darstellung im Rechner geeignet wäre.

2.6.2 BCD-Code

Der BCD-Code ermöglicht es, Dezimalzahlen in einem sehr bequemen Format zu speichern. BCD steht für *B*inary *C*oded *D*ecimal, also binär kodierte Dezimalzahl. Dabei wird nicht der Zahlenwert als solcher abgespeichert, sondern die einzelnen Ziffern der dezimalen Darstellung in kodierter Form. Für die Kodierung der Ziffern wird dann aber wieder das Dualsystem herangezogen (Genaugenommen handelt es sich hierbei nur um einen Spezialfall der BCD-Kodierung, der unter dem Namen nBCD-Kodierung (natürliche BCD-Kodierung) bekannt ist. Es ist aber die am häufigsten angewandte Variante und BCD wird deshalb oft als Synonym für diesen Spezialfall verwendet.):

Dezimalziffer	Binärcodewort	Hexadezimalcodewort
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9

Für die Kodierung der zehn Dezimalziffern sind mindestens 4 Bit notwendig, 3 Bit würden nur acht verschiedene Codewörter ergeben. 4 Bit ergeben aber insgesamt 16 Codewörter, also mehr als benötigt werden. Die sechs nicht in der obigen Tabelle enthaltenen 4-Bit-Wörter 1010 ... 1111 sind sogenannte Pseudotetraden (eine Tetrade ist eine Gruppe von 4 Bits) und nicht Bestandteil des Codes.

Da in den meisten Rechnern eine byteorientierte Speicherung erfolgt, unterscheidet man das ungepackte und das gepackte BCD-Format. Beim ungepackten Format wird pro Byte eine Dezimalziffer gespeichert: Die niederwertigen vier Bit enthalten das

Binärcodewort nach obiger Tabelle, die höherwertigen vier Bit bleiben Null. Bei der gepackten Darstellung finden zwei Dezimalziffern in einem Byte Platz, je eine in der niederwertigen und der höherwertigen Tetrade.

Beispiel: BCD-Kodierung der Dezimalzahl 39

Dezimalzahl	→	BCD-Kodierung			
39	→	0000 0011 0000 1001 B	=	0309 H	(ungepackt)
39	→	0011 1001 B	=	39 H	(gepackt)

Wegen der effektiveren Speicherung wird natürlich häufig das gepackte Format verwendet. Der Vorteil der BCD-Kodierung wird deutlich, wenn „nahe" an der dezimalen Zahlendarstellung gearbeitet wird, also z.B. bei der Ein- und Ausgabe von Dezimalzahlen. Die ziffernweise Umsetzung ist einfacher und schneller als die sonst notwendige Konvertierung von Dualzahlen in die dezimale Darstellung. Bereits in der hexadezimalen Schreibweise von BCD-kodierten Zahlen besteht kein Unterschied mehr zur dezimalen Ziffernfolge.

Nachteilig bei der BCD-Kodierung ist der, durch die nichtgenutzten Pseudotetraden bedingte, eingeschränkte Wertebereich im Vergleich zu Dualzahlen. Während mit einer (vorzeichenlosen) 16 Bit breiten Dualzahl von 0 bis 65535 gezählt werden kann, erreicht man mit dem gepackten BCD-Format nur die Zahlen von 0 bis 9999.

Mit dem hier vorgestellten nBCD-Code können auch arithmetische Operationen, insbesondere Addition und Subtraktion durchgeführt werden. Dazu werden die BCD-kodierten Zahlen zunächst wie (vorzeichenlose) Dualzahlen behandelt. Allerdings müssen die unter Unständen auftretenden Pseudotetraden berücksichtigt werden, d.h. es müssen Korrekturen an den (Zwischen-)Ergebnissen durchgeführt werden. Die notwendigen Anpassungen sind beim nBCD-Code besonders einfach und werden vom Z80-Prozessor mit dem Befehl DAA im Zusammenhang mit den Flags C, H und N unterstützt.

2.6.3 ASCII-Code

Während der BCD-Code zur Darstellung von Zahlenwerten dient und damit noch in den Bereich „numerischer Information" fällt, handelt es sich bei der Speicherung und Verarbeitung von Buchstaben, Ziffern und Sonderzeichen als Grundbausteine von Texten um „richtige" nichtnumerische Informationen. Codes zu deren Darstellung werden auch *alphanumerische Codes* genannt.

Der ASCII-Code (American Standard Code for Information Interchange, CCITT-Code No. 5) ist das wichtigste Mitglied einer Serie von 7-Bit-Codes, die zur Datenübertragung entwickelt wurden. Da es sich bei dieser Version des Codes um die nationale amerikanische Version handelt, wird er oft auch als US-ASCII bezeichnet. Der ASCII-Code wird mit kleinen Modifikationen auch im KC-System verwendet.

Auf Grund der Informationseinheit von 7 Bits können insgesamt 128 Zeichen kodiert werden. Da als rechentechnische Grundeinheit 8 Bits gebräuchlich sind, wird der 7-Bit-Code eingebettet in 8-Bit-Bytes. Das zur Kodierung nicht benötigte Bit – das MSB – wird mitunter zur Fehlerkontrolle eingesetzt (z.B. als Paritätsbit).

Da Buchstaben, Ziffern und Sonderzeichen „von Natur aus" keine Zahlenwerte zugeordnet sind, kann eine Kodierung nur über eine Tabelle erfolgen, die jedes dieser Zeichen auf genau einen Zahlenwert abbildet. Auch im Rechner selbst werden solche Tabellen verwendet.

Wie die Tabelle im Anhang 2 zeigt, wurde der ASCII-Code trotzdem sehr systematisch aufgebaut. Die ersten 32 Zeichen (00 H ... 1F H) sowie das letzte Zeichen (7F H) stellen sogenannte Steuerzeichen dar. Die restlichen Zeichen sind „druckbare" Zeichen, d.h. Zeichen, die im Gegensatz zu den Steuerzeichen lesbar bzw. druckbar sind.

2.6.4 Bitmuster und grafische Information

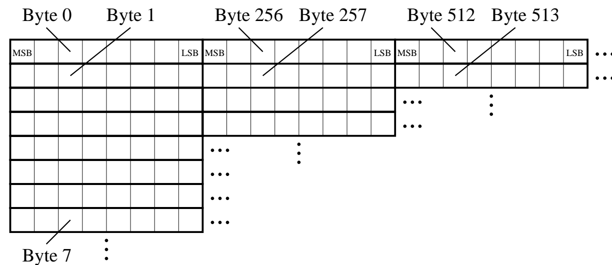
Zum Abschluß des kleinen Rundganges durch die Welt der Kodierung sollen noch einmal Bitmuster betrachtet werden, und zwar als Grundbausteine der grafischen Darstellung.

Das Fernseh- bzw. Monitorbild des KC-Systems besteht bekanntlich aus einer Pixelmatrix mit 320 mal 256 Bildpunkten. Jedes dieser Pixel kann im allgemeinen unter Beachtung gewisser Randbedingungen eine von mehreren Farben annehmen. Im folgenden soll allerdings nur der zweifarbige Fall betrachtet werden, in dem alle Pixel nur eine von zwei Farben annehmen können. Dann kann man eine dieser Farben als Hintergrundfarbe, die andere als Vordergrundfarbe bezeichnen (nach dem Einschalten des KC-Systems ist z.B. Blau als Hintergrundfarbe und Weiß als Vordergrundfarbe eingestellt). Die Information für jedes Pixel ist damit in einem Bit darstellbar, wobei die folgende Zuordnung gilt:

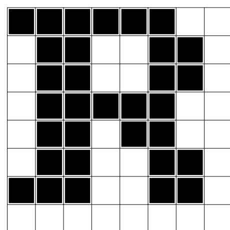
Bit = 0: Pixel zeigt Hintergrundfarbe
Bit = 1: Pixel zeigt Vordergrundfarbe

Beginnend am linken Rand der Pixelmatrix werden jeweils acht nebeneinanderliegende Pixel zu einer Achtergruppe – also einem Byte – zusammengefaßt, wobei das MSB dem am weitesten links liegenden Pixel entspricht, das LSB entsprechend dem an weitesten rechts liegenden. Eine Pixelzeile mit 320 Bildpunkten wird so in eine

Zeile mit 40 Bytes zerlegt und die Pixelmatrix kann nun als eine Matrix aus 40 mal 256 Bytes aufgefaßt werden. Nun werden die Bytes in dieser Matrix noch durchnumeriert, und zwar spaltenweise, beginnend bei Null. Untereinanderliegende Bytes haben damit aufeinanderfolgende Nummern; zwischen zwei nebeneinanderliegenden Bytes besteht eine Differenz von 256. Diese Numerierung entspricht auch genau der Reihenfolge, in der diese Bytes später im Speicher abgelegt werden. Die folgende Abbildung soll das verdeutlichen, sie zeigt die linke obere Ecke des Bildschirmes:



Um nun eine Grafik auf dem KC-Bildschirm darzustellen, muß für jedes gesetzte bzw. nicht gesetzte Pixel das dazugehörige Bit im richtigen Byte gesetzt bzw. zurückgesetzt werden. Für die Darstellung von Textzeichen wird eine Matrix aus Cursorfeldern über die Pixelmatrix gelegt. Der CAOS-Bildschirm besteht z.B. aus 40 mal 32 solchen Cursorfeldern, wovon jedes genau 8 mal 8 Pixel groß ist ($8 \cdot 40$ mal $8 \cdot 32$ ergibt genau 320 mal 256 Pixel). Diese Cursormatrix wird dabei sinnvollerweise so über die Pixelmatrix gelegt, daß jeweils acht aufeinanderfolgende Bytes ein Cursorfeld ergeben. Für die Darstellung eines Zeichens (Buchstabe, Ziffer oder Sonderzeichen) mußsen nun nur noch an der richtigen Stelle acht aufeinanderfolgende Bytes entsprechend dem Zeichenbild des darzustellenden Zeichens belegt werden. Das Zeichenbild wird dabei als Muster bereits irgendwo (unsichtbar) im Speicher bereitstehen, und zwar als Folge von acht Bytes. Für den Buchstaben „R" wäre das z.B. die folgende Bytefolge: FC H, 66 H, 66 H, 7C H, 6C H, 66 H, E6 H, 00 H.



1111 1100 B = FC H
0110 0110 B = 66 H
0110 0110 B = 66 H
0111 1100 B = 7C H
0110 1100 B = 6C H
0110 0110 B = 66 H
1110 0110 B = E6 H
0000 0000 B = 00 H

Soviel zum Thema Grafik und Bildschirmdarstellung am KC, das eigentlich viel komplexer ist und deshalb hier nur angerißen wurde. Mehr dazu wird es an geeigneter Stelle im weiteren Verlauf des Programmierkurses noch geben.

Grafische Information werden also als Bitmuster dargestellt. Anders als bem BCD- oder ASCII-Code liegt die eigentliche Information hier wirklich in den einzelnen Bits und die Interpretation als dualer Zahlenwert macht außer zur kompakteren Schreibweise keinen Sinn. Dementsprechend verschieden sind auch die Operationen, die üblicherweise auf Bitmuster angewendet werden. Während Zahlenwerte gewöhnlich zusammen mit numerischen Operationen (wie z.B. Addition und Subtraktion) verwendet werden, sind bei Bitmustern vorrangig logische Operationen (logische Verknüpfungen AND, OR und XOR sowie Verschiebe- und Rotationsbefehle) anzutreffen.

2.7 Übungsaufgaben für Teil 2

1. Man wandle die folgenden Dezimalzahlen in die Binär-, Hexadezimal- und Oktaldarstellung um (vorzeichenlose Darstellung):

   $$23, \quad 78, \quad 127, \quad 234$$

2. Welcher (vorzeichenlosen) Dezimalzahl entsprechen die folgenden Zahlen?

   1010 1001 B, 3A H, 127 O

3. Man wandle die folgenden vorzeichenbehafteten Dezimalzahlen in die Binärdarstellung unter Verwendung des Zweierkomplements um! Es soll eine 8 Bit breite Darstellung verwendet werden:

   $$27, \quad -27, \quad 123, \quad -123$$

4. Wie sieht die Binärdarstellung im Zweierkomplement der folgenden Zahlen aus, wenn als Breite einmal 8 Bit und einmal 12 Bit verwendet werden sollen:

   $$33, \quad -47$$

   Wie könnte die Regel lauten, wenn Zahlen im Zweierkomplement in eine breitere Binärdarstellung überführt werden sollen (die zusätzlichen Bits sollen dabei jeweils links angefügt werden)?

5. Man führe die folgenden Rechnungen mit Binärzahlen im Zweierkomplement durch:

   $$33 + 67, \quad 74 - 27, \quad -41 + 55, \quad -13 - 38$$

Teil 3 – Der Z80

von Jörg Linder

Die hardwaremäßigen Unterschiede zwischen Grundgerät und Floppy Disk Basis sind zwar immens, aber in einem Punkt sind die Systeme identisch: Beide verwenden als Prozessor einen Z80 (oder eine kompatible CPU). In diesem Abschnitt wird der Z80-Prozessor beschrieben und somit der Grundstein für alle späteren Teile des Kurses gelegt.

3.1 Prozessorarchitektur

Das Blockschaltbild (Bild 2) zeigt den logischen, internen Aufbau des Z80-Prozessors. Es enthält die wichtigsten Bestandteile der CPU, auf die sich die nachfolgenden Erläuterungen beziehen.

3.1.1 CPU-Register

Ein Z80-Prozessor beinhaltet 208 Bits Schreib-/Lesespeicher in Form von Registern, auf die der Programmierer zugreifen kann. Bild 3 gibt einen Überblick, wie dieser Speicher in achtzehn 8-Bit-Register und vier 16-Bit-Register aufgeteilt ist. Es wird unterschieden zwischen zwei Sätzen mit je sechs allgemeinen Register, die als einzelne 8-Bit-Register oder paarweise als 16-Bit-Register verwendet werden können. Außerdem stehen zwei Sätze mit Akkumulator und Flag-Register zur Verfügung. Darüber hinaus gibt es sechs spezielle Register.

3.1.1.1 Spezielle Register

Program Counter (PC), Programmladezähler

Der Programmladezähler enthält die 16-Bit-Adresse des aktuellen Befehls, der aus dem Arbeitsspeicher gelesen wurde. Nach der Ausgabe des Inhalts auf die Adreßleitungen wird der Programmladezähler automatisch erhöht. Durch eine Sprunganweisung wird der neue Wert in den Programmladezähler unter Umgehung der Erhöhungsautomatik geladen.

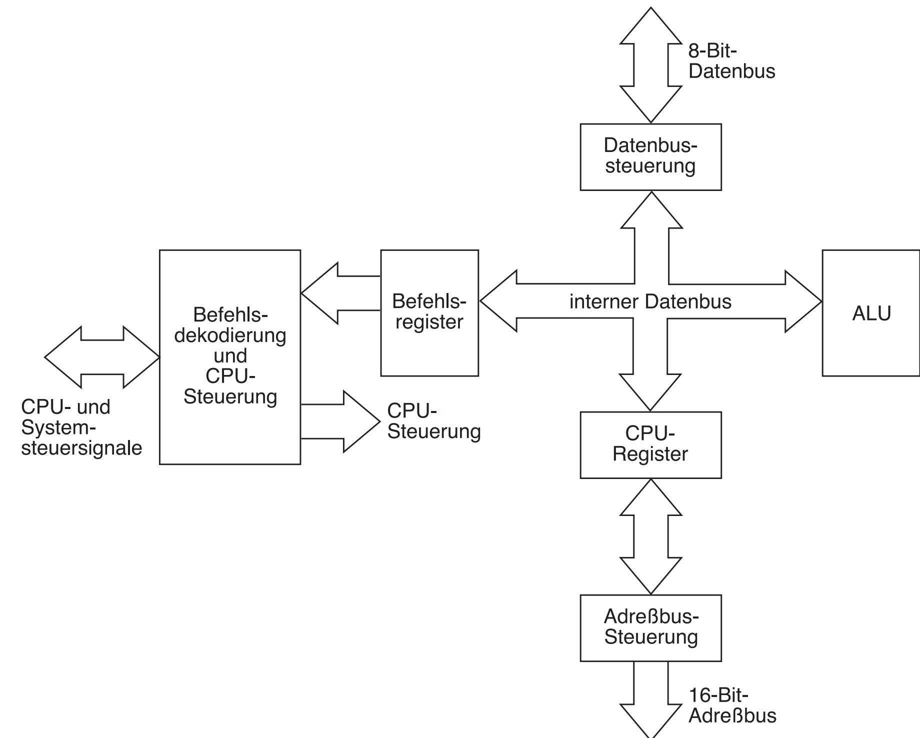


Bild 2: Blockschaltbild des Z80-Prozessors

Stack Pointer (SP), Kellerspeicherzeiger

Im Stack Pointer (Pointer = Zeiger) steht die 16-Bit-Adresse der momentan obersten Speicherzelle des Stack (Top of Stack), die sich an einer beliebigen Stelle im Arbeitsspeicher des Systems befinden kann. Der Stack wird nach dem LIFO-Prinzip verwaltet (last-in-first-out, der zuletzt abgelegte Wert wird als erstes zurückgegeben; manchmal auch als Kellerspeicherprinzip bezeichnet, weshalb der Stack mitunter auch „Kellerspeicher“ genannt wird). Daten können mittels PUSH- bzw. POP-Anweisung aus CPU Registern auf dem Stack abgelegt oder von dort in CPU Register geholt werden. Die vom Stack geholten Daten (POP) sind stets die zuletzt dort abgelegten (PUSH). Der Stack gestattet auf einfache Weise die Einrichtung mehrerer Interruptebenen, die nahezu unbegrenzte Verschachtelung von Subroutinen und die Manipulation von Daten.

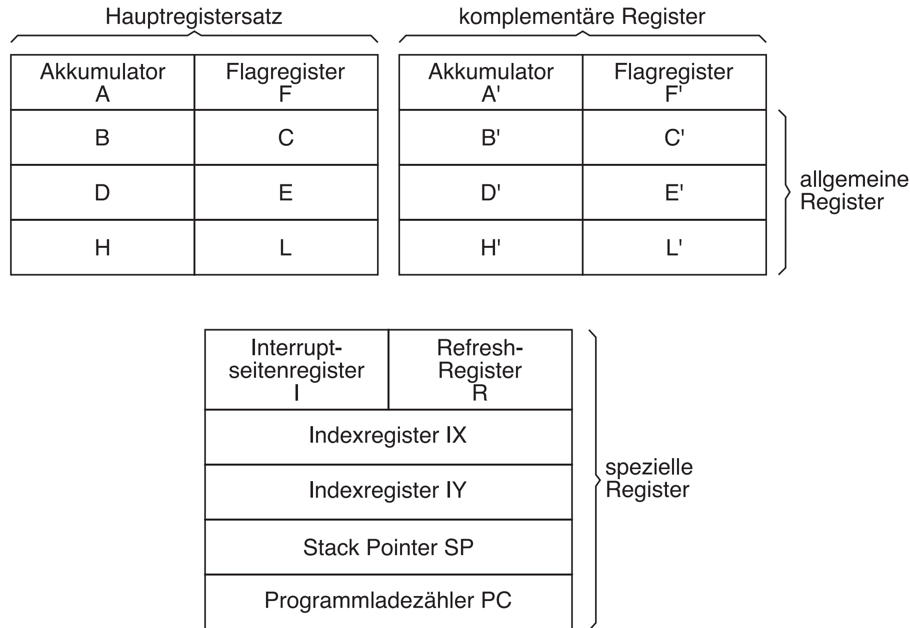


Bild 3: Registersatz des Z80-Prozessors

Indexregister (IX und IY)

Die beiden von einander unabhängigen Indexregister enthalten 16-Bit-Basisadressen, die für indirekte Adressierungen benutzt werden. Dabei dient ein Indexregister als Zeiger auf eine Speicherregion, aus der Daten gelesen oder dort abgelegt werden sollen. Während einer Indexoperation wird mit Hilfe eines zusätzlichen Bytes die Distanz („Entfernung“) der gewünschten Speicherstelle von der Basisadresse angegeben, also adressiert. Derartige Adressierungen vereinfachen in zahlreichen Programmen den Zugriff auf Daten, insbesondere bei Verwendung von Tabellen.

Interrupt Page Address Register (I), Interruptseitenregister

Die Z80 CPU läßt sich in eine spezielle Betriebsart versetzen, in der bei Auftreten eines Interrupts ein indirekter Call zu einer beliebigen Speicherzelle ausgelöst wird. In diesem Fall werden dem Interruptseitenregister I die oberen acht Bits für die Adresse entnommen, während von der interruptauslösenden Einheit die unteren acht Bits für die Adresse bereitgestellt werden. Dadurch ist es möglich, die Interruptserviceroutinen bei minimaler Zugriffszeit auf beliebigen Speicherstellen im Arbeitsspeicher zu plazieren.

Memory Refresh Register (R), Speicherauffrischregister

Anstelle von statischem RAM läßt sich mit der Z80 CPU auch dynamischer RAM verwenden. Die notwendige Auffrischung des Speicherinhalts wird mit Hilfe des Registers R vorgenommen. Nach jeder Befehlserkennung werden die unteren sieben Bits dieses 8-Bit-Registers automatisch um 1 erhöht. Das achte Bit bleibt aus dem Ergebnis eines LD R,A Befehls erhalten. Während die CPU den erkannten Befehl dekodiert und ausführt, werden die Daten des Registers R auf den niederwertigen Teil des Adreßbusses bei aktiviertem Refresh Control Signal ausgegeben. Der Inhalt des Registers I bildet den höherwertigen Teil der Adresse. Dieser Refresh-Modus ist für den Programmierer vollkommen transparent und beeinträchtigt die Verarbeitungsgeschwindigkeit der CPU in keiner Weise. Das Register R kann mit einem speziellen Wert geladen werden, jedoch ist diese Funktion eher zu Testzwecken gedacht und findet in Programmen üblicherweise keine Verwendung.

3.1.1.2 Akkumulator und Flagregister

Im Z80 Prozessor sind zwei von einander unabhängige 8-Bit-Akkumulatoren mit dazugehörigem Flagregister vorhanden. Im Akkumulator wird das Ergebnis einer arithmetischen oder logischen 8-Bit-Operation gespeichert. Mit den Bits des Flagregisters werden spezielle Zustände von 8- oder 16-Bit-Operationen angezeigt (z.B. wenn das Ergebnis einer Operation gleich Null ist). Mit einem Befehl läßt sich das Akkumulator-Flag-Paar des einen Registersatzes gegen das des anderen Registersatzes unabhängig von den anderen Registern tauschen, so daß der Programmierer quasi parallel mit beiden Paaren arbeiten kann.

3.1.1.3 Allgemeine Register

Es sind zwei Sätze mit je sechs allgemeinen Registern vorhanden, die entweder einzeln als 8-Bit-Register oder zusammengefaßt als 16-Bit-Registerpaare benutzt werden können. Sie sind daher vielseitig verwendbar. Die Register des einen Satzes werden mit BC, DE und HL bezeichnet, die Register des komplementären Satzes mit BC',

DE' und HL'. Innerhalb eines Programmes kann zwischen den Sätzen jederzeit gewechselt werden. Dabei wird per Befehl der komplette Satz getauscht; einzelne Register oder Registerpaare lassen sich nicht tauschen. Sind in einem System sehr schnelle Interrupt-Antwortzeiten notwendig, so kann beispielsweise ein Registersatz und ein Akkumulator-Flag-Paar für diese Aufgaben reserviert werden. Anstelle einer umfangreichen Prozedur für das Sichern der Registerinhalte auf dem externen Stack genügen zwei Austauschbefehle, um den jeweils anderen Registersatz zu benutzen.

3.1.2 Arithmetik-Logik-Einheit

Alle arithmetischen und logischen 8-Bit-Operationen der CPU werden von der Arithmetik-Logik-Einheit (Arithmetic Logic Unit, ALU) ausgeführt. Über den internen Datenbus des Prozessors wird die Arithmetik-Logik-Einheit mit Informationen aus den Registern oder vom externen Datenbus versorgt. Von der ALU werden folgende Funktionen ausgeführt:

- Addition
- Subtraktion
- logische UND-Verknüpfung
- logische ODER-Verknüpfung
- logische XODER-Verknüpfung
- Vergleich
- Verschiebung oder Rotation (links- und rechtsherum, arithmetisch und logisch)
- Inkrementieren (um 1 erhöhen)
- Dekrementieren (um 1 verringern)
- Bit setzen
- Bit löschen
- Bit testen

3.1.3 Befehlsregister und CPU-Steuerung

Ein Befehl wird aus dem Arbeitsspeicher zunächst in das Befehlsregister übertragen und dort dekodiert. Dieser Vorgang wird von der Steuereinheit veranlaßt, welche anschließend auch die notwendigen Steuersignale – sowohl interne Signale für die ALU oder das Schreiben und Lesen von Registerdaten als auch alle externen Signale – generiert.

3.2 Befehlssatz

Unter dem Befehlssatz versteht man alle zulässigen Befehle, die der Prozessor verarbeiten kann. Beim Z80 umfaßt der Befehlssatz 158 verschiedene Befehlstypen. Diese sind aber bei weitem nicht so komfortabel wie die Kommandos einer Hochsprache. Einen Befehl für „Ausgabe auf den Bildschirm“ (analog PRINT in BASIC) gibt es hier nicht. Alle Werte bzw. Informationen werden in die Register geladen und dort verarbeitet. Nach Befehlsausführung befindet sich das Ergebnis ebenfalls in einem Register oder wird in den Arbeitsspeicher übertragen. Die Befehle verändern also lediglich die Inhalte der Register und Speicherzellen. Für die Realisierung von Funktionen wie z.B. die Zeichenausgabe auf den Bildschirm ist (fast) immer eine ganze Reihe von Befehlen notwendig.

In der einschlägigen Z80-Literatur wird der Befehlssatz oftmals in verschiedene Funktionsgruppen unterteilt. Dies ist sowohl beim Erlernen der Assemblersprache als auch bei der Suche nach einem bestimmten Befehl sehr hilfreich. Im Anhang dieses Teils unseres Programmierkurses ist eine Übersicht der Befehle zu finden. Sie ist in folgende Gruppen unterteilt, mit denen wir uns anschließend kurz befassen wollen:

- Lade- und Austauschbefehle
- Blocktransfer- und Suchbefehle
- arithmetische und logische Befehle
- Verschiebungs- und Rotationsbefehle
- Bitmanipulationsbefehle
- Sprung-, Ruf- und Rückkehrbefehle
- Eingabe- und Ausgabebefehle
- Steuerbefehle

3.2.1 Befehlsgruppen

Mit den Ladebefehlen werden Daten intern von einem Register zum anderen bewegt oder zwischen einem Register und dem Arbeitsspeicher. Dazu wird stets die Angabe der Quelle (woher) und des Ziels (wohin) benötigt. Die Quelle wird von einem Ladebefehl nicht verändert. Im Gegensatz dazu werden bei den Austauschbefehlen die Inhalte der jeweiligen Register miteinander ausgetauscht.

Eine herausragende Funktionalität bieten die Blocktransfer- und Suchbefehle des Z80. Mit einem einzigen Befehl ist es möglich, den Inhalt eines Speicherbereiches

beliebiger Größe in einen anderen Speicherbereich zu übertragen. Ebenfalls mit einem einzigen Befehl kann in einem Speicherbereich beliebiger Größe nach einem 8-Bit-Wert gesucht werden. Sobald der Wert gefunden oder das Ende des Speicherbereichs erreicht wurde, wird die Suche automatisch beendet. Sowohl die Blocktransfer- als auch die Suchbefehle können per Interrupt unterbrochen werden, so daß die CPU in solchen Fällen nicht für längere Zeit blockiert bleibt.

Die arithmetischen und logischen 8-Bit-Operationen beziehen sich auf die Daten im Akkumulator und in den allgemeinen Registern oder externen Speicherzellen. Nach Ausführung des Befehls liegt das Ergebnis im Akkumulator vor und die entsprechenden Flags werden gesetzt. Analog verhält es sich bei den 16-Bit-Operationen.

Mit Hilfe der Verschiebungs- und Rotationsbefehle kann der Inhalt eines beliebigen Registers oder einer beliebigen Speicherzelle bitweise nach links oder rechts verschoben bzw. rotiert werden. Dabei kann das Carry-Flag mit einbezogen und die Operation sowohl arithmetisch als auch logisch ausgeführt werden.

Zum Setzen, Löschen oder Testen eines Bits im Akkumulator, in einem allgemeinen Register oder in einer beliebigen Speicherzelle stehen die zahlreichen Bitmanipulationsbefehle zur Verfügung.

Die Sprung-, Ruf- und Rückkehrbefehle werden benutzt, um innerhalb eines Programmes während der Abarbeitung zwischen bestimmten Speicherbereichen zu wechseln. Diese Befehlsgruppe benutzt verschiedene Techniken, um eine neue Adresse in den Programmladezähler zu übertragen. Eine Besonderheit stellen die Restart-Befehle dar, die nur 1 Byte umfassen. Sie führen zu definierten Speicherzellen im unteren Adreßbereich.

Für die Kommunikation mit peripheren Geräten oder Baugruppen stehen die Eingabe- und Ausgabebefehle zur Verfügung. Sie ermöglichen sowohl die byte- als auch die blockweise Datenübertragung zwischen einem sogenannten I/O-Port und den Registern oder dem Arbeitsspeicher.

Mit den Steuerbefehlen können verschiedene Optionen und Betriebsarten der CPU eingestellt werden. So kann der Programmierer beispielsweise festlegen, ob und wie die Interruptannahme erfolgen soll.

3.2.2 Mnemonik

In der Übersicht ist zu jedem Befehl der entsprechende Maschinencode in hexadezimaler Form aufgeführt. Da sich aber kein (normaler) Mensch die Bedeutung von Bytekombinationen wie z.B. „ED B0“ merken kann, werden spezielle Kürzel - Mnemonik genannt - verwendet. Art und Umfang der Mnemonik sind im wesentlichen vom Prozessortyp und vom Assembler abhängig. Die von Zilog für den Z80-Prozessor veröffentlichte Mnemonik entstand in Anlehnung die englische Sprache. (Vermutlich werden jetzt einige Leser die Augen verdrehen...)

Sämtliche Quelltexte unsererseits basieren auf der offiziellen Zilog-Mnemonik, was mit einigen Assemblern eventuell zu Problemen führen kann. In diesem Fall sollte man entweder das Handbuch seines Assemblers zu Rate ziehen oder besser gleich die auf der Beilagen-Diskette befindlichen Programme benutzen. (Manchmal sind die Unterschiede nur marginal, wie z.B. im Fall der Robotron K-Mnemonik. Mitunter können aber auch Welten dazwischenliegen, wie z.B. bei der Intel i8080-Mnemonik.)

Abgesehen davon, daß die Kürzel verständlicher sind als der Maschinencode, bringt die Programmierung in Assembler einen weiteren, entscheidenden Vorteil mit sich: Das Programm kann unabhängig von der aktuellen Speicheradresse geschrieben und geändert werden, indem symbolische statt reale Adressen und mnemonische Befehle statt Maschinencode verwendet wird. Im Abschnitt „5.## Assemblernotation“ werden wir genauer darauf eingehen, an dieser Stelle genügt es zu wissen, daß Maschinencode und Assemblersprache nicht ein und dasselbe sind. Ein Z80-Befehl in mnemonischer Darstellung setzt sich aus dem Operationscode (kurz: Opcode) und bis zu zwei Operanden zusammen.

3.2.3 Z80 Statusindikatoren (Flags)

Die Flagregister (F und F’) enthalten Informationen zum Ergebnis der letzten ausgeführten Befehle, wobei die Bits (Flags) innerhalb des Flagregisters eine spezielle Bedeutung haben. Sie sind wie folgt angeordnet:

7	6	5	4	3	2	1	0
S	Z	X	H	X	P/V	N	X

Bild 4: Belegung des Flagregisters

C	Carry-Flag, Übertrag
N	Add/Substract, Addition/Subtraktion
P/V	Parity/Overflow Flag, Parität/Überlauf
H	Half Carry Flag, Halb-Byte-Übertrag
Z	Zero Flag, Nullergebnis
S	Sign Flag, Vorzeichen
X	nicht benutzt

Jeweils sechs Bits der Flagregister werden durch CPU Operationen gesetzt (= 1) oder zurückgesetzt (= 0), die anderen Bits (3 und 5) werden nicht benutzt. Vier Flags (C, P/V, Z und S) können unmittelbar getestet werden und für bedingte Sprung-, Aufruf- oder Rückkehrbefehle verwendet werden. Zwei Flags (H und N) lassen sich nicht unmittelbar testen. Sie werden für BCD-Arithmetik benutzt.

C-Flag (Carry, Übertrag)

Das C-Flag wird in Abhängigkeit von der ausgeführten Operation beeinflußt. Es wird gesetzt, wenn bei Additionen ein Übertrag entsteht oder bei Subtraktionen ein Bit geborgt wird. Anderenfalls wird das C-Flag zurückgesetzt. Dies kann man sich für arithmetische Routinen mit erhöhter Genauigkeit zunutze machen. Auch vom Befehl 'DAA' wird das C-Flag gesetzt, falls die Bedingungen zur Durchführung der Dezimalkorrektur erfüllt waren.

Bei den Verschiebungs- und Rotationsbefehlen wird das höchstwertige oder niederwertigste Bit in das C-Flag übertragen und kann somit als Verbindungsstelle zwischen diesen Bitpositionen von Registern und Speicherstellen dienen.

Logische Befehle (AND, OR und XOR) setzen das C-Flag zurück.

Darüber hinaus kann das C-Flag mit Befehlen gesetzt (SCF) oder umgekehrt (CCF) werden.

N-Flag (Add/Substract, Addition/Subtraktion)

Dieses Flag wird für die Dezimalkorrektur des Akkumulators (Befehl 'DAA') zur Unterscheidung zwischen Addition und Subtraktion benötigt. Bei Additionsbefehlen wird das N-Flag zurückgesetzt und bei Subtraktionsbefehlen gesetzt.

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

Der Status des P/V-Flags hängt von der ausgeführten Operation ab. Bei arithmetischen Befehlen zeigt dieses Flag einen Überlauf an, wenn das Ergebnis im Akkumulator größer als die maximal mögliche Zahl (+127) oder kleiner als die minimal mögliche Zahl (-128) ist. Anhand der Vorzeichenbits der Operanden kann der Zustand des P/V-Flags vorherbestimmt werden.

Additionen von Operanden mit unterschiedlichen Vorzeichen erzeugen keinen Überlauf. Werden hingegen zwei Operanden mit gleichem Vorzeichen addiert und das Ergebnis hat ein anderes Vorzeichen, dann wird der Überlauf durch das gesetzte P/V-Flag angezeigt.

```
    +120 = 01111000
(+) +105 = 01101001
    ---------------
    +225 = 11100001 (-95)
```

Das Resultat der beiden positiven Summanden ist größer als +127 und wird als negative Zahl (-95) dargestellt. Dies ist natürlich nicht korrekt. Aus diesem Grund wird das P/V-Flag gesetzt.

Bei Subtraktionen verhält es sich genau umgekehrt. Operanden mit gleichen Vorzeichen erzeugen keinen Überlauf, jedoch kann dies bei Operanden mit unterschiedlichen Vorzeichen der Fall sein.

```
    +127 = 01111111
(-)  -64 = 11000000
    ---------------
    +191 = 10111111
```

Im oben gezeigten Beispiel ist das Ergebnis nicht korrekt, weil das Vorzeichen des Minuenden von positiv zu negativ wechselte. Das gesetzte P/V-Flag zeigt dies an.

Das P/V-Flag dient nicht nur zur Anzeige des Überlaufs bei arithmetischen Operationen, sondern wird auch dazu benutzt, die Parität des Ergebnisses von logischen Operationen und Rotationsbefehlen anzuzeigen. Dazu wird die Anzahl der gesetzten Bits (= 1) im Byte festgestellt. Ist sie ungerade (engl.: odd), wird das P/V-Flag zurückgesetzt (= 0). Demzufolge wird bei gerader Parität (engl.: even) das P/V-Flag gesetzt (= 1).

Wird mittels IN r oder IN (C) Befehl ein Byte von einem Ein-/Ausgabegerät gelesen, zeigt das P/V-Flag ebenfalls die Parität der gelesenen Daten an.

Während der Blocktransfer- und Blocksuchbefehle gibt das P/V-Flag den Status des Registerpaars BC wieder. Ergibt sich durch die Verringerung Zählers der Wert Null, dann wird das P/V-Flag zurückgesetzt (= 0), anderenfalls ist es gesetzt (= 1).
Die Befehle LD A,I und LD A,R laden den Wert des Interruptfreigabe-Flip-Flops (IFF2) in das P/V-Flag. Diese Funktion ist allerdings eher zu Testzwecken gedacht und findet in Programmen normalerweise keine Anwendung.

H-Flag (Half Carry, Halb-Byte-Übertrag)

Das H-Flag wird gemäß dem Übertrag bzw. dem Borgen zwischen Bit 3 und Bit 4 bei einer arithmetischen Operation gesetzt. Außerdem wird das H-Flag für die Dezimalkorrektur (Befehl 'DAA') nach Addition oder Subtraktion gepackter BCD-Zahlen herangezogen.
Es wird gesetzt (= 1), wenn bei Additionen ein Übertrag von Bit 3 zu Bit 4 erfolgt bzw. bei Subtraktionen von Bit 4 geborgt wird. Tritt bei Additionen kein Übertrag von Bit 3 zu Bit 4 auf oder wird bei Subtraktionen von Bit 4 nicht geborgt, dann wird das H-Flag zurückgesetzt (= 0).

Z-Flag (Zero, Nullergebnis)

Mit dem Z-Flag wird angezeigt, ob das Ergebnis einer Operation Null ist. Es wird gesetzt (= 1), wenn nach einer arithmetischen oder logischen Operation im Akkumulator das Byte 00 steht. Bei jedem anderen Wert wird das Z-Flag zurückgesetzt (= 0).
Analog verhält es sich, wenn ein Byte von einem Ein-/Ausgabegerät per IN r oder IN (C) Befehl gelesen wird. Ist der Wert des gelesenen Bytes gleich Null, dann wird dies durch das gesetzt Z-Flag angezeigt. Anderenfalls ist das Z-Flag zurückgesetzt.
Bei der Übertragung eines Bytes zwischen einer Speicherstelle und einem Ein-/Ausgabegerät (Befehle INI, IND, OUTI und OUTD) wird das Z-Flag gesetzt, wenn der Inhalt des Registers B (Zähler) beim Verringern den Wert Null erreicht.
Das Z-Flag wird bei Vergleichs- und Suchoperationen gesetzt, wenn eine Übereinstimmung mit dem Wert im Akkumulator auftritt.
Wird der Zustand eines Bits in einem Register oder einer Speicherstelle mittels BIT-Befehl getestet, dann nimmt das Z-Flag den komplementären Zustand dieses Bits an: Ist das getestete Bit gesetzt (= 1), wird das Z-Flag zurückgesetzt (= 0) und umgekehrt.

S-Flag (Sign, Vorzeichen)

Der Zustand des höchstwertigen Bits (Bit 7) des Akkumulators widerspiegelt sich im S-Flag. Bei arithmetischen Operationen mit vorzeichenbehafteten Zahlen wird das Bit 7 für das Vorzeichen benutzt. Bei positiven Zahlen ist das Bit 7 zurückgesetzt (= 0), bei negativen gesetzt (= 1). Die Zahl selbst - also der Wert - ist in den Bits 0 bis 6 gespeichert. Daraus ergibt sich ein Wertebereich von 0 bis 127 für positive und von -1 bis -128 für negative Zahlen.
Das S-Flag zeigt auch während der Datenübernahme von einem Ein-/Ausgabegerät mittels IN r oder IN (C) Befehl an, ob es sich bei dem gelesenen Byte um einen positiven (S = 0) oder negativen (S = 1) Wert handelt.

3.3 Befehlssatz – Tabellarische Übersichten

8-Bit-Ladebefehle

Quelle (Spalten) / Ziel (Zeilen)

Ziel		implizit		Register							Register indirekt			indiziert		erweit. Adr.	direkt
		I	R	A	B	C	D	E	H	L	(HL)	(BC)	(DE)	(IX+d)	(IY+d)	(nn)	n
Register	A	ED 57	ED 5F	7f	78	79	7A	7B	7C	7D	7E	0A	1A	DD 7E d	FD 7E d	3A nn	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
Register indirekt	(HL)			77	70	71	72	73	74	75	76						36 n
	(BC)			02													
	(DE)			12													
indiziert	(IX+d)			DD 77 d	DD 70 d	DD 71 d	DD 72 d	DD 73 d	DD 74 d	DD 75 d	DD 76 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 76 d						FD 36 d n
erweit. Adr.	(nn)			32 nn													
implizit	I			ED 47													
	R			ED 4F													

16-Bit-Ladebefehle

Quelle (Spalten) / Ziel (Zeilen)

Ziel		Register							erweit. Direkt-wert	erweit. Adr.	indir. Reg.-Adr.
		AF	BC	DE	HL	SP	IX	IY	nn	(nn)	(SP)
Register	AF										F1
	BC								01 nn	ED 48 nn	C1
	DE								11 nn	ED 58 nn	D1
	HL								21 nn	2A nn	E1
	SP				F9		DD F9	FD F9	31 nn	ED 78 nn	
	IX								DD 21 nn	DD 2A nn	DD E1
	IY								FD 21 nn	FD 2A nn	FD E1
erweit. Adr.	(nn)		ED 43 nn	ED 53 nn	22 nn	ED 73 nn	DD 22 nn	FD 22 nn			
indir. Reg.-Adr. (PUSH Befehle →)	(SP)	F5	C5	D5	E5		DD E5	FD E5			

↑ POP Befehle (Spalte (SP))

Achtung: Die PUSH- und POP-Befehle verändern den SP-Inhalt nach jeder Ausführung

Austauschbefehle

		implizite Adressierung				
		AF'	BC', DE' und HL'	HL	IX	IY
implizit	AF	08				
implizit	BC DE und HL		D9			
implizit	DE			EB		
indir. Reg.-Adr.	(SP)			E3	DD E3	FD E3

Block-Ladebefehle

Ziel			indir. Reg.-Adr. (HL)	
Ziel	indir. Reg.-Adr.	(DE)	ED A0	"LDI" – Laden (DE) <-- (HL) Erhöhen HL und DE, Verringern BC
			ED B0	"LDIR" – Laden (DE) <-- (HL) Erhöhen HL und DE, Verringern BC, wdh. bis BC = 0
			ED A8	"LDD" – Laden (DE) <-- (HL) Verringern HL und DE, Verringern BC
			ED B8	"LDDR" – Laden (DE) <-- (HL) Verringern HL und DE, Verringern BC, wdh. bis BC = 0

Registerpaar HL – bezeichnet die Quelle
DE – bezeichnet das Ziel
BC – ist der Bytezähler

Block-Vergleichsbefehle

SUCH-Adresse

indir. Reg.-Adr. (HL)	
ED A1	"CPI" Erhöhen HL, Verringern BC
ED B1	"CPIR" Erhöhen HL, Verringern BC wdh. bis BC = 0 oder bis Übereinstimmung gefunden
ED A9	"CPD" Verringern HL und BC
ED B9	"CPDR" Verringern HL und BC wdh. bis BC = 0 oder bis Übereinstimmung gefunden

Registerpaar HL – bezeichnet den Speicherplatz, der mit dem Akkumulatorinhalt verglichen werden soll
BC – ist der Bytezähler

Ausgabebefehle

			Quelle							
			Registeradressierung							indir. Reg.
			A	B	C	D	E	H	L	(HL)
OUT	Direkt-wert	n	D3 n							
OUT	indir. Reg.	(C)	ED 79	ED 41	ED 49	ED 51	ED 59	ED 61	ED 69	
Ausgabe und Erhöhen HL, Verringern B OUTI	indir. Reg.	(C)								ED A3
Ausgabe und Erhöhen HL, Verringern B wdh. bis B = 0 OTIR	indir. Reg.	(C)								ED B3
Ausgabe und Verringern HL und B OUTD	indir. Reg.	(C)								ED AB
Ausgabe und Verringern HL und B whd. bis B = 0 OTDR	indir. Reg.	(C)								ED BB

Ziel-Kanaladresse

Block-Ausgabe-Befehle

Eingabebefehle

Ziel			Kanaladresse: Direktwert n	Kanaladresse: indir. Reg. (C)
Eingabe IN	Registeradressierung	A	DB n	ED 78
		B		ED 40
		C		ED 48
		D		ED 50
		E		ED 58
		H		ED 60
		L		ED 68
Eingabe und Erhöhen HL, Verringern B INI	indirekt Register	(HL)		ED A2
Eingabe und Erhöhen HL, Verringern B, wdh. bis B = 0 INIR				ED B2
Eingabe und Verringern HL und B IND				ED AA
Eingabe und Verringern HL und B, wdh. bis B = 0 INDR				ED BA

Block-Ausgabe-Befehle (INI, INIR, IND, INDR)

8-Bit-Arithmetikbefehle

Quelle	Registeradressierung							indir. Reg.-Adr.	indizierte Adressierung		Direkt-wert
	A	B	C	D	E	H	L	(HL)	(IX+d)	(IY+d)	n
Addition ADD	87	80	81	82	83	84	85	86	DD 86 d	FD 86 d	C6 n
Addition mit Übertrag ADC	8F	88	89	8A	8B	8C	8D	8E	DD 8E d	FD 8E d	CE n
Subtraktion SUB	97	90	91	92	93	94	95	96	DD 96 d	FD 96 d	D6 n
Subtraktion mit Übertrag SBC	9F	98	99	9A	9B	9C	9D	9E	DD 9E d	FD 9E d	DE n
UND-Verknüpfung AND	A7	A0	A1	A2	A3	A4	A5	A6	DD A6 d	FD A6 d	E6 n
exklusives ODER XOR	AF	A8	A9	AA	AB	AC	AD	AE	DD AE d	FD AE d	EE n
ODER-Verknüpfung OR	B7	B0	B1	B2	B3	B4	B5	B6	DD B6 d	FD B6 d	F6 n
Vergleich CP	BF	B8	B9	BA	BB	BC	BD	BE	DD BE d	FD BE d	FE n
Erhöhen um 1 INC	3C	04	0C	14	1C	24	2C	34	DD 34 d	FD 34 d	
Verringern um 1 DEC	3D	05	0D	15	1D	25	2D	35	DD 35 d	FD 35 d	

16-Bit-Arithmetikbefehle

Ziel		Quelle: BC	DE	HL	SP	IX	IY
ADD	HL	09	19	29	39		
ADD	IX	DD 09	DD 19		DD 39	DD 29	
ADD	IY	FD 09	FD 19		FD 39		FD 29
Addition mit Übertrag und Flagsetzer ADC	HL	ED 4A	ED 5A	ED 6A	ED 7A		
Subtraktion mit Übertrag und Flagsetzer SBC	HL	ED 42	ED 52	ED 62	ED 72		
Erhöhen um 1	INC	03	13	23	33	DD 23	FD 23
Verringern um 1	DEC	0B	1B	2B	3B	DD 2B	FD 2B

AF-Operationen und Steuerbefehle

Befehl		Code
Einstellen der Dezimale des Akkumulators (Dezimalkorrektur)	DAA	27
Komplement des Akkumulators (Einerkomplement)	CPL	2F
Vorzeichenumkehr des Akkumulators (Zweierkomplement)	NEG	ED 44
Komplement des C-Flags	CCF	3F
Setzen des C-Flags	SCF	37

Befehl	Code
NOP	00
HALT	76
INT abweisen DI	F3
INT annehmen EI	FB
Interruptmodus 0 setzen IM0	ED 46
Interruptmodus 1 setzen IM1	ED 56
Interruptmodus 2 setzen IM2	ED 5E

Bitbefehle

Test "BIT"	Registeradressierung							indir. Reg.	indiziert	
	A	B	C	D	E	H	L	(HL)	(IX+d)	(IY+d)
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	CB 75	CB 76	DD CB d 76	FD CB d 76
7	CB 7F	CB 78	CB 79	CB 7A	CB 7B	CB 7C	CB 7D	CB 7E	DD CB d 7E	FD CB d 7E

Bitbefehle

		Registeradressierung							indir. Reg.	indiziert	
		A	B	C	D	E	H	L	(HL)	(IX+d)	(IY+d)
Löschen “RES”	0	CB 87	CB 80	CB 81	CB 82	CB 83	CB 84	CB 85	CB 86	DD CB d 86	FD CB d 86
	1	CB 8F	CB 88	CB 89	CB 8A	CB 8B	CB 8C	CB 8D	CB 8E	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 B6	DD CB d B6	FD CB d B6
	7	CB BF	CB B8	CB B9	CB BA	CB BB	CB BC	CB BD	CB BE	DD CB d BE	FD CB d BE

Bitbefehle

		Registeradressierung							indir. Reg.	indiziert	
		A	B	C	D	E	H	L	(HL)	(IX+d)	(IY+d)
Setzen “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

Verschiebebefehle

Quelle und Ziel

	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		

Sprung-/Rücksprungbefehle und Unterprogrammaufruf

Bedingung

			un-bedingt	Übertrag C	kein Übertrag NC	Null Z	nicht Null NZ	Parität ungerade PE	Parität gerade PO	negativ M	positiv P	Reg. B ≠ 0
Sprung JP	erweit. Direktwert	nn	C3 nn	DA nn	D2 nn	CA nn	C2 nn	EA nn	E2 nn	FA nn	F2 nn	
Sprung JR	relativ	PC + e	18 e-2	38 e-2	30 e-2	28 e-2	20 e-2					
Sprung JP	indirektes Register	(HL)	E9									
Sprung JP		(IX)	DD E9									
Sprung JP		(IY)	FD E9									
CALL	erweit. Direktwert	nn	CD nn	DC nn	D4 nn	CC nn	C4 nn	EC nn	E4 nn	FC nn	F4 nn	
Verringern B, Sprung bei B ≠ 0 DJNZ	relativ	PC + e										10 e-2
Rücksprung RET	indirektes Register	(SP) (SP + 1)	C9	D8	D0	C8	C0	E8	E0	F8	F0	
Rücksprung von Interrupt RETI	indirektes Register	(SP) (SP + 1)	ED 4D									
Rücksprung von nichtmaskierbarem Interrupt RETN	indirektes Register	(SP) (SP + 1)	ED 45									

Achtung: Einige Flags haben mehr als eine Bedeutung.

Restart-Befehle

	CALL Adresse	OP CODE	
CALL Adresse	0000 H	C7	RST 0
	0008 H	CF	RST 8
	0010 H	D7	RST 16
	0018 H	DF	RST 24
	0020 H	E7	RST 32
	0028 H	EF	RST 40
	0030 H	F7	RST 48
	0038 H	FF	RST 56

Teil 4 – Assembler

von Ralf Kästner

4.1 Einleitung

Nachdem Euch Frank und Jörg in den drei ersten Teilen unseres Kurses mit den leider recht trockenen theoretischen Grundlagen „gelangweilt“ haben, wird es nun langsam ernst. Nach einer Einführung zu unserem zukünftigen Werkzeug – dem Assembler – schreiben wir heute unser erstes Programm in Assemblersprache und erzeugen daraus ein Maschinenprogramm, welches der Z80 direkt abarbeiten kann. Ich werde versuchen, diesen ersten praktischen Schritt ganz langsam und relativ ausführlich zu erklären, da man zwar beispielsweise im Handbuch zum M027 DEVELOPMENT ab Seite 68 ein Beispiel nachlesen kann, leider fehlt dort jeglicher Kommentar nur die entsprechenden Bildschirmausgaben wurden abgedruckt. Mir ging es am Anfang so, daß ich niemand zum Fragen hatte und vieles mühsam durch Ausprobieren herausfinden mußte, das schult zwar unheimlich, kostet aber auch jede Menge (unnötiger) Zeit, vielleicht geht es ja bei dem Einen oder Anderen nach dem Durcharbeiten der heutigen Zeilen etwas schneller als damals bei mir!

Wie die vorhergehenden Abschnitte gezeigt haben, ist es bei der Programmierung in Assembler gar nicht so einfach einen Anfang zu finden, da man sich zunächst sehr viel Grundlagenwissen aneignen müßte, um halbwegs gute Programme zu schreiben. Wenn man dann nach einer gewissen Zeit langsam anfängt „binär zu denken“ und den Befehlssatz des Z80 nachts um 2 Uhr zu 80 Prozent im Schlaf aufsagen kann, hat man es fast schon geschafft, bei mir hat das etwa 2 Jahre gedauert. Ich will keineswegs Panik machen aber um die Grundlagen kommt man früher oder später nicht mehr herum und die Z80-Befehle sollte man schon größtenteils kennen, je früher desto besser – das ist schließlich das Handwerkszeug, wenn man beim Programmieren ist und den Quelltext eintippt, es würde ziemlich lange dauern, wenn man bei jedem Befehl erst in einer Tabelle nachschlagen müßte.

Wenn das also alles so schwierig ist, warum machen wir das überhaupt – es gibt doch BASIC, TURBO-PASCAL usw.? Nun diese Frage läßt sich ziemlich schnell beantworten. Unser KC wird mit 1,75 MHz getaktet, ist also im Vergleich zu anderen Heimcomputern ziemlich langsam, ganz zu schweigen von heute üblichen Taktfrequenzen, welche sich im Bereich von mehreren 100 MHz bewegen. Nur mit der Assemblersprache hat man „den direkten Draht“ zur CPU – dem Herz jedes Mikrorechners. Man bestimmt mit seinem Programm unmittelbar die Abläufe, Arbeitsweise und damit die Ausführungsgeschwindigkeit, da keine Schicht (z.B. BASIC-Interpreter im HCBASIC) wie in einer Hochsprache zwischen Prozessor und Programm liegt und nur dadurch läßt sich die physikalisch mögliche Geschwindigkeit aus dem Computer herausholen.

Der zweite Punkt wären die programmtechnischen Möglichkeiten, es gibt im Prinzip keinerlei Einschränkungen, man ist der absolute und uneingeschränkte „Herrscher“ über den Rechner, er macht genau das, was man ihm sagt – nicht mehr und auch nicht weniger (das ist häufig der Grund, daß er nach dem Start eines selbstgeschriebenen Programmes scheinbar gar nichts mehr macht bzw. irgendetwas, was zunächst nicht so genau erklärbar ist...).

Drittens sind in Assembler programmierte Programme für vergleichbare Probleme i.d.R. kürzer als die gleiche Lösung geschrieben in einer Hochsprache. Das letztendlich in beiden Varianten entstehende ausführbare Maschinenprogramm benötigt bei Assemblerprogrammierung weniger Speicher, da man als Programmierer bestimmt, was übersetzt wird und so direkten Einfluß auf die Programmgröße nehmen kann. In Hochsprachen hängen beispielsweise Compiler benutzte Bibliotheken komplett an das Programm mit an, auch wenn nur eine Bibliotheksroutine genutzt wird.

Der vierte wichtige Grund für die Programmierung speziell des KC 85 in Assembler liegt im modularen Grundkonzept des KC-Systems. Da wäre z.B. der Hauptspeicher – der KC hat einen 16 Bit breiten Adressbus, womit 64 kB Hauptspeicher direkt angesprochen werden können. Davon benötigt das Betriebssystem mind. 8 kB, so daß unter günstigsten Umständen max. 56 kB nutzbar sind. Zieht man weitere 8 kB für das Betriebssystem ab (CAOS ROM C – wird von CAOS automatisch geschaltet und steht somit nicht uneingeschränkt zur Verfügung), nochmal 10 kB für den Bildwiederholspeicher (IRM) und knapp 5 kB Notizspeicher für das Betriebssystem kommt man unter ungünstigsten Umständen auf einen uneingeschränkten Hauptspeicher von etwa 33,3 kB, was nicht gerade berauschend ist! Die Speicherbereiche in zusätzlich gesteckten RAM-Modulen bzw. die 2 bereits vorhandenen RAM8-Blöcke bieten zwar erhebliche Speicherreserven aber so ohne Weiteres kann man mit ihnen nicht arbeiten. Sie müssen für eine Benutzung in eigenen Programmen erst in den 64kB Hauptspeicherbereich des Prozessors eingeblendet werden – an dieser Stelle kommt man um die Programmierung in Maschinencode auch kaum herum. Das gleiche Problem besteht oft bei Programmen für die anderen Module, speziell Ein- bzw. Ausgabeschnittstellen, dort sind eigentlich immer Maschinencodeprogramme für eine effektive Ausnutzung aller Möglichkeiten notwendig.

Soviel zur Einstimmung – Lohn aller Mühe ist ein schnelles fehlerfreies und schönes Programm, das ist auch oder gerade mit dem Assembler möglich, zunächst ist hier

aber erst mal die Klärung von einigen Begriffen notwendig, damit wir alle die gleiche Sprache sprechen.

4.2 Begriffsdefinitionen

Die Erstellung von Maschinenprogrammen unterscheidet sich nicht wesentlich von der Programmerstellung in anderen Programmiersprachen, z.B. in BASIC.

Zunächst sind die einzelnen Befehle des Programmes in der Programmiersprache, welche heute nun die ASSEMBLERSPRACHE ist, zu erfassen. Dazu verwendet man üblicherweise ein Textprogramm. Den fertig geschriebenen Text bezeichnet man als QUELLPROGRAMM, QUELLTEXT oder QUELLCODE (engl.: Source-Program oder Source-Code):

CAOS: *NAME.ASM* CP/M: *NAME.MAC*

Das Quellprogramm versteht der Z80 natürlich noch nicht, es ist ja zunächst nur Text. Es muß von einem Programm, das man ASSEMBLER-PROGRAMM oder auch nur kurz ASSEMBLER nennt, erst noch übersetzt werden. Dieses Assembler-Programm übersetzt also die einzelnen Befehle des Quellprogramms, die in der jeweiligen Assemblersprache geschrieben sein müssen, in die entsprechenden Maschinenbefehle des Z80 und gibt das Ergebnis i.d.R. in Form einer neuen Datei auf einen externen Datenträger (unter CP/M und CAOS) oder auch direkt in den Hauptspeicher (nur unter CAOS) aus. Die Begriffe Assembler und Assemblersprache sind also nicht das Gleiche, obwohl sie oft synonym verwendet werden. Zusätzlich gibt es auch noch verschiedene Varianten von Assemblersprachen, welche sich in der Schreibweise der einzelnen Befehle unterscheiden. Ein bestimmtes Assembler-Programm akzeptiert meist auch nur eine Schreibweise einer Assemblersprache, beide müssen also zusammenpassen.

Die Assembler unter CP/M erzeugen einen sog. ZWISCHENCODE (im Speicher frei verschiebliche Folge von Maschinenbefehlen mit relativen, auf eine Leitadresse bezogenen Speicheradressen), der noch nicht dem fertigen Maschinenprogramm entspricht:

CAOS: *entfällt* CP/M: *NAME.REL*

Dieser Zwischencode muß erst noch mit einem anderen zweiten Programm, dem LINKER (deutsch: BINDER, PROGRAMMVERBINDER), in das fertige Maschinenprogramm übersetzt werden, wobei dann wiederum eine neue Datei entsteht.

Das fertig übersetzte Programm wird in der Literatur unterschiedlich bezeichnet, unter CAOS werden wir es als MASCHINENPROGRAMM oder MASCHINENCODE (kurz MC) bezeichnen. Unter CP/M ist der Begriff OBJEKTCODE üblich, wobei vielfach ebenfalls Maschinencode verwendet wird. Fertige und für den Z80 direkt ausführbare Programme haben abhängig von der KC-Betriebsart eine festgelegte Dateierweiterung:

CAOS: *NAME.KCC* CP/M: *NAME.COM*

Diese Dateien stellen damit das Ziel all unserer Wünsche dar, das jeweils laufende Betriebssystem oder diverse Dienstprogramme können diese Dateien in den Hauptspeicher unseres Mikrorechners laden und die CPU, unser Z80, kann sie anschließend direkt ausführen, ohne daß weitere Programme, wie Laufzeitbibliotheken (BRUN.COM unter CP/M) oder Interpreter (HCBASIC unter CAOS oder BASI unter CP/M) notwendig wären. Selbst das Betriebssystem des jeweiligen Mikrorechners ist nach dem Laden und Starten eines Maschinenprogrammes nicht mehr erforderlich, da das Programm selbst durch entsprechende Befehle an die CPU den Speicher und die Ein-/Ausgabebausteine kontrollieren und steuern kann – das Betriebssystem ist nämlich auch nur ein Maschinenprogramm, das irgendwann mal von einem Programmierer geschrieben wurde.

Dies ist natürlich nicht Sinn und Zweck der Assemblerprogrammierung, in der Regel nutzt man die dokumentierten Software-Schnittstellen des Betriebssystems in den eigenen Programmen, da sie einem bereits sehr viel immer wiederkehrende Probleme, wie Bildschirmausgaben oder Tastaturabfragen abnehmen. Bei sehr verbreiteten Standards, wie dem Betriebssystem CP/M, kann das sogar noch erheblich weiter gehen. Nutzt man ausschließlich dokumentierte Schnittstellen des Betriebssystems und spricht evtl. zusätzlich vorhandene Hardware (D001 des KC unter CP/M) gar nicht an, laufen Maschinenprogramme auch auf anderen Mikrorechnern, wenn diese ebenfalls CP/M-kompatibel (gleichwertige Version vorausgesetzt) sind. Dahinter kann sich eine vollkommen andere Hardware-Plattform verstecken, nur ein Z80 muß als CPU eingebaut sein – der vom Assemblerprogramm aus dem Quellprogramm erzeugte Maschinencode ist immer auf einen bestimmten Prozessor bezogen und wenn wir Maschinencode für den Z80 erzeugen, läuft dieses Maschinenprogramm auch nur auf einem Computer mit eingebautem Z80.

Wer in Assemblersprache programmiert, hat also die folgenden Grundeigenschaften dieser Sprache zu akzeptieren:

- Sie ist an einen bestimmten Prozessortyp gebunden, so daß nur eine maschinenbezogene Programmierung möglich ist. Jede Änderung der Gerätetechnik bedeutet auch eine unmittelbare Änderung des Programmes.

- Sie spiegelt den Befehlssatz und die Architektur (Register, Organisation des Statusspeichers und des Unterbrechungssystems, Adressierungsarten usw.) des Prozessors wider.
- Die Programmstruktur bleibt beim Übersetzungsvorgang unverändert, da jede in Assemblersprache geschriebene Anweisung genau einem Maschinenbefehl entspricht.

4.3 Assembler-Entwicklungssysteme

Im vorhergehenden Abschnitt wurde gezeigt, wie ein ausführbares Maschinenprogramm entsteht und genau diese Schritte sind jedesmal als MINIMUM notwendig! Was macht man aber, wenn das erzeugte Programm fehlerhaft arbeitet oder bestimmte häufig benutzte (fertige) Programmroutinen nicht jedesmal neu übersetzt werden sollen, sondern beispielsweise als Toolsammlung (SYSLIB) zusammengefaßt in einer Datei zur Verfügung gestellt werden sollen, wo man bei Bedarf dann darauf zugreift?

Für diese Probleme gibt es weitere Programme, die den Assemblerprogrammierer bei seiner Arbeit unterstützen.

Der BIBLIOTHEKAR dient zum Bilden und Verwalten einer Bibliothek aus übersetzten Zwischencode (s.o.), Programm-Modulen, also *.REL-Dateien. Der Verfasser dieser Bibliothek gibt die Modulnamen und Aufrufbedingungen seiner Bibliotheksroutinen bekannt und jeder andere Programmierer kann sie dann in seinen Programmen einfach aufrufen und so mitbenutzen. Beim Verbinden der einzelnen *.REL-Dateien durch den LINKER (s.o.) ergibt sich das fertige Programm dann aus der REL-Datei des eigenen Programms und den REL-Dateien der benutzten Bibliotheksroutinen.

Der DEBUGGER (deutsch: Entwanzer) unter CP/M oder TESTMONITOR unter CAOS dient zum Austesten von fertigen Maschinenprogrammen. Er stellt verschiedene Funktionen zur Verfügung, wie:

- Vergleich, Suche und Modifikation von Daten im Hauptspeicher
- Echtzeittest von Programmen und Manipulation der CPU-Register
- Reassemblierung (Rückübersetzung) von Maschinencode in Assemblersprachbefehle
- Aufruf von Systemunterprogrammen
- Hilfsfunktionen, wie hexadezimale Addition/Subtraktion oder Adressberechnungen von Hand

Die Gesamtheit all dieser verschiedenen Programme und Hilfsmittel wird als ASSEMBLER-ENTWICKLUNGSSYSTEM bezeichnet. In der nachfolgenden Übersicht ist alles noch mal getrennt für die beiden Betriebsarten des KC zusammengefaßt, es gibt dort einige Unterschiede:

CP/M – ASM oder M80

TP.COM bzw. EDIT.COM	Textprogramm zum Erfassen des Quellprogramms (*.MAC)
ASM.COM	Assemblerprogramm zum Übersetzen des Quelltextes (*.MAC) in Zwischencode (*.REL)
LIB.COM	Bibliothekar für die Verwaltung von Zwischencodemodulen
LINK.COM	Linker zum Verbinden von Zwischencode-Modulen (*.REL) in ausführbare Maschinenprogramme (*.COM)
DU.COM	Debugger für das Testen von fertigen Maschinenprogrammen
DISASS.COM	Rückübersetzungsprogramm, das aus einem Maschinenprogramme (*.COM) Assemblerquelltext (*.MAC) erzeugt
HIST.UTL, TRACE.UTL	Hilfsprogramme für DU.COM, das verschiedene zusätzliche Funktionen bereitstellen (Erzeugung eines Histogrammes der relativen Ansprechhäufigkeit von Adressen in einem zu testenden Programm, Protokollierung des Programmverlaufes beim Test)

CAOS – EDAS

Unter CAOS werden die Einzelbestandteile u.a. durch das Modul M027 DEVELOPMENT zur Verfügung gestellt, welches 1987 von Mühlhausen für den KC 85/3 herausgegeben wurde. Die jeweiligen Programme befinden sich auf mehreren EPROM-Speicherschaltkreisen und sind nicht so scharf abgegrenzt, wie unter CP/M. Diese Vermischung und einige weitere später noch behandelte Einschränkungen machen die Programmierung etwas gewöhnungsbedürftig, nachfolgend kurz die Programme bzw. Kommandos:

EDIT (EDAS)	Textprogramm zum Erfassen des Quelltextes (*.ASM)
ASM (EDAS)	Assemblerprogramm zum Übersetzen des Quelltextes in ein ausführbares Maschinenprogramm

Bemerkung: Der Quelltext muß sich zum Zeitpunkt der Übersetzung bereits im Hauptspeicher befinden. Die Erzeugung von relativem Zwischencode und alle damit verbundenen weiteren Möglichkeiten entfallen. Es gibt demzufolge auch keinen speziellen Linker oder Bibliothekar unter CAOS.

TEMO	Debugger für das Testen von fertigen Maschinenprogrammen
DISASS	Rückübersetzungsprogramm, das aus einem Maschinenprogramm, welches sich im Hauptspeicher befinden muß, Assemblerquelltext erzeugt und auf dem Bildschirm ausgibt
CDISASS	wie DISASS, jedoch wird der erzeugte Quelltext in Form einer *.ASM-Datei auf Kassette/Diskette geschrieben und kann mit EDAS weiterbearbeitet werden

4.4 EDAS – Assemblerprogrammierung unter CAOS

Nachdem wir zum diesjährigen Treffen das Für und Wider diskutiert hatten, sind wir zu der Ansicht gelangt, daß der Einstieg in die Assemblerprogrammierung mit Hilfe von EDAS unter CAOS einfacher ist als unter CP/M. In einem späteren Teil unseres Kurses soll aber auch über das CP/M-System etwas geschrieben werden. Ich setze ein System mit einem KC 85/4 voraus, prinzipiell gilt zwar auch alles für den KC 85/2 oder /3, da ich aber keinen besitze, kann ich Beispiele nicht auf diesen Rechnern abtesten.

Wie in den vorangegangenen Abschnitten zu lesen war, sind unter CAOS prinzipiell nur zwei Schritte für die Erzeugung von ausführbaren Maschinenprogrammen notwendig:

1. Erfassen des Quelltextes mit einem Textprogramm
2. Übersetzen der Quelle mit dem Assemblerprogramm in ausführbaren Maschinencode

Beide Programme (Textprogramm, Assemblerprogramm) und weitere Funktionen sind im Programmpaket EDAS (EDitor/ASsembler) enthalten, das über eine dem CAOS-Betriebssystem ähnliche Menüleiste bedient wird. Wir werden allerdings NICHT das M027 DEVELOPMENT nutzen, da es mittlerweile dank Mario Leubner eine wesentlich verbesserte Version von EDAS gibt – dies ist EDAS164A.KCC, welches mit der zugehörigen Beschreibung EDAS164.DOK auch noch mal auf der heutigen Beilagendiskette zu finden ist, für den KC 85/2,3 mit CAOS 3.1 ist EDAS161.KCC zu verwenden.

Zum Treffen tauchte das Handbuch zum M027 in Form einer TPKC-Textdatei auf. Dort wird EDAS in der Version 1.4 beschrieben. Es ist deshalb notwendig, sich beim Ausprobieren unserer Beispiele entweder nach den in den folgenden Abschnitten genannten Hinweisen zu richten oder die beiliegende Datei EDAS164.DOK zu Rate zu ziehen, am zweckmäßigsten druckt man sie zuerst mal aus, damit man bei der Arbeit mit EDAS 1.6 dort sofort nachschlagen kann. Erst wenn man in den beiden genannten Quellen nichts findet, sollte man im Handbuch zum M027 nachlesen, wie bereits gesagt, es gibt einige Unterschiede zwischen Version 1.6 und 1.4. Hervorragend für die Programmierung mit EDAS ist allerdings der gesamte Abschnitt 3.5 des M027 geeignet. Dort findet man eine für die Assemblerprogrammierung unbedingt notwendige Liste aller zulässigen Z80-Befehle mit der richtigen Schreibweise (Syntax) für EDAS, nebst Anzahl der Maschinenzyklen und Beeinflussung der Flags (Register F, siehe Punkt 3.2.3.) durch jeden Befehl. Sucht man bei der Programmierung nach einem bestimmten Befehl oder will man seinen Einfluß auf bestimmte Flags herausfinden, kommt man damit schneller zum Ziel als mit den Tabellen aus Abschnitt 3.3.

4.4.1 Laden und Starten von EDAS

Es ergibt sich damit erst einmal die Frage, wie wird EDAS 1.6 auf dem eigenen System gestartet? Benutzer des D004-Floppy-Disk-Aufsatzes starten natürlich erst mal die CAOS-Betriebsart des Systems.

CAOS 4.3	Wiederum glücklich können sich alle Benutzer von CAOS 4.3 mit erweitertem ROMC-Bereich schätzen, dort befindet sich EDAS 1.6 bereits in einem ROMC-Block und ist nach Eingabe von %EDAS sofort arbeitsbereit (und kann durch „wildgewordene selbstverschriebene“ Programme auch nicht versehentlich zerstört werden!).
CAOS 3.1	Beim KC 85/3 ist zunächst mit dem Befehl %SWITCH 2 0 der BASIC-ROM auf Adresse 0C000H zu deaktivieren, alle weiteren Schritte entsprechen CAOS 4.1/4.2, auch nach dem Betätigen der RESET-Taste ist dieser Schritt notwendig!
CAOS 4.1/4.2	EDAS 1.6 benötigt 8 kB Hauptspeicher von 0C000H bis 0DFFFH.

Im Normalfall hat der KC 85/4 dort ein „Speicherloch“, der CAOS-ROMC wird intern von CAOS 4.x nur dann ein- und auch wieder ausgeschaltet, wenn dort befindliche CAOS-Routinen benötigt werden, der BASIC-ROM auf dieser Adresse ist beim KC 85/4 nur eingeschaltet, wenn BASIC aktiv ist – in diesem Bereich ist also im Normalfall gar kein Speicher vorhanden. Wir müssen deshalb VOR DEM LADEN von EDAS 1.6 erst einmal RAM-Speicher zur Verfügung stellen, sonst wird das Programm ins Leere geladen! Dazu eignen sich alle externen RAM-Module, die sich auf Adresse 0C000H aktiv schalten lassen, wie sonst auch, sind die Modulprioritäten zu beachten, Beispiele:

`%SWITCH 8 C3`	16 kB RAM-Modul M022 in Schacht 08
`%SWITCH 1C C3`	64 kB RAM-Modul M011 in Schacht 1C

Damit ist im gewünschten Bereich RAM vorhanden und wir können EDAS nun laden mit:

```
%FLOAD
Name:EDAS164A
```

Der Nachteil der RAM-Variante wurde oben schon angedeutet, wird der Speicherbereich, wo sich EDAS befindet, unbeabsichtigt durch falsche Programmierung oder abgestürzte Programme überschrieben, funktioniert auch EDAS nicht mehr richtig oder ist (nach RESET) ganz aus dem CAOS-Menü verschwunden! In gewisser Weise kann man sich davor schützen, indem man NACH dem Laden von EDAS den Schreibschutz des verwendeten RAM-Moduls aktiviert und damit hardwaremäßig Schreibzugriffe blockiert:

`%SWITCH 8 C1`	16 kB RAM-Modul M022 in Schacht 08
`%SWITCH 1C C1`	64 kB RAM-Modul M011 in Schacht 1C

Nachdem dies alles erledigt ist, rufen wir mit %MENU das CAOS-Menü auf und dort müßten jetzt ganz oben zwei neue Befehle zu lesen sein.

`%EDAS`	Kaltstart von EDAS
`%REEDAS`	Warmstart von EDAS

Bevor man %REEDAS benutzen kann, muß %EDAS bereits einmal aufgerufen worden sein. Man kann nach dem Verlassen von EDAS mittels *QUIT (s.u.) mit Hilfe von %REEDAS den unveränderten Quelltext weiterbearbeiten, da keine Initialisierung des von EDAS verwendeten Speichers erfolgt. Da wir eben EDAS 1.6 neu geladen hatten, beginnen wir mit der Eingabe von:

```
%EDAS
```

Der Bildschirm wird gelöscht und EDAS ermittelt im Hintergrund, wieviel Speicher zur Verfügung steht. Die maximal möglichen Grenzen schlägt EDAS nun zur Bestätigung vor, womit man für die begonnene Sitzung die Speicherbelegung festlegt:

Top of Text: 0200	Festlegen der unteren RAM-Grenze, dort beginnt der Speicher für den Quelltext
End of Text: C000	Festlegen der oberen RAM-Grenze + 1, dort endet die Markentabelle (s.u.) des Assemblers
Start of MC: 0200	Modifizierung des standardmäßigen Maschinencodebereiches, wenn das übersetzte Programm direkt in den Hauptspeicher assembliert wird
ASM-Offset: 0000	Modifizierung des Assembleroffsets für die Option „O“ (s.u.)

Die physische Speicheradresse, das ist die Adresse, auf der das erzeugte Programm abgearbeitet werden kann, des mit dem Assembler erzeugten Programmes ergibt sich aus logischer Adresse („Start of MC“) plus angegebenen Assembleroffset. Man kann damit Quellprogramme für beliebige Adressen assemblieren, es wird aber immer vom Assembler an der gleichen, mit „Start of MC“ festgelegten Adresse im Hauptspeicher abgelegt, dort läßt es sich aber nicht abarbeiten, wenn man einen Offset ungleich 0000 angegeben hat!

Keine Angst, wenn Ihr das eben nicht verstanden habt, eine Änderung der von EDAS vorgeschlagenen Werte ist nur in speziellen Fällen notwendig, ich selbst habe es noch nie benötigt! In 99,9 % aller Fälle betätigt man vier mal die Taste >ET< und übernimmt damit die von EDAS ermittelten Vorschläge, es ergibt sich beim KC 85/4 dann die nachfolgend dargestellte Aufteilung des Hauptspeichers:

Nach dem Kaltstart ist natürlich weder Quelltext noch Markentabelle vorhanden, der gesamte benutzte Speicher ist noch frei – zum besseren Verständnis der später behandelten Arbeitsweise soll dies an dieser Stelle aber schon mal mit dargestellt werden. An der oberen Grenze erkennen wir einen wesentlichen Vorteil von EDAS 1.6 gegenüber früheren Versionen – es arbeitet mit RAM von 200H bis maximal 0C000H. Dazu wird der IRM, der sich normalerweise von Adresse 08000H bis 0BFFFH im Hauptspeicher der CPU befindet, bei Zugriff auf den dort liegenden RAM weggeschalten. Dadurch stehen fast 48 kB Hauptspeicher für die Arbeit mit EDAS zur Verfügung, was für kleine bis mittlere Projekte in der Regel vollkommen ausreicht, der BASIC-Interpreter hat auch nicht mehr Speicher zur Verfügung, also eine tolle Leistung von Mario!

Nachdem die Speicherabfragen des Kaltstarts erledigt sind, wird der Bildschirm gelöscht und in den KOMMANDOMODUS von EDAS übergegangen:

```
>>> Edas V1.6a <<<   Frei: BDFE   Lw:EF
---------------------------------------
*MENU
*QUIT
*CLEAR
*SAVE
*LOAD
*VERIFY
*PRINT
*KEY
*ASM
*FIND
*TOP
*BOTTOM
*EDIT
*_
```

Beim Start von EDAS bzw. Verlassen des Editors erscheint das gezeigte verkürzte Menü. Wie bereits dargestellt, ist vor den EDAS-Kommandos ein STERN und kein Prozentzeichen einzugeben, ansonsten entspricht die Bedienung vollkommen der vom CAOS-Menü des Betriebssystems.

In der Statuszeile wird in der Mitte der verfügbare EDAS-Speicher hexadezimal angezeigt, rechts kann man in Abhängigkeit vom eigenen System sehen, ob das Ausgabemedium Kassette oder Diskette ist. Arbeitet im D004 eine DEP-Version ab 2.0, dann ist dort Laufwerk und eventuell Userbereich (wenn ungleich 0) ersichtlich.

Als erste Handlung gebt nun *MENU ein und bestätigt mit der >ET<-Taste. Nun wird das vollständige Menü angezeigt, das in der Version 1.6 immerhin 27 Kommandos enthält, wenn man ein D004 angeschlossen hat.

Im folgenden Text werden diese Kommandos funktionell in mehrere Gruppen aufgeteilt und ihr Sinn und Zweck kurz erklärt. Man erhält so zunächst einen Überblick, welche Möglichkeiten bei der Lösung einer bestimmten Aufgabe zur Verfügung stehen, mitunter werden Begriffe verwendet, welche erst später genau erläutert werden.

4.4.2 Steuerkommandos

Diese Kommandogruppe dient der Steuerung von EDAS bzw. des verwendeten eigenen KC-Systems.

*MENU — Das vollständige EDAS-Menü wird ausgeschrieben, es reicht auch die Eingabe von *M und >ET<.

*QUIT — Damit verläßt man den Kommandomodus von EDAS und kehrt zu CAOS zurück. Das CAOS-Fenster wird auf volle Bildschirmgröße eingestellt.

*KEY (n) — Mit diesem Kommando kann man sich die Belegung der Funktionstasten auflisten lassen oder Funktionstasten neu belegen, nachfolgend die Optionen für den Parameter n:

ohne n	KEYLIST (Aufruf Menükommandoroutine wie in CAOS)
n = 0	Grundbelegung des F-Tastenspeichers herstellen
1 ≤ n ≤ C	KEY (Aufruf Menükommandoroutine wie in CAOS)
n = D5	Grundbelegung für D005-Tastatur herstellen, (jetzt ist F5 Tabulator und F6 Find-Funktion, die anderen F-Tasten sind mit den entsprechenden Zeichencodes belegt)

Bei angeschlossenem Floppy erscheinen im vollständigen Menü weitere Menüworte, die der Diskettenarbeit dienen. Das ehemals notwendige Programm DEVEX.KCC (macht EDAS 1.4 vom DEVELOPMENT-Modul M027 diskettenfähig) brauchen wir nicht mehr! Die dort enthaltenen Funktionen wurden von Mario bereits fest in EDAS

1.6 integriert, so daß die Arbeit an dieser Stelle erheblich vereinfacht wird, u.a. erkennt EDAS 1.6 automatisch, ob mit D004 gearbeitet wird.

DIR Es erfolgt die Anzeige des Inhaltsverzeichnisses der eingelegten Diskette, und der freien Speicherkapazität. Ein „“ nach dem Dateinamen zeigt an, daß die Datei schreibgeschützt ist. Die Tasten BRK und STOP sind während der Ausgabe wirksam.

*ERA Löschen einer Datei, deren vollständiger Name einzugeben ist. Vor Ausführung erfolgt noch die Schutzabfrage „Erase ?“, die mit „Y“ zu beantworten ist, wenn die Datei wirklich gelöscht werden soll.

*REN Datei umbenennen, es ist der alte und neue Dateiname vollständig mit Dateityp anzugeben.

*TAPE / *DISK
Umschaltung auf Kassette bzw. auf Diskette (und das letzte aktuelle Laufwerk).

*DRIVE Nach Aufforderung ist das neue Laufwerk (Buchstabe) und eventuell noch dahinter der User-Bereich (Hexzahl) einzugeben. Das Ergebnis ist in der Statuszeile zu sehen, User 0 wird dort nicht angezeigt.

4.4.3 Erfassen und Bearbeiten des Quelltextes

Diese Kommandogruppe faßt alle Befehle zusammen, mit denen der Quelltext eines Programmes verändert werden kann. Er muß sich immer in einem zusammenhängenden Teil des Hauptspeichers des KC befinden. Der verwendete Speicherbereich wird beim Kaltstart durch die Abfragen „Top of Text“ und „End of Text“ festgelegt. Will man während einer EDAS-Sitzung davor oder dahinter Teile des Speichers anderweitig verwenden, sind die vorgeschlagenen Werte der Abfragen entsprechend zu verändern. Wird der maximal mögliche Speicher von ca. 48 kB verwendet, lassen sich Maschinenprogramme von durchschnittlich 6 kB Länge erzeugen. Reicht dies nicht aus, ist der Quelltext entsprechend sinnvoll aufzuteilen und die Teile einzeln zu übersetzen. Mit dem CAOS-Kommando %FLOAD können die übersetzten Teilprogramme nacheinander in den Hauptspeicher geladen werden und über %FSAVE dann als Gesamtprogramm gespeichert werden. Dies entspricht einem Linken „von Hand“ und wenn man an diese Grenze stößt, sollte man auf die CP/M-Assembler umsteigen, wo es keine derartigen Speicherbeschränkungen gibt.

Der in EDAS integrierte Editor stellt das Textprogramm dar, welches man sonst für eine Bearbeitung von Quelltexten benötigt. Im EDITIERMODUS befindet man sich, wenn der Quelltext auf dem Bildschirm angezeigt wird. Es stehen dann verschiedene Tastenfunktionen für die Bearbeitung des Quelltextes zur Verfügung, eine genaue Aufstellung ist in der beigefügten EDAS-Beschreibung zu finden. Der Editiermodus wird mit >BRK< beendet, man gelangt wieder in den Kommandomodus und das verkürzte Menü wird angezeigt.

Es gibt Kommandos, die nach Aufruf direkt in den Editiermodus führen und Kommandos, welche nach Aufruf den Quelltext unsichtbar bearbeiten und sofort wieder in den Kommandomodus zurückkehren. Um das Ergebnis der zweiten Kommandoart zu sehen, ruft man nach Ausführung ein Kommando der ersten Art auf. Dies klingt vielleicht zu Anfang etwas kompliziert, aber man gewöhnt sich mit der Zeit daran. Man sollte sich aber wirklich die Zeit nehmen und mit der zweiten Gruppe etwas üben, man kann durch die Nutzung dieser Kommandos viel Zeit und Tipparbeit einsparen!

*EDIT / *TOP / *BOTTOM / *FIND (s)
Diese vier Kommandos bilden die erste Gruppe, nach Aufruf befindet man sich im Editiermodus und der Quelltext wird angezeigt.

Mit *EDIT kommt man an die Stelle, an der man den Editiermodus das letzte Mal abgebrochen hat. *TOP führt zum Textanfang und die erste Seite des Quelltextes wird angezeigt, *BOTTOM stellt das Gegenteil dar, die letzte Seite des Quelltextes wird angezeigt.

Mit Hilfe von *FIND (s) kann man nach einer Zeichenkette bzw. Teilen davon (nach Aufruf hinter Text: anzugeben) im Quelltext suchen lassen. Der Parameter s (beliebiger Wert) stellt einen Schalter dar, im Normalfall, wenn man nichts angibt, wird im Quelltext von ganz vorn gesucht, gibt man s an, ab Seitenanfang der zuletzt bearbeiteten Quelltextseite. Wird die gesuchte Zeichenkette im Quelltext gefunden, sieht man die betreffende Seite auf dem Bildschirm und in der 3. Zeile blinkt der Cursor direkt auf dem 1. Zeichen der gefundenen Kette. EDAS merkt sich immer die zuletzt eingegebene Suchkette und man kann innerhalb des Editiermodus (Quelltext wird angezeigt!) jederzeit mit >F2< (D005: >F6<) ab der aktuellen Cursorposition die Suchoperation wiederholen, auch wenn man zwischendurch ganz andere Tätigkeiten durchgeführt hat. Wird die Zeichenkette nicht gefunden, so geht der Cursor zum Textende.

*PDEL / *PCOPY
Diese beiden Kommandos erlauben das Löschen bzw. Kopieren von markierten Textblöcken. Ein Textblock wird durch zwei „@“ eingegrenzt, welche in der ersten Spalte einer Quelltextzeile stehen müssen. Bewährt hat

sich folgende Vorgehensweise: Man ruft zunächst mit einem o.g. Kommando den Editiermodus auf. Dann stellt man den Cursor auf die letzte Zeile vor dem zu markierenden Textblock und betätigt >ET<. Dadurch wird eine Leerzeile erzeugt, die dem Blockanfang entspricht; man braucht jetzt nur noch nach ganz links zu gehen und das erste „@“ zu setzen. Am Blockende stellt man sich auf die letzte Zeile des zu markierenden Blockes und verfährt ebenso mit dem zweiten „@“. Durch die zwei Leerzeilen mit „@“ in der ersten Spalte kann man den Block optisch gut erkennen und man übersieht Beginn und Ende auch beim Durchblättern nicht so schnell.

Mit dem Kommando *PDEL wird der zuvor markierte Textblock gelöscht. Hat man mal ein „@“ vergessen zu setzen, hat der Befehl keine Wirkung (Schutzfunktion). Hat man dagegen beispielsweise aus einer vorhergehenden anderen Blockaktion noch weitere „@“ im Quelltext, kann das ins Auge gehen, die erste gefundene Markierung im gesamten Text beginnend von ganz vorn ist immer der Blockanfang, die zweite das Blockende, alle weiter hinten liegenden interessieren EDAS nicht. Da alle Blockbefehle aus dem Kommandomodus heraus gestartet werden, sieht man nicht direkt, was wirklich passiert. Die einzige sichere Kontrolle VOR Blockaktionen besteht darin, daß man mit der *FIND-Funktion den Quelltext nach den ersten beiden „@“ absucht (zur Erinnerung: nach der gefundenen ersten Markierung >F2< bzw. mit D005 >F6< betätigen). Insbesondere wenn man sich nicht ganz sicher ist, sollte man diese Kontrolle wirklich durchführen, es geht schneller als sich den mühevoll entwickelten Quelltext der letzten Arbeitsstunde noch mal auszudenken und einzugeben!

Mit dem Kommando *PCOPY kann man schließlich den markierten Textblock an eine andere Stelle kopieren, der zuvor markierte Block UND SEINE ANFANG- bzw. ENDE-MARKIERUNG bleiben unverändert erhalten, so daß man den Befehl auch mehrmals direkt hintereinander aufrufen kann. So lassen sich bequem ähnliche Quelltextteile zur Einsparung von Texteingaben vervielfältigen, es geht meist schneller, Vorhandenes zu verändern, als alles noch mal komplett neu einzugeben. Zur Vermeidung der eben genannten Gefahren empfiehlt es sich, sofort nach Abschluß der PCOPY-Aktion die beiden Markierungen zu löschen oder mit *PDEL den gesamten Block, wenn man ihn nicht mehr benötigt! Diese Kombination von *PCOPY mit nachfolgendem *PDEL muß man auch für Textverschiebungen benutzen, da es dafür keinen eigenen Befehl gibt!

Bleibt noch ein Problem – wie sage ich EDAS bei *PCOPY, wo das Ziel dieser Aktion liegt bzw. wohin wird der Block kopiert? Zum Treffen mußte ich etwas verwundert feststellen, daß das selbst Leuten Sorgen bereitet, die bereits längere Zeit mit EDAS arbeiten! Das „Geheimnis“ dieser ganzen Geschichte liegt darin, daß EDAS beim Einfügen von zusätzlichem Quelltext (Bildschirm)seitenorientiert arbeitet, sprich die Cursorposition im Quelltext ist total egal, das einzig Entscheidende ist die erste und damit oberste Bildschirmzeile, die man (nur) im Editiermodus sehen kann. Vor dieser Zeile liegt die Trennstelle, wo der neue Quelltext eingefügt wird!!! Was muß man also tun, um das Ziel von *PCOPY festzulegen, man begibt sich per Blättern im Text oder mit Hilfe der *FIND-Funktion erst einmal auf die Seite, wo der Block landen soll. Dann scrollt man mit >CUD< (wenn der Text zuende ist, einfach den Cursor auf die letzte Textzeile bewegen und mit >ET< Leerzeilen erzeugen!) die Seite solange nach oben, bis die Zeile, welche vor dem einzufügenden neuen Block stehen soll, oben herausgerutscht ist, dann müßte die erste nach dem einzufügenden neuen Block stehende Zeile die oberste auf dem Bildschirm sein. Hat man das geschafft, geht es mit >BRK< in den Kommandomodus, nun kann man *PCOPY eintippen, bestätigen und mit *EDIT wieder zurück in den Editiermodus an die gleiche Stelle gehen – nun müßte der eingefügte Textblock zu sehen sein.

*REPL (s)

REPL kommt von REPLACE und bedeutet soviel wie Ersetzen; der Befehl dient zum Ersetzen von Zeichenketten durch eine andere (ohne Schutzabfrage!). Es wird nach Aufruf zuerst die zu suchende (ersetzende) Zeichenkette abgefragt und anschließend die Ersatzkette, die dafür eingesetzt wird. Die Anzahl der Ersetzungen wird dezimal ausgegeben. Besonders nützlich ist diese Möglichkeit bei massiven Markenumbenennungen, z.B. nach Durchlauf von Maschinencode durch den Markenreassembler QMR. Wird der Schalter s nicht angegeben, wird von vorn mit dem Ersetzen begonnen, mit Schalter s ab aktuellem Seitenanfang.

*CLEAR

Diese Funktion löscht den von EDAS benutzten Hauptspeicher wieder, wo sich der Quelltext und die Markentabelle (s.u.) befinden können. Durch zwei unabhängige Fragen kann auch nur der Text- bzw. nur der Markenspeicher gelöscht werden.

4.4.4 Speichern, Laden und Ausdrucken des Quelltextes

Diese Kommandogruppe dient zur Kommunikation von EDAS mit externen Geräten, wie der Kassette, Diskette oder dem Drucker. Die Druckerdaten werden über den USER-Ausgabekanal 2 (Systemschnittstelle UOUT1) ausgegeben, wo man auch eigene Druckertreiberroutinen eintragen kann. Wird mit dem CAOS-Druckertreiber gearbeitet, lassen sich über >SH+CLR< alle Bildschirmausgaben protokollieren, was man bei bestimmten Funktionen vorteilhaft nutzen kann, z.B. Ausdruck einer kompletten Markentabelle mit Hilfe von *LBLIST (s.u.). Mittels *TAPE bzw. *DISK kann man die Ein-/Ausgaben von/auf Kassette bzw. Diskette leiten (s.o.), nachfolgend wird lediglich auf die Diskette/Festplatte eingegangen.

*SAVE / *PSAVE

*SAVE gibt immer den kompletten im Hauptspeicher befindlichen Quelltext als Datei auf Diskette aus. Nach Eingabe der Dateibezeichnung und Bestätigung wird erst eine gleichnamige Datei gesucht, und wenn vorhanden zur Abfrage „(O)verwrite, (B)ackup, (S)kip ?“ übergegangen. Diese ist mit dem gekennzeichneten Buchstaben zu beantworten. (O für überschreiben, B für Anlegen einer *.BAK-Datei und S für Abbruch). Die Erzeugung der Backupdatei wird in folgenden Teilschritten ausgeführt:

- Löschen einer eventuell vorhandenen Datei NAME.BAK
- Umbenennen der Datei NAME.ASM in NAME.BAK
- Abspeichern der neuen Datei NAME.ASM

Abgespeichert wird im CAOS-Format mit Vorblock. Der letzte verwendete Dateiname wird wieder vorgegeben. Als Dateityp wird stets ASM vorgegeben, er kann (sollte aber nicht für eine eindeutige Kennzeichnung der Datei!) überschrieben werden.

Mit *PSAVE kann der aktuelle Textblock (s.o.) abgespeichert werden, welcher durch die ersten beiden „@“ im Quelltext eingegrenzt wird. Bei nur einem „@“ wird ab dieser Stelle bis zum Textende gespeichert. Mit Hilfe dieser Funktion kann man sich eine Sammlung (Bibliothek) nützlicher Routinen selbst erzeugen, sie lassen sich dann über *LOAD (s.u.) beliebig oft an die benötigte Stelle in andere Quelltexte einbauen.

*VERIFY

Die gleichartige CAOS-Routine (bei Quelltexten mit mehr als 0FFH Blöcken wird der Vergleich systembedingt bei FF> abgebrochen) für die Überprüfung der fehlerfreien Speicherung von Kassettenaufzeichnungen wird aufgerufen.

*LOAD

Mit dieser Funktion werden abgespeicherte Quelltexte wieder in den Hauptspeicher des KC eingeladen. Durch den gegenüber den Vorgängerversionen von EDAS vergrößerten Textspeicher sind Blocknummern über FFH erforderlich. Das Dateiende wird deshalb nur noch bei Erkennen des Codes 03H ausgewertet. Eingelesen werden können sowohl CAOS-Dateien mit Vorblock als auch solche ohne CAOS-Vorblock, die Unterscheidung erfolgt automatisch.

Beim Aufruf von *LOAD ist der momentane Zustand des Editors in Bezug auf den Quelltextspeicher zu beachten. Nach einem Kaltstart von EDAS ist alles klar, der Textspeicher ist leer, entweder man tippt direkt neuen Quelltext ein oder lädt einen gespeicherten Text mit Hilfe von *LOAD für die weitere Bearbeitung. In beiden Fällen beginnt der Text bei TOP, dem Beginn des Textspeichers.

Ist man jetzt mitten in der Arbeit und hat bereits im Editiermodus gearbeitet, ist beim Aufruf von *LOAD zu beachten, daß zu ladender Quelltext IMMER vor die erste Textzeile der zuletzt bearbeiteten Bildschirmseite eingefügt wird!!!

Das ist genau die gleiche Handhabung, wie eben bei den Blockbefehlen erläutert wurde. Daraus ergeben sich dann zwei Varianten, entweder man will einen Text ganz neu bearbeiten, dann löscht man vor *LOAD mit *CLEAR den kompletten Textspeicher oder man will den zu ladenden Text in den bereits vorhandenen einfügen, dann ist wieder die betreffende Textseite so zu scrollen, daß die Einfügestelle vor der obersten Bildschirmzeile liegt – die Cursorposition im Quelltext ist vollkommen egal (im Handbuch zum M027 auf Seite 7 falsch beschrieben!).

*PRINT (n) / *PPRINT (n)

Mit diesen beiden Befehlen kann man den Quelltext bzw. Teile davon auf einem Drucker zu Papier bringen. Der Parameter n ist optional und bewirkt folgendes:

ohne n	Ausgabe des Quelltextes auf den Bildschirm
n = 0	Ausgabe auf Drucker im Endlosdruck
n = 1...99	Ausgabe auf Drucker mit Blattwechsel nach n Zeilen. (n ist dezimal anzugeben)

PRINT schickt den kompletten Text an den Drucker, PPRINT gibt nur den durch 2 mal „@“ eingegrenzten Textblock aus.

4.4.5 Übersetzen des Quelltextes in ein Maschinenprogramm

Nachdem man sich mühevoll mit dem Editor von EDAS vertraut gemacht und seine Gedanken in Form von Assemblerbefehlen unter Aufbietung aller geistigen Kräfte eingetippt hat, fehlt nun noch die Krönung – die Erzeugung des ausführbaren Maschinenprogrammes durch das Assemblerprogramm. Dieses Programm versteckt sich in EDAS hinter dem Kommando *ASM und wird wie jede andere Funktion einfach aus dem Kommandomodus heraus aufgerufen. Der zu übersetzende Quelltext muß sich im Hauptspeicher befinden, ist also vorher einzugeben oder einzuladen. Findet der Assembler keinen Quelltext, landet man sofort wieder im Kommandomodus. Direkt hinter *ASM sind zunächst keine Parameter anzugeben, nachdem man aber *ASM aufgerufen hat, fragt der Assembler nach diversen Optionen. Diese sind nun hinter dem Doppelpunkt anzugeben, wobei gleichzeitig auch mehrere Optionen zulässig sind. Diese Optionen steuern den Assembler und sagen ihm, wie er den Übersetzungsvorgang ausführen soll. Nachfolgend eine unkommentierte Aufstellung aller Optionen, unbekannte Begriffe werden später an passender Stelle erklärt:

- \+ eine bereits vorhandene Markentabelle wird vor dem Assemblerlauf nicht gelöscht.
- 1 es wird nur der Pass 1 (Erzeugung der Markentabelle) ausgeführt.
- 2 es wird nur der Pass 2 ausgeführt, die bestehende Markentabelle wird ohne Aktualisierung wiederverwendet. Die Option „+“ braucht dabei nicht extra angegeben werden.
- B (Brief – Kurzform) alle Ausgaben (Listing und Fehleranzeigen) werden auf Bildschirmformat (39 Zeichen) verkürzt.
- L (Listing) Ausgabe des kompletten Assemblerlistings auf Bildschirm oder Drucker (mit Option P).
- O (Output) Der erzeugte Maschinencode wird direkt in den Hauptspeicher des KC geschrieben. Ein Überschreiben der von Edas benötigten Speicherbereiche hat der Programmierer durch geeignete Wahl der ORG-Adresse oder der Parameter beim Start von Edas zu verhindern.
- P (Printer) Ausgabeumschaltung auf den Drucker. Dazu muß ein Druckertreiber auf der Schnittstelle UOUT1 eine Zeichenausgabe eingetragen haben, CAOS trägt dort den Druckertreiber für einen K6313/14 und dazu kompatible Geräte ein.
- S (Save) Nach Durchlauf von Pass 1 und 2 wird ein weiterer Pass ausgeführt, wo der generierte Maschinencode auf Kassette/Diskette ausgegeben wird. In diesem Fall darf der Quelltext nur eine ORG-Anweisung am Anfang enthalten. Der Pass 3 (Ausgabe des MC) wird jedoch nur erreicht, wenn der Pass 2 fehlerfrei gelaufen ist.

In den wenigsten Fällen hat der Assembler nichts zu meckern, nach dem kompletten Übersetzungsvorgang (der Umfang hängt von den angegebenen Optionen ab) wird die Gesamtfehlerzahl, die Endadresse+1 des soeben übersetzten Programmes und eventuell eine vorhandene Startadresse ausgegeben. Der Assemblerlauf kann an jeder Stelle mit >BRK< abgebrochen werden.

Bei der Option 'S' wird der Dateityp KCC erzeugt, dieser kann bei Bedarf überschrieben werden. Eine Backup-Abfrage ist bei der Ausgabe des Maschinencodes auf Diskette nicht vorgesehen, da der Quelltext und der MC in den meisten Fällen den gleichen Namen haben werden. Eine neue BAK-Datei würde hier die BAK-Datei des Quelltextes überschreiben! Ist auf der Diskette bereits eine gleichnamige Datei vorhanden, wird diese ohne gesetzten Schreibschutz in in jedem Fall überschrieben!

Die nachfolgenden drei Kommandos beziehen sich unmittelbar auf den Assembler bzw. auf seine Arbeit und werden deshalb an dieser Stelle mit aufgeführt.

*HELP Nachdem Pass 1 des Assemblers abgelaufen ist (Erzeugung einer Tabelle aller Marken des zu übersetzenden Programmes), folgt der zweite Pass, wo nun das eigentliche Maschinenprogramm erzeugt wird. In diesem Pass gibt der Assembler bereits die gefundenen Fehler aus, was man auf dem Bildschirm verfolgen kann. In der ersten Spalte steht die entsprechende Fehlernummer, gefolgt von der fehlerhaften Quelltextzeile. Mit Hilfe des Kommandos *HELP werden diese Fehlernummern mit einer kurzen Erklärung auf dem Bildschirm angezeigt.

*LABEL / *LBLIST (n)

Zu den Marken kommen wir im folgenden Abschnitt, hier nur die Erklärung der beiden diesbezüglichen EDAS-Kommandos. Bevor man diese beiden Kommandos benutzen kann, muß der Assembler natürlich schon mal gelaufen sein, anders gesagt, es muß sich auch eine Markentabelle im Hauptspeicher befinden.

Nach Aufruf von *LABEL ist der Name der gewünschten Marke einzugeben und zu bestätigen, daraufhin erfolgt die Ausgabe des Wertes dieser Marke.

Mit Hilfe von *LBLIST erfolgt die Ausgabe der kompletten Markentabelle in n (sonst 3) Druckspalten auf den Bildschirm, hat man vorher über >SH+CLR< auch das Druckerprotokoll aktiviert, kann man sie auch ausdrucken lassen, danach mit >SH+CLR< den Drucker wieder abschalten!

4.5 Assembleranweisungen (Statements)

Bevor wir uns nun endlich ins Getümmel der Bits und Bytes stürzen können, müssen wir leider noch einige Dinge besprechen, welche die in der Assemblersprache benutzbaren Befehle bzw. Anweisungen betreffen und wie diese Elemente letztendlich in einer für das Assemblerprogramm lesbaren und damit übersetzbaren Form mit Hilfe des Editors geschrieben werden müssen.

Wie in Abschnitt 3.2.2 MNEMONIK bereits zu lesen war, bestehen alle Assemblersprachbefehle aus symbolischen Abkürzungen, welche das Assemblerprogramm „versteht“ und dafür beim Übersetzen den binären Maschinenbefehl für den Z80 erzeugt, es schreibt z.B. bei der Anweisung „LD D,6“ im Quelltext die beiden Bytes „16 06“ in die übersetzte Datei, welche in den Hauptspeicher oder auf Diskette ausgegeben wird. Im Abschnitt 3.3 Befehlssatz – Tabellarische Übersichten kann man alle beim Z80 möglichen binären Maschinenbefehle innerhalb (Schnittpunkte von Zeile und Spalte) der einzelnen Tabellen sehen. Ist im Schnittpunkt von Zeile und Spalte kein Binärcode (in hexadezimaler Schreibweise!) vorhanden, dann gibt es diesen Befehl auch nicht beim Z80!

Dazu ein kleines Beispiel aus Tabelle 1, 8-Bit-Ladebefehle: Der durchaus denkbare Befehl „LD B,(DE)“, welcher sich im Schnittfeld von Zeile 2 und Spalte 12 befinden müßte, ist nicht vorhanden! Verwendet man ihn innerhalb des Quelltextes trotzdem, ist dieses Beispiel besonders gemein, da der EDAS-Assembler diese Anweisung nicht als Fehler erkennt, er interpretiert die Anweisung als „LD A,(DE)“ und erzeugt den zugehörigen binären Befehlscode „1A“. Was dies für den Ablauf des eigentlich fehlerfrei übersetzten Programmes bedeutet, brauche ich sicher nicht zu erläutern.

Man darf also nur erlaubte Befehle benutzen, wovon es beim Z80 über 600 gibt, leider ist der Befehlssatz unsymmetrisch aufgebaut, was das Auswendiglernen etwas erschwert. Jörg wird aber neben den bereits vorhandenen Tabellen auch noch eine komplette Befehlsliste zur Verfügung stellen, wo alle zulässigen Befehle in einer etwas besser lesbaren Form enthalten sein werden (siehe auch Punkt 4.9 Literatur).

Die zulässigen Z80-Befehle resultieren aus der Hardware des Prozessors und sind innerhalb dieser fest verdrahtet. Die zweite Gruppe von Assemblersprachbefehlen sind die sogenannten PSEUDOBEFEHLE. Diese Befehle haben keine binäre Entsprechung im Z80, sondern wurden vom Entwickler des Assemblerprogrammes in dieses integriert, damit bestimmte Abläufe beim Übersetzungsvorgang realisiert werden können. Sie sind damit für den Prozessor auch nicht direkt ausführbar und werden nicht ins entstehende Maschinenprogramm geschrieben, da es ja keinen zugehörigen Maschinenbefehl im Befehlssatz des Z80 gibt. Sie können aber sehr wohl dazu führen, daß das Assemblerprogramm auf Grund eines Pseudobefehls beispielsweise Datenbereiche innerhalb des übersetzten Programmes schon beim Übersetzen mit entsprechenden Werten belegt und diese Daten werden dann natürlich auch in die übersetzte Datei geschrieben. Da die Pseudobefehle softwaremäßig durch den Entwickler des Assemblerprogrammes realisiert werden, gibt es bei unterschiedlichen Assemblersprachversionen auch unterschiedliche Pseudobefehle, sie dienen beispielsweise unter EDAS zur

- Definition von Daten und Symbolen
- Verwaltung des Speicherplatzes
- Steuerung des Übersetzungsvorganges des Assemblers.

Nachfolgend die unkommentierte Liste der zulässigen Pseudobefehle von EDAS 1.6. Für 'Ausdruck' kann eine Marke, eine Zahl oder eine Verknüpfung stehen:

ORG Ausdruck
: Setzt den Adresszähler auf den Wert des 'Ausdrucks', meistens wird damit die erste Speicheradresse (Beginn) des übersetzten Maschinenprogrammes definiert, sollte nur einmal im gesamten Quelltext verwendet werden und als erste Anweisung im Quelltext stehen;

Marke EQU Ausdruck
: Weist der 'Marke' den Wert des 'Ausdrucks' zu, damit kann im Quelltext anstatt mit realen Adressen bzw. Konstanten mit symbolischen Bezeichnungen gearbeitet werden (EQU kommt von „EQUate“ und bedeutet so viel wie „Zuweisen“);

DEFM 'Text'
: Definiere Text, erzeugt die ASCII-Werte der durch 'Text' definierten Zeichenkette von links nach rechts und gibt sie in das übersetzte Programm (MC) aus; die Zeichenkette muß in Hochkommas eingeschlossen sein;

DB x,'Text', ...
: Definiere Byte und/oder Text, als Operanden können Bytes und Zeichen-

ketten gemischt und mit Komma voneinander getrennt angegeben werden; durch die Verarbeitung von Zeichenketten in DB kann dieser Befehl auch anstelle von DEFM Anwendung finden;

DEFB x
: Definiere 1 Byte, das Byte kann auch als „1-Byte-Zeichenkette“ angegeben (z.B.: DEFB 'X') und mit Rechenzeichen verbunden werden (z.B.: DEFB 'y'+80H); das ist bei DB nicht möglich und führt zur Fehlermeldung „8“;

DS x,y
: Definiere Bytes, 'x' gibt an, wieviele gleiche Bytes erzeugt werden sollen, wird 'y' weggelassen, so werden Nullbytes, ansonsten Bytes mit dem Wert 'y' erzeugt;

DEFS x,y
: wie DS;

DW xx,...,xx
: Definiere Wort(e), erzeugt 16-Bit-Worte (zuerst Low-Teil, dann High-Teil); durch Komma getrennt können mehrere Worte in einer Zeile erzeugt werden;

DEFW xx
: wie DW;

END
: Bewirkt die Beendigung der Übersetzung ab dieser Stelle; nachfolgender Text wird nicht mehr beachtet; der „END“-Befehl wurde aus Kompatibilitätsgründen zu anderen Assemblern eingeführt; er ist z.B. beim Testen von Programmen einsetzbar, wenn noch nicht bis zum Ende übersetzt werden soll.

Die Gesamtheit aller ausführbaren (Z80-Befehle) und nicht ausführbaren Anweisungen (Pseudobefehle) ergeben dann unser Quellprogramm, das der Assembler in ein Maschinenprogramm übersetzt, das unser Z80 wiederum direkt ausführen kann.

Das Quellprogramm für den Assembler ist immer zeilenweise aufgebaut, jede Zeile nimmt genau EINE Anweisung auf, in der Literatur wird eine solche Anweisung auch als STATEMENT bezeichnet.

4.6 Syntax der Assemblersprache

Wie wir eben kennengelernt haben, steht jede Assembleranweisung innerhalb des Quelltextes auf einer eigenen Zeile. Der Editor von EDAS arbeitet mit dem VIDEO-RAM-Bereich des IRM im KC zusammen, so daß es folgende Einschränkung gibt, eine Zeile des Quelltextes ist maximal so lang wie eine Bildschirmzeile, reicht dieser Platz nicht aus, z.B. bei der Definition von Datenbytes oder langen Zeichenketten, sind die Daten auf zwei aufeinanderfolgende Quelltextzeilen zu verteilen, wobei die notwendige Assembleranweisung in jeder Zeile stehen muß.

Eine Zeile ist damit auch der kleinste Baustein eines Quelltextes. Damit das Assemblerprogramm beim Übersetzen nicht durcheinander kommt, hat sie einen genau festgelegten Aufbau. Eine Zeile besteht aus bis zu 4 Feldern, dem Markenfeld, dem Operationscodefeld (dort stehen die mnemonischen Bezeichnungen), dem Operandenfeld und dem Kommentarfeld.

Beispiel:

Markenfeld	Operationscodefeld	Operandenfeld	Kommentarfeld
BEGIN:	LD	A,6	;Akkumulator laden

Je nach Art eines Befehls können oder müssen einzelne dieser Felder wegfallen. Die einzelnen Felder müssen durch eine beliebige Anzahl (mindestens 1!) von Leerzeichen oder Tabulatoren voneinander getrennt sein.

Im Quelltext wird außer innerhalb von Zeichenketten (zwischen den Hochkommas) nicht zwischen Groß- und Kleinschreibung unterschieden, so daß man beides benutzen kann.

4.6.1 Marken

Marken sind symbolische Bezugspunkte innerhalb des Programmes, welche durch den Programmierer festgelegt und bezeichnet werden müssen. Dadurch ist es möglich anstatt reale Speicheradressen des KC-Hauptspeichers in den Quelltext zu schreiben, die Namen der Marken zu verwenden, wodurch sich weitestgehend adressunabhängige Programme schreiben lassen.

Marken werden verwendet, um in einer anderen Anweisung auf den momentanen Befehlszählerstand (PC = Program-Counter, siehe Punkt 3.1.1.1), auf eine andere Marke oder auf eine Konstante Bezug nehmen zu können. Eine Marke muß in der

ersten Spalte einer Programmzeile beginnen und sollte 6 Zeichen nicht überschreiten. Das erste Zeichen muß ein Buchstabe sein, als weitere sind Buchstaben, Ziffern, der Punkt und das Unterstreichungszeichen zulässig. Marken können bei EDAS mit einem Doppelpunkt abgeschlossen werden, müssen aber nicht.

Die Bezeichnung für eine Marke kann der Programmierer unter Beachtung der eben genannten Einschränkungen vollkommen frei festlegen, verboten für Marken sind die folgenden Zeichenketten, da sie innerhalb des Assemblerprogrammes für die Bezeichnung der CPU-Register (siehe Punkt 3.1.1) bzw. Flagbedingungen (siehe Punkt 3.2.3) reserviert sind:

Registernamen:	A B C D E H L I R AF BC DE HL IX HX LX IY HY LY SP
Flagbedingungen:	C NC Z NZ M P PE PO

EDAS 1.6 verwendet eine Marke mit dem Namen „START“ als Startadresse für selbststartende Programme, diese besondere Marke ist also unbedingt an der richtigen Stelle im Quelltext zu plazieren oder wegzulassen (dann ist das übersetzte Programm nicht selbststartend). Die durch den Assembler beim Übersetzen ermittelte Adresse wird als dritter Parameter bei der Assembler-Option ’S’ mit ausgegeben.

Bewährt hat sich bei mir eine Mindestlänge von 3 Zeichen für Markenbezeichnungen, damit geht man der versehentlichen Benutzung der eben genannten reservierten Zeichenketten aus dem Wege, was mich schon mal 2 Tage Fehlersuche im „eigentlich“ fehlerfreien Programm gekostet hat!

4.6.2 Operationscodes

Im Operationscodefeld steht eine der ZILOG-Maschinenbefehlsmnemoniken (siehe Punkt 3.3) oder eine Assembler-Pseudoanweisung (siehe Punkt 4.5), sie teilen dem Assembler mit WAS zu tun ist (Z80-Befehl erzeugen oder Pseudooperation einleiten).

Das Operationscodefeld beginnt frühestens in der 2. Spalte der Programmzeile, d.h., wenn das Markenfeld leer ist, muß vor dem Operationscode mindestens ein Trennzeichen (Leerzeichen oder Tabulator) stehen.

4.6.3 Operanden

Je nach Operationscode muß das Operandenfeld entweder leer sein (wenn keine Daten zu verarbeiten oder Bedingungen zu beachten sind, z.B. bei Befehlen wie RET, NOP, SCF, EXX, DI, END) oder es enthält einen oder zwei (durch ein Komma getrennte) Operanden, die

- eine Adresse (Speicher, Register oder Ein-/Ausgabe-Kanal),
- eine Konstante oder
- eine Flagbedingung

repräsentieren. Die Operanden ergänzen die Operationscodes durch eine Information darüber, WOMIT, also mit welchen Parametern die Operation durchzuführen ist.

Folgende Schlüsselwörter sind wieder für die Operandenfelder reserviert und dürfen vom Programmierer nicht anders verwendet werden:

8-Bit CPU-Register:	A B C D E F H L I R HX LX HY LY
16-Bit CPU-Register:	AF AF’ BC DE HL IX IY SP
Flagbedingungen:	C NC Z NZ M P PE PO

Als Adressen oder Konstanten können arithmetische Ausdrücke stehen, die durch nachfolgend aufgeführte Rechenzeichen verbunden sein können. Die Reihenfolge der Abarbeitung ist dabei stets von links nach rechts ohne Berücksichtigung von mathematischen Regeln! Die Verwendung von Klammern ist nicht gestattet.

+	Addition
−	Subtraktion
*	Multiplikation
/	ganzzahlige Division ohne Rest
%	Rest bei ganzzahliger Division (Modulo-Division)

In den Ausdrücken können Dezimalzahlen, Hexadezimalzahlen, Markenbezeichnungen, Zeichenkettenkonstanten und die Zeichen „$“ oder „#“ verwendet werden. Innerhalb eines Ausdruckes sind keine Leerzeichen gestattet, alle Bestandteile folgen also unmittelbar aufeinander!

Die Zeichen „$“ und „#“ haben eine spezielle Bedeutung, sie stehen stellvertretend für den momentanen Befehlszählerstand, der Befehl

```
LD HL,$
```

lädt in das Doppelregister HL die aktuelle Adresse des ersten Befehlsbytes dieses Befehls, welcher insgesamt eine Länge von 3 Bytes hat.

In Abschnitt 2.2 Zahlensysteme hat Frank verschiedene Zahlensysteme vorgestellt, EDAS „versteht“ direkt Dezimalzahlen, die ohne zusätzliche Kennzeichnungen im Quelltext verwendet werden können. Weiterhin können Hexadezimalzahlen benutzt werden, diesen Zahlen ist zur Kennzeichnung ein 'H' nachzustellen und sie müssen immer mit einer Ziffer (0...9) beginnen, damit sie nicht mit Markenbezeichnungen verwechselt werden können, d.h., ist das erste Zeichen der Hexadezimalzahl ein Buchstabe, muß eine führende '0' (der Wert der Zahl ändert sich ja dadurch nicht) vorangestellt werden, z.B. 200H oder 3FFFH, aber 0E000H!

Zeichenkettenkonstanten sind in Hochkommas (Apostroph) einzuschließen. Sie stehen für den ASCII-Wert der jeweiligen Zeichen, z.B. steht 'A' für 41H oder 'NO' für 4E4FH.

Tritt bei der Berechnung der Ausdrücke durch den Assembler ein arithmetischer Überlauf (siehe Abschnitt 2.4.4) auf, so wird dieser nicht berücksichtigt und geht verloren:

Beispiel: 0E000H+32768 ergibt 6000H.

4.6.4 Kommentare

Kommentare dienen zu Dokumentationszwecken und zur Erhöhung der Übersichtlichkeit der Quellprogramme. Sie sind kein funktioneller Bestandteil des übersetzten Programmes und werden beim Assembliervorgang (Übersetzen) übersprungen.

Ein Kommentar darf in jeder Spalte der Programmzeile beginnen und er endet mit dem Zeilenende. Das erste Zeichen und damit die Kennzeichnung für den Beginn eines Kommentars ist immer ein Semikolon „;“.

4.7 Das erste Programm „Hallo KC-Club !“

Für alle, die es geschafft haben bis hierher zu lesen und sich dabei auch noch einige Sachverhalte merken konnten, kommt jetzt die gute Nachricht – damit ist es erstmal überstanden! Sicher müßt Ihr im weiteren Verlauf unseres Kurses noch oft in den Abschnitten 2 bis 4 nachlesen, aber das macht überhaupt nichts. In Anlehnung an andere berühmte Sprüche „Der KC wurde auch nicht an einem Tag gebaut oder CAOS innerhalb einer Woche programmiert“!

Bis hierher seid Ihr aber zumindest theoretisch fit für den Umgang mit den verschiedenen Zahlensystemen, mit unserer CPU – dem Z80 – und mit einem Assembler in Form von EDAS. Also können wir endlich auch mal praktisch zur Sache gehen. Wir werden jetzt unser erstes Assemblerprogramm schreiben, übesetzen und ausführen. Es ist bewußt sehr einfach gehalten, damit Ihr überhaupt erst mal ein Gefühl für den Umgang mit der Materie bekommt. Damit auch etwas zu sehen ist, werden wir einen Text auf den Bildschirm ausgeben, welcher in der Überschrift schon verraten wurde. Dieses Beispiel findet Ihr nicht auf der Beilagendiskette, da man sich i.d.R. die Tätigkeiten am besten merkt, welche man selbst schon mal durchgeführt hat. Um mit dem Editor von EDAS klarzukommen, ist darüber hinaus schon etwas Übung notwendig. Dazu solltet Ihr auch die Tastenfunktionen des Editiermodus (vorher ausdrucken!) in schriftlicher Form bei der Hand haben.

Zunächst also ran an den KC, einschalten und nach dem Starten der CAOS-Betriebsart durch die D004-Besitzer EDAS 1.6 laut der Anleitung im Punkt 4.4 entsprechend dem eigenen System laden.

Los geht es immer mit %EDAS und >ET<, alle Abfragen des Kaltstarts quittieren wir ebenfalls mit >ET< und übernehmen dadurch die Vorschläge von EDAS, anschließend gelangen wir in das verkürzte Menü und der Cursor blinkt gelangweilt vor sich hin ... aber nicht lange, wir gehen auf *EDIT, bestätigen und befinden uns anschließend im Editiermodus, wo uns eine leere blaue Seite begrüßt.

Wie fängt man nun an? Ein ordentlicher Programmierer dokumentiert sein Programm mit entsprechenden Informationen über Zweck, Bearbeitungsstand und Autor, es hat sich bewährt, dies gleich immer als Kopf vor Beginn der eigentlichen Anweisungen zu setzen, dies sind also technisch gesehen Kommentare, welche wir ganz links mit einem Semikolon beginnen lassen:

```
;***************************************
; Ausgabe von "Hallo KC-Club !" in CAOS
;
;               01.08.1998
;
;           Autor: R. Kaestner
;***************************************
```

Nun müssen wir uns zunächst Gedanken machen, wo unser Programm später mal im Hauptspeicher des KC abgearbeitet werden soll, als Assemblerprogrammierer muß man sich nämlich um alles selbst kümmern u.a. auch um die Speicherverwaltung.

Die Abfrage 'Start of MC' haben wir mit 200H und 'ASM-Offset' mit 0000H beantwortet. Ohne weitere Maßnahmen benutzt der Assembler diese Angaben bei der Übersetzung und das übersetzte Programm (*.KCC-Datei) wird ab Adresse 200H von den CAOS-Kommandos %LOAD bzw. %FLOAD in den Hauptspeicher geladen. Das lassen wir auch erst mal so, der Pseudobefehl ORG 'Ausdruck' dient ansonsten zur Festlegung einer anderen (beliebigen) Anfangsadresse des Maschinenprogrammes (siehe 4.4.), ist er vorhanden, bestimmt er diese Adresse und die Abfragewerte des Kaltstarts sind dann ohne Bedeutung, die Anweisung ORG 4000H würde den Assembler beispielsweise dazu veranlassen unser Programm ab der Adresse 4000H zu übersetzen.

Nun zum eigentlichen Programm, wir wollen, daß es über einen Eintrag in der CAOS-Menüleiste beliebig oft aufgerufen werden kann. Dafür ist vor den eigentlichen Programmbefehlen, welche die Funktion des Programmes ausmachen, ein sog. Header oder Vorspann zu definieren, welcher CAOS mitteilt, daß es die Headerangaben entsprechend auswerten soll und in sein Menü einträgt. Dies ist eine CAOS-spezifische Angelegenheit und wird im Systemhandbuch unter Punkt 3.4. ausführlich beschrieben. Die folgenden Anweisungen kommen als nächste Zeilen in unser Quellprogramm, dadurch erscheint das Kommando %HALLO als letzter Befehl in der CAOS-Menüleiste und darüber wird unser Programm auch aufgerufen:

```
DB      7FH,7FH ;Standardprolog
DB      'HALLO' ;Menuewort
DB      1       ;Epilog
```

Anschließend kann es dann endlich mit dem eigentlichen Inhalt des Programmes weitergehen. Wie Jörg schon beispielhaft bemerkte, gibt es keinen direkten Assemblerbefehl analog 'PRINT' unter BASIC für den Z80. Aber schließlich gibt es CAOS, unser Betriebssystem, was erstens über sehr viele Funktionen und zweitens vor allem über klar definierte Schnittstellen verfügt, die sich in Form von sogenannten Systemrufen einfach benutzen lassen.

Diese beiden Tatsachen machen die Assemblerprogrammierung unter CAOS zu einer relativ einfachen Übung, wenn man erst mal weiß, wie es geht und solange die Betriebssystemfunktionen für die Lösung der zu programmierenden Aufgaben ausreichend sind, erst dann muß man selbst eigene Lösungen entwickeln, was in Assembler etwas komplizierter als beispielsweise in BASIC ist.

Zurück zu CAOS, im Systemhandbuch beschäftigt sich Punkt 3.5 mit den CAOS-Systemschnittstellen, wir benötigen heute im Punkt 3.5.3 die Aussagen zu Programmverteiler I und im Punkt 3.5.4 zu Unterprogramm OSTR mit der Nummer 23H, diese beiden Sachen werden wir in unserem Programm benutzen.

Der Programmverteiler I ist ein Unterprogramm von CAOS, das wie in BASIC mit dem Befehl CALL gerufen wird und auf Adresse 0F003H beginnt. Ihm ist unmittelbar nach dieser Anweisung die gewünschte CAOS-Unterprogrammnummer mitzuteilen, welche als nächste Assembleranweisung im Quelltext folgen muß, deshalb kommen nun folgende zwei Zeilen in den Quelltext:

```
CALL    0F003H  ;CAOS PV I rufen
DB      23H     ;UP-Nr. 23H
```

Jetzt weiß CAOS, welches Unterprogramm genutzt werden soll, nun müssen wir noch diesem Unterprogramm seine Parameter mitteilen. Das Unterprogramm OSTR gibt Zeichenketten auf dem Bildschirm aus, die im Quelltext unmittelbar auf den Aufrufbefehl folgen müssen. Damit es das Ende der Zeichenkette erkennen kann, ist als Abschluß der Zeichenkette ein Nullbyte in den Quelltext zu schreiben, also folgt jetzt:

```
DB      'Hallo KC-Club !',0
                ;Zeichenkette + 0
```

Damit ist unser Programm fast fertig, die gewünschte Aufgabe hat es bereits erfüllt und muß nun noch ordnungsgemäß beendet werden. Jedes aus der CAOS-Menüleiste aufgerufene Programm wird als Unterprogramm mit CALL gerufen, wie in BASIC muß deshalb unser Programm mit einem Rückkehrbefehl abgeschlossen werden:

```
RET             ;Programmende
```

Damit gibt unser Programm die Steuerung wieder an CAOS zurück und der Cursor müßte wieder blinken, was die Arbeitsbereitschaft der CAOS-Menüschleife signalisiert.

Wenn Ihr alle Schritte mitgemacht habt, müßte Euer Bildschirm also folgendermaßen aussehen:

```
;***************************************
; Ausgabe von "Hallo KC-Club !" in CAOS
;
```

```
;               01.08.1998
;
;           Autor: R. Kaestner
;***************************************
        DB      1       ;Epilog
        DB      7FH,7FH ;Standardprolog
        DB      'HALLO' ;Menuewort
        DB      1       ;Epilog
        CALL    0F003H  ;CAOS PV I rufen
        DB      23H     ;UP-Nr. 23H
        DB      'Hallo KC-Club !',0
                        ;Zeichenkette+0
        RET             ;Programmende
```

Eine leidige Eigenschaft von jeglichen Quelltexten ist, daß sie festgelegten Regeln 100%-ig entsprechen müssen, schaut Euch Euer Kunstwerk also noch mal genau an, bevor sich der Assembler über Schreibfehler beschwert. Danach beenden wir den Editiermodus mit >BRK< und kehren in den Kommandomodus zurück.

Die erste Bürgerpflicht am Computer lautet Datensicherung von neuen Eingaben bzw. Veränderungen, wir speichern unser Quellprogramm also erst mal ab, indem wir *SAVE aufrufen und als Namen beispielsweise HALLO1.ASM verwenden.

Nun machen wir uns eine besondere Eigenschaft des Assemblerprogrammes zunutze: Ruft man *ASM ohne Angabe von Optionen auf, wird nichts übersetzt aber das Quellprogramm wird auf syntaktische Fehler getestet. Also führen wir das mit dem soeben eingegebenen Programm durch und starten *ASM, die Abfrage nach den Optionen quittiert man mit >ET< ohne etwas einzugeben. Nachdem *ASM fertig ist, kommt die Ausgabe END PASS 1 und die Anzahl der gefundenen Fehler wird angezeigt. Nur wenn dort „ERRORS: 0000“ gemeldet wird, ist die Quelle syntaktisch fehlerfrei. Sind Fehler vorhanden, sind diese mit dem Editor zu beseitigen und die ganze Prozedur zu wiederholen, hat man dann ein syntaktisch korrektes Quellprogramm erstellt, speichert man es mit *SAVE auch noch einmal ab!!!

Nun können wir zum zweiten wichtigen Schritt kommen, dem Übersetzen des Quellprogramms in ein lauffähiges Maschinenprogramm. Dazu bemühen wir wiederum *ASM und >ET<, es erfolgt die Abfrage:

```
OPTIONS(+,1,2,B,L,O,P,S):S
```

Wie angegeben geben wir >S< und anschließend >ET< ein, damit übersetzt der Assembler unser Quellprogramm in Maschinencode und fordert uns anschließend zur Eingabe eines Namens für das fertige Programm auf, wir nehmen wieder HALLO1.KCC und bestätigen – jetzt wird das Programm auf die Diskette/Kassette ausgegeben.

Vor allem für die Fehlersuche bei größeren Programmen ist ein gedrucktes Assemblerprotokoll eine nützliche Angelegenheit. Neben unserem Quelltext steht dort auch der übersetzte Maschinencode drin, man kann also sehr schön den Zusammenhang zwischen den Assembleranweisungen und ihren binären Entsprechungen sehen. Leider sind nur die ersten 4 Bytes des erzeugten Maschinencodes pro Quelltextzeile zu sehen, der Rest wird abgeschnitten und nicht ausgedruckt. Anhand der Adressangabe in der ersten Zeile des Listings kann man aber sehen, daß Anweisungen, welche mehr Maschinencode erzeugen (in unserem Beispiel die DB Anweisungen mit 'HALLO' und 'Hallo KC-Club !') korrekt übersetzt werden. Ein solches Protokoll wird erzeugt, indem wir nochmals den Assembler aufrufen und folgende Optionen angeben:

```
OPTIONS(+,1,2,B,L,O,P,S):LP
```

'L' steht für Listing und 'P' für Ausgabe auf den Drucker (dieser muß natürlich angeschlossen sein und auf 'online' stehen, EDAS wartet sonst solange, bis der Drucker bereit ist!). Läßt man P weg, wird das Listing auf dem Bildschirm ausgegeben, das ist allerdings recht unübersichtlich, da eine Protokollzeile auf 2 Bildschirmzeilen verteilt wird, behelfsmäßig kann man das Protokoll für die Ausgabe auf den Bildschirm durch die Angabe von LB auf eine Bildschirmzeile verkürzen, dann fehlt allerdings auch immer der hintere Teil.

Damit hat EDAS seine Arbeit erst mal getan und wir verlassen es mit *QUIT.

Der Erfolg unserer programmtechnischen Künste soll natürlich auch noch betrachtet werden. Dazu ist das Maschinenprogramm mit dem CAOS-Kommando:

```
%FLOAD
Name:HALLO1
```

in den Hauptspeicher zu laden, danach gibt man %MENU ein und es erscheint ganz unten unser neues Menüwort %HALLO, worüber wir nun schließlich unser Programm starten können, die folgenden Ausschriften sollten dann erfolgen:

```
%HALLO
Hallo KC-Club !%_
```

Hinter unserem Text erscheint ein CAOS-Promptzeichen und direkt danach der Cursor und WENN er wirklich wieder zu sehen ist, habt Ihr soeben erfolgreich Euer erstes Programm in Assemblersprache geschrieben und dürft Euch ab sofort ASSEMBLER-PROGRAMMIERER nennen und das können nicht allzu viele Leute von sich sagen, die einen Computer bedienen...

Herzlichen Glückwunsch!

Zugegeben, unser Programm macht noch nicht allzu viel her, noch hat es irgendwelche besonderen Funktionen, aber als frischgebackene Assemblerprogrammierer ist es für Euch sicher kein Problem, entsprechende Modifikationen vorzunehmen. Folgende Anregungen möchte ich Euch noch mitgeben und lasse mich einfach mal überraschen bis zur nächsten Ausgabe der KC-News:

- Ausgabe des Textes an eine bestimmte Bildschirmposition
- Löschen des Bildschirms vor der Textausgabe
- Ändern der Text- und/oder Bildschirmfarben während/vor Ausgabe
- Erzeugen eines Textmusters durch die wiederholte Ausgabe des Textes auf verschiedenen Positionen oder in verschiedenen Farben usw.

Alle dafür notwendigen Informationen findet Ihr im Systemhandbuch des KC, welches ein Assemblerprogrammierer mindestens genauso gut wie den Befehlssatz des Z80 kennen sollte. Viel Spaß beim Experimentieren!

Die Beherrschung der soeben durchgeführten Handlungen ist Grundvoraussetzung für die erfolgreiche Assemblerprogrammierung mit EDAS unter CAOS und damit auch für eine aktive Teilnahme an unserem Kurs – nur dann kann man die Beispiele nachvollziehen und Übung macht bekanntlich den Meister!

Viele weitere Details wurden bereits mit genannt, sie sind zunächst aber nicht entscheidend, oft werden sie nur für besondere Einsatzfälle benötigt bzw. sie werden bei Notwendigkeit in den folgenden Abschnitten unseres Kurses an der entsprechenden Stelle erklärt. Alle User, auch die, welche bereits mit EDAS gearbeitet haben, denen bestimmte Sachverhalte nicht ganz klar sind, sind natürlich zur aktiven Mitarbeit aufgerufen, ich erkläre gern auch kompliziertere Vorgänge. Ich arbeite jetzt bereits 8 Jahre mit den verschiedenen EDAS Versionen und habe beispielsweise auch UNIPIC 2.0 mit Hilfe von EDAS 1.6 geschrieben, dessen Quelltexte aus 16 verschiedenen Dateien mit einem Gesamtumfang von ca. 800 kB bestehen – es ist machbar und funktioniert!

4.8 Hilfsmittel für die Assemblerprogrammierung

Frank deutete es schon an, mitunter ist es ganz nützlich, wenn man bestimmte, immer wieder notwendige Sachen hat oder weiß, wo man nachschlagen muß.

Insbesondere bei Nutzung der verschiedenen Zahlensysteme macht sich eine GEDRUCKTE Umrechnungstabelle der Zahlen von 0 bis 255 (alle Zustände, die ein Byte annehmen kann) schnell bezahlt.

Unter CAOS sollte ein Taschenrechner mit von der Partie sein, der dezimal, hexadezimal und idealerweise auch binär rechnen kann, unter CP/M nutzt man zweckmäßigerweise das hervorragende HPKC.COM, das sogar online per Tastendruck seine Dienste zur Verfügung stellt.

Für die schriftliche Fixierung von programmtechnischen Problemen oder anderen Skizzen benutzt man einen Bleistift mit Radiergummi – keinen Kugelschreiber! Grafische Skizzen gelingen am besten auf quadratisch kleinkarierten Papier (Programmenüs u.ä.) oder Millimeterpapier (Pixelgrafik), ich verwende mittlerweile ausschließlich UNIPIC 2.0 für solche Zwecke.

Alle zusammengehörigen Entwürfe sollte man nicht irgendwo ablegen, sondern in einem Register oder eigenen Schnellhefter zusammen aufbewahren, man spart viel Zeit beim Suchen oder Nachschlagen.

4.9 Literatur

Abschließend noch einige Literaturquellen, wo man sich weiterbilden kann oder Anregungen und Beispiele findet. Wie in jeder Programmiersprache sind die letztgenannten beiden Sachverhalte oft Auslöser für eigene Ideen oder der schon lange gesuchte Lösungsweg für bestimmte Probleme in eigenen Kreationen. Man muß das Fahrrad nicht ständig neu erfinden. Da der Z80 bzw. U880 zu den aussterbenden Spezies gehört, dürften die meisten Sachen nur noch auf Flohmärkten oder in einer Bibliothek zu finden sein.

Das *Systemhandbuch des KC 85/4* sollte jeder besitzen, da es zum Lieferumfang des KC-Systems gehörte. Dort findet man die unbedingt notwendigen Informationen zu CAOS und eine komplette Liste seiner universell nutzbaren Unterprogramme mit allen zugehörigen Parametern (beim KC 85/3 in den Übersichten!).

Die *Beschreibung zum M027 DEVELOPMENT* lag jedem M027 bei und enthält neben einer Beschreibung von EDAS Version 1.4 (viele Passagen wurden in den vorangegangenen Abschnitten daraus entnommen und durch die Weiterentwicklungen der Version 1.6 ergänzt) eine komplette Befehlsliste des Z80 für die Nutzung unter EDAS und die Beschreibungen der weiteren Programme des M027, wie Disassembler und Testmonitor. Wer es besitzt, sollte die dort enthaltene Befehlsliste für die Programmierung (Suche nach zulässigen Z80-Befehlen) verwenden, da sie in der Handhabung einfacher sprich eindeutiger als die Übersichtstabellen ist.

Im Buch *BASIC für Mikrorechner* von Dieter Werner, erschienen in der 2. Auflage 1987 in Berlin: Verlag Technik (ISBN: 3-341-00437-8) findet sich ab Seite 223 (Anhang) o.g. Umrechnungstabelle nebst zugrundeliegendem BASIC-Programm. Vielleicht schreibt jemand mal dieses Programm für den KC um, dann kann sich jeder die Tabelle selbst ausdrucken.

Als ich damals anfing, habe ich oft die Quelltexte von WordPro '86 studiert und viel daraus gelernt. Sie sind im Buch *Tips und Tricks für kleine Computer* von K. Schlenzig und S. Schlenzig zu finden, erschienen in der 1. Auflage 1988 in Berlin: Militärverlag der DDR (ISBN: 3-327-00555-9).

Ebenfalls optimal für die Einarbeitung in die Assemblerproblematik ist das Buch *8-Bit-Mikrorechentechnik* von H. Bäurich und H. Barthold, erschienen in der 1. Auflage 1988 in Berlin: Militärverlag der DDR (ISBN: 3-327-00668-7). Wer sich auch mal mit der Programmierung der Z80-Peripherieschaltkreise, wie PIO oder CTC beschäftigen will, kommt um dieses oder die nächsten beiden Bücher kaum herum, dort finden sich die notwendigen Informationen.

Die letztgenannte Quelle basiert auf den folgenden beiden Heften und behandelt zusammengefaßt in Form eines Buches die gleiche Problematik. Für mich wertvoller, weil noch ausführlicher, waren die beiden Doppelausgaben 222/223 und 224/225 der *electronica*-Reihe von den gleichen Autoren wie eben:

> *Mikroprozessoren – Mikroelektronische Schaltkreise und ihre Anwendung, Teil 1: Grundlagen der Mikrorechentechnik*, 3. überarbeitete Auflage, Berlin: Militärverlag der DDR, 1985 (electronica 222/223) und
>
> *Mikroprozessoren – Mikroelektronische Schaltkreise und ihre Anwendung, Teil 2: Periphere Schaltkreise/Programmbeispiele*, 3. überarbeitete Auflage, Berlin: Militärverlag der DDR, 1985 (electronica 224/225).

Diese beiden Hefte waren und sind beim Programmieren meine ständigen Begleiter.

Sie wurden in den letzten 2 Jahren nur ab und zu von einer anderen Quelle abgelöst, dem kommentierten Quelltext von CAOS 4.2 bzw. später CAOS 4.3, beide sind von Mario Leubner bei Interesse erhältlich und führen natürlich ideal in das Thema ein. Nicht alle Heimcomputerbesitzer haben diesen unschätzbaren Vorteil, daß die Quellen für das wichtigste Programm des Computers – das Betriebssystem – offenliegen und jederzeit studiert werden können. Dort kann man z.B. die programmtechnische Realisierung der Systemprogramme oder die Abfrage der peripheren Bausteine des KC-Systems anschauen und jede Menge Tips und Tricks entnehmen. Die Programmierung von Systemkomponenten gehört schon zu den gehobenen Aufgaben eines Assemblerprogrammierers, dementsprechend schwierig ist eine fehlerfreie Umsetzung und die sollte das Betriebssystem schon leisten.

Für spezielle Gebiete der Assemblerprogrammierung sind auch noch weitere Bücher erschienen, an dieser Stelle soll das aber erst mal für heute genügen.

Teil 5 – Einfache Datenstrukturen

von Jörg Linder

5.1 Bytes

5.1.1 Der LD-Befehl

Die kleinste Speichereinheit, die der Z80 mit einem Befehl zwischen internem und externem Speicher transportieren kann, ist ein Byte. Daher wird der Prozessor auch als byteorientiert bezeichnet. Mittels Assemblerbefehl LD (load; laden) kann ein Byte in eines der 8-Bit-Register des Z80 gebracht werden. Soll zum Beispiel der Akkumulator mit dem Wert 12H geladen werden, so lautet der entsprechende Befehl:

```
LD      A,12H   ; den Wert 12H ins Register A bringen
```

Der LD-Befehl hat stets zwei Argumente. Das erste Argument bezeichnet das Ziel, also wohin der Datenwert geschrieben werden soll (in unserem Beispiel der Akkumulator). Das zweite Argument gibt den Datenwert an.

Ein weiteres Beispiel: Der Datenwert C3H soll in das Register D geschrieben werden. Der Befehl dazu lautet:

```
LD      D,0C3H  ; den Wert C3H ins Register D bringen
```

Hex-Zahlen müssen für den Assembler stets mit einer Ziffer beginnen, damit sie von Namen unterscheidbar sind. Beginnt ein hexadezimaler Wert mit einem Buchstaben, muß eine Null vorangestellt werden.

Numerische Argumente von Befehlen können auch in dezimaler, binärer oder oktaler Notation angegeben werden. Statt des obigen Befehls könnte auch einer der drei folgenden Befehle geschrieben werden, der Assembler würde stets dasselbe Resultat erzeugen:

```
LD      D,195           ; den Wert C3H als Dezimalzahl ins
                        ; Register D bringen
LD      D,11000011B     ; den Wert C3H als Binaerzahl ins
                        ; Register D bringen
LD      D,303Q          ; den Wert C3H als Oktalzahl ins
                        ; Register D bringen
```

Natürlich sollte man sich das Leben nicht unnötig schwer machen und daher die Notation wählen, die sich am besten für den jeweiligen Anwendungsfall eignet.

Merke: Durch einen LD-Befehl wird kein Flag verändert!

5.1.2 Einfache Byte-Arithmetik

Ein Datenwert vom Typ „Byte“ kann als Darstellung einer vorzeichenlosen ganzen Zahl im Bereich 0 bis einschließlich 255 interpretiert werden (siehe Teil 2). Der Z80 verfügt deswegen über arithmetische Befehle, mit deren Hilfe man mit Werten vom Typ „Byte“ rechnen kann. Die Berechnungen erfolgen stets modulo 256, damit das Ergebnis der Operation wieder vom Typ „Byte“ ist.

(Eine ganze Zahl n modulo m rechnen bedeutet, daß zu der zu reduzierenden Zahl n solange m beziehungsweise $-m$ addiert wird, bis das Ergebnis im Bereich 0 bis $m-1$ liegt; es ist also der ganzzahlige Rest der Division n durch m; Beispiele: 23 modulo $4 = 3$, 12 modulo $3 = 0$, 17 modulo $5 = 2$)

Als erstes unterziehen wir die Addition einer näheren Betrachtung. Mit Hilfe des Befehls ADD (add; addieren) kann eine Konstante vom Typ „Byte“ zum Inhalt des Akkumulators addiert werden. Das Ergebnis wird im Akkumulator abgelegt. Soll der Inhalt des Akkumulators um 63H erhöht werden, so lautet der Befehl:

```
ADD     A,63H   ; Inhalt des Registers A um den
                ; Wert 63H erhoehen
```

Ist das Ergebnis größer als 255, dann wird es modulo 256 reduziert. Ob dieser Fall eintrat, ist nach der Ausführung des Befehls am Zustand des C-Flags (Übertrag) zu erkennen. Der ADD-Befehl setzt die Flags folgendermaßen (ein Flag ist gesetzt, wenn es den Inhalt 1 besitzt; zurückgesetzt oder gelöscht, wenn es den Inhalt 0 hat):

S	gesetzt, falls Bit 7 des Ergebnisses gesetzt
Z	gesetzt, falls Ergebnis gleich Null
H	gesetzt, falls Übertrag von Bit 3
P/V	gesetzt, falls Überlauf auftrat
N	zurückgesetzt
C	gesetzt, falls Übertrag von Bit 7

Wurde das Ergebnis also modulo 256 reduziert (das nicht reduzierte Ergebnis ist somit nicht als „Byte" darstellbar), so ist das C-Flag gesetzt, ansonsten ist es zurückgesetzt.

Merke: Der erste Operand des (8-Bit) ADD-Befehls ist stets der Akkumulator!

Ein Datenwert vom Typ „Byte" kann auch als Darstellung einer ganzen Zahl im Bereich von −128 bis +127 interpretiert werden. Die Darstellung erfolgt hierbei im Zweierkomplement (siehe Teil 2). Ein Byte stellt dabei genau dann eine negative Zahl dar, wenn das Bit 7 gesetzt ist.

Zahlen in derartiger Darstellung werden ebenfalls mit dem ADD-Befehl addiert. Kann das Ergebnis der Operation (vor einer Reduktion modulo 256) nicht als Zweierkomplement einer ganzen Zahl im Bereich −128 bis +127 gedeutet werden, so liegt ein Überlauf vor. Bei Überlauf wird das P/V-Flag gesetzt.

Anhand des folgenden Beispiels soll der Unterschied zwischen Übertrag und Überlauf verdeutlicht werden:

FDH + 20H = ?

Darstellung binär	vorzeichenlos	Zweierkomplement
1111 1101	253	−3
+ 0010 0000	32	+32
1 0001 1101	285	+29

Wie zu erkennen ist, reicht der darstellbare Wertebereich für vorzeichenlose ganze Zahlen nicht mehr aus (Ergebnis = 285), so daß der Übertrag durch das C-Flag angezeigt und das Ergebnis modulo 256 gerechnet wird (285 modulo 256 = 29). Für die Zweierkomplement-Darstellung war keine Reduktion modulo 256 des Ergebnisses notwendig; ein Überlauf ist nicht aufgetreten und das P/V-Flag daher auch nicht gesetzt.

Der Inhalt des Akkumulators (als vorzeichenlose ganze Zahl gedeutet) soll nun um den Wert 35H vermindert werden. Dies geschieht mit Hilfe des Befehls SUB (substract; subtrahieren):

```
SUB     35H      ; Inhalt des Registers A um den
                 ; Wert 35H vermindern
```

Merke: Der erste Operand des SUB-Befehls ist stets der Akkumulator, wird aber nicht explizit angegeben!

Wie schon beim ADD-Befehl wird auch beim SUB-Befehl das Ergebnis nötigenfalls modulo 256 reduziert. Ist der Wert im Akkumulator kleiner als der Wert des Arguments (beide als nicht-negative ganze Zahlen betrachtet), so muß zur Subtraktion „geborgt" werden. Es wird dann das C-Flag gesetzt.

Man kann die Arithmetik auch auf ganze Zahlen in Zweierkomplement-Darstellung anwenden. Ist das Ergebnis der Operation nicht als Zweierkomplement einer ganzen Zahl im Bereich −128 bis +127 interpretierbar, so tritt ein Überlauf auf. In diesem Fall wird das P/V-Flag gesetzt.

Der SUB-Befehl setzt die Flags folgendermaßen:

S	gesetzt, falls Bit 7 des Ergebnisses gesetzt
Z	gesetzt, falls Ergebnis gleich Null
H	gesetzt, falls Borgen von Bit 4 nötig war
P/V	gesetzt, falls Überlauf auftrat
N	gesetzt
C	gesetzt, falls Borgen nötig war

Mit dem folgenden Beispiel soll die Funktionsweise des SUB-Befehls illustriert werden:

Für das Verständnis der Arithmetik ist es ganz wichtig, sich klar zu machen, daß ein und derselbe ADD- beziehungsweise SUB-Befehl, angewandt auf vorzeichenlose ganze Zahlen oder auf vorzeichenbehaftete ganze Zahlen, jeweils das richtige Ergebnis in derselben Darstellung liefert. Der Z80 kann dabei gar nicht wissen, was die Darstellung bedeuten soll, er führt immer dasselbe Verfahren aus. Daß dies funktioniert, liegt an den Eigenschaften der Zweierkomplement-Darstellung, die bezüglich Addition und Subtraktion mit der Darstellung vorzeichenloser ganzer Zahlen verträglich ist. Als Programmierer weiß man natürlich (oder sollte es zumindest wissen), was die Darstellung bedeuten soll.

Mit Hilfe des Befehls NEG (negate; negieren) wird der Inhalt des Akkumulators (als Zweierkomplement einer ganzen Zahl im Bereich −128 bis +127 gedeutet) negiert. Das gleiche Ergebnis würde man erzielen, wenn man den Inhalt des Akkumulators von Null subtrahieren würde. Der NEG-Befehl hat keine Argumente:

```
        NEG             ; Inhalt des Registers A negieren
```

Die Flags werden wie folgt gesetzt:

S	gesetzt, falls Bit 7 des Ergebnisses gesetzt
Z	gesetzt, falls Ergebnis gleich Null
H	gesetzt, falls Borgen von Bit 4 nötig war
P/V	gesetzt, falls im Akku 80H vor der Operation
N	gesetzt
C	gesetzt, falls im Akku nicht 00H vor der Operation

Ist der Inhalt des Akkumulators vor der Operation 80H (das ist die Zweierkomplement-Darstellung von −128), so tritt ein Überlauf ein, weil das Ergebnis der Operation +128 ist und somit nicht in einem Byte als Zweierkomplement dargestellt werden kann. Es wird modulo 256 reduziert und als Ergebnis erhält man 80H.

Merke: Der NEG-Befehl bezieht sich immer auf den Akkumulator!

Zusammenfassend kann festgehalten werden: Ist das Resultat eines ADD-Befehls, SUB-Befehls oder NEG-Befehls negativ (eventuell nach einer Reduktion modulo 256), das Bit 7 hat also den Wert 1, so wird das S-Flag (Vorzeichen) gesetzt.

5.2 Zeichen

Unter einem Zeichen (engl. character) versteht man Buchstaben, Ziffern, das Leerzeichen (engl. space oder blank), Sonderzeichen wie Punkt, Komma und Klammern, sowie Steuerzeichen. Letztgenannte dienen zur Positionierung von Ausgabegeräten und zur Kommunikation zwischen Prozessor und Ein-/Ausgabe-Geräten.

5.2.1 Der ASCII-Code

Eine gebräuchliche Codierung für Zeichen ist der ASCII-Code (American Standard Code for Information Interchange). Der ASCII-Code ist ein 7-Bit-Code, das heißt zur Codierung eines Zeichens werden 7 Bits verwendet. Da der Z80 byteorientiert arbeitet, wird zur Darstellung eines Zeichens in ASCII normalerweise ein Byte verwendet. Bit 7 hat dann gewöhnlich entweder stets den Wert 0 oder (seltener) stets den Wert 1. Bei Datenübertragungen wird manchmal dieses Bit auch so gesetzt, daß die Gesamtzahl der Bits mit Wert 1 in einem Byte – die Parität – gerade (engl. parity even) oder ungerade (engl. parity odd) ist.

Die Codierung der einzelnen Zeichen kann der Tabelle im Anhang B.1.1 entnommen werden. (Ein stets zurückgesetztes Bit 7 wird dabei angenommen.) Zeichen mit den Codierungen 00H bis 1FH sowie 7FH sind Steuerzeichen. Der Prozessor unterscheidet nicht zwischen Steuerzeichen und sichtbaren Zeichen; dies tun nur die Ein-/Ausgabe-Geräte.

Je nach Ein-/Ausgabe-Gerät variiert die Bedeutung mancher Steuerzeichen. Die Tabelle im Anhang B.1.2 zeigt einige Zeichen, die auf fast allen Geräten eine feste Bedeutung haben.

Für die meisten Ausgabegeräte reichen die wenigen Steuerzeichen nicht aus. Man behilft sich damit, daß Steuerfunktionen durch eine Folge von Zeichen ausgelöst werden, die oftmals mit dem Zeichen ESC (escape) beginnt, sonst aber beliebige (auch sichtbare) Zeichen enthalten darf.

Um in verschiedenen wichtigen Sprachen einen angepaßten Zeichensatz verfügbar zu haben, gibt es Varianten des ASCII-Codes. Einige Beispiele sind in der Tabelle im Anhang B.1.3 enthalten.

Außer dem ASCII-Code werden gelegentlich auch andere Codes wie Baudot-Code (Fernschreib-Code) oder EBCDIC (Extended Binary Coded Decimal Interchange Code) verwendet. Erweiterungen des ASCII-Codes zum 8-Bit Code durch Hinzunahme von Graphikzeichen (IBM-Code) oder auch 2-Byte Code (UNICODE) sind ebenfalls gebräuchlich. Manche Geräte können keine Kleinbuchstaben oder keine Großbuchstaben verarbeiten. Diese Buchstaben sind dann im Code sinngemäß durch andere ersetzt.

5.2.2 Operationen mit Zeichen

ASCII-codierte Zeichen unterscheiden sich im Speicher nicht von numerischen Daten des Types „Byte“ und werden deshalb vom Prozessor auch wie solche behandelt. Der Assembler bietet die Möglichkeit, Konstanten vom Typ „Zeichen“ in der ASCII-Darstellung oder in einer numerischen Zahl-Darstellung zu notieren.

Die Befehle

```
        LD      A,3FH   ; Register A mit dem Wert 3FH laden
```

und

```
        LD      A,'?'   ; Register A mit dem ASCII-Code des
                        ; Fragezeichens laden
```

werden vom Assembler beide in den gleichen Objektcode umgewandelt, da der ASCII-Code des Fragezeichens 3F in hexadezimaler Darstellung entspricht.

Merke: Im Programmtext müssen Zeichenkonstanten in einfache Hochkommas eingeschlossen werden.

Da der Z80 mit ASCII-codierten Zeichen genauso verfährt wie mit numerischen Daten vom Typ „Byte", kann man auf sie arithmetische Operationen anwenden. Besonders praktisch ist dies bei den Dezimalziffern, deren ASCII-Codierungen eine lückenlos aufsteigende Folge bilden.

Ein Beispiel: Der Akkumulator enthält den numerischen Wert einer Dezimalziffer (beispielsweise 03H). Dieser soll in die entsprechende ASCII-Codierung der Dezimalziffer umgewandelt werden. Diese erhält man, indem 30H zum numerischen Wert der Ziffer addiert werden:

```
        ADD     A,30H   ; binaer codierte Dezimalziffer in
                        ; ASCII umwandeln
```

Analog die Umkehrung: Der Akkumulator enthält die ASCII-Codierung einer Dezimalziffer (beispielsweise 39H für die Ziffer „9"). Sie soll in ihren numerischen Wert umgewandelt werden. Diesen erhält man, wenn 30H von der Ziffer subtrahiert werden:

```
        SUB     30H     ; ASCII-codierte Dezimalziffer in
                        ; numerischen Wert umwandeln
```

Der Assembler erzeugt denselben Objektcode, wenn statt dessen – etwas besser kommentiert – geschrieben wird:

```
        SUB     '0'     ; ASCII-codierte Dezimalziffer in
                        ; numerischen Wert umwandeln. '0' geht
                        ; ueber in 0. Dezimalziffern sind
                        ; sowohl in Hex wie in ASCII
                        ; fortlaufend aufsteigend.
```

Eine Zeichenkonstante als Argument eines Befehls wird vom Assembler durch ihre ASCII-Codierung (vom Typ „Byte") ersetzt.

Ebenso leicht wie die Umwandlung von Ziffern ist die Umwandlung Großbuchstaben in Kleinbuchstaben. Sowohl Groß- als auch Kleinbuchstaben sind jeweils als lückenlose aufsteigende Folge codiert.

Beispiel: Im Akkumulator befindet sich die ASCII-Codierung eines Großbuchstabens (beispielsweise 41H für den Buchstaben „A"). Durch die Addition von 20H entsteht der entsprechende Kleinbuchstabe:

```
        ADD     A,20H   ; Grossbuchstaben in Kleinbuchstaben
                        ; umwandeln
```

Oder umgekehrt: Ein ASCII-codierter Kleinbuchstabe (beispielsweise 68H für den Buchstaben „h") steht im Akkumulator. Dieser soll in den entsprechenden Großbuchstaben umgewandelt werden. Durch Subtraktion von 20H wird dies erreicht:

```
        SUB     20H     ; Kleinbuchstaben in Grossbuchstaben
                        ; umwandeln
```

Bisher wurde der Abstand zwischen Groß- und entsprechendem Kleinbuchstaben immer als Konstante in hexadezimaler Darstellung („20H") angegeben. Man kann aber im Programm auch folgenden Befehl benutzen:

```
        SUB     'a'-'A' ; Kleinbuchstaben in Grossbuchstaben
                        ; umwandeln
```

Findet der Assembler für ein Befehlsargument nämlich einen Ausdruck statt einer Konstanten, so berechnet er den Wert des Ausdrucks und verwendet diesen als Argument. Zeichenkonstanten werden dabei durch ihre ASCII-Codierungen ersetzt. Bei

numerischen Operationen wird davon ausgegangen, daß die Operanden im Zweierkomplement dargestellt sind. Die Berechnung wird modulo $2^{\text{Argumentlänge}}$ durchgeführt, so daß das Ergebnis eine der Länge des Arguments angepaßte Größe (meist 8 Bits oder 16 Bits) besitzt.

Demzufolge unterscheidet sich der Objektcode der beiden zuletzt aufgeführten Befehle nicht.

5.3 Worte

Ein Wort (engl. word) ist eine Datenstruktur, bestehend aus zwei Bytes. Worten wird – wie schon von den Bytes her bekannt – oftmals eine Interpretation untergeschoben. Besonders häufig werden Worte als binär-codierte vorzeichenlose ganze Zahlen (Wertebereich von 0 bis 65535) oder als ganze Zahl in Zweierkomplement-Darstellung (Wertebereich von −32768 bis +32767) interpretiert.

Das Byte eines Wortes, welches den höherwertigen Anteil der dargestellten Zahl enthält, wird MSB (most significant byte) genannt, das Byte mit dem niederwertigen Anteil LSB (least significant byte). Im Speicher werden MSB und LSB normalerweise fortlaufend abgelegt, wobei das MSB üblicherweise die höhere Adresse erhält. Der Z80 führt Wort-Operationen stets in dieser Weise aus, weshalb im folgenden von dieser Anordnung ausgegangen wird.

(Anmerkung: Die Abkürzungen MSB und LSB werden auch für das höchstwertige Bit (MSB = most significant bit) bzw. das niederwertigste Bit (LSB = least significant bit) benutzt. Da es keinen verbindlichen Standard gibt, kann die Bedeutung nur aus dem Zusammenhang erkannt werden. In diesem Abschnitt werden die Abkürzungen ausschließlich als Bezeichnungen für die Bytes benutzt.)

Worte können nicht nur im Speicher sondern auch in einem Doppelregister, einem Indexregister oder im Stack Pointer untergebracht werden. (Auch im Programmladezähler (PC) kann ein Wort untergebracht werden; da der Inhalt des PC die Adresse des nächsten auszuführenden Befehls enthält, stellt das Laden des PC mit einem neuen Inhalt einen Sprung innerhalb des Programms dar.) Nachfolgend wird es jedoch nur um Worte gehen, die im Speicher oder in einem der Registerpaare BC, DE, HL untergebracht sind.

5.3.1 Ladebefehle für Worte

Wenn eines der genannten Registerpaare mit einem bestimmten 16-Bit-Wert geladen werden soll, kann man dies im Prinzip mit den bereits bekannten LD-Befehlen für 8-Bit-Register tun. Zum Beispiel soll der Wert 21F7H ins Registerpaar DE gebracht werden:

```
LD      D,21H
LD      E,0F7H      ; fuehrende 0 nicht vergessen!
```

Schneller (und sogar mit kürzerem Objektcode) erreicht man dies durch einen 16-Bit-Ladebefehl:

```
LD      DE,21F7H    ; Konstante 21F7H vom Typ Wort in
                    ; Registerpaar DE laden
```

Leider gibt es keine Befehle, um ein Registerpaar direkt in ein anderes Registerpaar umzuspeichern. Wenn zum Beispiel der Inhalt des Registerpaars DE ins Registerpaar HL übertragen werden soll, muß man schreiben:

```
LD      H,D         ; Inhalt des Registerpaars DE
LD      L,E         ; ins Registerpaar HL bringen
```

Merke: Bei Worten in Registerpaaren enthält das zuerst genannte Register das MSB, das anschließend genannte Register das LSB!

Das Laden eines Registerpaars mit einem Wort, das im Speicher steht, kann dagegen wieder durch einen einzigen Befehl realisiert werden. Dazu muß die Speicheradresse des LSB angegeben werden. Beispiel: Im Speicher steht unter der Adresse 31A7H der 8-Bit-Wert E9H, unter der Adresse 31A8H der 8-Bit-Wert 23H. Der Befehl:

```
LD      BC,(31A7H)  ; Lade Inhalt der Adresse 31A7H ins
                    ; Register C und Inhalt der Adresse
                    ; 31A8H ins Register B
```

bewirkt dann, daß in das Register B der Wert 23H und in das Register C der Wert E9H geladen wird. Als Registerpaar gesehen enthält BC damit den Wert 23E9H.

Interessanterweise belegt der Objektcode des Befehls

```
          LD      HL,(adresse)
```

nur 3 Bytes, während die Befehle

```
          LD      BC,(adresse)
          LD      DE,(adresse)
```

jeweils 4 Bytes Speicherplatz benötigen. Dies liegt daran, daß letztere beim direkten Vorgänger des Z80 – dem Prozessor i8080 von Intel – noch nicht vorhanden waren, im Gegensatz zum erstgenannten Befehl. Kürzere Objektcodes und schnellere Ausführungszeiten beim Registerpaar HL haben auch zu der Bezeichnung „16-Bit-Akkumulator“ geführt.

Soll jedoch der Inhalt eines Registerpaars in den Speicher kopiert werden, so benutzt man dazu einen Befehl der Form

```
          LD      (adresse),registerpaar
```

Hierbei wird das LSB in „adresse“ geschrieben, das MSB hingegen in „adresse+1“.

Die Pseudo-Operation EQU (siehe Abschnitt 4.5) kann auch zur Vereinbarung von 16-Bit-Konstanten benutzt werden. Dies sind je nach Anwendung Worte oder Adressen:

```
  UMSATZ  EQU     1E94H         ; Adresse
  STEUER  EQU     1E9BH         ; Adresse
  KOSTEN  EQU     23400         ; Konstante vom Typ Wort

          LD      (STEUER),BC   ; Wort in den Speicher bringen
          LD      HL,(UMSATZ)   ; Wort aus dem Speicher holen
          LD      DE,KOSTEN     ; Konstante in Registerpaar schreiben
```

5.3.2 Vereinbarung von Variablen durch Pseudo-Operationen

Bisher wurden bei Operationen, die auf den Speicher zugreifen, die Adressen mehr oder weniger willkürlich gewählt. Richtige Variablen gehören aber zu einem bestimmten Programm und werden in diesem mittels der Pseudo-Operationen DEFB (define byte) und DEFW (define word) vereinbart. Zum Beispiel bewirkt die (mit einer Marke versehene) Pseudo-Operation

```
 ZEICH:  DEFB    2AH     ; 1 Byte Speicherplatz unter dem Namen
                         ; ZEICH reservieren und mit dem Wert
                         ; 2AH belegen
```

daß ein Speicherplatz der Größe 1 Byte reserviert wird, der den Initialwert 2AH erhält und unter der symbolischen Adresse (Variablenname!) ZEICH angesprochen werden kann. Ebenso gut hätte der Befehl

```
 ZEICH:  DEFB    '*'     ; 1 Byte Speicherplatz unter dem Namen
                         ; ZEICH reservieren und mit dem Wert
                         ; '*' belegen
```

lauten können, denn der ASCII-Code für das Zeichen '*' ist 2AH und somit ergäbe sich der gleiche Objektcode.

Zur Reservierung eines Speicherplatzes für ein Wort mit dem Initialwert 1719H und der symbolischen Adresse DAUER muß die Pseudo-Operation lauten:

```
 DAUER:  DEFW    1719H   ; 1 Wort Speicherplatz unter dem Namen
                         ; DAUER reservieren und mit den Wert
                         ; 1719H belegen
```

An der Adresse DAUER wird der Wert 19H (LSB) abgelegt, an der Adresse DAUER+1 wird der Wert 17H (MSB) abgelegt.

Mit den Pseudo-Operationen DEFB und DEFW wurden bisher „initialisierte Variablen“ realisiert, also Variablen, denen vor dem Start des Programms bereits Werte zugewiesen wurden. Nun kommt es aber auch vor, daß Variablen im Programm benötigt werden, die zunächst keinen Wert haben sollen, sondern diesen erst im Verlauf des Programms zugewiesen bekommen. Diese uninitialisierten Variablen werden mit der Pseudo-Operation DEFS (define storage) geschaffen. Als Argument der Pseudo-Operation DEFS wird die Anzahl der Bytes angegeben, die für die Variable zur Verfügung gestellt werden sollen. Hier ein Beispiel:

```
ALPHA:  DEFS    1       ; Variable vom Typ Byte
BETA:   DEFS    2       ; Variable vom Typ Wort
GAMMA:  DEFS    8       ; Variable mit einem Speicherbedarf
                        ; von 8 Bytes, zum Beispiel fuer eine
                        ; Gleitpunktzahl
DELTA:  DEFS    256     ; Variable mit einem Speicherbedarf
                        ; von 256 Bytes, zum Beispiel fuer eine
                        ; Zeichenreihe oder einen Puffer
```

Relativ häufig müssen Texte in Variablen gespeichert werden. Zur Vereinbarung solcher initialisierter Text-Variablen gibt es die Pseudo-Operation DEFM (define memory), mit der eine Variable angelegt wird, deren Größe der angegebenen Zeichenkette angepaßt ist:

```
SPRUCH: DEFM    'Dies ist eine schoene Zeichenkette!'
```

Die Variable SPRUCH belegt nun 35 Bytes im Speicher.

Der Assembler vergibt die Adressen für den Programmcode und für die Datenspeicherplätze entsprechend der Auflistung im Quellcode fortlaufend. Befehle und Pseudo-Operationen können dabei in beliebiger Reihenfolge auftreten. Es ist jedoch guter Programmierstil, Daten und Code je in einem zusammenhängenden Bereich zu vereinbaren.

5.3.3 Arithmetik mit Worten

Arithmetische Operationen mit Worten werden stets im Registerpaar HL ausgeführt, so wie für Byte im Akkumulator. Dazu gleich ein paar Beispiele:

Mit Hilfe einer Routine soll eine vorzeichenlose 8-Bit-Zahl mit der Konstanten 10 multipliziert werden. Dabei kann ein Ergebnis entstehen, das nicht als 8-Bit-Größe darstellbar ist. Für die Aufnahme des Ergebnisses wird deshalb ein Speicherplatz zur Ablage eines Datenwertes vom Typ „Wort“ zur Verfügung gestellt. Für den Operanden wird ein Speicherplatz zur Aufnahme eines Datenwertes vom Typ „Byte“ eingerichtet. Die Multiplikation wird durch mehrmalige Addition ausgeführt. Vor Durchführung der Multiplikation wird der Operand vom Typ „Byte“ zum Typ „Wort“ expandiert und gleichzeitig in das Registerpaar HL übertragen. Eine Kopie des Operanden gelangt zusätzlich in das Registerpaar DE.

```
OPERAN: DEFS    1            ; Speicherplatz fuer Operand,
                             ; Wert wird spaeter eingesetzt
ERGEBN: DEFS    2            ; Speicherplatz fuer Ergebnis
MULT10: LD      A,(OPERAN)   ; Operand laden
        LD      L,A
        LD      H,0          ; Operand zu Wort expandieren
        LD      D,H          ; Kopie des Operanden erstellen
        LD      E,L
        ADD     HL,HL        ; Operand verdoppelt
        ADD     HL,HL        ; Operand vervierfacht
        ADD     HL,DE        ; Operand verfuenffacht
        ADD     HL,HL        ; Operand verzehnfacht
        LD      (ERGEBN),HL  ; Ergebnis abspeichern
```

Die weitere Arithmetik auf Größen vom Typ „Wort“ ist recht dürftig ausgelegt: Es gibt einen Befehl ADC (add with carry; addieren mit Übertrag), der wie der ADD-Befehl wirkt, jedoch auch noch den Inhalt des C-Flags zum Ergebnis addiert. Schließlich existiert noch der Befehl SBC (subtract with carry; subtrahieren mit Übertrag), der den Inhalt eines Registerpaars von HL subtrahiert und anschließend auch noch den Wert des C-Flags abzieht.

Um mit dem SBC-Befehl normale Subtraktionen durchzuführen, muß man sich folgender Befehle bedienen: SCF (set carry flag; C-Flag setzen), der den Wert 1 ins C-Flag bringt, und CCF (complement carry flag; C-Flag komplementieren), der den Wert des C-Flags durch sein Einerkomplement ersetzt. Hier ein Beispiel für den Ablauf einer Subtraktion von Worten:

```
OP1:    DEFS    2            ; Speicherplatz fuer ersten Operanden
OP2:    DEFS    2            ; Speicherplatz fuer zweiten Operanden
ERGEBN: DEFS    2            ; Speicherplatz fuer Ergebnis
SUBTRA: LD      HL,(OP1)     ; 1. Operand laden
        LD      DE,(OP2)     ; 2. Operand laden
        SCF                  ; C-Flag setzen
        CCF                  ; C-Flag umkehren (geloescht)
        SBC     HL,DE        ; normale Subtraktion durchfuehren
        LD      (ERGEBN),HL  ; Ergebnis abspeichern
```

Der Umgang mit dem ADC-Befehl ist ganz analog dazu. Ein Beispiel: Zwei vorzeichenlose ganze Zahlen, die als 32-Bit-Größen im Speicher stehen, sollen addiert und

das Ergebnis (modulo 2^{32}) ebenfalls im Speicher untergebracht werden. Es werden nacheinander zwei Additionen mit Worten ausgeführt. Zuerst werden die niederwertigen 16 Bits der beiden Zahlen addiert und als die niederwertigen 16 Bits des Ergebnisses abgespeichert. Anschließend werden die höherwertigen 16 Bits - unter Einbeziehung eines eventuell angefallenen Übertrages – addiert und als die höherwertigen 16 Bits des Ergebnisses im Speicher abgelegt. Der zuletzt angefallene Übertrag braucht nicht berücksichtigt zu werden, da das Ergebnis modulo 2^{32} reduziert werden soll. Das zugehörige Programm lautet nun:

```
OP1:    DEFS    4            ; Speicherplatz fuer ersten Operanden
OP2:    DEFS    4            ; Speicherplatz fuer zweiten Operanden
ERGEBN: DEFS    4            ; Speicherplatz fuer Ergebnis
ADD32:  LD      HL,(OP1)     ; niederwertige 16 Bits des
                             ; 1. Operanden laden
        LD      DE,(OP2)     ; niederwertige 16 Bits des
                             ; 2. Operanden laden
        ADD     HL,DE        ; niederwertige 16 Bits des
                             ; Ergebnisses berechnen
        LD      (ERGEBN),HL  ; niederwertige 16 Bits des
                             ; Ergebnisses abspeichern
        LD      HL,(OP1+2)   ; hoeherwertige 16 Bits des
                             ; 1. Operanden laden
        LD      DE,(OP2+2)   ; hoeherwertige 16 Bits des
                             ; 2. Operanden laden
        ADC     HL,DE        ; hoeherwertige 16 Bits des
                             ; Ergebnisses berechnen, dabei
                             ; Uebertrag aus vorhergehender Addition
                             ; beruecksichtigen
        LD      (ERGEBN+2),HL
                             ; hoeherwertige 16 Bits des
                             ; Ergebnisses abspeichern
```

Hierbei ist besonders die Adressierung der niederwertigen bzw. höherwertigen 16 Bits der jeweiligen Operanden zu beachten!

Während die Befehle ADC und SBC das C-Flag beeinflussen und somit auch für Zweierkomplement-Arithmetik einsetzbar sind, werden die Befehle ADD, INC und DEC (letztere werden im Abschnitt über Adressen und Zeiger behandelt) auf Worten hauptsächlich für Adreßberechnungen verwendet.

Merke: Arithmetische Operationen auf Worten finden stets im Registerpaar HL statt!

5.4 Übungsaufgaben für Teil 5

1. Ein Programm soll erstellt werden, das nacheinander folgende Additionen ausführt:

   41H + 20H
   3 + 48
   254Q + 221Q
   1110110B + 11010010B

2. Ein Programm soll erstellt werden, das nacheinander folgende Subtraktionen ausführt:

   68H − 20H
   57 − 48
   231Q − 116Q
   00110001B − 10100001B

3. Warum ist folgendes Programm sinnlos?

   LD 16,B

4. Folgende Zahlen sollen durch ein Programm negiert werden:

   80H, 0, 164Q, 10111001B

   Wie lauten die Ergebnisse in der jeweiligen Darstellung? Wie werden die Flags beeinflußt?

5. Welche allgemeinen Regeln lassen sich für die Beeinflussung der Flags durch die Befehle ADD, SUB und NEG aufstellen?

6. Was bedeutet folgende ASCII-Codierung einer Zeichenfolge?

   47H, 75H, 74H, 20H, 67H, 65H, 6DH, 61H, 63H, 68H, 74H, 21H, 0DH, 0AH

7. Folgender Satz soll in ASCII codiert werden:

   Der Assemblerkurs des KC-Clubs ist Spitze!

8. Großbuchstaben sollen in Ordnungszahlen umgewandelt werden, wobei „A“ = 1, „B“ = 2 usw. entspricht. Wie lautet das entsprechende Programm?

9. Ein Programm wird benötigt, das aus Ordnungszahlen Kleinbuchstaben herstellt. Dabei soll gelten: „a“ = 1, „b“ = 2 usw. Wie könnte das Programm aussehen?

10. Ein Programm soll erstellt werden, mit dem die Zahl 22131 in das Registerpaar BC geladen und anschließend dessen Inhalt in das Registerpaar DE kopiert wird.
11. Das ab Adresse 4256H stehende Wort soll auch unter Adresse 567BH abgespeichert werden.
12. Initialisierte Variablen für das Byte 45H, das Zeichen '+', das Wort 1700 und die Zeichenreihe „KC-Club? Was sonst!“ sollen vereinbart werden. Wieviel Speicherplatz belegen die einzelnen Variablen?
13. Mit welchen Anweisungen können uninitialisierte Variablen für ein Byte, ein Wort, eine 32-Bit-Zahl, eine Zeichenkette mit 7 Zeichen und einen Puffer mit einer Länge von 128 Bytes vereinbart werden?
14. Ein Programm soll erstellt werden, das das 12-fache einer mit 8 Bits im Speicher dargestellten vorzeichenlosen ganzen Zahl berechnet und das Ergebnis mit 16 Bits wieder im Speicher ablegt.
15. Wie lautet das Programm, welches zwei im Speicher stehende Binärzahlen mit je 64 Bits addiert und das Ergebnis (modulo 2^{64}) ebenfalls im Speicher ablegt?
16. Es soll ein Programm entwickelt werden, das eine in Zweierkomplement-Darstellung mit 64 Bits im Speicher stehende ganze Zahl von einer anderen solchen Zahl subtrahiert und das Ergebnis (ohne Berücksichtigung eines eventuell auftretenden Überlaufs) wieder mit 64 Bits im Speicher darstellt.

Teil 6 – Einfache Programmstrukturen

von Frank Dachselt

6.1 Befehlsabarbeitung

Im vorangegangenen Teil haben wir gesehen, wie der Prozessor mit einfachen Datenstrukturen umgeht, wie also Datenbytes und -worte vom Speicher in die Register des Prozessors transportiert werden und umgekehrt. Überhaupt ist die Abarbeitung eines Maschinenprogramms eine stetige Folge aus Lese- und Schreibvorgängen im Hauptspeicher des Rechners. Da werden Operationscode-Bytes vom Prozessor gelesen sowie Datenbytes gelesen und geschrieben.

Wenn wir uns das Ergebnis des Assemblers im Speicher anschauen, dann werden wir feststellen, daß im Speicher eine Folge von Operationscode- und Datenbytes steht. Als Beispiel betrachten wir das folgende Fragment eines Quelltextes:

```
LD   A,(2000H)
LD   B,35H
ADD  A,B
LD   (2002H),A
```

Die Funktion dieser Befehlsfolge können wir mit Hilfe des vorhergehenden Teils leicht nachvollziehen: Zum Inhalt der Speicherzelle mit der Adresse 2000H wird 35H addiert und das Ergebnis in die Speicherzelle mit der Adresse 2002H geschrieben. Wir wollen annehmen, daß der Assembler den Objektcode für dieses Programmstück ab der Adresse 700H im Speicher ablegt. Das erreichen wir z.B. durch eine geeignete ORG-Anweisung und die Assembleroption „O“ in EDAS. Mit Hilfe des DISPLAY-Kommandos können wir uns das Ergebnis im Speicher anschauen:

```
%DISPLAY 700 708
0700 3A 00 20 06 35 80 32 02  :. .5.2.
0708 20                        .......
```

Die vier Befehle haben eine Folge von neun Bytes im Speicher erzeugt. Wie wir bereits aus den vorangegangenen Teilen wissen, enthält der Befehlssatz des Z80 Befehle verschiedener Länge: Ein-, Zwei-, Drei- und Vierbytebefehle. Das erste Byte eines

jeden Befehls ist stets ein Operationscode-Byte (Opcode), danach folgen bei den Mehrbytebefehlen die Bytes für die Operanden oder evtl. weitere Opcode-Bytes. Der erste und der letzte Befehl unseres Beispiels ist jeweils ein Dreibytebefehl, nach dem Opcode (3AH bzw. 32H) folgen hier zwei Operandenbytes, die die beiden Bytes für die angegebenen Adressen enthalten (Reihenfolge der Bytes für die Adressen beachten: zuerst das niederwertige, dann das höherwertige Byte). Der zweite Befehl ist ein Zweibytebefehl, nach dem Opcode 06H folgt hier nur ein Byte für den Operanden. Der dritte Befehl schließlich ist ein Einbytebefehl, er besteht nur aus dem Opcode 80H.

Wenn wir nun „einfach mal so" in den Speicher schauen, wie erkennen wir dann, ob es sich bei einem Byte um ein Operationscode- oder Operandenbyte handelt? Nun, eigentlich überhaupt nicht! Ein beliebiges Byte kann sowohl ein Operationscode als auch Teil eines Operanden sein. Ohne bestimmte „Vorkenntnisse" können wir nicht sagen, welche Befehle und Operanden sich hinter einer willkürlich herausgeschnittenen Bytefolge im Speicher verbergen. Genauso ergeht es unserem Prozessor. Was wir brauchen, ist die Information über die Startadresse einer Befehlsfolge. Dieser Eintrittspunkt markiert das erste Byte des ersten Befehls und damit mit Sicherheit einen Operationscode. Dem Prozessor teilen wir diesen Startpunkt mit, indem wir das Register PC – den Programmzähler – mit der Adresse dieses Eintrittspunktes laden. Wie das geschehen kann, wird in den folgenden Abschnitten dieses Teils noch ausführlich behandelt.

Steht die richtige Startadresse im Register PC unseres Prozessors, dann beginnt dieser mit der Abarbeitung der Befehlsfolge, indem er zunächst das erste Opcodebyte ließt. Dieses Byte beinhaltet nicht nur Informationen über die Aktionen, die im Prozessor selbst ausgelöst werden sollen, sondern auch darüber, ob es sich bei diesem Befehl um einen Ein-, Zwei-, Drei- oder Vierbytebefehl handelt und wie die dem Opcode folgenden Bytes zu interpretieren sind. Wenn wir nun, um zu unserem Beispiel zurückzukehren, den Programmzähler mit der Adresse 700H geladen haben, dann wird der Prozessor zunächst das Byte 3AH aus dem Speicher lesen. Er erkennt sofort, daß es sich um einen Dreibytebefehl handelt und die beiden folgenden Bytes den Operanden enthalten. Diese werden anschließend ebenfalls aus dem Speicher gelesen und der Prozessor kann nun die eigentliche Operation – also den Inhalt der Speicherzelle 2000H ins Register A laden – ausführen. Nachdem dieser Befehl abgearbeitet wurde, erwartet der Prozessor auf der nächsten Adresse – in unserem Beispiel auf 703H – den Operationscode des nächsten Befehls und der Zyklus der Befehlsabarbeitung beginnt von vorn.

Wie wir sehen, wird der vom Assembler aus dem Quelltext erzeugte Objektcode nur dann die gewünschten Aktionen im Prozessor auslösen, wenn dieser den richtigen Eintrittspunkt in unsere Befehlsfolge kennt. So einfach und einleuchtend wie diese Aussage ist, so unangenehm können die Folgen sein, wenn diese Bedingung verletzt wird. Was passiert, wenn der Prozessor nicht auf der Adresse 700H sondern bei 701H mit der Befehlsabarbeitung beginnt, können wir mit Hilfe einer Befehlstabelle vielleicht erahnen: Zunächst erkennt er den Befehl NOP, danach einen bedingten relativen Sprung JRNZ 06, falls er jetzt noch in dieser Befehlsfolge ist, geht es mit DEC M weiter...

Diese Situation wird gewöhnlich Programm- oder Systemabsturz genannt, führt zu unkontrollierbaren Reaktionen des Rechners bis hin zum Verlust aller im Speicher befindlichen Daten oder – wie es das Systemhandbuch nennt – zu einem unbestimmten Systemzustand. An dieser Stelle hilft nur noch der Griff zur RESET-Taste...

Keinem Assemblerprogrammierer ist diese Situation unbekannt, auch langjährige Erfahrung schützt nicht davor, daß man ihr immer wieder unerwartet gegenübersteht. Es ist der Preis, den der Programmierer für die Freiheit, die der Assembler bietet, bezahlen muß. Der Anfänger muß an dieser Stelle lernen, nicht zu verzweifeln und statt dessen gezielt auf Fehlersuche zu gehen.

Nach diesem kurzen Abstecher in den Bereich der Befehlsabarbeitung wollen wir nun zum eigentlichen Kern dieses Teils kommen, den einfachen Programmstrukturen.

6.2 Sequenzen

Die Programmbeispiele in Teil 5 sowie das im ersten Abschnitt dieses Teils gehören zu den Sequenzen. Das sind Programmteile, die aus einer Folge von Befehlen bestehen, die – wenn sie sich einmal in der Abarbeitung befinden – stets in der gleichen Reihenfolge abgearbeitet werden. In einer Sequenz gibt es keine Möglichkeit, die Befehlsabarbeitung zu unterbrechen, um sie an einer anderen Stelle fortzusetzen oder Teile der Sequenz zu überspringen.

Um die Programmstrukturen in diesem Teil auch grafisch zu veranschaulichen, werde ich stark vereinfachte Ablaufpläne verwenden. Eine Sequenz ist in einem solchen Ablaufplan ein breiter senkrechter Strich, der von oben nach unter durchlaufen (abgearbeitet) wird.

Sequenzen bilden die Grundbausteine eines jeden Programmes und man sollte – insbesondere als Anfänger – immer erst einmal versuchen, kleinere Teilprogramme – soweit

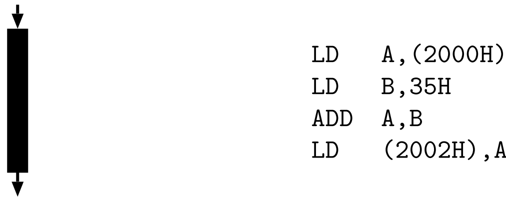


Bild 5: Vereinfachter Ablaufplan einer Sequenz

sie sich dazu eignen – als Sequenz zu programmieren. Sequenzen haben den Vorteil, sehr übersichtlich und „durchschaubar“ zu sein, was bei einer möglichen Fehlersuche sehr hilfreich sein kann. Die Optimierung der Programmlänge unter Verwendung komplexerer Programmstrukturen läßt sich im nachhinein immer noch vornehmen.

6.3 Verzweigungen

Mit einer Befehlssequenz allein wird es aber in den wenigsten Fällen getan sein. Wenn z.B. in Abhängigkeit von einer Eingabe oder eines Zwischenergebnisses verschiedene Befehlsfolgen abgearbeitet werden sollen, muß man den Programmablauf verzweigen. Um eine solche Programmverzweigung zu erreichen, bietet der Z80 in seinem Befehlssatz die bedingten Sprungbefehle, die ihrerseits wiederum absolute und relative Sprünge zulassen:

absolute bedingte Sprünge	relative bedingte Sprünge	Sprung, wenn
JP NZ,nn	JR NZ,e	Z-Flag rückgesetzt
JP Z,nn	JR Z,e	Z-Flag gesetzt
JP NC,nn	JR NC,e	C-Flag rückgesetzt
JP C,nn	JR C,e	C-Flag gesetzt
JP PO,nn		P/V-Flag rückgesetzt (Parity odd)
JP PE,nn		P/V-Flag gesetzt (Parity even)
JP P,nn		S-Flag rückgesetzt (positiv)
JP M,nn		S-Flag gesetzt (negativ)
	DJNZ e	B - 1 gleich Null

Die Entscheidung, ob der Sprung ausgeführt wird oder nicht, wird also stets auf die Auswertung eines Flags zurückgeführt. Die Bedeutung der einzelnen Flags kann im Abschnitt 3.2.3 nachgesehen werden. Liegt die gewünschte Sprungbedingung noch nicht in Form eines beeinflußten Flags vor, dann muß erst noch mittels einer geeigneten numerischen oder logischen Operation dafür gesorgt werden.

Die Verwendung eines absoluten oder relativen Sprunges richtet sich neben den zur Verfügung stehenden Sprungbedingungen vor allem danach, wie weit gesprungen werden soll. Mit einem absoluten Sprung kann an eine beliebige Stelle innerhalb des 64 K großen Speicherbereiches gesprungen werden; die anzuspringende Adresse wird direkt angegeben, was in der obigen Tabelle durch den symbolischen Operanden „nn“ (zwei Byte) angedeutet ist. Die absoluten Sprünge gehören damit zu den Dreibytebefehlen (ein Byte Opcode und zwei Byte Operand). Ein relativer Sprung ist dagegen ein Zweibytebefehl; der Operand besteht nur aus einem Byte „e“. Mit diesen Sprungbefehlen können wir nur in einem begrenzten Bereich um die „Absprungstelle“ herum bewegen. Der Operand „e“ wird als 8-Bit-Zahl im Zweierkomplement interpretiert und zum Befehlszähler PC addiert. Gemessen von der Adresse des folgenden Befehls können wir bis zu 127 Bytes nach vorn (in Richtung höherer Adressen) bzw. bis zu 128 Bytes zurück (in Richtung niedrigerer Adressen) springen (siehe auch Abschnitt 2.4.3.2).

Der begrenzte „Aktionsradius“ der relativen Sprünge ist in vielen Fällen ausreichend und man sollte stets versuchen, mit ihnen auszukommen. Neben den kürzeren Objektcode haben sie auch einen Geschwindigkeitsvorteil zur Folge, da ja nur ein Operandenbyte aus dem Speicher gelesen werden muß.

Bei der Verwendung relativer Sprünge macht uns der Assembler die Arbeit einfach, indem er die Sprungdistanz selbst berechnet, so daß wir wie bei absoluten Sprüngen mit einfachen symbolischen Adressen arbeiten können. Sollte das Sprungziel für einen relativen Sprung zu weit entfernt liegen, dann teilt uns das der Assembler durch eine Fehlermeldung mit. In diesem Fall müssen wir den relativen Sprung durch einen absoluten ersetzen, was aber immer möglich ist (siehe obige Tabelle).

Bild 6 zeigt einige Programmstrukturen mit Verzweigung. Bei einer Verzweigung wird also entweder der dem Sprungbefehl folgende Programmteil (Sprung nicht ausgeführt) oder ein entfernt liegender Programmteil (Sprung ausgeführt) abgearbeitet. Ein Spezialfall der Verzweigung ist die alternative Programmausführung, bei der nach der Abarbeitung des einen oder anderen Programmzweiges sich beide Zweige wieder treffen. Wenn man solche Programmstrukturen auf dem Papier mit Hilfe eines Ablaufplanes entwirft, dann muß man bei der Umsetzung in ein Programm daran denken, daß alle Befehle und Programmsequenzen auf einer Linie stehen müssen. In Bild 6 ist das in den ersten beiden Fällen angedeutet. Bei der alternativen Abarbeitung von Befehlsfolgen muß deshalb mindestens am Ende des einen Zweiges ein

unbedingter Sprung durchgeführt werden.

Wiederum ein Spezialfall davon ist die Alternative mit leerem Zweig. Hier wird ein Programmteil entweder abgearbeitet oder übersprungen. Im Gegensatz zur vollständigen Alternative ist hier kein unbedingter Sprung erforderlich.

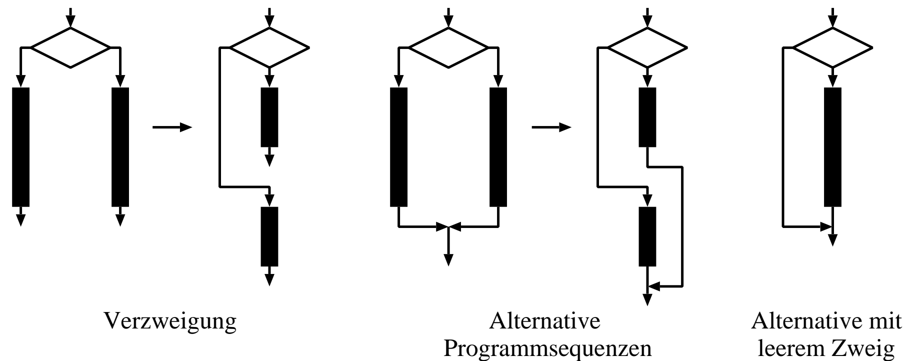


Bild 6: Programmstrukturen mit Verzweigungen

Wie erreicht man nun, daß die Flags, die bei bedingten Sprüngen ausgewertet werden, eine sinnvolle Sprungbedingung darstellen? Elemente, wie sie aus Hochsprachen bekannt sind, etwa „Sprung, falls A ungleich 10“ oder „Sprung, falls B größer als Null“, stehen uns als Maschinenbefehle nicht zu Verfügung. Bei der Assemblerprogrammierung müssen solche Bedingungen durch geeignete Auswahl der zur Verfüngung stehenden Maschinenbefehle nachgebildet werden. Dazu ist es zunächst einmal notwendig, daß man weiß, ob und wie die Maschinenbefehle die Flags beeinflussen, oder – wie bereits an anderer Stelle gesagt – daß man weiß, wo man nachschauen kann. Im Anhang B.2 ist in einer Tabelle die prinzipielle Beeinflussung der Flags durch die Z80-Befehle zusammengestellt. Die genauen Bedingungen zu deren Beeinflussung müssen dann den einzelnen Befehlsbeschreibungen entnommen werden. Oftmals genügt aber auch schon, die Funktion der einzelnen Flags aus dem Abschnitt 3.2.3 zu kennen, um zu wissen, wie diese durch eine Operation beeinflußt werden.

Mit den Maschinenbefehlen sind wir nicht nur in der Lage, alle aus Hochsprachen bekannten Bedingungen nachzubilden – schließlich sind diese Hochsprachenbefehle ja auch nur für uns nicht direkt sichtbare Maschinenroutinen –, es stehen uns vielmehr deutlich mehr und flexiblere Möglichkeiten zur Verfügung.

Eine der am häufigsten verwendeten Sprungbedingungen ist der numerische Vergleich von zwei Zahlenwerten, der also Aussagen wie „kleiner als“, „gleich“ oder „größer als“ liefert. Der Befehlssatz des Z80 bietet uns dafür den Befehl CP (compare; vergleichen). Er vergleicht den Inhalt des Akkumulators mit dem Inhalt eines anderen 8-Bit-Registers (r) oder einem Direktoperanden (n, als Zahl nach dem Befehl anzugeben), indem er die Differenz A - r bzw. A - n bildet und in Abhängigkeit des Subtraktionsergebnisses die Flags setzt. Das Subtraktionsergebnis selbst wird bei dieser Operation nirgends gespeichert und geht deshalb verloren.

Wenn wir annehmen, daß A für den Akkumulatorinhalt und B für den Vergleichswert (8-Bit-Register oder Direktoperand) steht, dann sind beim Vergleich von vorzeichenlosen Dualzahlen (der Zahlenbereich ist jetzt also 0...255) sechs verschiedene Aussagen möglich. Die folgende Tabelle stellt die Vergleichsergebnisse und die resultierenden Flags dar:

Vergleich vorzeichenloser Dualzahlen

Vergleichsergebnis		resultierende Flags für $A - B$
$A < B$	A kleiner als B	C
$A \leq B$	A kleiner als oder gleich B	C oder Z
$A > B$	A größer als B	NC und NZ
$A \geq B$	A größer als oder gleich B	NC
$A = B$	A gleich B	Z
$A \neq B$	A ungleich B	NZ

Diese Tabelle ist wie folgt zu lesen: Ist $A < B$, dann ist nach der Vergleichsoperation das C-Flag gesetzt (Zeile 1); im anderen Fall, also wenn $A \geq B$ gilt, ist das C-Flag zurückgesetzt (Zeile 4). Mitunter reicht ein Flag für die Auswertung nicht aus: Der Fall $A \leq B$ liegt dann vor, wenn entweder das C-Flag ODER das Z-Flag gesetzt sind (Zeile 2). Das Gegenteil, also $A > B$, gilt, wenn keines von beiden Flags gesetzt ist, also C-Flag UND Z-Flag rückgesetzt sind (Zeile 3). In solchen Fällen kann die Entscheidung nur durch zwei bedingte Sprungbefehle getroffen werden.

Beispiel 1: Sprung, falls Inhalt von A kleiner als Inhalt von B

```
        CP      B        ; Vergleich A - B
        JR      C,KLEIN  ; Sprung, falls A<B
GRGL:   ...              ; Sequenz fuer A>=B

KLEIN:  ...              ; Sequenz fuer A<B
```

Beispiel 2: Sprung, falls Inhalt von A kleiner als oder gleich 85H

```
        CP      85H     ; Vergleich A - 85H
        JR      C,KLGL  ; Sprung, falls A<85H
        JR      Z,KLGL  ; Sprung, falls A=85H
GROSS:  ...             ; Sequenz fuer A>85H

KLGL:   ...             ; Sequenz fuer A<=85H
```

Beispiel 3: Sprung, falls Inhalt von A größer als Inhalt von C

```
        CP      C        ; Vergleich A - C
        JR      NC,S1    ; Sprung, falls A>=C
        JR      KLGL     ; nein, dann A<C
S1:     JR      NZ,GROSS ; Sprung, falls A!=C
KLGL:   ...              ; Sequenz fuer A<=B

GROSS:  ...              ; Sequenz fuer A>B
```

Obwohl in beiden Fällen zwei Flags ausgewertet werden müssen, sind im Beispiel 2 nur zwei Sprunganweisungen notwendig, gegenüber drei im Beispiel 3. Das liegt an der einfachen Auswertbarkeit einer ODER-Verknüpfung – (C oder Z) – von Flagbedingungen. Die UND-Verknüpfung – (NC und NZ) – erfordert dagegen einen höheren Aufwand. Da beide Programmstücke die gleiche Unterscheidung treffen, ist es natürlich günstiger, das obere Beispiel zu verwenden; dazu müssen lediglich die angesprungenen Programmsequenzen miteinander vertauscht werden. Auf diese Weise ist eine UND-Verknüpfung von Flagbedingungen stets in die einfacher handhabbare ODER-Verknüpfung überführbar.

Die eben gezeigten Beispiele vergleichen vorzeichenlose Zahlen miteinander. Mit dem CP-Befehl ist es aber auch möglich, Zahlen in Zweierkomplement-Darstellung zu vergleichen. Da der Prozessor nicht weiß, mit welchem Format wir unseren Zahlen interpretieren, müssen zur Auswertung notwendigerweise andere Flags herangezogen werden. Die folgende Tabelle stellt die möglichen Vergleichsergebnisse und die resultierenden Flags dar:

Vergleich von Zahlen im Zweierkomplement

Vergleichsergebnis		resultierende Flags für $A - B$
$A < B$	A kleiner als B	(M und PO) oder (P und PE)
$A \leq B$	A kleiner als oder gleich B	(M und PO) oder (P und PE) oder Z
$A > B$	A größer als B	((P und PO) oder (M und PE)) und NZ
$A \geq B$	A größer als oder gleich B	(P und PO) oder (M und PE)
$A = B$	A gleich B	Z
$A \neq B$	A ungleich B	NZ

Für das Vergleichsergebnis ist zunächst das Vorzeichen der berechneten Differenz $A - B$ – angezeigt im S-Flag – entscheidend. Die Differenz von Zahlen im Zweierkomplement kann Werte zwischen -255 und 255 annehmen und damit den mit 8 Bit darstellbaren Zahlenbereich von -128 bis 127 weit überschreiten. Die berechnete Differenz und deren Vorzeichen sind nur dann korrekt, wenn sie sich im Bereich von -128 bis 127 bewegen. In diesem Fall tritt kein Überlauf auf und das P/V-Flag ist rückgesetzt. Tritt dagegen ein Überlauf auf (P/V-Flag ist gesetzt), dann ist zwar der Betrag der Differenz nicht bestimmbar, aber das resultierende Vorzeichen ist gerade das Inverse des tatsächlichen Vorzeichens der Differenz. Obwohl wir hier das P/V-Flag als Überlauf-Flag interpretieren müssen, wird bei der Notation der bedingten Sprünge die Flagbedingung nach ihrer Patitätsbedeutung benannt. Das ist aber nur eine Frage der Schreibweise: PO steht für ein rückgesetztes P/V-Flag, PE für ein gesetztes.

Zum Abschluß dieses Abschnitts betrachten wir noch das Beispiel einer alternativen Befehlssequenz.

Beispiel 4: Lade das Register B mit 23H, falls Z = 0, bzw. mit 33H, falls Z = 1

```
        JR      NZ,S0   ; Sprung, falls Z=0
        LD      B,33H   ; lade B mit 33H
        JR      S1
S0:     LD      B,23H   ; lade B mit 23H
S1:     ...             ; folgende Befehle fuer
                        ; beide Faelle
```

In Abhängigkeit vom Z-Flag wird hier genau einer der beiden Ladebefehle ausgeführt, so wie es der Verzweigung mit zwei alternativen Befehlssequenzen entspricht. Dieses Beispiel läßt sich jedoch leicht in eine Alternative mit leerem Zweig überführen, bei der, wie bereits gesagt, die Notwendigkeit des unbedingten Sprunges entfällt:

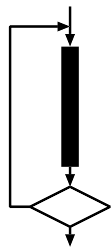
Beispiel 5: Alternative mit leerem Zweig

```
        LD      B,23H    ; lade B mit 23H
        JR      NZ,S0    ; Sprung, falls Z=0
        LD      B,33H    ; lade B mit 33H
S1:     ...              ; folgende Befehle fuer
                         ; beide Faelle
```

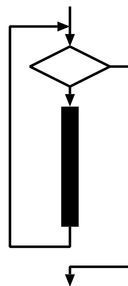
Hier haben wir die Befehlsfolge des einen Zweiges vor den bedingten Sprung verlagert. Das ist immer dann möglich, wenn beide Zweige die gleichen Register, Speicherzellen usw. verändern und somit das Ergebnis des einen Zweiges durch den anderen „überschrieben“ werden kann. Bei längeren Sequenzen steht dem jedoch die möglicherweise längere Abarbeitungszeit beim Durchlaufen beider Zweige gegenüber.

6.4 Schleifen

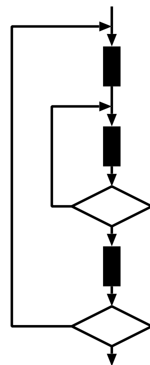
Eine Programmschleife wird benötigt, wenn bestimmte Befehlssequenzen mehrmals hintereinander ausgeführt werden sollen. Dazu erfolgt am Ende dieser Befehlssequenz ein Sprung zurück an deren Anfang. Zur Realisierung einer Schleife benötigt man



bodengesteuerte Schleife



kopfgesteuerte Schleife



zwei verschachtelte bodengesteuerte Schleifen

Bild 7: Schleifen

mindestens einen bedingten Sprungbefehl, der die Anzahl der Schleifendurchläufe und damit auch den Abbruch der Schleife steuert.

In Abhängigkeit davon, wo sich dieser bedingte Sprung befindet, unterscheidet man boden- und kopfgesteuerte Schleifen (Bild 7). Wichtigstes Merkmal dieser beiden Varianten ist die Mindestanzahl der Schleifendurchläufe. Während die bodengesteuerte Schleife stets mindestens einmal durchlaufen werden muß, ist bei der kopfgesteuerten Variante der Abbruch bereits vor dem ersten Durchlauf möglich. Letzteres scheint zwar zunächst einmal nicht sinnvoll, kommt aber in der Programmierpraxis oft vor, insbesondere dann, wenn die Anzahl der Schleifendurchläufe durch andere Programmteile berechnet wird.

In Schleifen, bei denen die Anzahl der Durchläufe bereits vor der Ausführung feststeht, verwendet man meist ein Register als sogenannten Schleifenzähler (in manchen Hochsprachen auch Laufvariable genannt). Solche Schleifen sollen in diesem Abschnitt im Vordergrund stehen.

Beispiel 6: bodengesteuerte Schleife mit 7 Durchläufen

```
        LD      A,0       ; setze Schleifenzaehler zurueck
LOOP:
        ...               ; die eigentlichen Befehle
                          ; der Schleife

        INC     A         ; Schleifenzaehler + 1
        CP      7         ; Schleifenzaehler = 7 ?
        JR      NZ,LOOP   ; wenn nicht, dann noch einmal
```

Anstatt bei der Erhöhung des Schleifenzählers den Befehl ADD 1 zu verwenden, werden in einfachen Schleifen meist die Inkrementier- bzw. die Dekrementierbefehle angewendet. Sie sind zum einem kürzer und damit schneller (Einbytebefehle) und zum anderen auf alle 8-Bit-Register anwendbar. Damit blockiert man nicht unnötigerweise den Akkumulator, der in der Schleife zumeist für andere Aufgaben benötigt wird.

Ein weiterer Nachteil des obigen Beispiels ist das Erhöhen des Schleifenzählers bei jedem Durchlauf. Wird der Wert des Schleifenzählers nicht unbedingt in dieser Form als Parameter in der Schleife benötigt, ist es günstiger, die Zählrichtung umzukehren:

Beispiel 7: bodengesteuerte Schleife mit 7 Durchläufen

```
        LD      B,7       ; setze Schleifenzaehler
LOOP:
        ...               ; die eigentlichen Befehle
                          ; der Schleife

        DEC     B         ; Schleifenzaehler - 1
        JR      NZ,LOOP   ; wenn nicht Null, dann noch einmal
```

Gegenüber Beispiel 6 entfällt in dieser Schleife der CP-Befehl. Den Test „Schleifenzähler gleich Null?“ brauchen wir nicht extra durchführen, da das bereits automatisch beim Dekrementieren – wie überhaupt bei jeder numerischen Operation – geschieht. Wir können deshalb das Flag sofort für den bedingten Sprung auswerten.

Ein weiterer Vorteil dieser Variante ist ihre leichte Konfigurierbarkeit für verschiedene Anzahlen von Durchläufen. Dazu brauchen wir nur den Schleifenzaehler zu verändern. Im Beispiel 6 wäre das nur durch „Opfern“ eines weiteren Registers für den Vergleichswert möglich.

Aber es geht noch einfacher: Der Befehlssatz des Z80 enthält den DJNZ-Befehl, der bereits in der Tabelle mit den bedingten Sprüngen aufgeführt ist. Dieser faßt das Dekrementieren von Register B sowie den bedingten Sprung zusammen und ermöglicht damit eine sehr kompakte Programmstruktur:

Beispiel 8: bodengesteuerte Schleife mit 7 Durchläufen

```
        LD      B,7      ; setze Schleifenzaehler
LOOP:
        ...              ; die eigentlichen Befehle
                         ; der Schleife

        DJNZ    LOOP     ; B = B - 1, Sprung, falls B!=0
```

Eine kopfgesteuerte Schleife mit 7 Durchläufen könnte dagegen wie folgt aussehen. Hier eignet sich der der DJNZ-Befehl nicht für den Schleifenabbruch:

Beispiel 9: kopfgesteuerte Schleife mit 7 Durchläufen

```
        LD      B,7      ; setze Schleifenzaehler
LOOP:   DEC     B        ; Schleifenzaehler - 1
        JR      Z,WEITER ; wenn Null, dann Abbruch

        ...              ; die eigentlichen Befehle
                         ; der Schleife

        JR      LOOP     ; zum naechsten Durchlauf
WEITER: ...
```

Wie bereits gesagt, unterscheiden sich boden- und kopfgesteuerte Schleifen in der Mindestanzahl der Schleifendurchläufe. Bei der hier gezeigten Verwendung eines 8-Bit-Registers als Schleifenzähler liegt die Anzahl der möglichen Durchläufe für die bodengesteuerte Variante im Bereich 1 ... 256, für die kopfgesteuerte Variante im Bereich 0 ... 255. Zwar können wir 256 nicht mit 8 Bit darstellen, aber wenn wir den Schleifenzähler mit 0 initialisieren, dann wird die bodengesteuerte Schleife genau 256 mal durchlaufen.

Mitunter reichen 256 Durchläufe für eine Schleife nicht aus. Dann kann man entweder zwei oder mehr Schleifen verschachteln (Bild 7) oder einen breiteren Schleifenzähler verwenden. Die Verschachtelung ist nur möglich, wenn sich die gewünschte Anzahl der Durchläufe durch die Multiplikation zweier bzw. mehrere kleinerer Zahlen angeben läßt, die dann als Initialwerte der Schleifenzähler verwendet werden.

Beispiel 10: zwei verschachtelte bodengesteuerte Schleifen zur Erzeugung von 320 Durchläufen der inneren Schleife ($5 \cdot 64 = 320$)

```
        LD      C,5       ; setze aeusseren Schleifenzaehler
LOOPA:  LD      B,64      ; setze inneren Schleifenzaehler
LOOPI:
        ...               ; die eigentlichen Befehle
                          ; der inneren Schleife

        DJNZ    LOOP      ; Abbruch der inneren Schleife
        DEC     C
        JR      NZ,LOOPA  ; Abbruch der aeusseren Schleife
```

Der DJNZ-Befehl existiert nur für das Register B; im Beispiel 10 verwenden wir es für die innere Schleife. Der Zähler für die äußere Schleife muß daher – wie im Beispiel 7 – mit einem extra DEC-Befehl heruntergezählt werden. Bei solchen Konstruktionen mit verschachtelten Schleifen, bei denen nur die Anzahl der Durchläufe der inneren Schleife interessiert, sollte man den Startwert des inneren Schleifenzählers möglichst hoch wählen, damit die äußere Schleife möglichst wenig durchlaufen wird. Das ist günstig für die Abarbeitungsgeschwindigkeit der gesamten Schleife.

Bei der Verwendung eines 16-Bit-Schleifenzählers können wir Schleifen mit bis zu $2^{16} = 65536$ Durchläufen programmieren. Allerdings stoßen wir hier schon an die Grenzen des Z80, der ja eigentlich nur ein 8-Bit-Rechenwerk besitzt. Zwar können wir 16-Bit-Register inkrementieren und dekrementieren, aber bei diesen Operationen werden die Flags nicht nach den bekannten Regeln beeinflußt. Ebenso steht uns kein Vergleichs-Befehl zur Verfügung, mit dem wir einen 16-Bit-Wert auf Gleichheit mit Null testen können. Zum Abbruch einer solchen Schleife müssen wir uns deshalb einen geeigneten Test einfallen lassen. Die eleganteste Methode zeigt das folgende Beispiel:

Beispiel 11: bodengesteuerte Schleife zur Erzeugung von 1333 Durchläufen mit einem 16-Bit-Schleifenzähler

```
        LD      BC,1333   ; setze Schleifenzaehler
LOOP:
        ...               ; die eigentlichen Befehle
                          ; der Schleife

        DEC     BC        ; Schleifenzaehler dekrementieren
        LD      A,B       ; A = B
        OR      C         ; A = B OR C
        JR      NZ,LOOP   ; Abbruch nur, wenn A und B gleich Null
```

Der Schleifenzähler BC enthält nach dem Dekrementieren genau dann den Wert Null, wenn auch die 8-Bit-Register B und C jeweils einzeln für sich Null sind. Wir könnten deshalb mittels zweier Vergleichsoperationen und zweier bedingter Sprünge den Schleifenabbruch realisieren. Noch einfacher ist aber der Test, ob das Bitmuster nach dem DEC-Befehl dem einer (16-Bit-)Null entspricht, ob also alle Bitstellen zurückgesetzt sind. Dazu werden die beiden 8-Bit-Register bitweise ODER-verknüpft; das geschieht mit dem OR-Befehl. Das Ergebnis im Akkumulator ist ein Byte, das genau dann Null ist, wenn die Inhalte von B und C Null sind. Wenn das der Fall ist, dann wird auch das Z-Flag gesetzt und kann für einen bedingten Sprung herangezogen werden.

Bei allen bisher betrachteten Schleifen war die Anzahl der Durchläufe vor dem Eintritt in die Schleife bekannt und der Abbruch der Schleife wurde durch einen Schleifenzähler gesteuert. Eine ganz andere Gruppe von Schleifen sind diejenigen, bei denen der Abbruch von einem „äußeren“ Ereignis abhängt. Solche Schleifen werden zum Beispiel dazu verwendet, um auf bestimmte Eingaben (z.B. von der Tastatur) zu warten und den Programmablauf danach fortzusetzen. Hier ist die Anzahl der Schleifendurchläufe vorher unbestimmt und in den meisten Fällen auch gar nicht von Interesse.

Beispiel 12: Schleife zum Warten auf die Betätigung der ENTER-Taste

```
LOOP:   CALL    KEYIN     ; Unterprogramm KEYIN liefert in A den
                          ; ASCII-Code der betaetigten Taste
        CP      0DH       ; Vergleich mit 0DH
        JR      NZ,LOOP   ; Abbruch nur, wenn A = 0DH
```

Da uns der genaue Mechanismus zur Bestimmung des ASCII-Codes einer gedrückten Taste auf der Tastatur nicht interessiert, begnügen wir uns an dieser Stelle damit, daß es ein Unterprogramm (siehe nächster Abschnitt) mit dem Namen KEYIN gibt, das uns diesen ASCII-Code im Akkumulator zur Verfügung stellt. Wir können uns daher auf den Vergleich dieses Wertes mit dem ASCII-Code des Steuerzeichens ENTER (siehe Tabelle im Anhang B.1) beschränken. Bei Übereinstimmung (Z-Flag ist gesetzt) verlassen wir die Schleife, ansonsten springen wir zurück zur Marke LOOP und fragen die Tastatur erneut ab. Dieser Vorgang wiederholt sich solange, bis die ENTER-Taste einmal gedrückt wird.

6.5 Unterprogramme

Wird in einem Programm eine bestimmte Befehlssequenz mehrmals und an verschiedenen Stellen benötigt, dann ist es günstig, diese Befehlssequenz in ein Unterprogramm zu verwandeln und dort, wo diese Sequenz gebraucht wird, mit einem Unterprogrammaufruf in den Ablauf des Programms einzufügen. Diese Vorgehensweise ist auch von Hochsprachen her bekannt und nennt sich dort Funktion bzw. Funktionsaufruf. Die Unterprogrammtechnik führt nicht nur zu kürzerem Quelltexten und

Programmen, sie hilft auch bei der Strukturierung von Programmen. Insbesondere große Quelltexte werden so übersichtlicher und leichter nachvollziehbar.

Für den Unterprogrammaufruf stellt uns der Z80 den CALL-Befehl zur Verfüngung. Das ist ein Dreibytebefehl, dessen Operand eine 16-Bit-Adresse enthält. Ein CALL-Befehl wirkt äußerlich zunächst wie ein JMP-Befehl, indem er die im Operanden stehenden Adresse in den Programmzähler PC lädt und damit bewirkt, daß als nächstes die Befehle ab dieser neuen Adresse abgearbeitet werden. Da die Befehle des Unterprogramms nur „eingeschoben" werden sollen, müssen wir nach der Abarbeitung des Unterprogramms wieder an die aufrufende Stelle im Programm zurückkehren, der nächste Befehl ist der dem CALL-Befehl folgende. Für die Rückkehr aus einem Unterprogramm gibt es den RET-Befehl. Das ist ein Einbytebefehl, der also keine Information darüber enthält, wo der Prozessor die Programmabarbeitung fortsetzen soll.

Wie weiß nun der Prozessor, wo er nach dem RET-Befehl weiterarbeiten soll? Schließlich kann das Unterprogramm von jeder beliebigen Stelle aus aufgerufen werden. Dazu hat sich der Prozessor die Stelle des Unterprogrammaufrufes gemerkt, indem er bei der Abarbeitung des CALL-Befehls die Adresse des dem CALL folgenden Befehls im Stack ablegt. Die genaue Funktionsweise dieses Bereiches im Hauptspeicher und weitere Möglichkeiten zu seiner Nutzung werden wir im nächsten Teil des Kurses genauer betrachten. An dieser Stelle begnügen wir uns damit, daß der RET-Befehl die im Stack zwischengespeicherte Adresse aus dem Hauptspeicher liest und sie in den Programmzähler lädt. Das Ablegen dieser Rückkehradresse im Stack sowie das zurücklesen geschehen dabei völlig ohne unser Zutun. Auf diese Weise ist auch die Verschachtelung von Unterprogrammen möglich, d.h. ein Unterprogramm kann weitere Unterprogramme aufrufen, ohne daß der Prozessor den Überblick über die Rückkehradressen verliert.

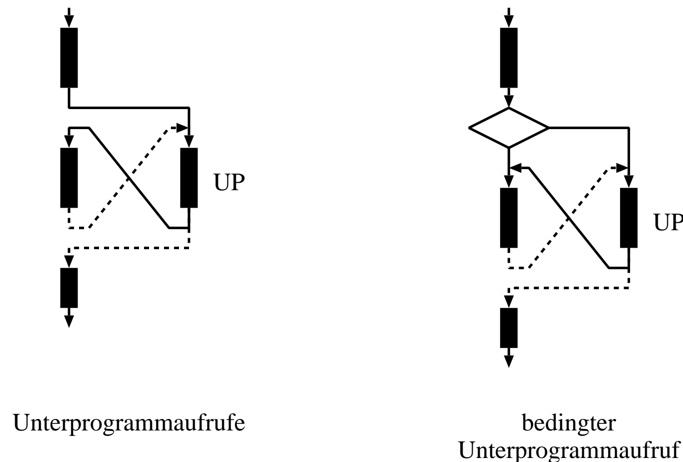


Bild 8: Unterprogrammaufrufe

Schauen wir uns das zunächst an einem einfachen Beispiel an: Aufgabe des Unterprogramms ADDM ist es, zum Inhalt der Speicherzelle 2000H den Inhalt des Registers B zu addieren und das Ergebnis in der Speicherzelle 2001H abzulegen.

Beispiel 13: Aufrufen eines Unterprogramms

```
        LD      B,25
        CALL    ADDM            ; Aufruf mit B = 25
        ...

        LD      B,73            ; Aufruf mit B = 73
        CALL    ADDM
        ...


ADDM:   LD      A,(2000H)       ; Unterprogramm ADDM
        ADD     B
        LD      (2001H),A
        RET
```

Das Unterprogramm ADDM erwartet im Register B den zu addierenden Wert. Dieses Register muß deshalb vor dem Unterprogrammaufruf im Hauptprogramm entsprechend gesetzt werden. Ebenso gibt das Unterprogramm zusätzlich im Register A das Ergebnis der Addition ans Hauptprogramm zurück, wo dieser Wert weiterverwendet werden kann. Diesen Vorgang nennt man Parameterübergabe. Durch Einführung von Parametern werden Unterprogramme flexibler einsetzbar und können somit auch öfter zum Einsatz kommen. Je nach Art und Umfang der Parameter kommen verschiedene Formen der Übergabe zur Anwendung, die an dieser Stelle nur kurz genannt werden sollen: Die *Übergabe von Registerinhalten* ist am schnellsten, bleibt aber auf wenige Bytes bzw. Worte beschränkt. Sind mehr Daten zu übergeben, die zudem noch längerfristig gültig sind, dann kann dazu ein vereinbarter *Bereich im Hauptspeicher* verwendet werden. Sind die zu übergebenden Daten unterschiedlich umfangreich oder möchte man sich das Umspeichern in vereinbarte Bereiche ersparen, dann verwendet

man *Zeiger* auf Datenbereiche (siehe nächsten Kursteil), die sich dann an wechselnden Stellen im Hauptspeicher befinden können.

Ein wichtiger Bestandteil eines Unterprogramms ist seine Dokumentation. Wie auch im übrigen Quelltext sollte man hier auf keinen Fall sparen. In größeren Programmen ist ein Unterprogramm nicht nur von lokaler Bedeutung, sondern es kann aus sehr unterschiedlichen Situationen heraus aufgerufen werden. Um das sichere Funktionieren zu gewährleisten, sollte man stets genau wissen, was in einem Unterprogramm vorsichgeht. Neben der Beschreibung eigentlichen Funktion sowie der Parameterübergabe beim Aufruf und bei Rückkehr, sind auch noch die vom Unterprogramm veränderten Register von Bedeutung. Sie geben Auskunft darüber, welche Register vor einem Unterprogrammaufruf gerettet werden müssen, falls sich in ihnen Daten befinden, die im weiteren Programmverlauf noch von Bedeutung sind. Oft benötigte Unterprogramme sorgen deshalb selbst dafür, daß sie so wenig wie möglich Registerinhalte verändern, indem sie diese vor der eigentlichen Funktion retten und danach wiederherstellen (siehe nächsten Kursteil).

Ein gutes Beispiel für die Dokumentation von Unterprogrammen bietet das Systemhandbuch des Grundgerätes. Ab Seite 72 steht dort die Liste der nutzbaren CAOS-Unterprogramme, die alle für den Programmierer relevanten Informationen enthält. Für das Unterprogramm aus Beispiel 14 könnte das wie folgt aussehen.

Beispiel 14: Dokumentation eines Unterprogramms

```
;-------------------------------------
; Unterprogramm ADDM
; Fkt:  <2001H> = <2000H> + B
; PE:   B ... zu addierende Zahl
; PA:   A ... Summe
;       F ... Flags nach der Addition
; VR:   AF
;-------------------------------------
ADDM:   LD      A,(2000H)
        ADD     B
        LD      (2001H),A
        RET
```

Eine besondere Gruppe von Unterprogramme sind die bereits erwähnten und im Teil 4 auch schon verwendeten Systemunterprogramme, die das Betriebssystem CAOS bereithält. Ihre Anwendung entbindet uns von einer Vielzahl von ständig wiederkehrenden Aufgaben, insbesondere in Bereich der Ein- und Ausgabe. Jeder CAOS-Assemblerprogrammierer sollte sich deshalb mit diesen Unterprogrammen und ihrer Anwendung vertraut machen. In den Beispielprogrammen zu diesem Kursteil finden bereits einige dieser Unterprogramme Verwendung. Aus den angegebenen Kommentaren und der Beschreibung im Systemhandbuch sollte jeder die Funktion in diesen Beispielen nachvollziehen können.

Der Aufruf von CAOS-Systemunterprogrammen – oder kurz CAOS-UPs – unterscheidet sich etwas vom Aufruf der „normalen“ Unterprogramme, die wir bis jetzt betrachtet haben. Das liegt daran, daß die CAOS-UPs über einen Programmverteiler (siehe folgende Kursteile) aufgerufen wird. Dazu wird bei jedem Unterprogrammaufruf die gleiche Adresse (0F003H) angesprungen, die eigentliche Unterscheidung der auszuführenden Funktionen geschieht durch ein Byte direkt nach dem CALL-Befehl. Dieses Byte ist kein Befehlscode im eigentlichen Sinne, sondern vielmehr ein Datenbyte, das vom Betriebssystem vor der Ausführung der gewünschten Funktion gelesen wird und die gewünschte Funktion auswählt. Das Betriebssystem sorgt bei diesen Unterprogrammaufrufen auch dafür, daß der RET-Befehl am Ende jeder dieser Funktionen nicht auf das Byte direkt nach dem CALL-Befehl (wie wir es für ein normales Unterprogramm erwarten würden), sondern auf das darauffolgende Byte zurückführt. Dieses Byte erst ist der Opcode des nächsten Befehls.

Beispiel 15: Aufruf eines CAOS-UP

```
CALL    0F003H
DB      0EH     ; UP-Nummer, hier
                ; Tastenstatusabfrage
...             ; der naechste Befehl
```

Zum Schluß dieses Abschnitts kommen wir noch zu einer Gruppe von Befehlen, die ich an dieser Stelle nur der Vollständigkeit halber erwähnen möchte. Es handelt sich dabei um die bedingten Unterprogrammaufrufe und die bedingten Rückkehrbefehle. In der gleichen Weise, wie wir es bei den bedingten Sprungbefehlen gesehen haben, können wir auch den Aufruf eines Unterprogramms bzw. die Rückkehr aus einem solchen von Flags abhängig machen. Bild 8 zeigt das auf der rechten Seite am Beispiel eines bedingten Unterprogrammaufrufes. Die Möglichkeiten, die sich mit dieser Befehlsgruppe eröffnen, können leicht aus dem Abschnitt über Verzweigungen ersehen werden. Die folgende Tabelle zeigt alle Befehle dieser Gruppe.

bedingter Unterprogrammaufruf		bedingte Rückkehr		Ausführung, wenn
CALL	NZ,nn	RET	NZ	Z-Flag rückgesetzt
CALL	Z,nn	RET	Z	Z-Flag gesetzt
CALL	NC,nn	RET	NC	C-Flag rückgesetzt
CALL	C,nn	RET	C	C-Flag gesetzt
CALL	PO,nn	RET	PO	P/V-Flag rückgesetzt (Parity odd)
CALL	PE,nn	RET	PE	P/V-Flag gesetzt (Parity even)
CALL	P,nn	RET	P	S-Flag rückgesetzt (positiv)
CALL	M,nn	RET	M	S-Flag gesetzt (negativ)

6.6 Beispielprogramme und Übungsaufgaben

Zum praktischen Kennenlernen der in diesem Teil vorgestellten Programmstrukturen habe ich ein paar Quelltexte vorbereitet, die auch als Grundlage für einige Übungsaufgaben dienen. Zunächst möchte ich aber die Verwendung von einigen CAOS-spezifischen Arbeitszellen erläutern, die wir bei dieser Gelegenheit erstmalig verwenden. Um das folgende besser zu verstehen, sollte man am besten gleichzeitig im Systemhandbuch des Grundgerätes die Seite 88 aufschlagen.

Die CAOS-Arbeitszellen, die wir hier verwenden, befinden sich in einem Teil des IRM, also im Speicherbereich zwischen 8000H und BFFFH. Hier speichert das Betriebssystem veränderliche Systemparameter und nimmt gleichzeitig von gerade laufenden Programmen Veränderungen an diesen Parametern entgegen. In der Gegenrichtung können natürlich auch Anwenderprogramme Informationen zum Systemzustand aus diesen Arbeitszellen auslesen. Diese Arbeitszellen stehen auch in direkter Verbindung zu den oben bereits erwähnten CAOS-UPs, da sie vielfach zur Parameterübergabe in diese Unterprogramme verwendet werden.

Als erstes verwenden wir die Parameterübergabe an ein Programm über die CAOS-Kommandozeile. Beim Start eines Programms können wir nach dem Programmnamen eine Folge von durch Leerzeichen voneinander getrennten Zahlenwerten angeben (maximal 10). Diese Zahlen dürfen maximal vierstellig sein und werden vom CAOS als Hexadezimalzahlen interpretiert; es sind also neben den Ziffern 0...9 auch die Buchstaben A...F erlaubt. Beim Programmaufruf wandelt das Betriebssystem diese Eingabe Zahl für Zahl in 16-Bit-Dualzahlen um und übergibt sie dem aufgerufenen Programm in einem Teil der erwähnten Arbeitszellen. Die Speicherzelle mit der Adresse B781H enthält dabei die Anzahl der erkannten und umgewandelten Zahlenwerte. In den folgenden Zellen stehen die gewandelten 16-Bit-Zahlen: die erste in den Zellen B782H und B783H, die zweite in den Zellen B784H und B785H usw. Dabei ist zu beachten, daß das niederwertige Byte auf der jeweils niederen Adresse und das höherwertige Byte auf der jeweils höheren Adresse steht. Gültig sind dabei nur so viele 16-Bit-Zahlen, wie in der Zelle B781H angegeben, der Inhalt der weiteren Zellen ist dann unbestimmt. Wenn wir nur Bytes in der Kommandozeile übergeben wollen – das geschieht z.B. im Programm COMP.ASM – dann brauchen wir nur die jeweils niederwertige Zelle eines jeden Parameterwertes zu verarbeiten; das höherwertige Byte wird ignoriert. Mit der Kommandozeile

```
%COMP 35 67
```

übergibt das Betriebssystem in der Zelle B781H den Wert 02H sowie in den Zellen B782H und B784H die Hexadezimalzahlen 35H bzw. 67H.

Die zweite Arbeitszelle, die wir hier verwenden wollen, beinhaltet die aktuelle Cursorposition. Immer wenn wir dem Betriebssystem Zeichen zur Ausgabe auf dem Bildschirm übergeben, werden diese an der aktuellen Cursorposition ausgegeben. Nach der Ausgabe hat sich diese Cursorposition um die Anzahl der ausgegeben Zeichen nach rechts verschoben – natürlich unter Berücksichtigung eines eventuellen Zeilenumbruchs oder sogar eines Bildschirmscrollens. Zu beachten ist dabei, daß diese Cursorposition die relative Position im aktuellen Fenster angibt – nicht etwa die absolute Position auf dem Bildschirm. Da wir heute nur mit dem Standardfenster (gesamter Bildschirm) arbeiten, stimmen beide Interpretationen trotzdem überein. Die aktuelle Cursorposition steht in den Zellen mit den Adressen B7A0H und B7A1H, getrennt nach Zeile und Spalte; Adresse B7A0H beinhaltet die Spalte, Adresse B7A1H die Zeile. Mit dem Befehl

```
        LD      HL,(0B7A0H)
```

wird also die aktuelle Zeile ins Register H und die aktuelle Spalte ins Register L gebracht. Beide Werte können nun unabhängig voneinander verändert und mit dem Befehl

```
        LD      (0B7A0H),HL
```

wieder dem Betriebssystem übergeben werden. Ab der nächsten Bildschirmausgabe kommen die neuen Werte zur Anwendung.

Nun aber zu den eigentlichen Programmen, die sich im Archiv ASMKURS6.PMA befinden, und den damit verbundenen Übungsaufgaben:

FLAGS.ASM Dieses Programm ist zum Experimentieren mit der Flagbeeinflussung bei numerischen und logischen Operationen gedacht. In einer Schleife mit einstellbaren Start- und Endwert (Parameter nach dem Programmnamen) wird mit dem Wert des Schleifenzählers im Akkumulator eine bestimmte Operation durchgeführt. Dabei werden der Wert vor und nach der Operation, sowie die aus der Operation resultierenden Flags angezeigt. Nach jedem Schleifendurchlauf wartet das Programm auf die Betätigung einer Taste, mittels <BRK> kann dabei das Programm abgebrochen werden. In der beiliegenden Version ist die Operation ADD 33H enthalten. An der im Quelltext gekennzeichneten Stelle kann jede beliebige numerische oder logische Funktion, die mit dem Akkumulator arbeitet, eingesetzt werden, wie z.B. SUB n, CP n, INC A, DEC A, AND n, OR n usw. Wer also genau wissen möchte, wie die Flags im Z80 von den genannten Operationen beeinflußt werden, der sollte mit diesem Programm etwas experimentieren.

COMP.ASM Dieses Programm vergleicht zwei vorzeichenlose 8-Bit-Dualzahlen miteinander und gibt das Ergebnis als Zeichenkette „größer“, „gleich“ oder „kleiner“ aus. Die zu vergleichenden Zahlen werden als Parameter nach dem Programmnamen angegeben.

Aufgabe 1: Man schreibe unter Verwendung der Tabelle auf Seite 62 ein Programm COMPZ.ASM, das den Vergleich für 8-Bit-Zahlen im Zweierkomplement durchführt!

LINE.ASM Dieses Programm gibt mittels einer Schleife eine senkrechte Linie aus „H“-Zeichen auf dem Bildschirm aus.

Aufgabe 2: Durch Variieren der Initialisierung der Schleifenvariablen und der Abbruchbedingung sowie unter evtl. Verwendung verschachtelter Schleifen programmiere man Schleifen, die

- eine waagerechte Linie,
- eine Diagonale von links oben nach rechts unten,
- eine Diagonale von links unten nach rechts oben,
- eine zentrierte Fläche aus 20 mal 25 (Höhe mal Breite) Zeichen

auf dem Bildschirm ausgeben!

Aufgabe 3: In den beiliegenden Programmen mit Parameterübergabe von der CAOS-Kommandozeile fehlt bisher noch jeweils ein Test, ob eine ausreichende Anzahl von Parametern übergeben wurde. Unter Verwendung des Inhaltes der Zelle B781H füge man in den entsprechenden Programmen jeweils einen solchen Test ein, der das Programm mit einer Fehlermeldung abbricht, falls weniger als die benötigte Anzahl von Parametern angegeben wurden!

Teil 7 – Komplexe Datenstrukturen

von Jörg Linder

7.1 Adressen und Zeiger

Mit „Adresse“ wird eine Speicheradresse bezeichnet, beim Z80 handelt es sich dabei also um eine Größe vom Typ „Wort“. Adressen lassen sich unterscheiden in Code-Adressen und Daten-Adressen. Code-Adressen sind Anfangsadressen von Befehlen; man benutzt sie, um einen bestimmten Befehl anzuspringen. Daten-Adressen sind Adressen von Datenwerten im Speicher.

Ein Zeiger ist ein Verweis auf einen Datenwert, der von den eigentlichen Daten unabhängig ist. Auf dem Z80 repräsentiert man einen Zeiger am einfachsten durch die Anfangsadresse der (eventuell kompliziert strukturierten) Datenwerte. Zeiger dienen häufig zur Verkettung der Elemente veränderlicher (dynamischer) Datenstrukturen.

7.1.1 Indirekte Sprünge

Code-Adressen wurden bereits im Abschnitt „6.3 Verzweigungen“ in Form direkter Sprungadressen des JP-Befehls verwendet. Beispiele dafür wären JP 3A4BH oder JP START.

Enthält das Registerpaar HL eine Code-Adresse, so kann damit ein indirekter Sprung auf diese Adresse ausgeführt werden. Der Befehl dazu lautet:

```
JP      (HL)    ; Programmladezaehler mit dem
                ; Inhalt des Registerpaares HL laden
                ; d.h. Sprung auf den Befehl, dessen
                ; Anfangsadresse in HL steht
```

In diesem Fall könnte man HL als „Code-Adreß-Register“ bezeichnen. Dazu ein Beispiel: Es soll ein Programmstück angesprungen werden, dessen Anfangsadresse ab der Adresse 6F90H im Speicher steht. Also:

```
LD      HL,(6F90H)      ; Code-Adresse holen
JP      (HL)            ; Programmstueck anspringen
```

Die Indexregister IX und IY können ebenfalls auf diese Weise benutzt werden.

```
JP      (IX)    ; Programmladezaehler mit dem
                ; Inhalt des Indexregisters IX laden
                ; d.h. Sprung auf den Befehl, dessen
                ; Anfangsadresse in IX steht

JP      (IY)    ; Programmladezaehler mit dem
                ; Inhalt des Indexregisters IY laden
                ; d.h. Sprung auf den Befehl, dessen
                ; Anfangsadresse in IY steht
```

Die Indexregister werden erst in einem späteren Abschnitt einer genaueren Betrachtung unterzogen, weshalb für den Umgang mit Indexregistern an dieser Stelle auf diesen Abschnitt verwiesen wird.

7.1.2 Indirekte Adressierung von Daten

Auch Daten-Adressen in Form direkter Adressierung sind bereits verwendet worden. So zum Beispiel in LD A,(1010H) oder in LD (ERGEBN),HL. Zur indirekten Adressierung von Datenwerten werden die Registerpaare BC, DE und HL, der Stackpointer SP sowie die Indexregister IX bzw. IY benutzt. (Auf den Stackpointer SP wird im Abschnitt „8.1 Stack“ näher eingegangen. Zu den Indexregistern siehe obige Anmerkung.)

Die Einsatzmöglichkeiten der Registerpaare BC und DE als Daten-Adreß-Register unterliegen erheblichen Beschränkungen. Es gibt je zwei Lade-Befehle:

```
LD      A,(BC)  ; Inhalt der Speicherzelle,
                ; deren Adresse im Registerpaar BC
                ; steht, in den Akkumulator bringen
LD      A,(DE)  ; Inhalt der Speicherzelle,
                ; deren Adresse im Registerpaar DE
                ; steht, in den Akkumulator bringen
LD      (BC),A  ; Inhalt des Akkumulators in
                ; die durch das Registerpaar BC
                ; adressierte Speicherzelle bringen
LD      (DE),A  ; Inhalt des Akkumulators in
```

```
                        ; die durch das Registerpaar DE
                        ; adressierte Speicherzelle bringen
```

Dazu gleich ein Anwendungsbeispiel: Eine im Speicher stehende ganze Zahl in Zweierkomplement-Darstellung soll negiert und das Ergebnis in einer anderen Speicherzelle untergebracht werden. Die Adresse des Operanden wird im Registerpaar BC erwartet, die Adresse des Ergebnisses im Registerpaar DE. Das Programm sieht folgendermaßen aus:

```
LD      A,(BC)  ; Operand aus dem Speicher holen
NEG             ; Operation durchfuehren
LD      (DE),A  ; Ergebnis abspeichern
```

Darüber hinaus gibt es noch spezielle Befehle für blockweises Bewegen von Daten, in denen das Registerpaar DE als Daten-Adreß-Register eingesetzt wird. Diese Befehle werden im Abschnitt „7.3 Felder“ behandelt.

Für die Daten-Adressierung ist das Registerpaar HL jedoch weitaus wichtiger. In vielen Befehlen kann anstelle eines 8-Bit-Registers auch der Inhalt einer Speicherzelle (ebenfalls 8 Bit groß) auftauchen, indem die Speicherzelle durch das Registerpaar HL indirekt adressiert wird. Hier ein paar Beispiele:

```
LD      C,(HL)   ; Inhalt der Speicherzelle,
                 ; deren Adresse im Registerpaar HL
                 ; steht, in Register C bringen
LD      (HL),E   ; Inhalt des Registers E
                 ; in die durch das Registerpaar HL
                 ; adressierte Speicherzelle bringen
LD      (HL),24H ; den Wert 24H in die durch
                 ; das Registerpaar HL adressierte
                 ; Speicherzelle bringen
ADD     A,(HL)   ; Inhalt der Speicherzelle,
                 ; deren Adresse im Registerpaar HL
                 ; steht, zum Akkumulator addieren
SUB     (HL)     ; Inhalt der Speicherzelle,
                 ; deren Adresse im Registerpaar HL
                 ; steht, vom Akkumulator abziehen
CP      (HL)     ; Inhalt der Speicherzelle,
                 ; deren Adresse im Registerpaar HL
                 ; steht, mit dem Inhalt des
                 ; Akkumulators vergleichen
```

Weitere Anwendungsmöglichkeiten des Registerpaares HL als Daten-Adreß-Register können in den Tabellen und Übersichten im Anhang nachgeschlagen werden.

Im folgenden soll ein Problem betrachtet werden, bei dem drei Daten-Adreß-Register gleichzeitig benötigt werden: Zwei im Speicher stehende vorzeichenlose ganze Zahlen mit je 8 Bits sollen addiert und das Ergebnis – nötigenfalls reduziert modulo 256 – im Speicher deponiert werden. Die drei Speicheradressen sollen jeweils über ein Registerpaar spezifiziert werden.

Bei solchen Operationen mit drei Adressen muß normalerweise der zweite Operand durch das Registerpaar HL angegeben werden, damit er direkt (also ohne Zwischenspeicherung in einem Register) in die Operation einbezogen werden kann. Die Zuordnung der Registerpaare BC und DE zum ersten Operanden bzw. zum Ergebnis ist dagegen beliebig. (Im Beispiel wird das Registerpaar BC als Adreß-Register für das Ergebnis, das Registerpaar DE als Adreß-Register für den ersten Operanden verwendet.):

```
LD      A,(DE)  ; ersten Operanden holen
ADD     A,(HL)  ; zweiten Operanden direkt
                ; in die Operation einbeziehen
LD      (BC),A  ; Ergebnis abspeichern
```

Im Abschnitt „5.3.3 Arithmetik mit Worten“ wurde gezeigt, wie man mit Größen vom Typ „Wort“ rechnen kann. Diese Methoden können zur Berechnung von Daten-Adressen benutzt werden.

Meist stehen jedoch die Daten fortlaufend im Speicher, entweder weil mehrere Datenwerte logisch zusammengehören, oder weil ein Datenwert mehr als ein Byte Speicherplatz benötigt. In diesem Fall liegen die verwendeten Adressen nahe beieinander; der Z80 unterstützt diesen Sachverhalt durch die Befehle INC (increment; erhöhen) und DEC (decrement; verringern), deren Verwendung weniger umständlich als die Berechnung mittels Wort-Arithmetik ist.

Der INC-Befehl erhöht den Inhalt eines Registers oder eines Registerpaares um 1, der DEC-Befehl verringert ihn um 1, also zum Beispiel:

```
        INC     HL      ; Inhalt des Registerpaares HL
                        ; um 1 erhoehen
        DEC     BC      ; Inhalt des Registerpaares BC
                        ; um 1 verringern
```

Diese Befehle sind schneller, kürzer und klarer als Arithmetik-Befehle. Es wird auch kein zweites Register(paar) zur Berechnung benötigt. Daß INC und DEC keine echten Arithmetik-Befehle sind, erkennt man daran, daß die Flags nicht verändert werden. (Im allgemeinen werden sie aber trotzdem zu den Arithmetik-Befehlen gezählt.)

Es soll nun die Überlegenheit der indirekten Adressierung gegenüber der direkten Adressierung an einem Beispiel demonstriert werden. Eine vorzeichenbehaftete ganze Zahl mit 8 Bits in Zweierkomplementdarstellung soll von einer anderen solchen Zahl subtrahiert werden. Die beiden Datenwerte sollen an festen Adressen im Speicher stehen, das Ergebnis (ohne Berücksichtigung von Überlauf) soll ebenfalls in den Speicher gebracht werden. Der Datenbereich sieht dann so aus:

```
OP1:    DEFS    1       ; Speicherplatz fuer
                        ; ersten Operanden
OP2:    DEFS    1       ; Speicherplatz fuer
                        ; zweiten Operanden
ERG:    DEFS    1       ; Speicherplatz fuer Ergebnis
```

Zuerst die Realisierung mit direkter Adressierung:

```
        LD      A,(OP2) ; zweiter Operand muss zuerst
                        ; geholt werden, da als Ziel-
                        ; register nur der Akkumulator
                        ; zur Verfuegung steht
        LD      D,A     ; zweiten Operanden hilfsweise
                        ; in Register sichern
        LD      A,(OP1) ; ersten Operanden holen
        SUB     D       ; Operation durchfuehren
        LD      (ERG),A ; Ergebnis speichern
```

Hier nun eine Lösung mit indirekter Adressierung:

```
        LD      HL,OP1  ; Anfangsadresse des Daten-
                        ; bereichs in das Daten-Adress-
                        ; Register laden
        LD      A,(HL)  ; ersten Operanden holen
        INC     HL      ; auf zweiten Operanden zeigen
        SUB     (HL)    ; Operation durchfuehren
        INC     HL      ; auf Speicherplatz fuer
                        ; Ergebnis zeigen
        LD      (HL),A  ; Ergebnis abspeichern
```

Während das erste Programmstück 11 Bytes belegt und von den fest vorgegebenen Speicheradressen abhängig ist, werden beim zweiten Programmstück nur 8 Bytes benötigt. Darüber hinaus kann es durch Abändern des ersten Befehls auf andere Datenbereiche mit gleicher Struktur angepaßt werden. Läßt man den ersten Befehl weg, so entsteht ein Programmstück, das auf alle Problemstellungen mit obiger Speicherstruktur angewandt werden kann. Die jeweilige Adresse des ersten Operanden muß dann allerdings vor Ausführung des Programmstücks in das Registerpaar HL gebracht werden.

Beim ersten Programmstück ist die Umstellung der Reihenfolge, in der die Operanden bearbeitet werden müssen, besonders ärgerlich. Insbesondere bei aufeinanderfolgenden Operationen, bei denen das Ergebnis einer Operation erster Operand der nächsten Operation ist, kann sich dies negativ bemerkbar machen.

Ein Nachteil der zweiten Lösung soll allerdings nicht verschwiegen werden: Sie ist von der oben gezeigten Speicherstruktur abhängig.

7.2 Bit-Manipulationen

Bei einer Bit-Manipulation werden ein oder mehrere Bits einer größeren Einheit (zum Beispiel eines Bytes oder Wortes) unabhängig von den übrigen Bits untersucht oder verändert. Bit-Manipulationen im weiteren Sinn sind auch solche Operationen, bei denen der neue Wert eines Bits von den alten oder neuen Werten benachbarter Bits abhängt (Rotations- und Schiebe-Operationen).

7.2.1 Untersuchung einzelner Bits

Gesetzt der Fall, im Registerpaar DE steht eine ganze Zahl in Zweierkomplement-Darstellung. Um herauszufinden, ob die Zahl negativ ist, muß nur das höchste Bit untersucht werden, da dieses das Vorzeichen angibt. Zur Durchführung des Tests kann man sich des Befehls BIT bedienen:

```
BIT    7,D      ; hoechstes Bit des Registerpaares DE testen
JP     NZ,NEGAT ; Bit 7 gesetzt, bedeutet negative Zahl
```

Die Sprungmarke NEGAT ist selbstverständlich im übrigen Programm geeignet zu definieren.

Der BIT-Befehl bringt das Einerkomplement des untersuchten Bits ins Z-Flag. Ein gesetztes Bit führt also zu einem zurückgesetzten Z-Flag, ein zurückgesetztes Bit zu einem gesetzten Z-Flag.

Natürlich hätte der Test auch mit einem der bekannten arithmetischen Befehle erfolgen können. Dazu müßte aber der Inhalt des Registers D beziehungsweise eine geeignete Prüfgröße erst in den Akkumulator gebracht werden, wodurch dessen Inhalt wiederum zerstört worden wäre. Der BIT-Befehl hat gegenüber den arithmetischen Befehlen den Vorteil, daß er auf jedes beliebige Bit eines der Register A, B, C, D, E, H, L angewandt werden kann, und das überdies der Inhalt des untersuchten Registers nicht verändert wird.

Der gleiche Test kann auch auf eine im Speicher stehende Größe vom Typ „Byte“ angewendet werden, wenn diese über das Registerpaar HL indirekt adressiert wird. Möchte man zum Beispiel prüfen, ob die betreffende Größe (als nichtnegative ganze Zahl interpretiert) gerade ist, so genügen die Befehle:

```
BIT    0,(HL)   ; letztes Bit der Zahl untersuchen
JP     Z,GERADE ; Bit 0 zurueckgesetzt, also Zahl gerade
```

Auch den schon bekannten Test, ob ein ASCII-codierter Buchstabe groß oder klein ist, kann man mit dem BIT-Befehl effizient durchführen. Dabei wird ausgenutzt, daß sich Groß- und Kleinbuchstaben genau durch den Wert von Bit 5 unterscheiden (das Zeichen soll diesmal im Register C stehen):

```
BIT    5,C      ; signifikantes Bit untersuchen
JP     NZ,KLEIN ; Bit 5 gesetzt, also Kleinbuchstabe
```

7.2.2 Setzen und Zurücksetzen einzelner Bits

Wie im vorhergehenden Beispiel gesehen, unterscheiden sich ASCII-codierte Großbuchstaben nur durch den Wert von Bit 5 von entsprechenden Kleinbuchstaben. Um aus einem Buchstaben den entsprechenden Kleinbuchstaben zu erhalten, muß lediglich das Bit 5 gesetzt werden. Dies kann man mit Hilfe des Befehls SET (set; setzen) tun, wobei der Buchstabe wieder im Register C erwartet wird:

```
SET     5,C     ; ASCII-codierten Buchstaben in
                ; Kleinbuchstaben umwandeln
```

Ebenso einfach erhält man einen Großbuchstaben durch Zurücksetzen von Bit 5 mittels des Befehls RES (reset; zurücksetzen):

```
RES     5,C     ; ASCII-codierten Buchstaben in
                ; Grossbuchstaben umwandeln
```

Wie beim BIT-Befehl kann auch beim SET-Befehl und beim RES-Befehl ein beliebiges 8-Bit-Hauptregister (außer dem Flag-Register) oder eine durch das Registerpaar HL indirekt adressierte Speicherzelle angegeben werden. Im folgenden Beispiel wird davon ausgegangen, daß eine ganze Zahl in Zweierkomplement-Darstellung im Register L steht. Für gerade Zahlen erhält man folgendermaßen die nächstgrößere ungerade Zahl:

```
SET     0,L     ; naechstgroessere ungerade Zahl erzeugen
```

7.2.3 Speichern von Zuständen

Man kommt des öfteren in die Situation, einen Test durchführen zu müssen, dessen Ergebnis wiederholt benötigt wird. Da die Flags durch viele Befehle verändert werden, muß der Zustand eines oder mehrerer Flags irgendwie gespeichert werden. Eine Methode ist, durch bestimmte Bits in einem bestimmten Register den Zustand der interessierenden Flags wiederzugeben. Es soll zum Beispiel der Zustand des S-Flags (Vorzeichen) in Bit 4 des Registers E gespeichert werden. Ein typisches Programmstück dazu könnte folgendermaßen aussehen:

```
; Zustand des S-Flags in Bit 4 des Registers E speichern

        SET     4,E             ; Zustand mit 1 vorbesetzen
        JP      M,FERTIG        ; Zustand ist richtig vorbesetzt
        RES     4,E             ; Zustand mit 0 besetzen
FERTIG: :                       ; Fortsetzung des Programmlaufs
        :

; Zustand des gespeicherten Flags benutzen
        BIT     4,E             ; Zustand testen
        JP      NZ,NEGAT        ; Vorzeichen-Flag war gesetzt
                                ; bei urspruenglichem Test
```

7.2.4 Maskieren

Ein Größe vom Typ „Byte“ kann man durch zwei Hex-Ziffern darstellen. Dabei entspricht die niederwertige Hex-Ziffer den Bits 0 bis 3, die höherwertige den Bits 4 bis 7. Man hat es also mit „Halb-Bytes“ zu tun; im Englischen nennt man ein Halb-Byte meist „Nibble“.

Ist man am niederwertigen Nibble eines Bytes interessiert, dann muß dieses vom höherwertigen Nibble isoliert werden. Dazu wird das Byte erst einmal in den Akkumulator gebracht. Dann werden die Bits 4 bis 7 gelöscht. Dies kann mit RES-Befehlen erfolgen:

```
        RES     4,A     ; Bit 4 loeschen
        RES     5,A     ; Bit 5 loeschen
        RES     6,A     ; Bit 6 loeschen
        RES     7,A     ; Bit 7 loeschen
```

Da mehrere Bits gleichzeitig verändert werden sollen, bedient man sich besser einer Technik, die „Maskieren“ genannt wird. Dabei verknüpft man den Inhalt des Akkumulators bitweise mit einem Datenwert vom Typ „Byte“. Dieser besitzt hier die spezielle Funktion einer „Maske“. Die Maske wird oft in binärer Schreibweise angegeben, um ihre Funktion klarer darzustellen. Als Verknüpfungen stehen die logischen Funktionen AND (Und-Verknüpfung, Konjunktion), OR (Oder-Verknüpfung, Disjunktion) und XOR (Entweder/Oder-Verknüpfung, exklusive Disjunktion) zur Verfügung. Die Verknüpfungstabelle für ein Bit sieht dabei so aus:

1. Operand	2. Operand	AND	OR	XOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Merke: Die logischen Befehle beziehen sich stets auf den Akkumulator. Er beinhaltet den ersten Operanden und anschließend das Ergebnis der Operation.

Das oben genannte Problem läßt sich nun durch den Befehl AND lösen:

```
        AND     00001111B       ; Bits 4 bis 7 wegmaskieren
```

Die Maske für einen AND-Befehl wird stets nach folgender Methode konstruiert: Soll ein bestimmtes Bit gelöscht werden, so steht in der Maske an dieser Stelle eine Null; soll der Wert des Bits erhalten bleiben, dann steht in der Maske eine Eins.

Daraus kann man schlußfolgern, daß sich mit Hilfe eines AND-Befehls und einer geeigneten Maske die Befehle RES bit,A und BIT bit,A ersetzen lassen. Um den RES-Befehl zu ersetzen, baut man eine Maske auf, in der alle Bits bis auf das zurückzusetzende Bit den Wert Eins erhalten. Für den BIT-Befehl ist eine Maske zu erzeugen, in der nur das zu testende Bit den Wert Eins hat. Das Ergebnis der logischen Operation ist damit genau dann Null, wenn das zu testende Bit den Wert Null hat; da auch genau in diesem Fall das Z-Flag gesetzt wird, ist die beschriebene Methode voll kompatibel zur Verwendung des BIT-Befehls.

Der Vorteil bei Substitution von RES- und BIT-Befehlen durch AND-Befehle besteht darin, daß der zweite Operand des AND-Befehls auch eines der 8-Bit-Hauptregister (außer dem Flag-Register) oder eine durch das Registerpaar HL indirekt adressierte Speicherzelle sein kann. Diese als Masken dienenden Datenwerte – und damit die aktuellen Bitnummern – können im Programm berechnet werden, während dagegen die Bitnummern in RES- und BIT-Befehlen bei der Assemblierung des Programm feststehen und nur durch Änderung des Programms selbst verändert werden können. (Finger weg von selbstmodifizierenden Programmen!!!)

Zu diesen Substitutionen noch ein Beispiel: Die bereits bekannte Umwandlung von ASCII-Buchstaben in Großbuchstaben läßt sich folgendermaßen realisieren, wenn sich der Buchstabe im Akkumulator befindet:

```
AND     11011111B       ; Buchstaben in Grossbuchstaben
                        ; umwandeln
```

Das Setzen mehrerer Bits gelingt mit dem Befehl OR, der ganz analog zum AND-Befehl verwendet wird. In der Maske steht dabei eine Eins, falls das entsprechende Bit gesetzt werden soll; eine Null kennzeichnet ein Bit, dessen Wert erhalten bleiben soll. Will man beispielsweise die Bits 4 und 5 des Akkumulators setzen, so genügt:

```
OR      00110000B       ; Bits 4 und 5 des Akkumulators
                        ; setzen
```

Bei genauer Betrachtung erkennt man, daß dies die Umwandlung binär codierter Dezimalziffern in ihre ASCII-Darstellung bewirkt.

Wie der AND-Befehl für RES- und BIT-Befehle, so kann auch der OR-Befehl als Ersatz für den SET-Befehl benutzt werden. In der Maske trägt dabei genau das Bit den Wert Eins, das gesetzt werden soll.

Die Umkehr (invertieren) von Bits geschieht mit dem Befehl XOR. Die Maske enthält für jedes Bit, das invertiert werden soll, eine Eins; für jedes Bit, dessen Wert erhalten bleiben soll, eine Null. Zum Beispiel invertiert man das Bit 0 des Akkumulators durch den Befehl

```
XOR     00000001B      ; Bit 0 des Akkumulators invertieren
```

und erhält damit die nächstgrößere ungerade bzw. nächstkleinere gerade ganze Zahl.

Aus der Verknüpfungstabelle der logischen Operationen ist zu erkennen, daß die Operationen symmetrisch bezüglich der Reihenfolge der Operanden sind. die Maske kann somit auch im Akkumulator stehen, während der zweite Operand den zu maskierenden Wert liefert (als direkten Datenwert, Registerinhalt oder Speicherinhalt). Das Ergebnis steht jedoch stets im Akkumulator.

Für die gleichzeitige Invertierung aller Bits des Akkumulators (Einerkomplement) gibt es noch den Befehl CPL (complement):

```
CPL             ; Einerkomplement des Akkumulators
```

Dieser Befehl bezieht sich ausschließlich auf den Akkumulator und hat deshalb keine Operanden.

Ein beliebter Programmiertrick besteht darin, statt des Befehls LD A,0 den Befehl XOR A zu verwenden, da dieser einen um ein Byte kürzeren Objekt-Code besitzt. Die beiden Befehle unterscheiden sich aber bezüglich der Flags in ihrer Wirkung. Während der LD-Befehl die Flags nicht verändert, werden bei allen logischen Befehlen die Flags folgendermaßen beeinflußt:

S	gesetzt, falls Bit 7 des Ergebnisses gesetzt
Z	gesetzt, falls Ergebnis Null ist
H	gesetzt für AND, zurückgesetzt für OR und XOR
P/V	gesetzt, falls Parität des Ergebnisses gerade ist
N	zurückgesetzt
C	zurückgesetzt

Die Parität einer Bitfolge ist die Anzahl der darin vorkommenden Bits mit dem Wert 1. Die Parität kann man zur Prüfung von übertragenen Daten ausnutzen (mehr dazu in einem späteren Abschnitt). Aus der Doppelfunktion des P/V-Flags als Paritäts-Flag und als Überlauf-Flag erklären sich auch die merkwürdigen Bezeichnungen PO (parity odd) und PE (parity even), wie sie bereits bei den Sprüngen verwendet wurden.

Ein weiterer Trick besteht darin, statt des Befehls CP 0 einen der Befehle AND A oder OR A zu benutzen. Das Z-Flag (Null) und das S-Flag (Vorzeichen) werden dabei wie durch den CP-Befehl gesetzt. Der Objekt-Code ist wiederum ein Byte kürzer; außerdem läßt sich die Parität am P/V-Flag erkennen.

Da bei allen logischen Befehlen das C-Flag (Übertrag) zurückgesetzt wird, verwendet man statt der bereits bekannten Befehlsfolge

```
SCF             ; C-Flag setzen
CCF             ; C-Flag invertieren
```

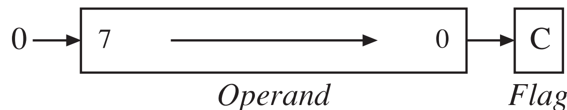
meist einen der Befehle AND A oder OR A, da diese in Bezug auf das C-Flag genauso wirken, aber einen kürzeren Objekt-Code ergeben. Der Inhalt des Akkumulators bleibt dabei erhalten, nicht jedoch die Zustände der übrigen Flags.

Die beschriebenen Programmiertricks sollte man mit Vorsicht verwenden, da die Befehle in ihrer Wirkung eben doch nicht genau übereinstimmen. Am besten schreibt

man erst die längeren, aber klareren Befehle ins Programm, um in einem späteren Optimierungslauf zu prüfen, ob sie gefahrlos durch die kürzeren ersetzt werden können. Auf jeden Fall sollten solche Optimierungen genau dokumentiert werden!

7.2.5 Verschieben und Rotieren

Als nächstes soll die Aufgabe gelöst werden, den höherwertigen Nibble eines Bytes zu isolieren. Durch Maskieren kann zwar der niederwertige Nibble entfernt werden, die entstehende Größe vom Typ „Byte" stellt jedoch – als ganze Zahl interpretiert – nicht die Binärcodierung der höherwertigen Hex-Ziffer dar. Das gewünschte Ziel kann man jedoch erreichen, indem der Inhalt des Akkumulators viermal nach rechts verschoben und dabei links jeweils eine Null nachgezogen wird (Bit 7 soll dabei ganz links stehen, Bit 0 ganz rechts). Eine derartige Operation heißt „logische Rechts-Verschiebung"; sie wird durch den Befehl SRL (shift right logically) realisiert. Die logische Rechts-Verschiebung kann man folgendermaßen darstellen:



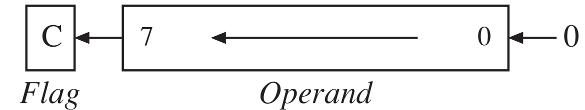
Logische Rechts-Verschiebung (SRL-Befehl)

Die geforderte Aufgabe wird durch folgendes Programmstück gelöst:

```
SRL     A       ; Inhalt des Akkumulators um ein Bit
                ; logisch rechts-verschieben
SRL     A       ; Inhalt des Akkumulators um ein Bit
                ; logisch rechts-verschieben
SRL     A       ; Inhalt des Akkumulators um ein Bit
                ; logisch rechts-verschieben
SRL     A       ; Inhalt des Akkumulators um ein Bit
                ; logisch rechts-verschieben
```

Als Nebeneffekt des SRL-Befehls wird der jeweilige Inhalt von Bit 0 ins C-Flag übertragen. (Bei diesem Beispiel befindet sich also letztendlich der ehemalige Wert des Bits 4 im C-Flag.) Als Operand des SRL-Befehls ist jedes der Register A, B, C, D, E, H, L oder eine durch das Registerpaar HL indirekt adressierte Speicherzelle zulässig.

Soll umgekehrt eine binärcodierte Hex-Ziffer in den höherwertigen Nibble eines Bytes gebracht werden, so kann dazu der Befehl SLA (shift left arithmetically; arithmetische Links-Verschiebung) benutzt werden, der um ein Bit nach links verschiebt, rechts eine Null nachzieht und den alten Inhalt von Bit 7 ins C-Flag überträgt. Im Bild:



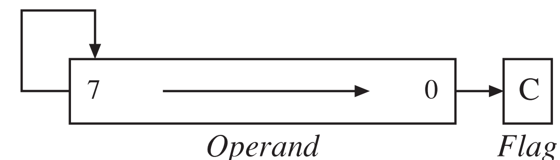
Arithmetische Links-Verschiebung (SLA-Befehl)

Das entsprechende Programmstück würde also lauten (diesmal mit Register C):

```
SLA     C       ; Inhalt des Registers C um ein Bit
                ; arithmetisch links-verschieben
SLA     C       ; Inhalt des Registers C um ein Bit
                ; arithmetisch links-verschieben
SLA     C       ; Inhalt des Registers C um ein Bit
                ; arithmetisch links-verschieben
SLA     C       ; Inhalt des Registers C um ein Bit
                ; arithmetisch links-verschieben
```

Die Befehle SLA A und ADD A,A unterscheiden sich nur bezüglich ihrer Wirkung auf das P/V-Flag (Überlauf) und das H-Flag (Hilfs-Überlauf). Besteht kein Interesse am Zustand dieser Flags, kann der kürzere und schnellere ADD-Befehl verwendet werden.

Die „arithmetische Rechts-Verschiebung" unterscheidet sich von der logischen Rechts-Verschiebung dadurch, daß der Zustand von Bit 7 stets erhalten bleibt. Schematisch dargestellt:



Arithmetische Rechts-Verschiebung (SRA-Befehl)

Die arithmetische Rechts-Verschiebung wird durch den Befehl SRA (shift right arithmetically) realisiert, hier am Beispiel einer indirekt adressierten Speicherstelle:

```
SRA     (HL)    ; Inhalt der durch das Registerpaar HL
                ; adressierten Speicherzelle um ein Bit
                ; arithmetisch rechts-verschieben
```

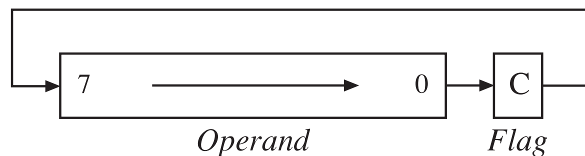
Das Zusammensetzen eines Bytes aus den beiden Nibbles erfolgt durch den OR-Befehl. Bei diesem Beispiel befindet sich ein Nibble im Akkumulator und der andere im Register C:

```
OR      C       ; zwei Nibbles zu Byte zusammensetzen
```

Natürlich hätte auch der Befehl ADD A,C dazu verwendet werden können. Der logische Befehl unterscheidet sich aber bezüglich der Wirkung auf die Flags; außerdem ist das Zusammensetzen dem Charakter nach keine arithmetische, sondern eine logische Operation.

Mit den beschriebenen Techniken können nun beliebige Ausschnitte aus einem Byte isoliert beziehungsweise ein Byte aus verschiedenen Abschnitten zusammengesetzt werden. Soll dasselbe mit einem Wort durchgeführt werden, so muß beim Verschieben zwischen den beiden beteiligten 8-Bit-Registern ein Bit übertragen werden. Mit den Verschiebe-Befehlen ist das nicht möglich. Daher bedient man sich der sogenannten Rotations-Befehle, wobei der Austausch eines Bits über das C-Flag erfolgt.

Der Befehl RR (rotate right; Rechts-Rotation) verschiebt wie der SRA-Befehl und der SRL-Befehl um ein Bit nach rechts, Bit 7 erhält dabei aber den alten Wert des C-Flags. Die Rechts-Rotation sieht damit folgendermaßen aus:



Rechts-Rotation (RR-Befehl)

Eine logische Rechts-Verschiebung des Registerpaares DE würde damit so aussehen:

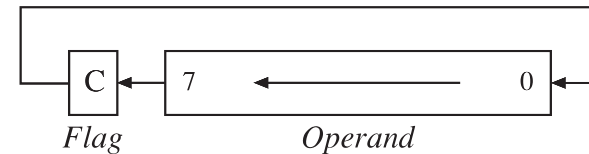
```
SRL     D       ; Register D logisch rechts-verschieben
RR      E       ; Register E rechts-verschieben, dabei
                ; Bit 0 von Register D uebernehmen
```

Genau betrachtet rotiert der RR-Befehl eigentlich nicht das betreffende Register, sondern eine Verbindung des C-Flags (als 1-Bit-Register interpretiert) mit dem Register. Im obigen Beispiel werden also (D_7, D_6, ..., D_0, C-Flag) rotiert. Entsprechendes gilt für die anderen Register.

Eine arithmetische Rechts-Verschiebung des Registerpaares HL läßt sich so realisieren:

```
SRA     H       ; Register H arithmetisch rechts-verschieben
RR      L       ; Register L rechts-verschieben, dabei
                ; Bit 0 vom Register H uebernehmen
```

Die Links-Verschiebung eines Registerpaares erhält man mit Hilfe des Befehls RL (rotate left; Links-Rotation). Dieser bewirkt eine Rotation der Verbindung aus C-Flag und Register linksherum. Bildlich dargestellt:



Links-Rotation (RL-Befehl)

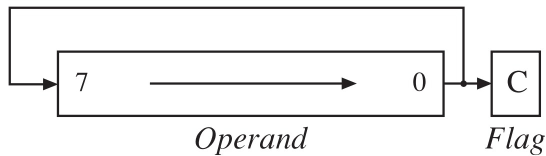
Zum Beispiel für das Registerpaar BC:

```
SLA     C       ; Register C links-verschieben
RL      B       ; Register B links-verschieben, dabei
                ; Bit 7 von Register C uebernehmen
```

Es ist stets darauf zu achten, in welcher Reihenfolge die Register bearbeitet werden und wie Verschieben und Rotieren ineinander greift.

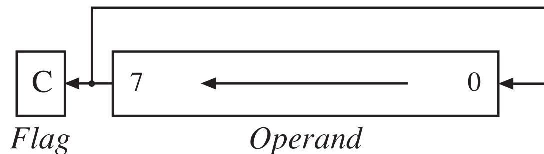
Die Links-Verschiebung des Registerpaares HL führt man meist mit dem kürzeren und schnelleren Befehl ADD HL,HL durch, der aber die Flags anders beeinflußt.

Soll statt der Verbindung aus Register und C-Flag nur das Register allein nach rechts rotiert werden, so benutzt man den Befehl RRC (rotate right circular; zirkuläre Rechts-Rotation). Der alte Wert von Bit 0 wird dabei zusätzlich ins C-Flag kopiert:



Zirkuläre Rechts-Rotation (RRC-Befehl)

Entsprechend gibt es für die zirkuläre Links-Rotation den Befehl RLC (rotate left circular), der den alten Wert von Bit 7 ins C-Flag bringt:



Zirkuläre Links-Rotation (RLC-Befehl)

Wie bei den Verschiebe-Befehlen kann auch bei den Rotations-Befehlen statt eines 8-Bit-Hauptregisters (außer Flag-Register) eine durch das Registerpaar HL indirekt adressierte Speicherzelle angegeben werden.

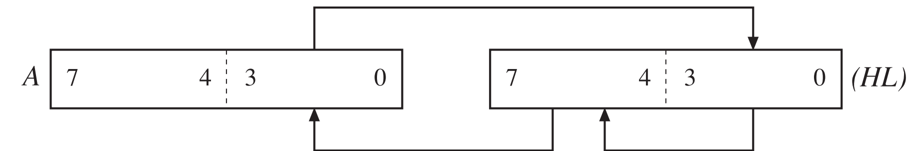
Zum Rotieren des Akkumulators gibt es vier weitere Befehle, die im Prinzip durch die bereits gezeigten Befehle realisiert werden könnten, die sich aber durch einen kürzeren Objekt-Code und eine kürzere Ausführungszeit sowie durch eine andere Behandlung der Flags von diesen unterscheiden. Es handelt sich dabei um:

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

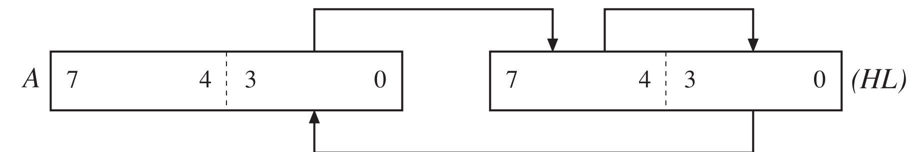
Mit Hilfe dieser Befehle kann das eingangs gezeigte Beispiel – das Isolieren und Rechts-Verschieben des höherwertigen Nibbles eines Bytes – noch effizienter gelöst werden:

```
AND    11110000B  ; niederwertigen Nibble ausblenden
RRCA              ; hoeherwertigen Nibble rechts-verschieben
RRCA              ; hoeherwertigen Nibble rechts-verschieben
RRCA              ; hoeherwertigen Nibble rechts-verschieben
RRCA              ; hoeherwertigen Nibble rechts-verschieben
```

Weil das Arbeiten mit Nibbles recht häufig vorkommt (insbesondere wegen der BCD-Codierung), besitzt der Z80 zwei spezielle Befehle für das Rotieren von Nibbles. Dies sind die beiden Befehle RLD (rotate left digit; zirkuläre Links-Rotation von Nibbles) und RRD (rotate right digit; zirkuläre Rechts-Rotation von Nibbles). Beide stellen eine zirkuläre Rotation um vier Bits auf den unteren Bits des Akkumulators A_3, A_2, A_1, A_0 und der durch das Registerpaar HL indirekt adressierten Speicherzelle (HL) dar:



Zirkuläre Links-Rotation von Nibbles (RLD-Befehl)



Zirkuläre Rechts-Rotation von Nibbles (RRD-Befehl)

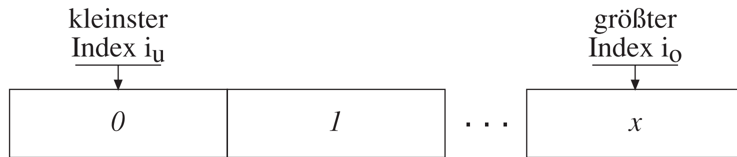
Von diesen beiden Befehlen wird beim Rechnen mit ganzen Zahlen und mit Gleitpunktzahlen intensiv Gebrauch gemacht.

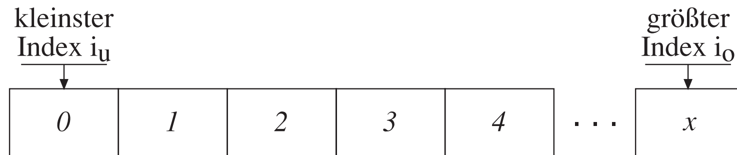
7.3 Felder

Ein (eindimensionales) Feld ist eine fortlaufend indizierte Menge von Datenwerten gleichen Typs (z.B. vom Typ Byte, Wort, Bit, Nibble). Die Indizies i sind ganze Zahlen. Es gibt einen kleinsten Index i_u und einen größten Index i_o. Jedem Index im Bereich i_u bis i_o ist ein Feldelement zugeordnet (siehe auch Bild 9).

7.3.1 Implementierung von Feldern

Nachfolgend wird davon ausgegangen, daß alle Feldelemente lückenlos im Speicher untergebracht sind (was in der Praxis fast immer stimmt). Anfangs sollen Felder





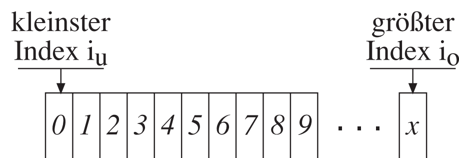


Bild 9: Felder mit Elementen verschiedener Größe

betrachtet werden, deren Elemente jeweils ein oder mehrere Bytes belegen (eine glatt durch 8 teilbare Anzahl von Bits). Ist L die Länge eines einzelnen Feldelements, so benötigt das ganze Feld $(i_o - i_u + 1) \cdot L$ Bytes Speicherplatz. Für ein uninitialisiertes Feld kann dieser Bereich durch die Pseudo-Operation DEFS bereitgestellt werden, zum Beispiel für ein Feld von Worten mit dem Indexbereich 0 bis 4:

```
IU      EQU     0                  ; kleinster Index
IO      EQU     4                  ; groesster Index
LAENGE  EQU     2                  ; Laenge eines Elements
WFELD:  DEFS    (IO-IU+1)*LAENGE   ; uninitialisiertes Feld
                                   ; von Worten reservieren
```

Soll ein initialisiertes Feld angelegt werden, so schreibt man die Elemente mittels der Pseudo-Operationen DEFB und DEFW hintereinander:

```
WFELD:  DEFW    19738   ; erstes Feldelement
        DEFW    7895    ; zweites Feldelement
        DEFW    -3396   ; drittes Feldelement
        DEFW    12987   ; viertes Feldelement
        DEFW    24745   ; fuenftes Feldelement
```

Sind die Feldelemente weder Bytes noch Worte, so setzt sich die Vereinbarung eines Elements aus mehreren Pseudo-Operationen zusammen, also zum Beispiel für ein 4-elementiges Feld, dessen Elemente je 3 Bytes belegen:

```
FELD:   DEFB    22H     ; erstes Feldelement
        DEFB    45H
        DEFB    8AH
        DEFB    0AH     ; zweites Feldelement
        DEFB    19H
        DEFB    0C7H
        DEFB    31H     ; drittes Feldelement
        DEFB    86H
        DEFB    73H
        DEFB    00H     ; viertes Feldelement
        DEFB    1FH
        DEFB    27H
```

Für Zeichen und Zeichenketten verwendet man die Pseudo-Operationen DEFB bzw. DEFM.

Belegen die Elemente eines Feldes je L Bits und ist L nicht ohne Rest durch 8 teilbar, so gibt es zwei Möglichkeiten: Entweder verschenkt man pro Element einige Bits und teilt jedem Element so viele Byte zu, wie zur Aufnahme von L Bits benötigt werden; die Darstellung des Feldes geschieht dann wie oben beschrieben. Oder man faßt den Speicher als eine Folge von Bits auf und teilt jedem Element genau L Bits Speicherplatz zu; die Grenzen der Feldelemente fallen dann nicht immer mit den Grenzen von Bytes zusammen.

Im folgenden soll der zweite Fall betrachtet werden. Als gesamten Speicherplatz des Feldes wählt man die kleinste Anzahl von Bytes, die zur Aufnahme von $(i_o - i_u + 1) \cdot L$ Bits notwendig sind. Beispielsweise vereinbart man ein uninitialisiertes Feld von 17 Nibbles mittels

```
NFELD:  DEFS    9       ; Feld mit 17 Nibbles reservieren
```

und ein Bit-Feld mit 116 Elementen mittels

```
BFELD:  DEFS    15      ; Feld mit 116 Bits reservieren
```

Initialisierte Felder dieser Art werden wieder mit DEFB und DEFW vereinbart, also zum Beispiel für das Nibble-Feld (4, 7, 9, A, F):

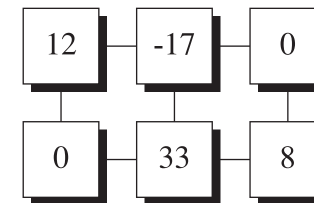
```
NFELD:  DEFB    74H     ; initialisiertes Feld mit 5
        DEFB    0A9H    ; Nibbles vereinbaren
        DEFB    0FH
```

Hierbei ist insbesondere darauf zu achten, wie die einzelnen Elemente als höherwertige und niederwertige Anteile von Bytes plaziert werden!

Bisher waren lediglich eindimensionale Felder Gegenstand der Betrachtungen. Es gibt jedoch auch mehrdimensionale Felder, zum Beispiel stellt man eine Matrix durch ein zweidimensionales Feld dar. Mehrdimensionale Felder werden durch Paare (i, j), Tripel (i, j, k) usw. indiziert. Selbstverständlich sind Felder mit größeren Dimensionen realisierbar, doch wir beschränken uns auf den für die Praxis relevanten Spezialfall mit zwei Dimensionen.

Stellt man sich ein zweidimensionales Feld als Darstellung einer Matrix von Elementen vor, so gibt der eine Index eine Zeile, der andere eine Spalte der Matrix an. Damit gibt es prinzipiell zwei Möglichkeiten, wie das Feld organisiert sein kann: Entweder stehen alle Elemente einer Zeile direkt hintereinander (zeilenorientiertes Feld) oder alle Elemente einer Spalte (spaltenorientiertes Feld).

Man kann das zweidimensionale Feld im ersten Fall als eindimensionales Feld von Zeilen, im zweiten Fall als eindimensionales Feld von Spalten interpretieren, wobei Zeilen und Spalten wiederum eindimensionale Felder von Elementen darstellen. Um dies zu verdeutlichen, hier ein kleines Beispiel. Die Matrix soll folgendermaßen aussehen:



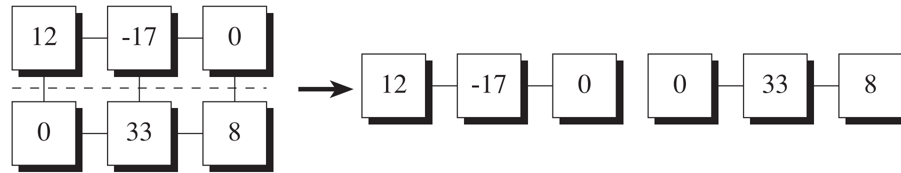
Ein zeilenorientiertes Byte-Feld wird vereinbart durch

```
ZFELD:  DEFB    12      ; erste Zeile
        DEFB    -17
        DEFB    0
        DEFB    0       ; zweite Zeile
        DEFB    33
        DEFB    8
```

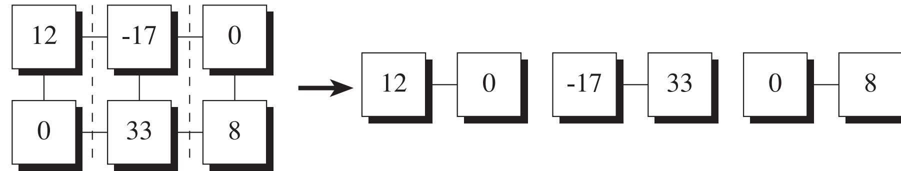
und ein spaltenorientiertes Byte-Feld durch

```
SFELD:  DEFB    12      ; erste Spalte
        DEFB    0
        DEFB    -17     ; zweite Spalte
        DEFB    33
        DEFB    0       ; dritte Spalte
        DEFB    8
```

Für andere Basistypen (Wort, Bit, Nibble) oder größere Dimensionen funktioniert dies ganz analog.



zeilenorientiertes Byte-Feld



spaltenorientiertes Byte-Feld

Die Dimension eines Feldes, die Unter- und Obergrenzen der Indexbereiche und der Typ der Elemente gehören eigentlich untrennbar von den Elementen selbst (bzw. den Speicherplätzen dafür) zur Definition eines Feldes. Diese Informationen sind meist im Programm versteckt: als Konstanten, die im Algorithmus auftauchen, wenn nur ein Feld mit unveränderlichen Abmessungen bearbeitet wird; als Registerinhalte, die zur Laufzeit des Programms berechnet werden, wenn sich die Indexgrenzen während des Programmlaufs verändern. Wenn man Programme schreiben will, die auf Felder beliebiger Größe arbeiten sollen, so faßt man die Strukturinformationen zu einem sogenannten Deskriptor zusammen, der vor den Elementen plaziert wird. Im Falle der Matrix ZFELD würde dies beispielsweise so aussehen:

```
ZFELD:
TYP:    DEFB    1       ; jedes Element belegt 1 Byte
DIM:    DEFB    2       ; Dimension d = 2
IU1:    DEFB    1       ; Zeilenindex laeuft
IO1:    DEFB    2       ; von 1 bis 2
IU2:    DEFB    1       ; Spaltenindex laeuft
IO2:    DEFB    3       ; von 1 bis 3
ELEM:   DEFB    12      ; erste Zeile
        DEFB    -17
        DEFB    0
        DEFB    0       ; zweite Zeile
        DEFB    33
        DEFB    8
```

Um die Deskriptoreinträge richtig zu interpretieren, muß natürlich die Struktur und die Bedeutung des Deskriptors bekannt sein (z.B. muß bekannt sein, daß als erstes die Länge eine Elements in Bytes angegeben wird). Außerdem muß für obiges Beispiel die Vereinbarung getroffen sein, daß die Matrix zeilenorientiert ist.

7.3.2 Adressierung einzelner Feldelemente

Zunächst sollen wieder eindimensionale Felder behandelt werden, deren Elemente je ein ganzzahliges Vielfaches von 8 Bit belegen. Um ein einzelnes Element mit Index zu bearbeiten, benötigt man die Anfangsadresse des Elements (grundsätzlich geht es auch mit der Endadresse).

Geht man von einer Funktion a aus, die den Speicher abbildet und zu jedem Index i die Anfangsadresse $a(i)$ des i-ten Feldelements liefert, dann läßt sich aus der im vorhergehenden Abschnitt vereinbarten Speichertechnik schlußfolgern: $a(i) = a(0) + i \cdot L$, wobei L wiederum die Länge eines Elements ist.

Meist ist jedoch die Größe $a(0)$ nicht bekannt, sondern die Anfangsadresse $a_u = a(i_u)$ des Feldes. Damit ergibt sich $a(i) = a_u + (i - i_u) \cdot L$.

Vom Index i zur Adresse $a(i)$ gelangt man nun in drei Schritten:

1. Berechnung des relativen Index $i - i_u$ (entfällt im Spezialfall $i_u = 0$).
2. Berechnung der relativen Adresse $(i - i_u) \cdot L$ (entfällt im Spezialfall $L = 1$, also bei Byte-Feldern).
3. Berechnung der absoluten Adresse $a_u + (i - i_u) \cdot L$.

Wurde die Adresse $a(i)$ schließlich ermittelt, so kann das Feldelement mit den bekannten Methoden der indirekten Adressierung bearbeitet werden.

Als Zeiger auf ein einzelnes Feldelement eignet sich besonders das Registerpaar HL, bedingt auch die Registerpaare BC oder DE. Im folgenden wird stets das Registerpaar HL zur Datenadressierung benutzt. Des weiteren wird davon ausgegangen, daß der Index i als vorzeichenlose ganze Zahl im Akkumulator steht. Anpassungen der im weiteren folgenden Programme an Indizies in Zweierkomplement-Darstellung und/oder Indizies in einem Registerpaar sind leicht zu bewerkstelligen. Die drei Schritte lauten dann schematisch:

$A = A - i_u$
$HL = A \cdot L$
$HL = a_u + HL$

Sind i_u oder L als Konstanten gegeben, so kann man sie direkt in den Algorithmen einbauen. Ansonsten taucht an ihrer Stelle ein geeignetes Register auf. Für den letzten Schritt muß ein weiteres Registerpaar benutzt werden, das die Adresse a_u aufnimmt. Ist L eine Konstante, so läßt sich der zweite Schritt als Sequenz ausführen; anderenfalls benutzt man eine Multiplikationsschleife.

Beginnen wir mit einem Byte-Feld ($L = 1$), wobei i_u und a_u als Konstanten vorgegeben sind:

```
IU      EQU     12      ; kleinster Index
AU      EQU     4200H   ; Anfangsadresse des Feldes
        SUB     IU      ; relativen Index berechnen
        LD      L,A     ; relativer Index = relative Adresse
        LD      H,0     ; wegen L = 1
        LD      DE,AU   ; Anfangsadresse des Feldes
        ADD     HL,DE   ; Adresse des Feldelements
```

Nun dasselbe für ein Wort-Feld:

```
IU      EQU     5       ; kleinster Index
AU      EQU     5E00H   ; Anfangsadresse des Feldes
        SUB     IU      ; relativen Index berechnen
        LD      L,A     ; relativen Index ins
        LD      H,0     ; Registerpaar HL bringen
        ADD     HL,HL   ; relative Adresse berechnen
        LD      DE,AU   ; Anfangsadresse des Feldes
        ADD     HL,DE   ; Adresse des Feldelements
```

Jetzt noch eine Routine für die Indizierung, wenn die Kenngrößen des Feldes in Registern übergeben werden. Dabei wird der Index i im Akkumulator, der kleinste Index i_u im Register C, die Länge L im Register E und die Anfangsadresse a_u des Feldes im Registerpaar HL erwartet. Der Wert a_u muß temporär im Speicher abgelegt werden, da alle Registerpaare für die Multiplikation benötigt werden.

```
; Berechnung der Anfangsadresse eines Feldelements
; A = Index i
; C = kleinster Index iu
; E = Laenge L eines Feldelements
; HL = Anfangsadresse des Feldes

ADRESS: LD      (BASIS),HL      ; Registerinhalt temporaer sichern

; relativen Index berechnen

        SUB     C       ; relativer Index jetzt im Akkumulator

; Multiplikation (i-iu)*L durchfuehren, das heisst
; relative Adresse des Feldelements berechnen

        LD      HL,0    ; Registerpaar loeschen
        LD      D,H     ; Multiplikand zu 16-Bit-Groesse erweitern
        LD      B,8     ; Schleifenzaehler mit Laenge des
                        ; Multiplikators (in Bits) besetzen
MULTI:  ADD     HL,HL   ; Inhalt verdoppeln
        RLCA            ; hoechstes Bit des Multiplikators ins
                        ; C-Flag bringen, gleichzeitig
                        ; Multiplikator links-rotieren
        JP      NC,NULL ; keine Addition erforderlich,
                        ; wenn anstehendes Bit des
                        ; Multiplikators 0 ist
        ADD     HL,DE   ; Multiplikand addieren
NULL:   DJNZ    MULTI   ; naechstes Bit des Multiplikators
                        ; verarbeiten

; Anfangsadresse des Feldelements berechnen

        LD      DE,(BASIS)      ; Anfangsadresse des Feldes
        ADD     HL,DE           ; Adresse des Elements berechnen

; Datenbereich

BASIS:  DEFS    2               ; Hilfs-Speicherplatz
```

Hier noch ein Tip: Bei eigenen Programmen sollte man sich die Zeit nehmen, sie so sorgfältig zu dokumentieren – es lohnt sich wirklich!

Bisher wurde stets angenommen, daß der Index i die Bedingung $i_u \leq i \leq i_o$ erfüllt. Durch Programmierfehler, fehlerhafte Daten oder unsinnige Benutzereingaben kann es aber durchaus vorkommen, daß i außerhalb des erlaubten Bereiches liegt. Daher sollten dynamische Kontrollen eingebaut werden, welche die Einhaltung der Indexgrenzen überwachen. Die Prüfung besteht aus zwei Teilen:

1. Prüfung, ob $i \geq i_u$ (entfällt für $i_u = 0$).
2. Prüfung, ob $i \leq i_o$ (entfällt für $i_o = 255$ bei 8-Bit-Indizies).

Ersetzt man den SUB-Befehl zu Beginn des eben gezeigten Programms durch folgendes Programmstück, so kann nichts mehr schiefgehen (i_o wird im Register D übergeben):

```
DYNKON: CP      D               ; i mit io vergleichen
        JP      C,OK            ; i < io, Index korrekt
        JP      NZ,UEBERL       ; Indexueberlauf i > io
OK:     SUB     C               ; relativen Index berechnen
        JP      C,UNTERL        ; Indexunterlauf i < iu
```

Manchmal wird statt des größten zulässigen Index die Adresse $a_o = a_u + (i_o - i_u + 1) \cdot L$ angegeben, die das erste Byte hinter dem Speicherbereich des Feldes angibt. Einen Indexüberlauf kann dann nach Berechnung der Adresse des Feldelements durch folgendes Programmstück prüfen, wobei ins Registerpaar DE zunächst die Adresse a_o geladen werden muß:

```
        EX      DE,HL           ; Operanden vertauschen zwecks
                                ; einfacherem Vergleich
        SCF                     ; C-Flag setzen, ebenfalls zwecks
                                ; einfacherem Vergleich
        SBC     HL,DE           ; ao > Adresse des Feldelements?
        JP      C,UEBERL        ; Indexueberlauf
        EX      DE,HL           ; Operanden zuruecktauschen
```

Diese Vorgehensweise wird auf jeden Fall notwendig, wenn statt des Index i bereits der relative Index $i - i_u$ übergeben wird und die Untergrenze i_u nicht bekannt ist. (Bei vorzeichenlosen ganzzahligen Indizies kann dann kein Unterlauf eintreten.)

Der neue Befehl EX (exchange; tauschen) tauscht die Inhalte der Registerpaare DE und HL wechselseitig aus. Für Adressierungsvorgänge wird er häufig verwendet.

Nun zu den mehrdimensionalen Feldern. Ein zweidimensionales Feld mit den Indexgrenzen $i_{u_1} - i_{o_1}$ und $i_{u_2} - i_{o_2}$ kann man sich im Speicher wie ein eindimensionales Feld mit den Indexgrenzen $i_u = 0$, $i_o = (i_{o_1} - i_{u_1} + 1) \cdot (i_{o_2} - i_{u_2} + 1) - 1$ vorstellen. Ist das Feld zeilenorientiert und gibt der erste Index die Zeile, der zweite die Spalte an, so entspricht der Adresse $a(i_1, i_2)$ eines Feldes in der Zeile i_1 und der Spalte i_2 im eindimensionalen Feld das Element mit der Adresse $a(i)$ mit $i = (i_1 - i_{u_1}) \cdot (i_{o_2} - i_{u_2} + 1) + (i_2 - i_{u_2})$. Ist der korrespondierende Index des eindimensionalen Feldes erst einmal bestimmt, so können die dafür entwickelten Methoden angewandt werden.

Für das weitere Verständnis ist es sehr wichtig, daß diese Zusammenhänge genau erkannt und begriffen wurden! Vorher sollte nicht weitergelesen werden.

Diese Abbildung eines d-dimensionalen Feldes auf ein eindimensionales Feld funktioniert natürlich nicht nur für Byte-Felder, sondern ebenso für beliebige Nibble- und Bit-Felder. Daher werden nur noch eindimensionale Felder betrachtet (ohne Deskriptor; Indexgrenzüberwachung entfällt ebenfalls aus Platzgründen).

Als nächstes sollen die Nibble-Felder genauer betrachtet werden. Da jedes Byte zwei Nibble enthält, kann das i-te Element eines Nibble-Feldes durch die Adresse $a(i)$ des Bytes, in dem der Nibble liegt, und die Nibble-Adresse $n(i)$ gekennzeichnet werden. Der niederwertige Nibble (Bit 0 – 3) besitzt die Nibble-Adresse 0, der höherwertige Nibble (Bit 4 – 7) die Nibble-Adresse 1. Es gilt damit:

$$\begin{aligned} 2 \cdot a(i) + n(i) &= 2 \cdot a(0) + n(0) + i \\ &= 2 \cdot a_u + n_u + (i - i_u) \end{aligned}$$

wobei $a_u = a(i_u)$ die Adresse des ersten Feldelements ist und $n_u = n(i_u)$ dessen Nibble-Adresse. Aus diesen Beziehungen läßt sich nun errechnen:

$$a(i) = a_u + (n_u + (i - i_u))/2$$

und

$$n(i) = (n_u + (i - i_u)) \bmod 2$$

Als Ergebnis der Division ist dabei nur der ganzzahlige Anteil (ohne Rest) zu nehmen. Die Berechnung der Adresse $a(i)$ und der Nibble-Adresse $n(i)$ geht nun so vor sich:

1. Berechnung des relativen Index $i - i_u$.
2. Berechnung des korrigierten relativen Index $n_u + i - i_u$.
3. Gleichzeitige Berechnung von $(n_u + (i - i_u))/2$ und $(n_u + (i - i_u)) \bmod 2$.
4. Berechnung der Adresse $a_u + (n_u + (i - i_u))/2$.

Berechnung von ganzzahligem Anteil und Rest der Division durch 2 läßt sich durch eine Rechts-Verschiebung realisieren. Dabei fällt der ganzzahlige Anteil im verschobenen Register an, der Rest im C-Flag. Der Wert des Flags muß gesichert werden, damit er durch Schritt 4 nicht zerstört wird.

Nachfolgend ein Programmstück für die Realisierung der Schritte 1 bis 3 (Schritt 4 dürfte mittlerweile wohl klar sein). Es wird davon ausgegangen, daß sich alle benötigten Größen in Registern befinden, und zwar

i	Akkumulator
i_u	Register B
n_u	Register C

Am Ende der Routine soll die Relativ-Adresse $(n_u + (i - i_u))/2$ im Akkumulator und die Nibble-Adresse $n(i)$ im Register D stehen.

Weiterhin soll gelten, daß $i - i_u < 256$ ist (sonst wird die Arbeit ein wenig umständlicher). Trotzdem kann natürlich bei der Addition $n_u + (i - i_u)$ ein Übertrag anfallen. Damit beim Rechts-Verschieben das Ergebnis nicht verfälscht wird, gelangt das C-Flag dabei einfach in Bit 7 – der Akkumulator wird nach rechts rotiert! Das ins C-Flag herausgeschobene Bit 0 wird dann durch eine Links-Rotation des Registers D auf dessen Bit 0. Damit steht der Rest richtig im Register D. Das Programmstück lautet also:

```
SUB     B        ; relativen Index berechnen
ADD     A,C      ; korrigierten Index berechnen
RRA              ; Division durch 2 und Restbildung
                 ; im C-Flag
LD      D,0      ; vorbereiten zur Speicherung
                 ; der Nibble-Adresse
RL      D        ; Nibble-Adresse abspeichern
```

Das Ganze funktioniert auch noch, wenn i und/oder i_u in Zweierkomplement-Darstellung vorliegen, da ja die Differenz $i - i_u$ laut Definition des Feldes nicht negativ sein kann.

Der Zugriff auf das i-te Element eines Nibble-Feldes geschieht nun folgendermaßen: Soll der Nibble gelesen werden, so wird mittels indirekter Adressierung das ihn enthaltende Byte mit der Adresse $a(i)$ geholt. Anschließend wird der Nibble mit den gelernten Techniken isoliert, wobei je nach Wert der Nibble-Adresse $n(i)$ der niederwertige oder höherwertige Nibble isoliert werden muß.

Soll der Nibble dagegen beschrieben werden, so muß darauf geachtet werden, daß der andere Nibble des Bytes nicht zerstört wird. Dazu holt man am besten erst einmal das ganze Byte mit der Adresse $a(i)$ in ein Register, maskiert den zu ersetzenden Nibble aus, maskiert den neuen Wert des Nibbles wieder ein und bringt das Byte in den Speicher zurück. Eine solche Schreiboperation wird im folgenden Programmstück gezeigt, wobei die Adresse $a(i)$ im Registerpaar HL, die Nibble-Adresse $n(i)$ im Register D und der einzufügende Nibble im Register E erwartet wird:

```
SNIBB:  LD      A,(HL)   ; beide alten Nibble holen
        DEC     D        ; Nibble-Adresse testen
        JP      Z,EINS   ; Sprung, wenn n(i) = 1
NULL:   AND     OFOH     ; hinteren Nibble ausmaskieren
        OR      E        ; hinteren Nibble einfuegen
        JP      WEITER   ; weiter an gemeinsamer
                         ; Fortsetzungsstelle
EINS:   AND     OFH      ; vorderen Nibble ausmaskieren
        SLA     E        ; neuen Nibble nach vorn bringen
        SLA     E
        SLA     E
        SLA     E
        OR      E        ; vorderen Nibble einfuegen
WEITER: LD      (HL),A   ; beide Nibbles zurueckschreiben
```

Abschließend sollen nun die Bit-Felder (oder Bitketten) betrachtet werden. Ein einzelnes Bit mit dem Index i wird durch die Adresse $a(i)$ des Bytes, in dem das Bit enthalten ist, und durch die Bit-Adresse/Bit-Nummer $b(i)$ gekennzeichnet. Bezeichnet man mit $a_u = a(i_u)$ die Adresse und mit $b_u = b(i_u)$ die Bit-Adresse des ersten Feldelements, so gilt die Beziehung

$$8 \cdot a(i) + b(i) = 8 \cdot a_u + b_u + (i - i_u)$$

Daraus ergibt sich dann

$$a(i) = a_u + (b_u + (i - i_u))/8$$

und

$$b(i) = (b_u + (i - i_u)) \bmod 8$$

wobei das Ergebnis der Division durch 8 wieder nur den ganzzahligen Anteil bezeichnet. Die Berechnung der Adresse $a(i)$ und der Bit-Adresse $b(i)$ erfolgt wieder in 4 Schritten:

1. Berechnung des relativen Index $i - i_u$.
2. Berechnung des korrigierten relativen Index $b_u + i - i_u$.
3. Gleichzeitige Berechnung von $(b_u + (i - i_u))/8$ und $(b_u + (i - i_u)) \bmod 8$.
4. Berechnung der Adresse $a_u + (b_u + (i - i_u))/8$.

Es ist damit alles wie bei der Adressierung von Nibble-Feldern, nur daß eben die Größen b_u und $b(i)$ je drei Bits belegen und daß statt einer Division durch 2 eine Division durch 8 durchgeführt wird. Die Algorithmen für die Schritte 1, 2 und 4 können daher als bekannt vorausgesetzt werden und lediglich die Realisierung 3. Schrittes soll nun näher betrachtet werden.

Eine Division durch 8 kann man als dreimalige Division durch 2 interpretieren. Es kommt also wieder die Schiebe-Technik zum Einsatz. Die drei Bits, die beim Rechts-Schieben von $b_u+(i-i_u)$ herausfallen, müssen für $b(i)$ gesammelt und stellenwertkorrekt abgelegt werden. Sie werden also nach rechts in das Register geschoben, welches $b(i)$ aufnehmen soll. Durch drei weitere zirkuläre Links-Rotationen gelangen sie auf den richtigen Platz. (Nicht verzweifeln!).

Geht man davon aus, daß $b_u + (i - i_u)$ im Akkumulator geliefert wird und der ganzzahlige Anteil bei der Division dort stehen bleiben soll, während der Rest $b(i)$ in das Register B kommt, dann sieht das zugehörige Programmstück so aus:

```
DIVMOD: LD      B,0       ; Register B zur Aufnahme
                          ; von b(i) vorbereiten
        SRL     A         ; durch 2 dividieren
        RR      B         ; herausgeschobenes Bit sichern
        SRL     A         ; durch 2 dividieren
        RR      B         ; herausgeschobenes Bit sichern
        SRL     A         ; durch 2 dividieren
        RR      B         ; herausgeschobenes Bit sichern
        RLC     B         ; Bits an den richtigen Platz bringen
        RLC     B
        RLC     B
```

Um einen Befehl einzusparen, kann das zuletzt herausgeschobene Bit auch gleich auf Bit-Position 0 gebracht werden:

```
DIVMOD: LD      B,0       ; Register B zur Aufnahme
                          ; von b(i) vorbereiten
        SRL     A         ; durch 2 dividieren
        RR      B         ; herausgeschobenes Bit sichern
        SRL     A         ; durch 2 dividieren
        RR      B         ; herausgeschobenes Bit sichern
        SRL     A         ; durch 2 dividieren
        RL      B         ; herausgeschobenes Bit sichern
        RL      B         ; Bits an richten Platz bringen
        RL      B
```

Die Bearbeitung des Feldelements geschieht am besten mit den bereits gezeigten Maskierungstechniken. Mit Hilfe der Größe $b(i)$, die genau deswegen ins Register B gebracht wurde, erstellt man eine Maske für die gewünschte Operation (testen, setzen, zurücksetzen, invertieren). Für das Testen, Setzen und Invertieren (alle drei Masken sehen hierbei gleich aus – vgl. Abschnitt 7.2.4):

```
        LD      A,10000000B     ; Bit 7 setzen, Rest loeschen
        INC     B               ; wegen DJNZ-Befehl und moeglichem
                                ; Inhalt 0
SETZEN: RLCA                    ; gesetztes Bit wandern lassen
        DJNZ    SETZEN          ; bis es am richtigen Platz ist
```

Für das Löschen wird nur eine andere Ausgangsmaske benötigt:

```
        LD      A,01111111B     ; Bit 7 loeschen, Rest setzen
        INC     B               ; wegen DJNZ-Befehl und moeglichem
                                ; Inhalt 0
LOESCH: RLCA                    ; geloeschtes Bit wandern lassen
        DJNZ    LOESCH          ; bis es am richtigen Platz ist
```

Steht die Maske endlich im Akkumulator, so wird die Operation mit der Adresse $a(i)$ im Registerpaar HL folgendermaßen ausgeführt:

```
; Bit testen
        AND     (HL)    ; alle Bits mit Maske verknuepfen
                        ; Z = 0, wenn Bit gesetzt

; Bit setzen
        OR      (HL)    ; alle Bits mit Maske verknuepfen
        LD      (HL),A  ; Bits zurueckschreiben

; Bit loeschen
        AND     (HL)    ; alle Bits mit Maske verknuepfen
        LD      (HL),A  ; Bits zurueckschreiben

; Bit invertieren
        XOR     (HL)    ; alle Bits mit Maske verknuepfen
        LD      (HL),A  ; Bits zurueckschreiben
```

Die ständige Neuberechnung der Masken ist bei häufiger Ausführung der Bit-Operationen zu aufwendig. Deshalb legt man besser zwei Byte-Felder mit jeweils 8 Elementen an, welche die beiden Typen von Masken enthalten:

```
; Masken zum Testen, Setzen oder Invertieren eines Bits

SMASKE: DEFB    00000001B       ; fuer Bit 0
        DEFB    00000010B       ; fuer Bit 1
        DEFB    00000100B       ; fuer Bit 2
        DEFB    00001000B       ; fuer Bit 3
        DEFB    00010000B       ; fuer Bit 4
        DEFB    00100000B       ; fuer Bit 5
        DEFB    01000000B       ; fuer Bit 6
        DEFB    10000000B       ; fuer Bit 7

; Masken zum Loeschen eines Bits

LMASKE: DEFB    11111110B       ; fuer Bit 0
        DEFB    11111101B       ; fuer Bit 1
        DEFB    11111011B       ; fuer Bit 2
        DEFB    11110111B       ; fuer Bit 3
        DEFB    11101111B       ; fuer Bit 4
        DEFB    11011111B       ; fuer Bit 5
        DEFB    10111111B       ; fuer Bit 6
        DEFB    01111111B       ; fuer Bit 7
```

Der Zugriff auf die Masken erfolgt nun mit Hilfe der Techniken, die für Byte-Felder gezeigt wurden. Dazu wird $b(i)$ nicht ins Register B sondern ins Register C gebracht (obige Routine läßt sich leicht abändern). Die Adresse $a(i)$, die sich im Registerpaar HL befindet, wird vorübergehend im Registerpaar DE gesichert, um HL zunächst zur Adressierung der Maskenfelder benutzen zu können. Die Maske (hier zum Setzen eines Bits; für die anderen Operationen analog) wird mit folgendem Programmstück beschafft:

```
        EX      DE,HL           ; a(i) temporaer sichern
        LD      HL,SMASKE       ; Anfangsadresse des Maskenfeldes
        LD      B,0             ; b(i) zu Wort erweitern
        ADD     HL,BC           ; Adresse der Maske berechnen
        LD      A,(HL)          ; Maske holen
        EX      DE,HL           ; a(i) ins Registerpaar HL bringen
```

Durch einen kleinen Trick läßt sich bei Verwendung von Maskenfeldern die Routine DIVMOD nochmals verkürzen. Das Links-Rotieren war notwendig, um die Bit-Adresse $b(i)$ nicht spiegelverkehrt zu erhalten. (Zuerst würde ja das letzte Bit ankommen und schließlich in Bit 2 stehen.) Durch Umstellen der Masken im Maskenfeld kann man aber auch mit dem spiegelverkehrten Wert von $b(i)$ arbeiten. Dabei hilft folgende Zuordnungstabelle:

$b(i)$	Spiegelbild
0	0
1	4
2	2
3	6
4	1
5	5
6	3
7	7

Als Maskenfeld für das Löschen eines Bits ergibt sich dann:

```
; Masken zum Loeschen eines Bits

LMASKE: DEFB    11111110B       ; fuer Bit 0
        DEFB    11101111B       ; fuer Bit 4
        DEFB    11111011B       ; fuer Bit 2
        DEFB    10111111B       ; fuer Bit 6
        DEFB    11111101B       ; fuer Bit 1
        DEFB    11011111B       ; fuer Bit 5
        DEFB    11110111B       ; fuer Bit 3
        DEFB    01111111B       ; fuer Bit 7
```

Die Routine DIVMOD ändert sich wie folgt:

```
DIVMOD: LD      C,0     ; Register C zur Aufnahme
                        ; des Spiegelbildes von b(i) vorbereiten
        SRL     A       ; durch 2 dividieren
        RL      C       ; herausgeschobenes Bit sichern
        SRL     A       ; durch 2 dividieren
        RL      C       ; herausgeschobenes Bit sichern
        SRL     A       ; durch 2 dividieren
        RL      C       ; herausgeschobenes Bit sichern
```

Zu Beginn wurde das Register C gelöscht, weshalb das C-Flag nach jedem der Befehle RL C gelöscht ist. Statt des Befehls SRL A kann aus diesem Grund der weniger aufwendige Befehl RRA benutzt werden:

```
DIVMOD: LD      C,0     ; Register C zur Aufnahme
                        ; des Spiegelbildes von b(i) vorbereiten
        SRL     A       ; durch 2 dividieren
        RL      C       ; herausgeschobenes Bit sichern
        RRA             ; durch 2 dividieren
        RL      C       ; herausgeschobenes Bit sichern
        RRA             ; durch 2 dividieren
        RL      C       ; herausgeschobenes Bit sichern
```

7.3.3 Bearbeiten ganzer Felder

Häufig kommt es vor, daß nicht nur auf einzelne Elemente eines Feldes zugegriffen wird, sondern daß alle Elemente eines Feldes fortlaufend – beginnend mit dem kleinsten oder dem größten Index – bearbeitet werden müssen. Das Berechnen der einzelnen Adressen der Feldelemente mit Hilfe der eben gezeigten Methoden wäre in diesem Fall umständlich, da die innere Struktur des Feldes dabei nicht gut ausgenutzt wird. Deshalb soll hier die Adressierung einer Menge von Feldelementen mit fortlaufenden Indizies aus einem großen zusammenhängenden Indexbereich gezeigt werden.

Dieses allgemeine Problem unterscheidet sich nicht wesentlich von der fortlaufenden Adressierung aller Feldelemente. Daher sollen hier nur zwei Typen der Adressierung genauer betrachtet werden: Fortlaufende Adressierung aller Feldelemente mit aufsteigenden Indizies (mit absteigenden Indizies geht es genauso!) und fortlaufende Adressierung von Feldelementen, beginnend beim kleinsten Index, bis zum Index eines Feldelements, das als erstes eine bestimmte Bedingung erfüllt. (Suchfunktionen rückwärts und/oder bis zum n-ten Feldelement mit einer bestimmten Eigenschaft lassen sich daraus leicht ableiten.)

Die einfachste Möglichkeit, alle Elemente eines Feldes fortlaufend zu adressieren, besteht darin, eine Zählschleife zu konstruieren, deren Schleifenkörper so oft durchlaufen wird, wie Indizies zu verarbeiten sind. Der Schleifenkörper sorgt dann für die Adressierung der Feldelemente und für deren Bearbeitung. Die Adressierung erfolgt indirekt über ein Daten-Adreß-Register; in den meisten Fällen wird dazu das Registerpaar HL gewählt. Zu Beginn wird das Adreß-Register auf die Adresse des ersten Feldelements gesetzt (in den folgenden Beispielen wird dies als bereits durchgeführt betrachtet). In jedem Durchlauf der Schleife wird dann dieser Zeiger auf das folgende Feldelement verschoben.

Nachfolgend wird stets angenommen, daß die Zahl der zu bearbeitenden Elemente zwischen 1 und 256 liegt und deshalb eine automatische Zählschleife benutzt werden kann. (Dies stellt keine wesentliche Einschränkung dar, denn ein Umstieg auf selbstgesteuerte Zählschleifen ist jederzeit möglich.) Die Anzahl der Feldelemente wird daher stets im Register B erwartet.

Anfangs sollen die Elemente eines Byte-Feldes negiert werden:

```
NEGIER: LD      A,(HL)  ; Feldelement beschaffen
        NEG             ; Element negieren
        LD      (HL),A  ; Feldelement zurueckschreiben
        INC     HL      ; auf naechstes Element zeigen
        DJNZ    NEGIER  ; naechster Schleifendurchlauf
                        ; bis alle Feldelemente bearbeitet wurden
```

Bei Byte-Feldern muß also nur der Zeiger inkrementiert werden. Bei Wort-Feldern genügt es, den Zeiger pro Schleifendurchlauf zweimal zu inkrementieren, ob dies am Schleifenende oder während der Bearbeitung des Feldelements geschieht, ist prinzipiell egal. Auch hierzu ein Beispiel: die Verdopplung aller Feldelemente eines Wort-Feldes. Die Adressierung ist ziemlich einfach, die Bearbeitung der Elemente dagegen kompliziert:

```
DOPPEL: LD      A,(HL)  ; niederwertiges Byte des
                        ; Feldelements holen
        ADD     A,A     ; niederwertiges Byte des Elements
                        ; verdoppeln
        LD      (HL),A  ; niederwertiges Byte des Elements
                        ; zurueckschreiben
        INC     HL      ; auf hoeherwertiges Byte zeigen
                        ; (C-Flag bleibt intakt)
        LD      A,(HL)  ; hoeherwertiges Byte des
                        ; Feldelements holen
        ADC     A,A     ; hoeherwertiges Byte des Elements
                        ; verdoppeln, dabei Uebertrag einbeziehen
        LD      (HL),A  ; hoeherwertiges Byte des Elements
                        ; zurueckschreiben
        INC     HL      ; auf naechstes Element zeigen
        DJNZ    DOPPEL  ; naechster Schleifendurchlauf
                        ; bis alle Feldelemente bearbeitet wurden
```

Der 8-Bit-Arithmetik-Befehl ADC (add with carry) wirkt wie der 8-Bit-ADD-Befehl, addiert jedoch den Wert des C-Flags (Übertrag) noch hinzu. (vgl. Abschnitt 5.3.3 16-Bit-ADC-Befehl).

Bei den Programmen wird zunächst davon abgesehen, daß die auszuführenden Operationen möglicherweise auf einen Fehler treffen (z.B. Überlauf). Hier handelt es sich also um den Prototyp einer Addition beliebig großer ganzer Zahlen.

Der nächste Schwierigkeitsgrad ist erreicht, wenn die Länge der Feldelemente ein fortwährendes Inkrementieren des Zeigers unrationell werden läßt. Dann berechnet man die Adresse des jeweils nächsten Feldelements durch Addition der Länge eines Elements zum aktuellen Zeiger. Voraussetzung ist dabei natürlich, daß dieser durch die Bearbeitung des Feldelements noch nicht verändert wurde. Als Beispiel soll der Wert 00H in das niederwertigste Byte jedes Elements geladen werden, wobei die Elemente eine Länge von 16 Bytes haben sollen:

```
        LD      DE,16   ; Laenge eines Feldelements
NULLEN: LD      (HL),0  ; niederwertigstes Byte des Elements
                        ; auf Null setzen
        ADD     HL,DE   ; auf naechstes Element zeigen
        DJNZ    NULLEN  ; naechster Schleifendurchlauf
                        ; bis alle Feldelemente bearbeitet wurden
```

An dieser Stelle einige Bemerkungen zu mehrdimensionalen Feldern. Im nächsten Beispiel wird von einem zweidimensionalen zeilenorientierten Feld ausgegangen. Wird das Feld zeilenweise bearbeitet, so können alle Elemente ihrer Speicherungsreihenfolge nach verwertet werden. Dies gilt ebenfalls, wenn nur die Elemente einer bestimmten Zeile bearbeitet werden sollen. Ist jedoch die Bearbeitung der Elemente einer bestimmten Spalte gewünscht, dann muß die Adresse von Element zu Element um $(i_{o_2} - i_{u_2} + 1) \cdot L$ Byte weitergeschaltet werden. Mit der eben gezeigten Technik kann man das bewerkstelligen.

Weitaus schwieriger wird es, wenn alle Elemente des Feldes spaltenweise bearbeitet werden sollen. Hier eine Adressierungs-Routine für ein zweidimensionales Byte-Feld mit 8 Zeilen und 12 Spalten (a_u wird in HL erwartet):

```
; Datenbereich

BASIS:  DEFS    2               ; Hilfs-Speicherplatz fuer
```

```
                                ; Zeiger auf erstes Element
                                ; einer Spalte

; Programmbereich

DIM2:   LD      C,12            ; Anzahl der Spalten
        LD      DE,12           ; Anzahl der Spalten mal
                                ; Laenge eines Feldelements
SPALTE: LD      (BASIS),HL      ; Zeiger auf erstes Element
                                ; einer Spalte sichern
        LD      B,8             ; Anzahl der Zeilen
NULLEN: INC     (HL)            ; Feldelement um 1 erhoehen
        ADD     HL,DE           ; auf naechstes Element
                                ; derselben Spalte zeigen
        DJNZ    NULLEN          ; alle Feldelemente einer
                                ; Spalte bearbeiten
        LD      HL,(BASIS)      ; Zeiger auf erstes Element
                                ; der alten Spalte holen
        INC     HL              ; auf erstes Element der
                                ; naechsten Spalte zeigen
        DEC     C               ; restliche Anzahl der Spalten
                                ; berechnen
        JP      NZ,SPALTE       ; alle Spalten bearbeiten
```

Bisher wurde stets nur ein Feld gleichzeitig bearbeitet. Oftmals werden aber die Elemente zweier Felder verknüpft und die Ergebnisse in die Elemente des ersten Feldes zurückgeschrieben. Deshalb sollen nun zwei Wort-Felder komponentenweise addiert und die Ergebnisse in das erste Feld zurückgeschrieben werden (Vektor-Addition). Neben dem Registerpaar HL wird ein weiteres Registerpaar als Daten-Adreß-Register benötigt; sinnvollerweise wählt man dazu am besten DE, da das Register B zum Zählen benutzt wird:

```
VEKADD: LD      A,(DE)  ; LSB eines Elements des
                        ; ersten Feldes holen
        ADD     A,(HL)  ; LSB eines Elements des
                        ; zweiten Feldes addieren
        LD      (DE),A  ; LSB der Summe zurueckschreiben
        INC     DE      ; auf MSB eines Elements
                        ; des ersten Feldes zeigen
        INC     HL      ; auf MSB eines Elements
                        ; des zweiten Feldes zeigen
        LD      A,(DE)  ; MSB eines Elements des
                        ; ersten Feldes holen
        ADC     A,(HL)  ; MSB eines Elements des
                        ; zweiten Feldes addieren, dabei
                        ; Uebertrag beruecksichtigen
        LD      (DE),A  ; MSB der Summe zurueckschreiben
        INC     DE      ; auf naechstes Element
                        ; des ersten Feldes zeigen
        INC     HL      ; auf naechstes Element
                        ; des zweiten Feldes zeigen
        DJNZ    VEKADD  ; alle Feldelemente bearbeiten
```

Mitunter kommt es vor, daß das Register B nicht als Zähler verwendet werden kann, zum Beispiel wenn man drei Felder simultan bearbeitet. In solchen Fällen wird die Zählgröße temporär im Speicher abgelegt. Folgende Subtraktion zweier Wort-Felder (Vektor-Differenz) mit Ablage des Ergebnisses in einem dritten Wort-Feld erläutert die Vorgehensweise (die Zeiger auf das erste, zweite, dritte Feld werden in DE, HL, BC erwartet, der Wert der Zählgröße im Speicherplatz ZAEHL):

```
; Datenbereich

ZAEHL:  DEFS    1               ; Hilfs-Speicherplatz
                                ; fuer Zaehlgroesse

; Programmbereich

VEKSUB: LD      A,(DE)          ; LSB eines Elements des
                                ; ersten Feldes holen
        SUB     (HL)            ; LSB eines Elements des
                                ; zweiten Feldes subtrahieren
        LD      (BC),A          ; LSB der Differenz in Element
                                ; des dritten Feldes schreiben
        INC     DE              ; auf MSB eines Elements des
                                ; ersten Feldes zeigen
        INC     HL              ; auf MSB eines Elements des
                                ; zweiten Feldes zeigen
        INC     BC              ; auf MSB eines Elements des
```

```
                                ; dritten Feldes zeigen
        LD      A,(DE)          ; MSB eines Elements des
                                ; ersten Feldes holen
        SBC     A,(HL)          ; MSB eines Elements des
                                ; zweiten Feldes subtrahieren,
                                ; dabei Uebertrag beruecksichtigen
        LD      (BC),A          ; MSB der Differenz in Element
                                ; des dritten Feldes schreiben
        INC     DE              ; auf naechstes Element des
                                ; ersten Feldes zeigen
        INC     HL              ; auf naechstes Element des
                                ; zweiten Feldes zeigen
        INC     BC              ; auf naechstes Element des
                                ; dritten Feldes zeigen
        LD      A,(ZAEHL)       ; Zaehlgroesse holen
        DEC     A               ; Zaehlgroesse verringern und
                                ; auf Null testen
        LD      (ZAEHL),A       ; Zaehlgroesse wieder sichern
        JP      NZ,VEKSUB       ; alle Feldelemente bearbeiten
```

Der 8-Bit-Arithmetik-Befehl SBC (subtract carry; Subtraktion mit Übertrag) funktioniert wie der SUB-Befehl; vom Ergebnis wird aber noch der Wert des C-Flags (Übertrag) abgezogen. Der SBC-Befehl für 16-Bit-Arithmetik wurde bereits im Abschnitt „5.3.3 Arithmetik mit Worten“ gezeigt.

Zum Abschluß der Felder, deren Elemente je eine bestimmte Anzahl von Bytes belegen, ein Beispiel für die Suche nach einem Element mit einer vorgegebenen Eigenschaft. Dieses Problem wird am besten durch eine Zählschleife mit Abbruch gelöst. Der Abbruch erfolgt, sobald das gewünschte Element gefunden ist. In einem Byte-Feld wird nach dem ersten Element größer 99 gesucht:

```
        LD      A,99            ; Testgroesse festlegen
FINDE:  CP      (HL)            ; Feldelement mit Testgroesse
                                ; vergleichen
        JP      C,GEFUND        ; Inhalt des Feldelements
                                ; groesser als Testgroesse
                                ; Suche abbrechen
        INC     HL              ; auf naechstes Element zeigen
        DJNZ    FINDE           ; moeglicherweise alle
                                ; Feldelemente pruefen
NICHT:  :                       ; Feld enthaelt kein Element
        :                       ; mit gewuenschter Eigenschaft
```

Kommen wir jetzt zu den Nibble-Feldern. Prinzipiell müssen dabei zum einen die Adressen fortlaufend weitergeschaltet, zum anderen innerhalb eines Bytes die Nibbles fortlaufend verarbeitet werden. Natürlich könnte man die Verarbeitung der beiden Nibbles eines Bytes als eine zusammengehörige Operation ansehen. Dies läßt sich mit den bisher gezeigten Methoden für die Bearbeitung von Byte-Feldern und den Bit-Manipulationstechniken leicht bewerkstelligen.

Andererseits sollte bei komplizierten Operationen auf Nibbles auch die interne Struktur eines Bytes ausgenutzt werden. Man schachtelt zwei Schleifen ineinander: Die äußere Schleife ist die Zählschleife, welche die fortlaufende Adressierung der Bytes gewährleistet. Die innere Schleife ist ebenfalls eine Zählschleife, die dafür sorgt, daß der Reihe nach beide Nibbles eines Bytes bearbeitet werden. Das Beschaffen der Nibbles erfolgt für aufsteigende Indizies durch den Befehl RRD, für absteigende Indizies durch den Befehl RLD.

Den Anfang macht die aufsteigende Bearbeitung eines Nibble-Feldes, dessen Grenzen mit Byte-Grenzen übereinstimmen. Dabei soll der erste Nibble mit dem Wert 0 ausfindig gemacht werden (a_u wird im Registerpaar HL erwartet, die Anzahl der Bytes im Register B):

```
        LD      A,0             ; der Test auf Null im RRD-Befehl
                                ; klappt nur, wenn der hoeher-
                                ; wertige Nibble des Akkumulators
                                ; geloescht ist
BYTE:   LD      C,2             ; 2 Nibbles pro Byte
NIBBLE: RRD                     ; Nibble in Akkumulator holen
        JP      Z,GEFUND        ; Nibble ist Null,
                                ; Suche abbrechen
        DEC     C               ; restliche Anzahl von Nibbles im
                                ; Byte berechnen und Test auf Null
        JP      NZ,NIBBLE       ; noch ein Nibble zu verarbeiten
        RRD                     ; Byte wiederherstellen
        INC     HL              ; auf naechstes Byte zeigen
        DJNZ    BYTE            ; alle Feldelemente pruefen
```

Bei diesem Algorithmus sind zwei Dinge wichtig:

1. Da das Nibble-Feld nicht beschädigt werden soll, muß durch dreimaliges Rotieren jedes Byte nach der Bearbeitung wiederhergestellt werden.
2. Die Suche wird abgebrochen, sobald ein Nibble mit der gewünschten Eigenschaft gefunden wurde. Das bearbeitete Byte muß aber wiederhergestellt werden, möglichst ohne die Zähler in Register B und C zu zerstören, denn diese geben ja an, wo sich der gefundene Nibble befindet.

Um den zweiten Teil zu erledigen, muß man noch folgendes Programmstück an der Stelle einfügen, die nach geglückter Suche angesprungen wird:

```
GEFUND: LD      D,C             ; Nibble-Zaehler umspeichern
REPARA: RRD                     ; Byte reparieren
        DEC     D               ; Anzahl der noch durch-
                                ; zufuehrenden Rotationen
                                ; berechnen
        JP      NZ,REPARA       ; genuegend Reparaturen
                                ; ausfuehren
```

Stimmen die Feldgrenzen nicht mit Byte-Grenzen überein, so muß ein „Vorlauf“ bzw. ein „Nachlauf“ gemacht werden. Der Vorlauf besteht darin, daß der nicht interessierende niederwertige Nibble des ersten Bytes zwar geholt, aber nicht getestet wird. Mit dem reduzierten Nibble-Zähler wird dann in die innere Schleife eingesprungen. Für den Nachlauf wird das letzte Byte des Feldes ohne Rotation geholt und der höherwertige Nibble wegmaskiert, weil das einfacher ist:

```
        LD      A,0             ; der Test auf Null im RRD-Befehl
                                ; klappt nur, wenn der hoeher-
                                ; wertige Nibble des Akkumulators
                                ; geloescht ist
VORLF:  RRD                     ; ersten Nibble nicht testen
        LD      C,1             ; Zaehler reduzieren
        JP      NIBBLE          ; in innere Schleife springen
BYTE:   LD      C,2             ; 2 Nibbles pro Byte
NIBBLE: RRD                     ; Nibble in Akkumulator holen
        JP      Z,GEFUND        ; Nibble ist Null,
                                ; Suche abbrechen
        DEC     C               ; restliche Anzahl von Nibbles im
                                ; Byte berechnen und Test auf Null
        JP      NZ,NIBBLE       ; noch ein Nibble zu verarbeiten
        RRD                     ; Byte wiederherstellen
        INC     HL              ; auf naechstes Byte zeigen
        DJNZ    BYTE            ; alle Feldelemente pruefen
NACHLF: LD      A,(HL)          ; letztes Byte des Feldes holen
        AND     0FH             ; hoeherwertigen Nibble
                                ; wegmaskieren
        JP      Z,GEFUND        ; Suche war beim letzten
                                ; Nibble erfolgreich
```

Es kann bei der Bearbeitung von Nibble-Feldern vorkommen, daß man weder Vorlauf noch Nachlauf benötigt, oder einen davon, oder schlimmstenfalls auch beide.

Die Bearbeitung von Nibble-Feldern mit absteigenden Indizies erfolgt ganz analog mit umgekehrter Rotationsrichtung.

Doch nun zum wohl kniffligsten Problem: die Bearbeitung ganzer Bit-Felder. Dabei sollte man stets versuchen, (zunächst) nur Byte-Operationen durchzuführen.

Beispiel: Eine häufige Anwendung von Bit-Feldern sind Pixelgrafiken wie im IRM des KC-Grundgerätes. Ein gesetztes Bit entspricht einem gezeichneten Pixel, ein gelöschtes Bit einem freien Pixel. Natürlich kann man auch zwei Bilder „übereinanderlegen“ und dadurch zu einem neuen Bild verschmelzen. Dabei gibt es prinzipiell zwei Möglichkeiten: Überschreiben und Transparenz.

Beim Überschreiben wird überall dort ein Punkt eingetragen, wo sich in mindestens einem der beiden Bilder ein Punkt befindet. Dies entspricht in der Bit-Darstellung der logischen Operation OR. Die Bits müssen also lediglich mittels OR-Befehl verknüpft werden. Im folgenden wird davon ausgegangen, daß HL und DE Zeiger auf zwei Bilder sind, und daß in BC die Länge des Feldes in Bytes steht. Das erste Bild wird mit dem zweiten überlagert:

```
UEBERL: LD      A,(DE)          ; 8 Bits des zweiten Feldes
                                ; auf einmal holen
        OR      (HL)            ; simultan 8 Bits des ersten
                                ; Feldes verknuepfen
        LD      (HL),A          ; resultierende Bits in erstes
```

```
                             ; Feld zurueckschreiben
        INC     HL           ; auf naechstes Byte des
                             ; ersten Feldes zeigen
        INC     DE           ; auf naechstes Byte des
                             ; zweiten Feldes zeigen
        DEC     BC           ; Anzahl der restlichen
                             ; Bytes berechnen
        LD      A,B          ; und auf
        OR      C            ; Null testen
        JP      NZ,UEBERL    ; alle Feldelemente bearbeiten
```

Im Transparent-Modus wird nur dort ein Punkt gesetzt, wo sich in genau einem der beiden Bilder ein Punkt befindet. Überschneidungen von Linien werden dadurch zum Beispiel besser sichtbar. Hier werden die Bits also durch den XOR-Befehl verknüpft. Um die Sache nicht ganz so einfach zu machen, soll zusätzlich die untere Feldgrenze nicht mit einer Byte-Grenze zusammenfallen. Die Bit-Adresse b_u des Feldes wird im Akkumulator übergeben. Ein Programm für diese Aufgabe könnte so aussehen:

```
; Datenbereich

ZEIGER: DEFS    2               ; Hilfs-Speicherplatz fuer
                                ; Zeiger auf erstes Feld
ZAEHL:  DEFS    2               ; Hilfs-Speicherplatz fuer
                                ; Byte-Zaehler
MASKEN: DEFB    11111111B       ; Maske fuer bu = 0
        DEFB    11111110B       ; Maske fuer bu = 1
        DEFB    11111100B       ; Maske fuer bu = 2
        DEFB    11111000B       ; Maske fuer bu = 3
        DEFB    11110000B       ; Maske fuer bu = 4
        DEFB    11100000B       ; Maske fuer bu = 5
        DEFB    11000000B       ; Maske fuer bu = 6
        DEFB    10000000B       ; Maske fuer bu = 7

; Programmbereich

VORLF1: LD      (ZEIGER),HL     ; Zeiger auf erstes Feld sichern
        LD      (ZAEHL),BC      ; Byte-Zaehler sichern
        LD      HL,MASKEN       ; Anfangsadresse des
                                ; Maskenfeldes laden
        LD      C,A             ; Bit-Adresse ins
        LD      B,0             ; Registerpaar BC bringen
        ADD     HL,BC           ; Adresse der richtigen
                                ; Maske berechnen
        LD      A,(DE)          ; erstes Byte des
                                ; zweiten Feldes holen
        AND     (HL)            ; nicht zum Bild gehoerige
                                ; Bits loeschen
        LD      HL,(ZEIGER)     ; Zeiger auf erstes Feld
                                ; wiederherstellen
        LD      BC,(ZAEHL)      ; Byte-Zaehler wiederherstellen
        JP      VORLF2          ; in Schleife einspringen
UEBERL: LD      A,(DE)          ; 8 Bits des zweiten Feldes
                                ; auf einmal holen
VORLF2: XOR     (HL)            ; simultan 8 Bits des
                                ; ersten Feldes verknuepfen
        LD      (HL),A          ; resultierende Bits in erstes
                                ; Feld zurueckschreiben
        INC     HL              ; auf naechstes Byte des
                                ; ersten Feldes zeigen
        INC     DE              ; auf naechstes Byte des
                                ; zweiten Feldes zeigen
        DEC     BC              ; Anzahl der restlichen Bytes
                                ; berechnen
        LD      A,B             ; und auf
        OR      C               ; Null testen
        JP      NZ,UEBERL       ; alle Feldelemente bearbeiten
```

Zum Schluß soll die Adresse und Bit-Adresse des ersten gesetzten Bits eines Bit-Feldes bestimmt werden. Der Einfachheit halber wird angenommen, daß die Feldgrenzen mit Byte-Grenzen zusammenfallen. Anstatt jedes Bit einzeln zu testen, wird erst (im Registerpaar HL) die Adresse des Byte, in dem das gesuchte Bit liegt, bestimmt und aus dem Byte selbst dann die Bit-Adresse (im Register B). Ob die Suche erfolgreich war, soll mit Hilfe des C-Flags angezeigt werden: Gesetztes C-Flag heißt erfolgreiche Suche; zurückgesetztes C-Flag bedeutet, daß alle Bits des Feldes zurückgesetzt sind.

```
BYTES:  LD      A,(HL)          ; 8 Feldelemente simultan
        OR      A               ; auf Null testen
        JP      NZ,GEFUND       ; mindestens ein Bit gesetzt
```

```
        INC     HL              ; auf naechstes Byte zeigen
        DEC     BC              ; Anzahl der restlichen
                                ; Bytes berechnen
        LD      A,B             ; Anzahl der restlichen Bytes
        OR      C               ; auf Null testen
        JP      NZ,BYTES        ; alle Feldelemente pruefen
        JP      FERTIG          ; kein gesetztes Bit gefunden,
                                ; C-Flag ist durch den
                                ; OR-Befehl zurueckgesetzt
GEFUND: LD      B,8             ; Anzahl der zu pruefenden Bits
BITS:   ADD     A,A             ; hoechstes Bit ins C-Flag bringen
        JP      C,KORREK        ; gesetztes Bit entdeckt,
                                ; C-Flag ist gesetzt
        DJNZ    BITS            ; alle Bits pruefen
KORREK: DEC     B               ; Korrektur fuer Bit-Adresse
FERTIG: :                       ; gemeinsame Fortsetzungsstelle
        :
```

7.3.4 Verschieben von Feldern

Bislang wurde bei der Bearbeitung ganzer Felder eine Operation ausgespart, die häufig vorkommt und für die es eine besonders elegante Lösung gibt: das Kopieren aller Feldelemente in ein anderes Feld. Da an den Inhalten der Feldelemente dabei gar kein Interesse besteht, sondern diese nur transportiert werden, kann man das Feld einfach als zusammenhängender Speicherbereich betrachten.

Die Aufgabe besteht also darin, eine bestimmte Anzahl von Bytes (bzw. Nibbles oder Bits), beginnend bei einer vorgegebenen Adresse, in einen anderen Speicherbereich zu übertragen, dessen Adresse ebenfalls vorgegeben ist. Grunsätzlich müssen drei verschiedene Situationen berücksichtigt werden:

1. Die beiden Speicherbereiche überlappen sich nicht.
2. Die beiden Speicherbereiche überlappen sich, und zwar am Ende des zu kopierenden Speicherbereichs.
3. Die beiden Speicherbereiche überlappen sich, und zwar am Anfang des zu kopierenden Speicherbereichs.

Im zweiten Fall können nicht einfach die Feldelemente, beginnend mit der niedrigsten Adresse, in das zweite Feld kopiert werden, weil dabei einige am Ende des ersten Feldes gelegene Elemente zerstört würden, bevor sie in das zweite Feld kopiert wurden. Beginnt der Kopiervorgang jedoch bei der höchsten Adresse, so wird das Feld korrekt kopiert.

Im dritten Fall ist die Situation umgekehrt. Hier muß man unbedingt bei der niedrigsten Adresse beginnen. (Der erste Fall bereitet keinerlei Probleme, weshalb sowohl mit der niedrigsten als auch mit der höchsten Adresse begonnen werden kann.)

Zunächst sollen Felder kopiert werden, deren Elemente je ein ganzzahliges Vielfaches an Bytes belegen. Beim Transport braucht dann weder die genaue Länge eines Feldelements noch die Dimension des Feldes bekannt zu sein. Es werden nur folgende Angaben benötigt: die Länge des Feldes in Bytes und die Anfangs- oder Endadressen der beiden Felder. Anfangs- und Endadresse hängen über die Beziehung

$$\text{Endadresse} = \text{Anfangsadresse} + \text{Länge} - 1$$

voneinander ab, so man daß je nach Bedarf die benötigte Größe berechnen kann.

Als erstes soll das Kopieren mit der höchsten Adresse beginnen. Dabei wird davon ausgegangen, daß die Endadresse des ersten Feldes im Registerpaar HL steht, im Registerpaar DE die des zweiten Feldes und im Registerpaar BC die Länge der Felder in Bytes. Der Z80 hat einen sehr effizienten Befehl, dessen Ausführung das gesamte Problem auf einen Schlag erledigt: LDDR (load, decrement and repeat; laden, verringern und wiederholen). Die Funktion des LDDR-Befehls kann wie folgt beschrieben werden:

wiederhole
    (DE) = (HL)
    DE = DE − 1
    HL = HL − 1
    BC = BC − 1
bis BC = 0

Dieser unscheinbare 2-Byte-Befehl ersetzt eine vollständige Zählschleife. Es ist lediglich die vorhergehende Besetzung der drei Registerpaare BC, DE und HL nötig.

Für das dritte Problem, mit der niedrigsten Adresse zu beginnen, gibt es ebenfalls einen entsprechenden Befehl: LDIR (load, increment and repeat; laden, erhöhen und wiederholen). Dessen Funktion so aussieht:

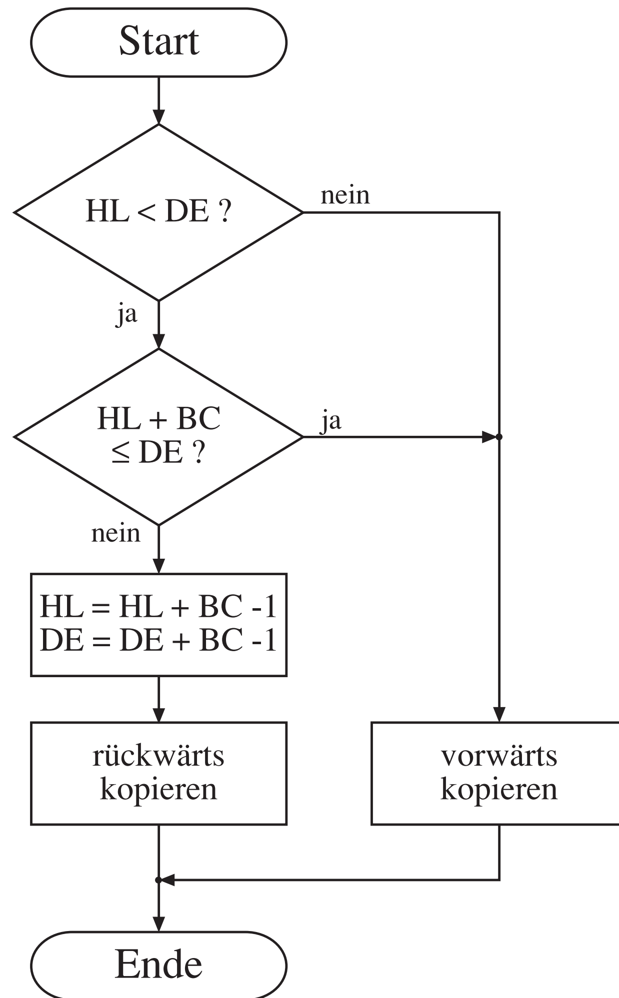


Bild 10: Flußdiagramm zum Kopieren von Feldern

wiederhole
    (DE) = (HL)
    DE = DE + 1
    HL = HL + 1
    BC = BC − 1
bis BC = 0

Im folgenden Beispiel wird angenommen, daß nur die Länge der beiden Felder und ihre Anfangsadressen übergeben werden. Bevor mit dem Kopieren begonnen werden kann, muß erst festgestellt werden, ob dies von der niedrigsten oder von der höchsten Adresse ab zu geschehen hat. Wird bei der niedrigsten Adresse begonnen, so können die Anfangsadressen der Felder benutzt werden, ansonsten müssen die Endadressen mit Hilfe der Anfangsadressen und der Länge erst berechnet werden. Die Lösung sieht nun so aus:

wenn Anfangsadresse1 < Anfangsadresse2 ≤ Endadresse1
dann kopiere rückwärts
sonst kopiere vorwärts

Das Flußdiagramm für diesen Ablauf ist in Bild 10 dargestellt. Das zugehörige Programm sieht so aus:

```
OR      A               ; C-Flag loeschen
SBC     HL,DE           ; Anfangsadressen vergleichen
JP      NC,VORW1        ; Anfangsadresse1 >=
                        ; Anfangsadresse2,
                        ; vorwaerts kopieren
ADD     HL,DE           ; Anfangsadresse1 restaurieren
ADD     HL,BC           ; Endadresse1 + 1 berechnen
SCF                     ; Endadresse1 mit
SBC     HL,DE           ; Anfangsadresse2 vergleichen
JP      C,VORW2         ; Endadresse1 < Anfangsadresse2,
                        ; vorwaerts kopieren
ADD     HL,DE           ; Endadresse1 berechnen
EX      DE,HL           ; und temporaer sichern
ADD     HL,BC           ; Endadresse2
DEC     HL              ; berechnen
EX      DE,HL           ; Endadressen tauschen
LDDR                    ; rueckwaerts kopieren
```

```
        JP      FERTIG          ; weiter an gemeinsamer
                                ; Fortsetzungsstelle
VORW1:  ADD     HL,DE           ; Anfangsadresse1 restaurieren
        JP      VORW            ; vorwaerts kopieren
VORW2:  ADC     HL,DE           ; Anfangsadresse2
        OR      A               ; wiederherstellen
        SBC     HL,BC           ;
VORW:   LDIR                    ; vorwaerts kopieren
FERTIG: :                       ; gemeinsame Fortsetzungsstelle
        :
```

Eine weitere Anwendungsmöglichkeit für den LDIR-Befehl liegt vor, wenn alle Elemente eines Feldes mit demselben Wert initialisiert werden sollen. Dazu wird der Initialwert in das erste Element des Feldes gebracht und von dort in das zweite Element kopiert, vom zweiten in das dritte, und so weiter. Die richtigen Parameter für den Vorgang sind dabei:

$$\begin{aligned} \text{Anfangsadresse2} &= \text{Anfangsadresse1} + \text{Länge eines Feldelements} \\ \text{Anzahl der Bytes} &= (\text{Anzahl der Feldelemente} - 1) \cdot \text{Länge eines Feldelements} \end{aligned}$$

Dies soll am Beispiel eines Wort-Feldes gezeigt werden, dessen Elemente mit dem Wert 0001H initialisiert werden sollen (HL ist Zeiger auf Anfang des Feldes, BC enthält die Anzahl der Feldelemente):

```
        LD      (HL),01H        ; niederwertiges Byte des ersten
                                ; Feldelements initialisieren
        INC     HL              ; auf hoeherwertiges Byte des
                                ; ersten Feldelements zeigen
        LD      (HL),00H        ; hoeherwertiges Byte des ersten
                                ; Feldelements initialisieren
        LD      D,H             ; Zeiger kopieren
        LD      E,L             ;
        INC     DE              ; auf zweites Feldelement zeigen
        DEC     HL              ; auf erstes Feldelement zeigen
        DEC     BC              ; Anzahl der zu kopierenden
                                ; Feldelemente
        SLA     C               ; Anzahl der zu kopierenden
        RL      B               ; Bytes
        LDIR                    ; kopieren
```

Will man die Kopiervorgänge abbrechen, sobald eine bestimmte Bedingung erreicht ist (zum Beispiel wenn das nächste zu kopierende Element eine bestimmten Wert besitzt), dann steht die automatische Wiederholung des Kopierens eines Bytes im Weg. Der Z80 besitzt deshalb zwei Befehle, welche die Funktion des LDIR- und des LDDR-Befehls ohne Wiederholung realisieren: LDI (load and increment; laden und erhöhen) und LDD (load and decrement; laden und verringern).

Beide Befehle löschen das P/V-Flag (Überlauf) genau dann, wenn das Registerpaar BC durch das Verringern zu Null wurde. Dies kann man nun dazu benutzen, um eine Zählschleife mit Abbruch zu bauen. Als Beispiel werden die Elemente eines Byte-Feldes in ein anderes Byte-Feld kopiert, solange diese größer als der Inhalt des Akkumulators sind. Die Anzahl der Feldelemente wird dabei im Registerpaar BC, der Zeiger auf das erste Feldelement des ersten Feldes im Registerpaar HL und der entsprechende Zeiger auf das zweite Feld im Registerpaar DE erwartet. Das Programm lautet damit:

```
KOPIE:  CP      (HL)            ; Feldelement mit Testgroesse
                                ; vergleichen
        JP      NC,FERTIG       ; Feldelement nicht groesser
                                ; als Testgroesse, abbrechen
        LDI                     ; Feldelement kopieren
                                ; und Zeiger erhoehen
                                ; sowie Zaehler verringern
        JP      PE,KOPIE        ; gesamtes Feld durchgehen
FERTIG: :                       ; gemeinsame Fortsetzungsstelle
        :
```

Kommen wir nun zum Verschieben von Nibble- und Bit-Feldern. Stimmen die Nibble-Adressen (beziehungsweise Bit-Adressen) der beiden Felder überein, so ist es bei größeren Feldern am einfachsten, Anfangs- und End-Byte, soweit sie nicht vollständig zum Feld gehören, separat zu bearbeiten und alle übrigen Feldelemente byteweise zu kopieren.

Stimmen die Nibble-Adressen (beziehungsweise Bit-Adressen) dagegen nicht überein, so empfiehlt es sich, daß Feld elementeweise zu kopieren und dabei Rotationsbefehle

einzusetzen. Der Vorgang ist für Nibble-Felder und Bit-Felder sehr ähnlich, weshalb er zunächst schematisch beschrieben wird:

1. Es wird eine Zählschleife aufgebaut, die bei jedem Durchlauf genau ein Feldelement transportiert. Dabei wird zuerst durch Rotieren das Element aus dem ersten Feld geholt und dann durch Rotieren ins zweite Feld kopiert. Der Schleifenkörper besteht somit aus zwei Teilen: Holen des Feldelements und Abspeichern des Feldelements.
2. Beim Holen des Feldelements ist zu berücksichtigen, daß dieses Feld nicht verändert werden darf. Daraus folgt, daß beim Kopieren von Bit-Feldern mit zirkulären Rotationen zu arbeiten ist, und daß jedes Byte 8mal rotiert wird. Für Nibbles gibt es leider keine zirkulären Rotationsbefehle, so daß mit normalen Rotationen gearbeitet werden muß. Jedes Byte wird deshalb dreimal rotiert. Außerdem muß man dabei den Nibble sichern, der beim nächsten Rotieren in das Byte gebracht werden soll (normalerweise ist dies der Nibble, der gerade herausrotiert wurde). Darf durch das Kopieren das Feld zerstört werden, so entfällt das dritte Rotieren und das Sichern des Nibbles.
3. Beim Abspeichern des Feldelements wird das zweite Feld zerstört. Alte Inhalte sind uninteressant und werden deshalb nirgendwo aufbewahrt. Das Einrotieren eines Feldelements geschieht durch gewöhnliches Rotieren; bei Bit-Feldern wird pro Byte 8mal rotiert, bei Nibble-Feldern zweimal.
4. Da die indirekte Adressierung für Rotationsbefehle nur durch das Registerpaar HL zu bewerkstelligen ist, wird für beide Felder HL als Adreß-Register verwendet. Nach Bearbeitung eines Feldelements müssen deshalb die Registerpaare DE und HL getauscht werden.
5. Das Erhöhen oder Verringern von Zeigern erfolgt immer beim Erreichen einer Byte-Grenze. Daher müssen Zähler mitgeführt werden, welche die aktuellen Nibble- beziehungsweise Bit-Adressen wiedergeben.
6. Als Schleifenzähler dient das Registerpaar BC (bei Bit-Feldern können damit bis zu 8 KByte bearbeitet werden).
7. Die Rotationsrichtung für aufsteigendes Kopieren ist rechts, für absteigendes Kopieren links.
8. Als Transporter für das Feldelement dient bei Nibbles der Akkumulator, bei Bits das C-Flag.
9. Fallen die Feldgrenzen nicht mit Byte-Grenzen zusammen, so müssen die Bytes am Anfang und/oder Ende der Felder zusätzlich behandelt werden.

Es werden zwei Speicherplätze für die Zähler eingerichtet, bei Nibble-Feldern zusätzlich noch ein Speicherplatz für den ausrotierten Nibble.

Zunächst also ein Programm zum aufsteigenden Verschieben von Nibble-Feldern. Dabei wird angenommen, daß die beiden Zählgrößen die Nibble-Adressen der ersten zu bearbeitenden Nibbles des jeweiligen Feldes angeben. Wenn die untere Feldgrenze nicht mit einer Byte-Grenze übereinstimmt, so wird vor Beginn des eigentlichen Kopierens das erste Byte des ersten Feldes nach rechts rotiert bzw. die beiden Nibbles des ersten Bytes des zweiten Feldes vertauscht. Wenn die obere Feldgrenze nicht mit einer Byte-Grenze übereinstimmt, so wird für das erste Feld noch zweimal nach rechts rotiert, während für das zweite Feld die beiden Nibbles getauscht werden müssen (viermal zirkulär rotieren):

```
; Datenbereich

ZAEHL1: DEFS    1       ; Hilfs-Speicherplatz fuer
                        ; Zaehlgroesse des ersten Feldes
ZAEHL2: DEFS    1       ; Hilfs-Speicherplatz fuer
                        ; Zaehlgroesse des zweiten Feldes
NIBBLE: DEFS    1       ; Hilfs-Speicherplatz fuer
                        ; Nibbles des ersten Feldes

; Programmbereich

VORLF:  LD      A,(ZAEHL1)      ; Nibble-Adresse fuer
                                ; erstes Feld holen
        NEG                     ; Anzahl der Nibbles im ersten
        ADD     A,2             ; Byte berechnen
        LD      (ZAEHL1),A      ; Zaehlgroesse fuer erstes
                                ; Feld abspeichern
        DEC     A               ; auf volles Byte testen
        JP      NZ,NROT         ; volles Byte, nicht rotieren
        RRD                     ; nicht benoetigten Nibble
                                ; ausrotieren
        LD      (NIBBLE),A      ; und sichern
NROT:   LD      A,(ZAEHL2)      ; Nibble-Adresse fuer
```

```
                                ; zweites Feld holen
        NEG                     ; Anzahl der Nibbles im zweiten
        ADD     A,2             ; Byte berechnen
        LD      (ZAEHL2),A      ; Zaehlgroesse fuer zweites
                                ; Feld abspeichern
        DEC     A               ; auf volles Byte testen
        JP      NZ,KOPIE        ; volles Byte, nicht rotieren
        EX      DE,HL           ; Zeiger tauschen
        RLC     (HL)            ; nicht benoetigten Nibble so
        RLC     (HL)            ; rotieren, dass er spaeter wieder
        RLC     (HL)            ; am alten Platz steht
        RLC     (HL)            ;
        EX      DE,HL           ; Zeiger wieder tauschen
KOPIE:  LD      A,B             ; Anzahl der restlichen
        OR      C               ; Feldelemente auf Null testen
        JP      Z,NACHLF        ; alle Feldelemente kopiert
        LD      A,(NIBBLE)      ; einzurotierenden Nibble holen
        RRD                     ; alten Nibble einrotieren,
                                ; neuen Nibble beschaffen
        LD      (NIBBLE),A      ; neuen Nibble fuer naechsten
                                ; Durchgang sichern
        EX      DE,HL           ; Zeiger tauschen
        RRD                     ; neuen Nibble einrotieren
        EX      DE,HL           ; Zeiger wieder tauschen
        LD      A,(ZAEHL1)      ; Zaehler fuer erstes Feld holen
        DEC     A               ; neuen Zaehlwert berechnen
                                ; und auf Null testen
        JP      NZ,NNULL1       ; Zaehler noch nicht Null
        LD      A,(NIBBLE)      ; einzurotierenden Nibble holen
        RRD                     ; Byte wiederherstellen
        INC     HL              ; auf naechstes Byte des
                                ; ersten Feldes zeigen
        LD      A,2             ; Zaehler neu laden fuer
                                ; 2 Nibbles
NNULL1: LD      (ZAEHL1),A      ; Zaehler wieder abspeichern
        LD      A,(ZAEHL2)      ; Zaehler fuer zweites Feld holen
        DEC     A               ; neuen Zaehlwert berechnen
        JP      NZ,NNULL2       ; Zaehler noch nicht Null
        INC     DE              ; auf naechstes Byte des
                                ; zweiten Feldes zeigen
        LD      A,2             ; Zaehler neu laden fuer
                                ; 2 Nibbles
NNULL2: LD      (ZAEHL2),A      ; Zaehler wieder abspeichern
        DEC     BC              ; restliche Anzahl von Nibbles
                                ; berechnen
        JP      KOPIE           ; alle Feldelemente kopieren
NACHLF: LD      A,(ZAEHL1)      ; Zaehler fuer erstes Feld holen
        DEC     A               ; auf volles Byte testen
        JP      NZ,NROT2        ; volles Byte, nicht rotieren
        LD      A,(NIBBLE)      ; einzurotierenden Nibble holen
        RLD                     ; entspricht zwei Rechtsrotationen
NROT2:  LD      A,(ZAEHL2)      ; Zaehler fuer zweites Feld holen
        DEC     A               ; auf volles Byte testen
        JP      NZ,FERTIG       ; volles Byte, nicht tauschen
        EX      DE,HL           ; Zeiger tauschen
        RRC     (HL)            ; hoeherwertigen und nieder-
        RRC     (HL)            ; wertigen Nibble des letzten
        RRC     (HL)            ; Bytes des Feldes tauschen
        RRC     (HL)            ;
        EX      DE,HL           ; Zeiger wieder tauschen
FERTIG: :                       ; gemeinsame Fortsetzungsstelle
        :
```

Das eben gezeigte Programm funktioniert allerdings nicht, wenn die Verschiebungsdistanz genau einen Nibble beträgt. In diesem speziellen Fall muß das Programm folgendermaßen aussehen (der Einfachheit halber wird davon ausgegangen, daß die untere Feldgrenze mit einer Byte-Grenze übereinstimmt und das Feld eine ungerade Anzahl von Nibbles enthält):

```
; Datenbereich

NIBBLE: DEFS    1               ; Hilfs-Speicherplatz fuer
                                ; ausrotierten Nibble

; Programmbereich

NIBB1:  INC     BC              ; Anzahl der zu
        SRL     B               ; bearbeitenden Bytes
```

```
        RR      C                   ; berechnen
KOPIE:  RLD                         ; alten Nibble ausrotieren,
                                    ; neuen Nibble einrotieren,
                                    ; inneren Nibble verschieben
        LD      (NIBBLE),A          ; ausrotierten Nibble sichern
        INC     HL                  ; auf naechstes Byte zeigen
        DEC     BC                  ; restliche Anzahl von Bytes
                                    ; berechnen
        LD      A,B                 ; und auf Null testen
        OR      C                   ;
        LD      A,(NIBBLE)          ; ausrotierten Nibble
                                    ; wieder holen
        JP      NZ,KOPIE            ; alle Feldelemente kopieren
```

Als letztes soll ein Bit-Feld (um mindestens 8 Bit) verschoben werden, beginnend bei dem kleinsten Index, wobei die Feldgrenzen beliebig zu Byte-Grenzen liegen. Die Bit-Adressen der unteren Grenzen der Felder werden in den Variablen ZAEHL1 und ZAEHL2, die Adressen in den Registerpaaren HL und DE, die Anzahl der Bits der Felder im Registerpaar BC erwartet.

```
; Datenbereich

ZAEHL1: DEFS    1       ; Hilfs-Speicherplatz fuer
                        ; Zaehlgroesse des ersten Feldes
ZAEHL2: DEFS    1       ; Hilfs-Speicherplatz fuer
                        ; Zaehlgroesse des zweiten Feldes
BHILF:  DEFS    1       ; Hilfs-Speicherplatz fuer
                        ; Wert des Registers B

; Programmbereich

VORLF:  LD      A,B                 ; Wert des Registers B
        LD      (BHILF),A           ; sichern
        LD      A,(ZAEHL1)          ; Zaehler fuer erstes Feld holen
        LD      B,A                 ; Anzahl der auszurotierenden Bits
        LD      A,8                 ; Maximalzahl von Feldelementen
                                    ; pro Byte
        SUB     B                   ; Anzahl der Feldelemente im
                                    ; ersten Byte berechnen
        LD      (ZAEHL1),A          ; Zaehlgroesse abspeichern
        INC     B                   ; abweisende Schleife modellieren
        JP      TEST1               ; in Schleife einspringen
VROT1:  RRC     (HL)                ; nicht benoetigtes Bit
                                    ; wegrotieren
TEST1:  DJNZ    VROT1               ; alle nicht benoetigten
                                    ; Bits wegrotieren
        EX      DE,HL               ; Zeiger tauschen
        LD      A,(ZAEHL2)          ; Zaehler fuer zweites Feld holen
        LD      B,A                 ; Anzahl der Zusatzrotationen
        LD      A,8                 ; Maximalzahl von Feldelementen
                                    ; pro Byte
        SUB     8                   ; Anzahl der Feldelemente im
                                    ; ersten Byte berechnen
        LD      (ZAEHL2),A          ; Zaehlgroesse abspeichern
        INC     B                   ; abweisende Schleife modellieren
        JP      TEST2               ; in Schleife einspringen
VROT2:  RRC     (HL)                ; Zusatzrotation ausfuehren
TEST2:  DJNZ    VROT2               ; alle Zusatzrotationen
                                    ; ausfuehren
        EX      DE,HL               ; Zeiger wieder tauschen
        LD      A,(BHILF)           ; alten Wert des Registers B
                                    ; beschaffen
        LD      B,A                 ; Register B wiederherstellen
KOPIE:  RRC     (HL)                ; naechstes Bit beschaffen
        EX      DE,HL               ; Zeiger tauschen
        RR      (HL)                ; neues Bit einrotieren
        EX      DE,HL               ; Zeiger wieder tauschen
        LD      A,(ZAEHL1)          ; Zaehler fuer erstes Feld holen
        DEC     A                   ; neuen Zaehlwert berechnen
                                    ; und auf Null testen
        JP      NZ,NNULL1           ; Zaehler noch nicht Null
        INC     HL                  ; auf naechstes Byte des
                                    ; ersten Feldes zeigen
        LD      A,8                 ; Zaehler neu laden fuer 8 Bits
NNULL1: LD      (ZAEHL1),A          ; Zaehler wieder abspeichern
        LD      A,(ZAEHL2)          ; Zaehler fuer zweites Feld holen
        DEC     A                   ; neuen Zaehlwert berechnen
        JP      NZ,NNULL2           ; Zaehler noch nicht Null
```

```
        INC     DE              ; auf naechstes Byte des
                                ; zweiten Feldes zeigen
        LD      A,8             ; Zaehler neu laden fuer 8 Bits
NNULL2: LD      (ZAEHL2),A      ; Zaehler wieder abspeichern
        DEC     BC              ; restliche Anzahl von Bits
                                ; berechnen
        LD      A,B             ; und auf Null testen
        OR      C               ;
        JP      NZ,KOPIE        ; alle Feldelemente kopieren
NACHLF: LD      A,(ZAEHL1)      ; Zaehler fuer erstes Feld holen
        LD      B,A             ; Anzahl der noch not-
                                ; wendigen Rotationen
NROT1:  RRC     (HL)            ; Nachlauf-Rotation ausfuehren
        DJNZ    NROT1           ; alle Nachlauf-Rotationen
                                ; ausfuehren
        LD      A,(ZAEHL2)      ; Zaehler fuer zweites Feld holen
        LD      B,A             ; Anzahl der noch not-
                                ; wendigen Rotationen
        EX      DE,HL           ; Zeiger tauschen
NROT2:  RRC     (HL)            ; Nachlauf-Rotation ausfuehren
        DJNZ    NROT2           ; alle Nachlauf-Rotationen
                                ; ausfuehren
        EX      DE,HL           ; Zeiger wieder tauschen
```

7.4 Zeichenketten

7.4.1 Implementierung von Zeichenketten

Es gibt sehr viele verschiedene Möglichkeiten, Zeichenketten darzustellen. Hier sollen deshalb nur einige grundsätzliche Implementierungstechniken gezeigt werden.

Die einfachste Form einer Zeichenkette ist ein Feld von Zeichen, das eine feste Länge besitzt. Diese ist von vornherein bekannt und kann daher in den Algorithmen als Konstante benutzt werden. Die Vereinbarung erfolgt beispielsweise mit der Pseudo-Operation DEFM:

```
KETTE:  DEFM    'Hallo!'        ; Zeichenkette
                                ; mit der festen Laenge 6
```

Ist der eigentliche Text kürzer als die Zeichenkette, so fügt man davor (rechtsbündiges Schreiben) oder dahinter (linksbündiges Schreiben) Leerzeichen ein. Obwohl Zeichenketten fester Länge recht unflexibel sind und folglich entsprechend wenig Operationen darauf sinnvoll sind, gibt es einen typischen Anwendungsbereich: Felder von Zeichenketten. Beim Aufbau von Feldern oder Tabellen von Zeichenketten können nur Elemente fester Länge benutzt werden, also entweder Zeichenketten fester Länger, oder statt der Zeichenkette selbst ihre Adresse (Deskriptor).

Eine sehr gebräuchliche Form ist die der Zeichenkette mit Längenangabe. Dabei wird entweder direkt vor den Text, der nun eine variable Länge besitzen kann, oder in einem späteren Deskriptor die aktuelle Länge der Zeichenkette – also die Anzahl der in ihr enthaltenen Zeichen – angegeben. Die Länge kann während der Laufzeit eines Programms variieren und unter Umständen sogar Null sein (leere Zeichenkette).

Zuerst die Implementierung einer Zeichenkette, deren Länge direkt vor dem Text steht:

```
KETTE:  DEFB    6               ; Laenge der Zeichenkette
        DEFM    'Hallo!'        ; eigentlicher Text
```

Manchmal begrenzt man die Länge (wie auch hier) auf 255 Zeichen, damit ein Byte für die Längenangabe genügt. Sonst muß man ein Wort dafür verwenden.

Nun ein Beispiel für eine Zeichenkette mit Deskriptor:

```
KETTE:  DEFB    6               ; Laenge der Zeichenkette
        DEFW    TEXT            ; Adresse des eigentlichen Textes
        :
        :
        :
TEXT:   DEFM    'Hallo!'        ; Text steht irgendwo im Speicher
```

Die zweite Form benötigt zwar mehr Speicherplatz, hat aber dafür den Vorteil, daß der Deskriptor eine feste Länge hat und damit zum Beispiel an einer festen Stelle des Datenbereichs stehen kann, obwohl der Text seine Länge permanent ändert. Solche Deskriptoren können insbesondere als Feldelemente verwendet werden.

Eine dritte Form markiert das Ende des Textes durch ein spezielles Zeichen, das im Text selbst dann natürlich nicht vorkommen darf. Meist handelt es sich um ein ASCII-Steuerzeichen, zum Beispiel 00H (NUL) oder 0DH (CR), mitunter aber auch um ganz normale ASCII-Zeichen, zum Beispiel 24H ('$' unter CP/M). Falls die Textzeichen aus dem 7-Bit-ASCII-Code stammen, kann auch ein Zeichen mit einem Code größer als 7FH verwendet werden. Um keinen Speicherplatz zu verschenken, bedient man sich dabei auch gern des folgenden Tricks: Bei Verwendung des 7-Bit-ASCII-Codes ist das letzte Zeichen durch Setzen von Bit 7 zu markieren. Nachfolgend einige Beispiel für Zeichenketten mit Ende-Markierung:

```
KETTE1: DEFM    'Hallo!'        ; eigentlicher Text
        DEFB    00H             ; Ende-Markierung
KETTE2: DEFM    'Hallo!'        ; eigentlicher Text
        DEFB    0DH             ; Ende-Markierung
KETTE3: DEFM    'Hallo!'        ; eigentlicher Text
        DEFM    '$'             ; Ende-Markierung
KETTE4: DEFM    'Hallo!$'       ; wie zuvor, aber
                                ; andere Schreibweise
KETTE5: DEFM    'Hallo'         ; eigentlicher Text
                                ; ohne letztes Zeichen
        DEFB    '!' + 80H       ; letztes Zeichen des Textes
                                ; mit Ende-Markierung
```

Die vierte Form ist eine Kombination aus einer Zeichenkette fester Länge und einer Zeichenkette mit Ende-Markierung. Dabei wird eine Maximallänge für den Text vorgegeben; ist der aktuelle Text kürzer, so wird sein Ende wie bei der dritten Form markiert. Hierzu ein Beispiel für die Maximallänge 6:

```
KETTE1: DEFM    'Hallo!'        ; Ende durch Laengenbegrenzung
KETTE2: DEFM    'Ende'          ; eigentlicher Text
        DEFB    00H             ; Ende durch Markierung
```

Es sei noch bemerkt, daß Hochsprachen-Compiler von dieser Form mitunter Gebrauch machen. Für den Assembler-Programmierer ist sie eher uninteressant, sollte aber der Vollständigkeit halber erwähnt werden.

Grundsätzlich können alle Formen von Zeichenketten als Byte-Felder interpretiert werden. Die Inhalte der Feldelemente haben aber je nach gewählter Form verschiedene Bedeutungen.

7.4.2 Kopieren von Zeichenketten und Längenberechnungen

Da Zeichenketten im Prinzip Felder von Zeichen sind, können sie mit den Verfahren für Byte-Felder kopiert werden. Man muß allerdings dazu die Länge des Feldes kennen. Bei Zeichenketten fester Länge ist diese bekannt, man kopiert sie am besten mit Hilfe der Befehle LDDR oder LDIR. Ein Beispiel für das Kopieren einer Zeichenkette der festen Länge 6 wäre deshalb (HL zeigt auf den Anfang der Zeichenkette, DE auf den neuen Ablageort):

```
KOPIE:  LD      BC,6    ; Anzahl der zu transportierenden Bytes
        LDIR            ; Transport durchfuehren
```

Beim Kopieren einer Zeichenkette mit Längenangabe muß man sich die aktuelle Länge der Zeichenkette erst holen, bevor der eigentliche Kopiervorgang gestartet werden kann. Für Zeichenketten, deren Längenangabe ein Byte benötigt, würde das so aussehen:

```
KOPIE:  LD      C,(HL)  ; Laenge des Textes holen
        LD      B,0     ; und zu Wort ergaenzen
        INC     BC      ; Laengenangabe mitzaehlen
        LDIR            ; Transport durchfuehren
```

Wichtig ist, daß die Längenangabe mitkopiert wird. Deswegen muß hier beim Transport ein Byte mehr angegeben werden, als im Text der Zeichenkette enthalten ist.

Beim Kopieren einer Zeichenkette mit Ende-Markierung ist keine Länge bekannt. Das Ende der Zeichenkette kann nur durch Überprüfung aller Zeichen vom Anfang der Zeichenkette her gefunden werden. Deshalb benötigt man einen völlig anderen Kopiermechanismus:

```
ENDE    EQU     0DH             ; Ende-Markierung,
                                ; beliebige Anpassung moeglich
KOPIE:  LD      A,(HL)          ; Zeichen holen
        LD      (DE),A          ; Zeichen kopieren
        INC     HL              ; auf naechstes Zeichen zeigen
        INC     DE              ; auf naechsten freien
                                ; Speicherplatz zeigen
        CP      ENDE            ; kopiertes Zeichen mit
```

```
                                ; Ende-Markierung vergleichen
        JP      NZ,KOPIE        ; alle Zeichen einschliesslich
                                ; der Ende-Markierung kopieren
```

Auch hier ist es wichtig, die Ende-Markierung mitzukopieren.

Wurde das Textende durch Setzen von Bit 7 im letzten Zeichen markiert, so vergleicht man nicht auf ein bestimmtes Ende-Zeichen, sondern testet den Wert von Bit 7 des eben kopierten Zeichens (durch einen geeigneten logischen Befehl oder durch einen Rotationsbefehl):

```
KOPIE:  LD      A,(HL)          ; Zeichen holen
        LD      (DE),A          ; Zeichen kopieren
        INC     HL              ; auf naechstes Zeichen zeigen
        INC     DE              ; auf naechsten freien
                                ; Speicherplatz zeigen
        RLA                     ; Bit 7 des Zeichens testen
        JP      NC,KOPIE        ; alle Zeichen kopieren
```

Beim Kopieren einer Zeichenkette mit Längenbegrenzung und Ende-Markierung müssen zwei Testkriterien gleichzeitig angewendet werden: Überschreitung der Maximallänge oder Erreichen des Ende-Zeichens. Dies wird am besten in Form einer Schleife mit Abbruch programmiert:

```
ENDE    EQU     0DH             ; Ende-Markierung,
                                ; beliebige Anpassung moeglich
MAXL    EQU     80              ; Maximallaenge,
                                ; beliebige Anpassung moeglich

        LD      B,MAXL          ; Maximalzahl von
                                ; zu kopierenden Zeichen
KOPIE:  LD      A,(HL)          ; Zeichen holen
        LD      (DE),A          ; Zeichen kopieren
        INC     HL              ; auf naechstes Zeichen zeigen
        INC     DE              ; auf naechsten freien
                                ; Speicherplatz zeigen
        CP      ENDE            ; kopiertes Zeichen mit
                                ; Ende-Markierung vergleichen
        JP      Z,WEITER        ; Ende, falls Ende-Markierung
                                ; gefunden und kopiert
        DJNZ    KOPIE           ; Ende, falls Maximallaenge
                                ; ueberschritten
WEITER: :                       ; gemeinsame Fortsetzungsstelle
        :
```

Neben dem Kopieren ist auch das Bestimmen der Länge einer Zeichenkette eine häufig benötigte Funktion. Das ist nicht die Anzahl der Bytes, die sie im Speicher belegt, sondern die Anzahl der Zeichen des eigentlichen Textes. Für Zeichenketten fester Länge ist diese stets bekannt und braucht nicht bestimmt zu werden. Für Zeichenketten mit Längenangabe holt man die Länge einfach aus dem ersten Byte (beziehungsweise aus den ersten beiden Bytes) der Zeichenkette. Für die übrigen Formen muß eine Suchschleife angelegt werden, zum Beispiel für Zeichenketten mit Ende-Markierung (HL soll wieder auf den Anfang der Zeichenkette zeigen):

```
ENDE    EQU     0DH             ; Ende-Markierung,
                                ; beliebige Anpassung moeglich

        LD      A,ENDE          ; Ende-Markierung zum Vergleichen
        LD      B,0             ; Zaehler fuer Laenge
SUCHE:  CP      (HL)            ; Zeichen auf Ende-Markierung
                                ; testen
        JP      Z,FERTIG        ; Ende-Markierung gefunden
        INC     B               ; Zaehler erhoehen
        INC     HL              ; auf naechstes Zeichen zeigen
        JP      SUCHE           ; alle Zeichen untersuchen
FERTIG: :                       ; Fortsetzungsstelle
        :
```

Für Zeichenketten mit Ende-Markierung im Bit 7 des letzten Zeichens läuft das Verfahren analog. Für Zeichenketten mit Längenbegrenzung und Ende-Markierung muß der Algorithmus etwas modifiziert werden:

```
ENDE    EQU     0DH             ; Ende-Markierung,
                                ; beliebige Anpassung moeglich
MAXL    EQU     80              ; Maximallaenge
```

```
        LD      A,ENDE          ; Ende-Markierung zum Vergleichen
        LD      B,MAXL          ; Maximallaenge von Zeichen
        LD      C,0             ; Zaehler fuer Laenge
SUCHE:  CP      (HL)            ; Zeichen auf Ende-Markierung
                                ; testen
        JP      Z,FERTIG        ; Ende-Markierung gefunden
        INC     C               ; Zaehler erhoehen
        INC     HL              ; auf naechstes Zeichen zeigen
        DJNZ    SUCHE           ; alle Zeichen untersuchen
FERTIG: :                       ; gemeinsame Fortsetzungsstelle
        :
```

Die Zeichenketten mit fester Länge besitzen gegenüber den Zeichenketten mit Längenangabe keine besonders gravierenden Unterschiede. Die Algorithmen sehen deshalb auch fast gleich aus. Deswegen wird in den folgenden Abschnitten nicht weiter auf Zeichenketten fester Länge eingegangen. Die Algorithmen für Zeichenketten mit Längenbegrenzung und Ende-Markierung lassen sich leicht aus denen für Zeichenketten mit Längenbegrenzung und für Zeichenketten mit Ende-Markierung ableiten. Die Zeichenketten mit Ende-Markierung im Bit 7 des letzten Zeichens unterscheiden sich nicht sehr stark von denen mit expliziter Ende-Markierung.

Im folgenden werden daher nur noch zwei verschiedene Formen von Zeichenketten betrachtet: Zeichenketten mit Längenangabe und Zeichenketten mit expliziter Ende-Markierung.

7.4.3 Suchen in Zeichenketten und Vergleichsoperationen

Unter dem Begriff „Suchen in Zeichenketten“ sind drei eng zusammengehörende Operationen zusammengefaßt:

1. Feststellen, ob ein bestimmtes Zeichen in einer Zeichenkette enthalten ist.
2. Index des ersten Zeichens einer Zeichenkette liefern, das mit einem bestimmten Zeichen übereinstimmt.
3. Zeiger auf das erste Zeichen einer Zeichenkette liefern, das mit einem bestimmten Zeichen übereinstimmt.

Vorweg sei bemerkt, daß die dritte Aufgabe beim Behandeln der ersten Aufgabe quasi nebenbei erledigt wird.

Die Suche in Zeichenketten mit Längenangabe führt man am besten mit dem Befehl CPIR (compare, increment and repeat; vergleichen, erhöhen und wiederholen) durch. Dieser Befehl vergleicht den Inhalt des Akkumulators mit dem Inhalt der Speicherzelle, die durch das Registerpaar HL indirekt adressiert wird. Die Flags werden wie durch den Befehl CP (HL) gesetzt. Anschließend wird das Registerpaar HL um 1 erhöht und das Registerpaar BC – das als Zähler dient – um 1 verringert. Falls der neue Wert von BC ungleich Null ist und beim Vergleich das Z-Flag zurückgesetzt wurde (das Zeichen der Zeichenkette war vom Testzeichen verschieden), so wird der Vorgang wiederholt.

Der CPIR-Befehl realisiert also eine Zählschleife (mit BC als Zählgröße), die abgebrochen wird, sobald das gewünschte Zeichen gefunden wurde. Ob die Suche erfolgreich war, läßt sich am Zustand des Z-Flags erkennen: gesetztes Z-Flag bedeutet gefunden, zurückgesetztes Z-Flag bedeutet nicht gefunden.

Somit wäre die erste Aufgabe gelöst. Bei erfolgreicher Suche braucht nur noch das Registerpaar HL um 1 verringert zu werden, damit es auf das gefundene Zeichen zeigt; schon ist auch die dritte Aufgabe gelöst. Ist man auch am Index des gefundenen Zeichens interessiert, so muß der Wert im Registerpaar BC von der Länge der Zeichenkette abgezogen werden (Indizies für Zeichenketten werden normalerweise ab 1 gezählt). Es können alle drei Aufgaben also durch folgendes Programm gelöst werden, das als Längenangabe ein Wort erwartet (HL zeigt auf die Zeichenkette, A enthält das zu suchende Zeichen):

```
        LD      C,(HL)          ; Laenge der Zeichenkette
        INC     HL              ; holen und
        LD      B,(HL)          ;
        INC     HL              ; auf Text zeigen
        LD      E,C             ; Laenge kopieren
        LD      D,B             ;
        CPIR                    ; Text absuchen
        SET     7,A             ; Zustand des Z-Flags
        JP      Z,NULL          ; in Bit 7 des
        RES     7,A             ; Akkumulators sichern
NULL:   DEC     HL              ; auf gefundenes Zeichen zeigen
        EX      DE,HL           ; Index des gefundenen
        OR      A               ; Zeichens berechnen
```

```
        SBC     HL,BC           ;
        EX      DE,HL           ;
```

Der Algorithmus funktioniert allerdings nur für nicht-leere Zeichenketten. Der Zustand des Z-Flags wurde in Bit 7 des Akkumulators gesichert, HL zeigt auf das gesuchte Zeichen und DE enthält den Index.

Außer dem CPIR-Befehl gibt es noch den Befehl CPDR (compare, decrement and repeat; vergleichen, verringern und wiederholen), der das Registerpaar HL in jedem Schritt um 1 verringert, sonst aber genauso wirkt.

Schwieriger wird es, wenn man Zeichenketten mit Ende-Markierung absucht. Hier muß man nach zwei Zeichen gleichzeitig sehen: Such-Zeichen und Ende-Zeichen. Eine automatische Wiederholung des Vergleichs durch den CPIR-Befehl kommt somit nicht in Frage. Hierzu baut man sich am besten selbst eine Schleife auf, bei der ein Schleifenzähler entfallen kann. Das Registerpaar HL zeigt wieder auf die Zeichenkette, im Register C befindet sich das Testzeichen. Am Ende der Suche soll das Z-Flag wieder das Ergebnis mitteilen, HL auf das gefundene Zeichen zeigen und DE den Index beinhalten:

```
ENDE    EQU     0DH             ; Ende-Markierung

        LD      DE,1            ; Index
SUCHE:  LD      A,(HL)          ; Zeichen holen
        CP      C               ; mit Testzeichen vergleichen
        JP      Z,WEITER        ; Zeichen gefunden
        INC     HL              ; auf naechstes Zeichen zeigen
        INC     DE              ; naechsten Index berechnen
        CP      ENDE            ; Zeichen mit Ende-Markierung
                                ; vergleichen
        JP      NZ,SUCHE        ; weitersuchen
        CP      C               ; Z-Flag loeschen
WEITER: :                       ; gemeinsame Fortsetzungsstelle
        :
```

Die Aufgabe kann man nun so modifizieren, daß nicht das erste Zeichen gesucht werden soll, das mit einem bestimmten Zeichen übereinstimmt, sondern das i-te Zeichen. Dazu wird in die Programme noch eine Zählschleife eingebaut. Die Initialisierung wird jeweils nur einmal durchgeführt. Zunächst für die Zeichenketten mit Ende-Markierung (die Anzahl i wird im Register B erwartet):

```
ENDE    EQU     0DH             ; Ende-Markierung

        LD      DE,1            ; Index
SUCHE:  LD      A,(HL)          ; Zeichen holen
        CP      C               ; mit Testzeichen vergleichen
        JP      NZ,NICHT        ; Zeichen nicht gefunden
        DEC     B               ; restliche Anzahl errechnen
        JP      Z,WEITER        ; Zeichen i-mal gefunden
NICHT:  INC     HL              ; auf naechstes Zeichen zeigen
        INC     DE              ; naechsten Index berechnen
        CP      ENDE            ; Zeichen mit Ende-Markierung
                                ; vergleichen
        JP      NZ,SUCHE        ; weitersuchen
        CP      C               ; Z-Flag loeschen
WEITER: :                       ; gemeinsame Fortsetzungsstelle
        :
```

Nun das entsprechende Programm für Zeichenketten mit Längenangabe (dabei werden zwei Hilfsvariablen benötigt):

```
; Datenbereich

HZEICH: DEFS    1       ; Hilfs-Speicherplatz
                        ; fuer Testzeichen
HZAEHL: DEFS    1       ; Hilfs-Speicherplatz
                        ; fuer Zaehlgroesse

; Programmbereich

        LD      (HZEICH),A      ; Testzeichen sichern
        LD      A,B             ; Zaehlgroesse
        LD      (HZAEHL),A      ; sichern
        LD      C,(HL)          ; Laenge der Zeichenkette
        INC     HL              ; holen und
        LD      B,(HL)          ;
        INC     HL              ; auf Text zeigen
        LD      E,C             ; Laenge kopieren
        LD      D,B             ;
SUCHE:  LD      A,(HZEICH)      ; Testzeichen holen
```

```
        CPIR                    ; Text absuchen
        JP      NZ,ENDE         ; Suche wird abgebrochen,
                                ; da Zeichenkette zu Ende
        LD      A,(HZAEHL)      ; Zaehlgroesse holen
        DEC     A               ; restliche Anzahl berechnen
        LD      (HZAEHL),A      ; Zaehlgroesse wieder sichern
        JP      NZ,SUCHE        ; Suche fortsetzen
NULL:   DEC     HL              ; auf gefundenes Zeichen zeigen
        EX      DE,HL           ; Index des
        OR      A               ; gefundenen Zeichens
        SBC     HL,BC           ; berechnen
        EX      DE,HL           ;
        XOR     A               ; Z-Flag setzen
ENDE:   :                       ; gemeinsame Fortsetzungsstelle
        :
```

Nun noch ein kompliziertes Problem: nämlich die Frage, ob eine Zeichenkette eine bestimmte andere Zeichenkette (deren Länge größer Null ist) enthält. Diese wird dann eine Teil-Zeichenkette (engl. substring) genannt.

Es wird davon ausgegangen, daß die beiden Zeichenketten in derselben Form vorliegen, daß HL auf die abzusuchende Zeichenkette zeigt und DE auf die Teil-Zeichenkette. Als erstes wird in der Zeichenkette nach dem ersten Zeichen der Teil-Zeichenkette gesucht. Wird dieses nicht gefunden, so ist die Frage bereits negativ beantwortet. Ansonsten wird der Zeiger auf dieses Zeichen gemerkt und die folgenden Zeichen mit dem Rest der Teil-Zeichenkette verglichen. Stimmen sie überein, so ist die Frage positiv beantwortet. Wenn nicht, wird der gesicherte Zeiger geholt und ab dieser Stelle erneut nach dem ersten Zeichen gesucht. Irgendwann bricht der Prozeß ab, weil die Teil-Zeichenkette gefunden wird, oder weil das Ende der Zeichenkette erreicht ist.

Für Zeichenketten mit Längenangabe:

```
; Datenbereich

ZEIGER: DEFS    2       ; Speicherplatz fuer Zeiger hinter
                        ; erstes gefundenes Zeichen
TEXT2:  DEFS    2       ; Speicher fuer Zeiger auf
                        ; eigentlichen Text der
                        ; Teil-Zeichenkette
LAENGE: DEFS    2       ; Speicherplatz fuer
                        ; Restlaenge der Zeichenkette
LAENG2: DEFS    2       ; Speicherplatz fuer
                        ; Restlaenge der Teil-Zeichenkette

; Programmbereich

        EX      DE,HL           ; Zeiger tauschen
        LD      C,(HL)          ; Laenge der
        INC     HL              ; Teil-Zeichenkette
        LD      B,(HL)          ; beschaffen und
        INC     HL              ; Restlaenge
        DEC     BC              ; in Variable
        LD      (LAENG2),BC     ; abspeichern
        LD      (TEXT2),HL      ; Zeiger auf eigentlichen
                                ; Text der Teil-Zeichenkette
                                ; sichern
        EX      DE,HL           ; Zeiger wieder tauschen
        LD      C,(HL)          ; Laenge der Zeichenkette
        INC     HL              ; beschaffen und auf
        LD      B,(HL)          ; eigentlichen Text zeigen
        INC     HL              ;
SUCHE:  LD      A,(DE)          ; erstes Zeichen der
                                ; Teil-Zeichenkette holen
        CPIR                    ; in Zeichenkette danach suchen
        JP      NZ,ENDE         ; Ende der Zeichenkette
                                ; erreicht, Teil-Zeichenkette
                                ; nicht enthalten
        LD      (ZEIGER),HL     ; Zeiger auf naechstes
                                ; Zeichen sichern
        LD      HL,(LAENG2)     ; Restlaenge der
        OR      A               ; Teil-Zeichenkette mit
        SBC     HL,BC           ; Restlaenge der Zeichen-
                                ; kette vergleichen
        JP      Z,WEITER        ; Laengen gleich?
        JP      NC,ENDE         ; Rest der Teil-Zeichenkette
                                ; laenger als Rest der
                                ; Zeichenkette, Teil-Zeichenkette
```

```
                                ; also nicht enthalten
                                ; Z-Flag ist zurueckgesetzt
WEITER: LD      (LAENGE),BC     ; Restlaenge der
                                ; Zeichenkette sichern
        LD      BC,(LAENG2)     ; Restlaenge der
                                ; Teil-Zeichenkette holen
        INC     DE              ; auf Rest der
                                ; Teil-Zeichenkette zeigen
        LD      HL,(ZEIGER)     ; Zeiger auf Rest der
                                ; Zeichenkette holen
VERGL:  LD      A,B             ; pruefen, ob
        OR      C               ; Teil-Zeichenkette abgearbeitet
        JP      Z,GEFUND        ; wenn ja fertig
        LD      A,(DE)          ; Test-Zeichen holen
        CPI                     ; und mit Zeichen vergleichen
        INC     DE              ; auf naechstes
                                ; Test-Zeichen zeigen
        JP      Z,VERGL         ; wenn Zeichen = Testzeichen,
                                ; Vergleich fortsetzen
        LD      HL,(ZEIGER)     ; Zeiger auf Rest der
                                ; Zeichenkette holen
        LD      DE,(TEXT2)      ; Zeiger auf Text der
                                ; Teil-Zeichenkette holen
        LD      BC,(LAENGE)     ; Restlaenge der
                                ; Zeichenkette holen
        JP      SUCHE           ; Suche von neuem beginnen
GEFUND: LD      HL,(ZEIGER)     ; Zeiger auf erstes
        DEC     HL              ; gefundenes Zeichen berechnen
ENDE:   :                       ; gemeinsame Fortsetzungsstelle
        :
```

Nun dasselbe für Zeichenketten mit Ende-Markierung:

```
; Datenbereich

ZEIGER: DEFS    2       ; Speicherplatz fuer Zeiger hinter
                        ; erstes gefundenes Zeichen
TEXT2:  DEFS    2       ; Speicher fuer Zeiger auf
                        ; Teil-Zeichenkette

; Programmbereich

ENDE    EQU     0DH             ; Ende-Markierung
        EX      DE,HL           ; Zeiger tauschen
        LD      (TEXT2),HL      ; Zeiger auf
                                ; Teil-Zeichenkette sichern
SUCHE:  LD      A,(DE)          ; Zeichen holen
        CP      (HL)            ; mit Testzeichen vergleichen
        INC     DE              ; auf naechstes Zeichen zeigen
        JP      Z,ZGEF          ; Zeichen stimmen ueberein
        CP      ENDE            ; auf Ende der
                                ; Zeichenkette pruefen
        JP      NZ,SUCHE        ; Zeichenkette nicht zu Ende
NGEF:   CP      (HL)            ; Z-Flag loeschen
        JP      FERTIG          ; zum Fortsetzungspunkt
ZGEF:   LD      (ZEIGER),DE     ; Zeiger auf naechstes
                                ; Zeichen sichern
        INC     HL              ; auf Rest der
                                ; Teil-Zeichenkette zeigen
VERGL:  LD      A,(HL)          ; pruefen, ob
        CP      ENDE            ; Teil-Zeichenkette abgearbeitet,
        JP      Z,GEFUND        ; wenn ja fertig
        LD      A,(DE)          ; Zeichen holen
        CP      (HL)            ; mit Testzeichen vergleichen
        INC     DE              ; auf naechstes Zeichen zeigen
        INC     HL              ; auf naechstes Testzeichen zeigen
        JP      Z,VERGL         ; wenn Zeichen = Testzeichen,
                                ; Vergleich fortsetzen
        LD      DE,(ZEIGER)     ; Zeiger auf Rest der
                                ; Zeichenkette holen
        LD      HL,(TEXT2)      ; Zeiger auf
                                ; Teil-Zeichenkette holen
        JP      SUCHE           ; Suche von neuem beginnen
GEFUND: LD      HL,(ZEIGER)     ; Zeiger auf erstes
        DEC     HL              ; gefundenes Zeichen berechnen
ENDE:   :                       ; gemeinsame Fortsetzungsstelle
        :
```

Im ersten Such-Algorithmus wurde der Befehl CPI (compare and increment; vergleichen und erhöhen) benutzt. Er entspricht dem CPIR-Befehl jedoch ohne automatische Wiederholung. (vgl. auch LDI/LDIR)

In den beiden Algorithmen kam bereits die Vergleichsoperation zwischen Zeichenketten vor. Dies läßt sich auf lexikalisches Vergleichen erweitern. So wie man auf Zahlen die Relationen $<$, $\leq$, $>$, $\geq$, $=$ und $\neq$ anwenden kann, läßt sich auch auf auf eine Reihe von Zeichenketten eine Anordnung einführen. Die Kleiner-Relation geht in die Relation „steht lexikalisch vor" über; die anderen Relationen lassen sich entsprechend ableiten.

7.4.4 Zeichenketten ausschneiden und anfügen

Bisher wurden bei der Arbeit mit Zeichenketten diese nicht verändert. Die Stärke der Zeichenketten-Verarbeitung liegt aber darin, aus Zeichenketten Teile herauszunehmen und zu neuen Zeichenketten zusammenzusetzen. Zunächst soll das Anfügen einer Zeichenkette an eine andere näher betrachtet werden.

Bei Zeichenketten mit Längenangabe müssen dabei die Summe der beiden alten Längen berechnet und die eigentlichen Texte aneinandergehängt werden. Es wird mit Längenangaben vom Typ „Wort" gearbeitet, wobei davon ausgegangen wird, daß die neue Länge wieder als Wort dargestellt werden kann, und daß hinter der einen Zeichenkette auch Platz für die Zeichen der anderen Zeichenkette ist. Das Registerpaar DE enthält einen Zeiger auf die Zeichenkette, an die angefügt werden soll, das Registerpaar HL enthält einen Zeiger auf die rechts davon anzufügende Zeichenkette:

```
; Datenbereich

ZEIGER: DEFS    2       ; Speicherplatz fuer
                        ; Zeiger auf Text der rechten
                        ; Zeichenkette
LAENGE: DEFS    2       ; Speicherplatz fuer
                        ; Laenge der linken Zeichenkette

; Programmbereich

        LD      C,(HL)          ; Laenge der
        INC     HL              ; rechten Zeichenkette
        LD      B,(HL)          ; holen und auf
        INC     HL              ; eigentlichen Text zeigen
        LD      A,B             ; auf leere Zeichenkette
        OR      C               ; pruefen
        JP      Z,FERTIG        ; rechte Zeichenkette leer,
                                ; nichts zu tun
        LD      (ZEIGER),HL     ; Zeiger auf Text der rechten
                                ; Zeichenkette sichern
        EX      DE,HL           ; Zeiger tauschen
        LD      E,(HL)          ; Laenge der
        INC     HL              ; linken Zeichenkette
        LD      D,(HL)          ; holen
        EX      DE,HL           ; Zeiger wieder tauschen
        LD      (LAENGE),HL     ; Laenge der linken
                                ; Zeichenkette sichern
        ADD     HL,BC           ; neue Laenge berechnen
        EX      DE,HL           ; Zeiger auf Gesamtlaenge
                                ; beschaffen
        LD      (HL),D          ; neue Laenge
        DEC     HL              ; eintragen
        LD      (HL),E          ;
        INC     HL              ; auf eigentlichen Text der
        INC     HL              ; linken Zeichenkette zeigen
        LD      DE,(LAENGE)     ; Laenge der linken
                                ; Zeichenkette holen
        ADD     HL,DE           ; hinter das Ende der linken
                                ; Zeichenkette zeigen
        EX      DE,HL           ; Zeiger sichern
        LD      HL,(ZEIGER)     ; Zeiger auf Text der rechten
                                ; Zeichenkette holen
        LDIR                    ; Text anfuegen
FERTIG: :                       ; gemeinsame Fortsetzungsstelle
        :
```

Wesentlich einfacher gestaltet sich das Anfügen, wenn die Zeichenketten mit Ende-Markierung versehen sind (auch hier wird genügend Platz vorausgesetzt):

```
ENDE    EQU     0DH             ; Ende-Markierung

SUCHE:  LD      A,(DE)          ; Zeichen der linken
```

```
                                ; Zeichenkette holen
        INC     DE              ; auf naechstes Zeichen der
                                ; linken Zeichenkette zeigen
        CP      ENDE            ; Zeichen mit Ende-Markierung
                                ; vergleichen
        JP      NZ,SUCHE        ; bis hinter das Ende der
                                ; linken Zeichenkette gehen
        DEC     DE              ; Ende-Markierung der linken
                                ; Zeichenkette wird ueberschrieben
KOPIE:  LD      A,(HL)          ; Zeichen der rechten
                                ; Zeichenkette holen
        LD      (DE),A          ; Zeichen an linke
                                ; Zeichenkette anf"ugen
        INC     HL              ; auf naechstes Zeichen der
                                ; rechten Zeichenkette zeigen
        INC     DE              ; auf naechstes Zeichen der
                                ; linken Zeichenkette zeigen
        CP      ENDE            ; Zeichen mit Ende-Markierung
                                ; vergleichen
        JP      NZ,KOPIE        ; weiter kopieren
```

Nun jedoch zu den Funktionen, mit denen Teile einer Zeichenkette isoliert werden. Dabei sind prinzipiell zwei Zugänge zu unterscheiden: die Teil-Zeichenkette kann durch Indizies bezeichnet werden; sie kann aber auch durch Zeiger gegeben sein. Im folgenden wird auf jegliche Fehlerbehandlungen verzichtet. Es ist jedoch eine gute Übung, in einige Programme eine solche einzubauen.

Ganz allgemein wird für alle folgenden Überlegungen vorausgesetzt, daß eine Zeichenkette bearbeitet werden soll, ohne diese zu verändern. Die Teil-Zeichenkette ist also eine eigenständige Zeichenkette, für die an einer vorgegebenen Stelle Speicherplatz freigehalten wird. Es wird stets angenommen, daß HL auf die Zeichenkette und DE auf den für die Teil-Zeichenkette reservierten Speicherplatz zeigt. Alle anderen Parameter variieren je nach Funktion.

Die erste Funktion liefert die ersten n Zeichen einer Zeichenkette (von links gezählt). Die Zahl n soll im Registerpaar BC stehen. Für Zeichenketten mit Längenangabe ist dies ein simpler Kopiervorgang mit vorgeschalteter Ablage der Länge der Teil-Zeichenkette:

```
        EX      DE,HL           ; Zeiger tauschen
        LD      (HL),C          ; Laenge der
        INC     HL              ; Teil-Zeichenkette ablegen
        LD      (HL),B          ; und auf eigentlichen
        INC     HL              ; Text zeigen
        EX      DE,HL           ; Zeiger wieder tauschen
        INC     HL              ; auf eigentlichen Text der
        INC     HL              ; Zeichenkette zeigen
        LDIR                    ; Text kopieren
```

Bei Zeichenketten mit Ende-Markierung erfolgt erst das Kopieren und dann das Markieren des Textendes der Teil-Zeichenkette:

```
ENDE    EQU     0DH             ; Ende-Markierung

        LDIR                    ; Text kopieren
        EX      DE,HL           ; Zeiger tauschen
        LD      (HL),ENDE       ; Ende der Teil-Zeichenkette
                                ; markieren
```

Ein ähnliches Problem liegt vor, wenn das Ende der Teil-Zeichenkette durch einen Zeiger bezeichnet ist (dieser zeigt normalerweise auf das nächste Zeichen hinter der gewünschten Zeichenkette). Am einfachsten ist es, mit Hilfe des Zeigers die Anzahl der Zeichen zu berechnen, die kopiert werden sollen. Anschließend lassen sich die oben gezeigten Algorithmen verwenden. Für Zeichenketten mit Längenangabe (als Wort) erfolgt diese Berechnung durch das folgende Programmstück (Zeiger wird in BC erwartet; die Berechnung lautet also: $BC = BC - HL - 2$):

```
        DEC     BC
        DEC     BC
        LD      A,C
        SUB     L
        LD      C,A
        LD      A,B
        SBC     A,H
        LD      B,A
```

Für Zeichenketten mit Ende-Markierung lautet das entsprechende Programmstück (mit der Wirkung $BC = BC - HL$):

```
        LD      A,C
        SUB     L
        LD      C,A
        LD      A,B
        SBC     A,H
        LD      B,A
```

Die nächste Operation liefert die Teil-Zeichenkette, die aus den letzten n Zeichen einer Zeichenkette besteht (n soll wieder in BC stehen). Zuerst für Zeichenketten mit Längenangabe:

```
; Datenbereich

ZEIGER: DEFS    2       ; Speicherplatz fuer
                        ; Zeiger auf letztes Zeichen
                        ; der Teil-Zeichenkette

; Programmbereich

        EX      DE,HL           ; Zeiger tauschen
        LD      (HL),C          ; Laenge der
        INC     HL              ; Teil-Zeichenkette
        LD      (HL),B          ; ablegen
        ADD     HL,BC           ; Zeiger auf letztes Zeichen
        LD      (ZEIGER),HL     ; der Teil-Zeichenkette
                                ; berechnen und sichern
        EX      DE,HL           ; Zeiger wieder tauschen
        LD      E,(HL)          ; Laenge der
        INC     HL              ; Zeichenkette besorgen
        LD      D,(HL)          ;
        ADD     HL,DE           ; Zeiger auf letztes Zeichen
                                ; der Zeichenkette berechnen
        LD      DE,(ZEIGER)     ; Zeiger auf letztes Zeichen der
                                ; Teil-Zeichenkette restaurieren
        LDDR                    ; Text kopieren
```

Entsprechend für Zeichenketten mit Ende-Markierung:

```
ENDE    EQU     0DH             ; Ende-Markierung

        LD      A,ENDE          ; Ende-Markierung holen
SUCHE:  CP      (HL)            ; Zeichen mit
                                ; Ende-Markierung vergleichen
        INC     HL              ; auf naechstes Zeichen zeigen
        JP      NZ,SUCHE        ; bis zur Ende-Markierung suchen
KOPIE:  DEC     HL              ; auf Ende-Markierung der
                                ; Zeichenkette zeigen
        EX      DE,HL           ; Zeiger tauschen
        ADD     HL,BC           ; auf Ende-Markierung der
                                ; Teil-Zeichenkette zeigen
        EX      DE,HL           ; Zeiger wieder tauschen
        INC     BC              ; Ende-Markierung wird
                                ; ebenfalls kopiert
        LDIR                    ; Text kopieren
```

Ein ähnliches Problem ist es, die Teil-Zeichenkette ab dem n-ten Zeichen (einschließlich) zu bilden. Hier exemplarisch nur die Variante für Zeichenketten mit Ende-Markierung:

```
ENDE    EQU     0DH             ; Ende-Markierung

        ADD     HL,BC           ; Zeiger auf erstes zu
        DEC     HL              ; kopierendes Zeichen berechnen
KOPIE:  LD      A,(HL)          ; Zeichen holen
        LD      (DE),A          ; Zeichen kopieren
        INC     HL              ; auf naechstes Zeichen der
                                ; Zeichenkette zeigen
        INC     DE              ; auf naechstes Zeichen der
                                ; Teil-Zeichenkette zeigen
        CP      ENDE            ; Zeichen mit Ende-Markierung
                                ; vergleichen
        JP      NZ,KOPIE        ; bis einschliesslich der
                                ; Ende-Markierung kopieren
```

Zum Abschluß die Beschaffung der Teil-Zeichenkette, die vom m-ten Zeichen (einschließlich) bis zum n-ten Zeichen (einschließlich) reicht. Hier die Variante für Zeichenketten mit Längenangabe (m steht in BC, n in der Variablen GRENZE):

```
; Datenbereich

GRENZE: DEFS    2       ; End-Index n
ZEIGER: DEFS    2       ; Speicherplatz fuer
                        ; Zeiger auf Teil-Zeichenkette

; Programmbereich

        ADD     HL,BC           ; Zeiger auf
        INC     HL              ; Teil-Zeichenkette
        LD      (ZEIGER),HL     ; berechnen und sichern
        LD      HL,(GRENZE)     ; oberen Index holen
        OR      A               ; Laenge der
        SBC     HL,BC           ; Teil-Zeichenkette
        INC     HL              ; berechnen
        LD      C,L             ; Laenge kopieren
        LD      B,H             ;
        LD      HL,(ZEIGER)     ; Zeiger auf Teil-Zeichenkette
                                ; restaurieren
        EX      DE,HL           ; Zeiger tauschen
        LD      (HL),C          ; Laenge der
        INC     HL              ; Teil-Zeichenkette ablegen
        LD      (HL),B          ; und auf Text
        INC     HL              ; des Ablagebereichs zeigen
        EX      DE,HL           ; Zeiger wieder tauschen
        LDIR                    ; Text kopieren
```

7.4.5 Einfügen, Löschen und Ersetzen in Zeichenketten

Nun sollen Änderungen an den Zeichenketten selbst vorgenommen werden. Eine wichtige Operation ist das Einfügen einer Zeichenkette in eine andere Zeichenkette an einer vorgegebenen Stelle. Hierzu wird angenommen, daß die Einfügestelle durch einen Zeiger markiert ist, und zwar zeigt dieser auf den Speicherplatz, in den das erste Zeichen der einzufügenden Zeichenkette geschrieben werden soll.

Man hat es also mit drei Zeigern zu tun: HL zeigt auf die Zeichenkette, in die eingefügt werden soll, DE zeigt auf die einzufügende Zeichenkette, und BC zeigt auf die Stelle, an der eingefügt werden soll. Zuerst muß man für das Einfügen Platz in der Zeichenkette schaffen, indem man den Text ab dem Zeiger BC nach hinten verschiebt. Dann kann der Text der neuen Zeichenkette eingefügt werden. Das folgende Programmstück zeigt dies für Zeichenketten mit Längenangabe:

```
; Datenbereich

ZEIGDE: DEFS    2       ; 1 + Inhalt von DE
ZEIGBC: DEFS    2       ; Inhalt von BC
LAENG1: DEFS    2       ; Laenge der Zeichenkette,
                        ; in die eingefuegt wird
LAENG2: DEFS    2       ; Laenge der einzufuegenden
                        ; Zeichenkette

; Programmbereich

        LD      (ZEIGBC),BC     ; Zeiger sichern
        EX      DE,HL           ; Zeiger tauschen
        LD      C,(HL)          ; Laenge der
        INC     HL              ; einzufuegenden
        LD      B,(HL)          ; Zeichenkette holen
        LD      (LAENG2),BC     ; und sichern
        LD      A,B             ; Laenge auf
        OR      C               ; Null testen
        JP      Z,FERTIG        ; nichts einzufuegen
        LD      (ZEIGDE),HL     ; 1 + alten Inhalt
                                ; von DE sichern
        EX      DE,HL           ; Zeiger tauschen
        LD      E,(HL)          ; Laenge der Zeichenkette,
        INC     HL              ; in die eingefuegt wird,
        LD      D,(HL)          ; holen,
        EX      DE,HL           ; verfuegbar machen
        LD      (LAENG1),HL     ; und sichern
        ADD     HL,BC           ; neue Laenge berechnen
        EX      DE,HL           ; Zeiger holen
        LD      (HL),D          ; neue Laenge
        DEC     HL              ; abspeichern
        LD      (HL),E          ;
        LD      DE,(LAENG1)     ; Zeiger auf Ende
        INC     HL              ; der Zeichenkette, in die
```

```
         ADD     HL,DE           ; eingefuegt wird, berechnen
         LD      D,H             ; und sichern
         LD      E,L             ;
         INC     HL              ; Laenge des zu verschiebenden
         LD      BC,(ZEIGBC)     ; Textes der Zeichenkette, in
         OR      A               ; die eingefuegt wird,
         SBC     HL,BC           ; berechnen
         LD      B,H             ; und sichern
         LD      C,L             ;
         LD      HL,(LAENG2)     ; Zeiger auf neues Ende der
         ADD     HL,DE           ; Zeichenkette berechnen
         EX      DE,HL           ; Zeiger tauschen
         LD      A,B             ; Laenge des zu verschiebenden
         OR      C               ; Speicherbereichs auf 0 testen
         JP      Z,EINFUE        ; nichts zu verschieben
         LDDR                    ; Platz schaffen
EINFUE:  LD      HL,(ZEIGDE)     ; Zeiger auf Ende
         LD      BC,(LAENG2)     ; der einzufuegenden
         ADD     HL,BC           ; Zeichenkette berechnen
         LDDR                    ; Einfuegen
FERTIG:  :                       ; gemeinsame Fortsetzungsstelle
         :
```

Die zweite wichtige Operation ist das Löschen einer vorgegebenen Teil-Zeichenkette aus einer Zeichenkette. Im folgenden Beispiel wird davon ausgegangen, daß DE auf die Teil-Zeichenkette zeigt und BC auf das nächste Zeichen hinter der zu löschenden Teil-Zeichenkette. Es wird die Variante für Zeichenketten mit Ende-Markierung gezeigt (dazu benötigt man keinen Zeiger auf die Zeichenkette, in der gelöscht wird):

```
ENDE     EQU     ODH             ; Ende-Markierung

KOPIE:   LD      A,(BC)          ; Zeichen holen
         LD      (DE),A          ; Zeichen kopieren
         INC     BC              ; beide Zeiger
         INC     DE              ; weiterbewegen
         CP      ENDE            ; Zeichen auf Ende testen
         JP      NZ,KOPIE        ; war nicht Ende, bis
                                 ; einschliesslich Ende kopieren
```

Nun zum letzten Problem, dem Umcodieren von Zeichen einer Zeichenkette. Dazu wird davon ausgegangen, daß zu jedem ASCII-Zeichen mit 8 Bits ein anderes ASCII-Zeichen (seine Codierung) definiert ist, welches das ursprüngliche Zeichen in der Zeichenkette ersetzen soll. Die Codierung wird an einer Zeichenkette mit Ende-Markierung ausgeführt, wobei BC auf den Anfang der Zeichenkette zeigt, DE auf den Anfang der Codierungstabelle:

```
ENDE     EQU     ODH             ; Ende-Markierung

CODIER:  LD      A,(BC)          ; Zeichen holen
         CP      ENDE            ; mit Ende-Markierung vergleichen
         JP      Z,FERTIG        ; Ende erreicht, fertig
         LD      L,A             ; Zeichen zu
         LD      H,0             ; Relativadresse machen
         ADD     HL,DE           ; Zeiger auf Codezeichen
                                 ; berechnen
         LD      A,(HL)          ; Codezeichen holen
         LD      (BC),A          ; und Codierung ausfuehren
         INC     BC              ; auf naechstes Zeichen zeigen
         JP      CODIER          ; alle Zeichen codieren
FERTIG:  :                       ; Fortsetzungsstelle
         :
```

Die Codierungstabelle ist ein Feld von Zeichen mit 256 Elementen, etwa folgendermaßen:

```
CODE:    DEFB    6BH     ; Codierung fuer 00H
         DEFB    9FH     ; Codierung fuer 01H
         :
         :
         DEFB    38H     ; Codierung fuer FFH
```

7.5 Übungsaufgaben (I)

Aufgaben zu Abschnitt 7.1

1. Es soll ein Programmstück angesprungen werden, dessen Anfangsadresse relativ zum Inhalt des Registerpaares HL durch den Inhalt des Registerpaares BC gegeben ist (eine Form des relativen indirekten Sprungs).
2. Mit einem Programmstück soll die Summe von drei hintereinander im Speicher stehenden vorzeichenlosen ganzen 8-Bit-Zahlen gebildet und das Ergebnis ohne Berücksichtigung eines Übertrags unmittelbar hinter den drei Zahlen wieder als Byte abgelegt werden. Die Speicheradresse der ersten Zahl soll dabei im Registerpaar HL stehen.
3. Ein Programmstück soll geschrieben werden, das die Summe von zwei im Speicher stehenden ganzen Zahlen in Zweierkomplement-Darstellung mit 8 Bits bildet und das Ergebnis vor den beiden Zahlen als Byte (ohne Berücksichtigung eines Überlaufs) ablegt. Die Speicheradresse der ersten Zahl soll dabei im Registerpaar HL stehen.
4. Im Speicher stehen zwei Folgen von je fünf vorzeichenlosen ganzen 8-Bit-Zahlen, die jeweils fortlaufend abgespeichert sind. Die beiden Folgen sollen komponentenweise addiert werden; die fünf Resultate sollen ebenfalls als Folge von Bytes lückenlos gespeichert werden. Die drei Speicherbereiche liegen dabei nicht unbedingt beieinander. Es ist ein Programmstück zu schreiben, das dieses Problem mittels indirekter Adressierung löst.
5. Ein Datenelement vom Typ „Wort“ soll im Speicher um 30 Bytes nach hinten verschoben werden. Die Speicheradresse des Wortes soll dabei in einem Adreß-Register stehen.

Aufgaben zu Abschnitt 7.2

6. Mit einer Verzweigung soll realisiert werden, daß eine vorzeichenlose ganze Zahl im Register E genau dann um 1 vermindert wird, wenn diese ungerade ist.
7. Ein ASCII-Buchstabe, der sich im Register H befindet, soll untersucht werden. Ist der Buchstabe klein, dann soll der Wert 0 ins Register D gebracht werden; ist er dagegen groß, dann soll der Wert −1 betragen. Wie könnte ein entsprechendes Programmstück aussehen.
8. Eine Verzweigung ist zu schreiben, mit der festgestellt werden kann, ob eine durch das Registerpaar HL indirekt adressierte ganze Zahl positiv ist.
9. Es sollen zwei Programme geschrieben werden, die im Speicher stehende ASCII-codierte Groß- und Kleinbuchstaben vertauschen. Es sollen einmal Bit-Manipulations-Befehle und einmal Maskierungen zum Einsatz kommen.
10. Wie könnte ein Programmstück aussehen, mit dem der Zustand des C-Flags (Übertrag) in einem Bit des Registers D gespeichert wird?
11. Der Zustand des Z-Flags (Null) soll in Bit 6 einer durch das Registerpaar HL indirekt adressierten Speicherzelle aufbewahrt werden. Ein entsprechendes Programmstück ist zu konstruieren.
12. Durch Maskieren sollen ASCII-codierte Dezimalzahlen in ihre Binärdarstellung umgewandelt werden.
13. ASCII-Codierung und Binärdarstellung von Dezimalziffern sollen durch Maskieren vertauscht werden.
14. Ein im Akkumulator stehendes Byte soll in zwei Nibbles umgewandelt und diese als ASCII-codierte Hex-Ziffern im Register B und Register C abgelegt werden.
15. In den Registern D und E befindet sich je eine ASCII-codierte Hex-Ziffer. Im Akkumulator soll das Byte zusammengesetzt werden, dessen höherwertiger Nibble im Register D und dessen niederwertiger Nibble im Register E codiert ist.
16. Die Register B, C, D und E sollen in dieser Reihenfolge als „32-Bit-Superregister“ aufgefaßt werden. Es soll je ein Programm für die arithmetische Rechts-Verschiebung, arithmetische Links-Verschiebung und logische Rechts-Verschiebung dieses Superregisters geschrieben werden.
17. Die Programme aus Aufgabe 14 und 15 sollen so modifiziert werden, daß statt des Akkumulators eine durch das Registerpaar HL indirekt adressierte Speicherzelle benutzt wird.

Aufgaben zu Abschnitt 7.3.1

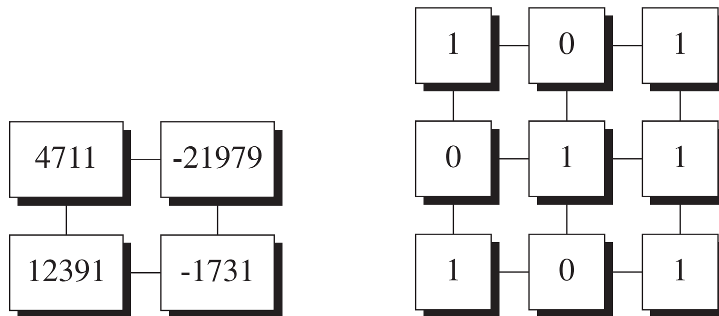
18. Folgende uninitialisierte eindimensionale Felder sollen vereinbart werden:

kleinster Index	größer Index	Basistyp/Länge
1	10	Wort
0	15	Byte
-5	5	4 Bytes
1	25	Bit
0	12	Nibble
0	31	2 Bits

19. Folgende initialisierte eindimensionale Felder sollen vereinbart werden:

Typ	Vektor
Byte	$(8, -13, 17, 99, -121, 44)$
Wort	$(12380, 16421, 246, -13131)$
Bit	(0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0)
Nibble	(3, A, 7, 0, B, F, 1, 4, D)

20. Folgende Matrizen sollen einmal als zeilenorientiertes und einmal als spaltenorientiertes Feld vereinbart werden:



Wieviel Speicherplatz benötigen die Matrizen? Für die spaltenorientierte Wort-Matrix soll eine Vereinbarung mit Deskriptor angegeben werden!

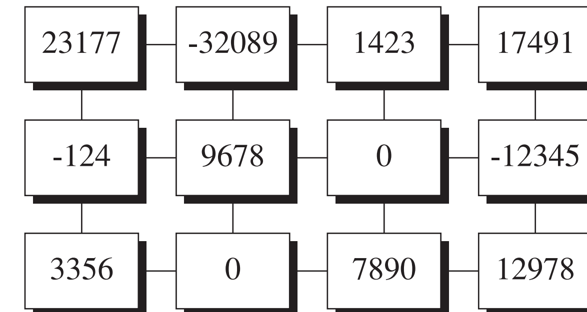
Aufgaben zu Abschnitt 7.3.2

21. Die Routine ADRESS aus dem Abschnitt 7.3.2 soll auf ganzzahlige Indizies, die in Zweierkomplement im Akkumulator geliefert werden (andere Größen entsprechend anpassen), erweitert werden.

22. Die Routine ADRESS aus dem Abschnitt 7.3.2 soll auf vorzeichenlose ganzzahlige Indizies, die im Registerpaar HL geliefert werden (andere Größen entsprechend anpassen), erweitert werden.

23. Wie könnte ein Deskriptorformat für eindimensionale Felder aussehen, so daß an die Routine ADRESS aus dem Abschnitt 7.3.2 nur die Adresse des Deskriptors und der Wert des Index übergeben werden muß? Die Routine ADRESS soll dementsprechend umgeschrieben und um eine Indexgrenzüberwachung ergänzt werden.

24. Für die folgende Wort-Matrix soll eine Adressier-Routine geschrieben werden:



Aufgaben zu Abschnitt 7.3.3

25. Ein Byte-Feld mit dem Indexbereich 0 – 255 soll als Ganzes bearbeitet werden; das i-te Element soll dabei den Wert i erhalten.

26. Es wird ein Programm benötigt, das zwei Wort-Felder mit derselben Anzahl von Elementen miteinander vergleicht und die Adressen der ersten nicht übereinstimmenden Elemente liefert.

27. Mit einem Programm soll der letzte Nibble eines Nibble-Feldes mit dem Wert 0FH gesucht werden. Dabei soll keine Feldgrenze mit einer Byte-Grenze übereinstimmen.

28. Durch ein Bit-Feld wurde eine Pixelgrafik realisiert, die nun invertiert werden soll. Die Feldgrenzen fallen nicht unbedingt mit Byte-Grenzen zusammen.

Aufgaben zu Abschnitt 7.3.4

29. Der Wert des letzten Feldelements eines Byte-Feldes soll in alle übrigen Feldelemente kopiert werden.

30. Die Elemente eines Byte-Feldes sollen von rückwärts solange in ein zweites Byte-Feld kopiert werden, bis eine Null erscheint.

Aufgaben zu Abschnitt 7.4.1

31. Folgende Zeichenketten sollen in den verschiedenen Formen der Darstellung implementiert werden (die spitzen Klammern kennzeichnen dabei ASCII-Steuerzeichen):

    - Eingabe
    - Guten Tag!
    - Seite 4<CR><LF>
    - (leere Zeichenkette)

    Als konstante Länge soll 9 verwendet werden, 00H bzw. 0DH als Ende-Markierung und 10 als Längenbegrenzung. Die jeweilige Größe des benötigten Speicherplatzes ist ebenfalls anzugeben!

Aufgaben zu Abschnitt 7.4.2

32. Ein Programm für das Kopieren von Zeichenketten mit Längenangabe ist zu schreiben, wobei die Längenangabe ein Wort belegt.

33. Zeichenketten mit Längenbegrenzung und Ende-Markierung sollen kopiert werden, bei denen die maximale Länge auch größer als 255 sein kann. Wie sieht das entsprechende Programm aus?

34. Mit welchem Programm läßt sich die Länge einer Zeichenkette mit Längenbegrenzung und Ende-Markierung berechnen, deren maximale Länge auch größer als 255 sein darf?

Aufgabe zu Abschnitt 7.4.3

35. Es ist ein Programm zu schreiben, das feststellt, ob ein bestimmtes Zeichen in einer Zeichenkette mit Längenbegrenzung und Ende-Markierung enthalten ist. Dabei soll der Befehl CPI benutzt werden.

Aufgaben zu Abschnitt 7.4.4

36. Mit Hilfe eines Programmes soll die Teil-Zeichenkette ab dem n-ten Zeichen einer Zeichenkette mit Längenangabe gebildet werden.

37. Aus einer Zeichenkette mit Ende-Markierung soll die Teil-Zeichenkette ab dem m-ten Zeichen bis einschließlich des n-ten Zeichens gebildet werden.

Aufgaben zu Abschnitt 7.4.5

38. Ein Programm soll für das Einfügen in eine Zeichenkette mit Ende-Markierung geschrieben werden.

39. Eine Teil-Zeichenkette soll mittels Programm aus einer Zeichenkette mit Längenangabe gelöscht werden.

7.6 Mengen

Eine Menge (engl. set) ist eine Zusammenfassung von Elementen. In einem Computer sind normalerweise nur Elemente desselben Typs zulässig. Beispiele für Mengen wären eine Zahlenmenge (0, 1, ..., 255), eine Buchstabenmenge (A, B, ..., Z), eine Menge von Namen (Müller, Meier, Schulze, Lehmann) oder Farbmenge (Rot, Grün, Blau, Schwarz, Weiß). Die beiden erstgenannten Mengen legen die Darstellung als Byte nahe, die übrigen müssen erst sinnvoll codiert werden.

Man muß grundsätzlich zwischen einer Menge und ihrer Repräsentation unterscheiden. Jede Menge kann nämlich – wie im folgenden gezeigt wird – mehrere Repräsentationen haben. Prinzipiell sind auch Repräsentationen möglich, in denen ein Element der dargestellten Menge auch mehrfach vorkommen darf. Darauf soll jedoch im weiteren verzichtet werden, da dies zu einer Aufblühung der Datenobjekte führt und in manchen Fällen auch die Algorithmen komplizierter werden.

Durch die Angabe einer Ordnungsrelation ist die Definition einer Reihenfolge der Elemente möglich. Ein Vorteil bei der Verarbeitung einer Repräsentation ergibt sich jedoch erst dann, wenn die Repräsentation diese Ordnung widerspiegelt. Man spricht dann von einer geordneten Repräsentation, wenn auf Menge und Repräsentation eine gemeinsame Ordnungsrelation definiert ist, ansonsten von einer ungeordneten Repräsentation.

Aus den Elementen einer Menge können neue, kleinere Mengen gebildet werden, sogenannte Teilmengen (engl. subsets). Im folgenden geht es immer um die Darstellung dieser Teilmengen – kurz Mengen genannt –, während die Grundmenge als fest und bekannt angesehen wird. Für den Umgang mit Mengen gibt es einige Standardoperationen, aus denen kompliziertere Operationen aufgebaut werden können. Diese Standardoperationen sollen nun zuerst einmal beschrieben werden:

Um überhaupt eine Menge aufbauen zu können, muß man in der Lage sein, ein einzelnes Element in eine schon bestehende Menge einzufügen. Außerdem muß man die *leere Menge* aufbauen können. Das ist eine Menge, in der sich kein Element befindet.

Weiterhin ist es sinnvoll, wenn man prüfen kann, ob sich ein bestimmtes Element in der Menge befindet.

Das Entfernen eines bestimmten Elements aus der Menge kann auf Schwierigkeiten treffen, wenn dieses Element gar nicht in der Menge enthalten ist. Deshalb muß zwischen zwei Formen des Entfernens unterschieden werden: das Entfernen eines vorhandenen Elements beziehungsweise der Versuch, ein bestimmtes Element zu entfernen. Auf jeden Fall befindet sich das Element hinterher nicht in der Menge.

Manchmal muß in Erfahrung gebracht werden, wieviele Elemente sich in der Menge befinden. Diese Anzahl nennt man die *Kardinalität* der Menge.

Zwei vorhandene Mengen kann man auf Gleichheit testen. Sie sind genau dann gleich, wenn sie dieselben Elemente enthalten. Ein spezieller Fall liegt vor, wenn man feststellen will, ob eine Menge die leere Menge ist.

Eine Menge ist eine Teilmenge einer anderen Menge, wenn sie ausschließlich Elemente aus dieser enthält. Zu einer Menge kann man das Komplement bilden. Das ist diejenige Menge, die genau die Elemente der Grundmenge enthält, die nicht in der ursprünglichen Menge enthalten waren.

Die *Vereinigung* zweier Mengen enthält alle Elemente, die in mindestens einer der beiden Mengen vorkommen. Ähnlich wie die Vereinigung funktioniert die *symmetrische Differenz* zweier Mengen. In ihr befinden sich nur diejenigen Elemente, die genau in einer der beiden Mengen vorkommen.

Die *Schnittmenge* zweier Mengen enthält nur die Elemente, die in beiden Mengen zugleich vorkommen. Die (gewöhnliche) *Differenz* zweier Mengen – hier kommt es auf die Reihenfolge der beiden Mengen an – enthält genau die Elemente der ersten Menge, die nicht in der zweiten Menge liegen.

Keine Operation, sondern eine Klasse von Operationen, bildet die Anweisung, eine bestimmte Aktion auf jedem einzelnen Element einer Menge auszuführen (zum Beispiel das Ausdrucken einer Menge, das Bilden der Summe einer Zahlenmenge, das Berechnen einer Farbmischung aus einer Menge von Farben).

Zu dieser Vielfalt von Operationen auf Mengen kommt noch eine große Freiheit in der Darstellung von Mengen. Primär werden zwei Formen der Darstellung unterschieden: Werden alle Elemente in irgendeiner Form aufgelistet, ihre Werte also gesammelt, so spricht man von einer Darstellung als Auflistung. In der zweiten Form wird jeder Menge ein spezielles Bit-Feld zugeordnet, der sogenannte Inzidenzvektor.

7.6.1 Darstellung durch Auflistung

Für die Auflistung der Elemente einer Menge gibt es verschiedene Möglichkeiten. Zunächst kann man die Form eines Feldes von Elementen wählen, wobei die Anzahl der Feldelemente die Kardinalität der Menge ist. Diese muß also flexibel sein, weshalb

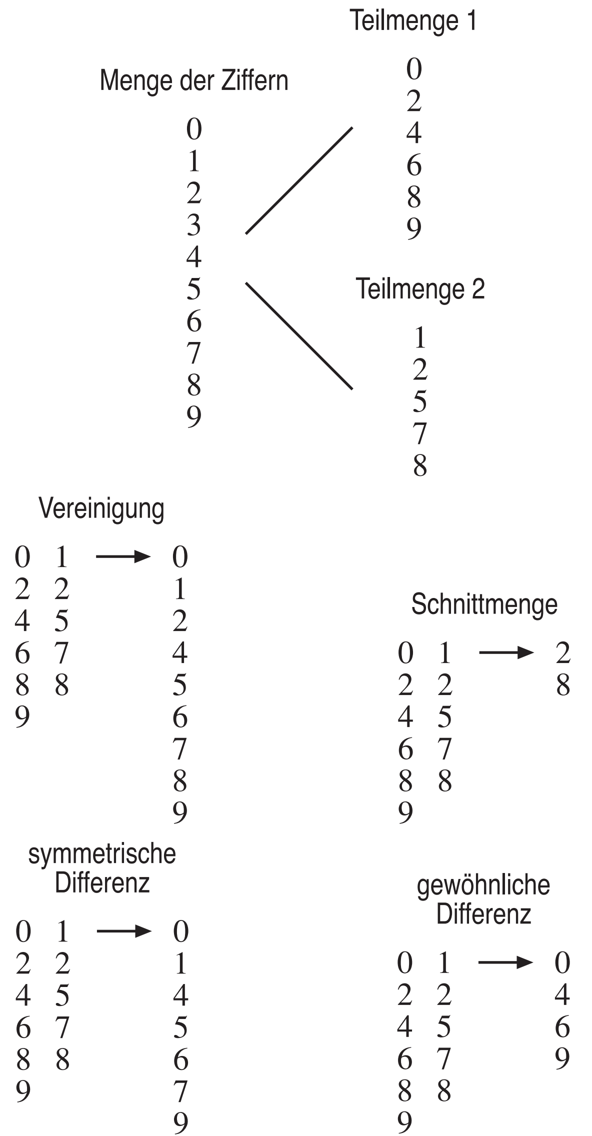


Bild 11: Operationen mit Mengen

ein Feld mit Deskriptor oder eine Tabelle (siehe Abschnitt 7.6) angemessen ist. Es kann aber auch die Form einer Liste gewählt werden.

Bei all diesen Darstellungen kann noch zwischen geordneter oder ungeordneter Auflistung gewählt werden, wobei die geordnete meist Vorteile bringt. Nachfolgend einige Hinweise für die Umsetzung der Mengenoperationen in Algorithmen:

Die leere Menge wird als Feld der Länge Null, leere Tabelle oder leere Liste dargestellt. Der Test auf leere Menge besteht dann im Prüfen der Länge. Die Kardinalität ist die Anzahl der Elemente der gewählten Struktur.

Beim Einfügen eines Elements muß (nach der oben getroffenen Festlegung) zuerst festgestellt werden, ob das Element schon vorhanden ist. In diesem Fall geschieht nichts. Ansonsten wird das neue Element angehängt (wenn ohne Anordnung gearbeitet wird) oder an der Stelle eingefügt, die der Anordnung entspricht (dies ist meist die Stelle, an der die Suche abgebrochen wurde). Die Kardinalität der Menge erhöht sich dabei um eins.

Beim Löschen eines Elements wird die gesamte Repräsentation durchsucht, falls keine Anordnung gegeben ist, ansonsten bricht die Suche meist früher ab. Ist das Element nicht in der Menge vorhanden, so bleibt die Operation ohne Wirkung. Anderenfalls wird das Element entfernt, wobei sich die Kardinalität um eins vermindert.

Die Prüfung, ob ein Element in der Menge vorhanden ist, besteht aus der Suche nach dem Element. Bei ungeordneten Repräsentationen ist die Suche einfacher als bei geordneten, erstreckt sich aber dafür auf die gesamte Repräsentation.

Am deutlichsten macht sich der Unterschied zwischen geordneten und ungeordneten Repräsentationen beim Vergleich zweier Mengen bemerkbar. Bei geordneten Repräsentationen geht man die Elemente simultan in gemeinsamer Reihenfolge durch, bis zwei Elemente differieren (die Mengen sind dann nicht gleich) oder beide Repräsentationen abgearbeitet sind (die Mengen stimmen überein). Bei ungeordneten Repräsentationen muß man dagegen für jedes Element der einen Menge prüfen, ob es in der anderen Menge enthalten ist. In beiden Fällen empfiehlt es sich, zuerst die Kardinalität der beiden Mengen zu vergleichen.

Bei gleichgroßen geordneten Repräsentationen mit n Elementen müssen höchstens n Vergleiche (wenn die beiden Mengen übereinstimmen), bei ungeordneten Repräsentationen mindestens n, unter Umständen aber sogar $n * (n + 1)/2$ Vergleiche durchgeführt werden. Schon für ein kleines n (z.B. $n = 10$) macht sich der Unterschied stark bemerkbar.

Für die Relation „Teilmenge von“ gilt ein ähnliches Argument, da auch hier geprüft werden muß, ob jedes Element der einen Menge in der anderen enthalten ist.

Das Komplement einer Menge wird gebildet, indem für jedes Element der Grundmenge festgestellt wird, ob es in der Menge enthalten ist (dann wird es weggelassen) oder nicht (dann kommt es in die Komplement-Menge).

Die Vereinigung zweier Mengen wird gebildet, indem sukzessive die Elemente der einen Menge zu der anderen Menge hinzugegeben wird. Dabei ist jedoch darauf zu achten, daß in der Repräsentation kein Element doppelt vorkommt. Bei der symmetrischen Differenz wird das Element sogar ganz weggenommen, wenn es in beiden Mengen vorkommt.

Die Schnittmenge zweier Mengen ermittelt man, indem sukzessive für jedes Element der einen Menge festgestellt wird, ob es auch in der anderen Menge vorkommt.

Bei der Bildung der Differenz zweier Mengen geht man sukzessive die Elemente der zweiten Menge durch und nimmt diese aus der ersten Menge fort, falls sie darin vorkommen.

Beim Ausführen einer bestimmten Aktion auf allen Elementen einer Menge kann diese für geordnete und ungeordnete Repräsentationen gleichermaßen in beliebiger Reihenfolge durchlaufen werden. Für die genaue Ausführung der Algorithmen sei an dieser Stelle auf die Abschnitte „Tabellen“ und „Verzeigerte Strukturen“ verwiesen. Im allgemeinen sind geordnete Repräsentationen den ungeordneten vorzuziehen, wenn die Kardinalität der typischerweise auftretenden Mengen einen Wert von 10 oder mehr annimmt.

7.6.2 Darstellung mit Hilfe von Inzidenzvektoren

Ein Inzidenzvektor einer Menge ist ein Bit-Feld. Jedes Element der Grundmenge korrespondiert zu genau einem Feldelement (es handelt sich also um eine geordnete Repräsentation). Das Element besitzt den Wert 1, wenn das Element in der Menge vorkommt, den Wert 0, wenn das Element nicht in der Menge vorkommt.

Inzidenzvektoren lassen sich recht einfach manipulieren. Dadurch schleichen sich weniger Fehler in die Algorithmen ein, außerdem benötigt man meist weniger Zeit für die Bearbeitung. Ob Inzidenzvektoren verwendet werden können, hängt von der Kardinalität der Grundmenge ab. Ist diese groß, so brauchen die Inzidenzvektoren sehr viel Speicherplatz, selbst wenn die auftretenden Mengen sehr klein sind. Ein Inzidenzvektor benötigt immer denselben Speicherplatz, unabhängig von der Menge, die er darstellt.

Als Beispiel soll die Menge der Zeichen (256 Elemente = 32 Byte Inzidenzvektor) betrachtet werden, für die Inzidenzvektoren noch tragbar sind; im Gegensatz zu Worten (2^{16} Elemente = 8 KByte Inzidenzvektor), wo ein einzelner Inzidenzvektor ein Achtel des gesamten Adreßraums des Z80 benötigen würde. Ein zweites Kriterium ist die Kardinalität der auftretenden Mengen. Ist diese sehr klein (im Beispiel mit den Zeichen vielleicht 10 Zeichen), so ist die Verwendung von Inzidenzvektoren aufwendiger als die Auflistung.

Im folgenden soll als Beispiel einer Grundmenge die Menge der Werte eines Nibbles (0 bis 15) gewählt werden. Jeder Inzidenzvektor belegt dann ein Wort. Es wird mit der Herstellen der leeren Menge begonnen, wobei HL auf den Inzidenzvektor zeigt:

```
XOR     A               ; Akkumulator loeschen
LD      (HL),A          ; 8 Elemente loeschen
INC     HL              ; auf die naechsten
                        ; 8 Elemente zeigen
LD      (HL),A          ; 8 Elemente loeschen
```

Es folgt der Test auf das Vorliegen der leeren Menge. Wenn HL auf die leere Menge zeigt, soll das Z-Flag gesetzt werden:

```
LD      A,(HL)          ; 8 Elemente laden
INC     HL              ; auf die naechsten
                        ; 8 Elemente zeigen
OR      (HL)            ; mit 8 Elementen verknuepfen
```

Als nächstes wird die Gleichheit zweier Mengen getestet, auf die HL beziehungsweise DE zeigen. Bei Gleichheit soll das Z-Flag gesetzt werden:

```
LD      A,(DE)          ; 8 Elemente holen
CP      (HL)            ; 8 Elemente testen
JP      NZ,FERTIG       ; Mengen sind verschieden
INC     HL              ; auf die naechsten
INC     DE              ; 8 Elemente zeigen
LD      A,(DE)          ; 8 Elemente holen
```

```
        CP      (HL)            ; 8 Elemente testen
FERTIG: :                       ; gemeinsame Fortsetzungsstelle
        :
```

Nun wird das Komplement einer Menge gebildet. Das Registerpaar HL zeigt auf die Menge, das Registerpaar DE auf das Komplement:

```
        LD      A,(HL)          ; 8 Elemente holen
        CPL                     ; Komplement bilden
        LD      (DE),A          ; 8 Elemente abspeichern
        INC     HL              ; auf die naechsten
        INC     DE              ; 8 Elemente zeigen
        LD      A,(HL)          ; 8 Elemente holen
        CPL                     ; Komplement bilden
        LD      (DE),A          ; 8 Elemente abspeichern
```

Die Vereinigung zweier Mengen wird durch die OR-Verknüpfung gebildet (HL beziehungsweise DE zeigen auf die beiden Mengen, BC auf die Vereinigung):

```
        LD      A,(DE)          ; 8 Elemente holen
        OR      (HL)            ; Vereinigung bilden
        LD      (BC),A          ; 8 Elemente abspeichern
        INC     HL              ; auf die naechsten
        INC     DE              ; 8 Elemente
        INC     BC              ; zeigen
        LD      A,(DE)          ; 8 Elemente holen
        OR      (HL)            ; Vereinigung bilden
        LD      (BC),A          ; 8 Elemente abspeichern
```

Die symmetrische Differenz wird hingegen mittels XOR-Verknüpfung realisiert:

```
        LD      A,(DE)          ; 8 Elemente holen
        XOR     (HL)            ; symmetrische Differenz bilden
        LD      (BC),A          ; 8 Elemente abspeichern
        INC     HL              ; auf die naechsten
        INC     DE              ; 8 Elemente
        INC     BC              ; zeigen
        LD      A,(DE)          ; 8 Elemente holen
        XOR     (HL)            ; symmetrische Differenz bilden
        LD      (BC),A          ; 8 Elemente abspeichern
```

Ebenso einfach erhält man durch die AND-Verknüpfung den Schnitt zweier Mengen:

```
        LD      A,(DE)          ; 8 Elemente holen
        AND     (HL)            ; Schnittmenge bilden
        LD      (BC),A          ; 8 Elemente abspeichern
        INC     HL              ; auf die naechsten
        INC     DE              ; 8 Elemente
        INC     BC              ; zeigen
        LD      A,(DE)          ; 8 Elemente holen
        AND     (HL)            ; Schnittmenge bilden
        LD      (BC),A          ; 8 Elemente abspeichern
```

Bei der Bildung der Differenz einer Menge A mit einer anderen Menge B $(A - B)$ ist zu beachten, daß die resultierende Menge auch als Schnittmenge von A mit dem Komplement von B dargestellt werden kann. Zeigt HL auf die Menge A, DE auf die Menge B, BC auf die Differenzmenge, so lautet das Programm:

```
        LD      A,(DE)          ; 8 Elemente holen
        CPL                     ; Komplement bilden
        AND     (HL)            ; Schnittmenge bilden
        LD      (BC),A          ; 8 Elemente abspeichern
        INC     HL              ; auf die naechsten
        INC     DE              ; 8 Elemente
        INC     BC              ; zeigen
        LD      A,(DE)          ; 8 Elemente holen
        CPL                     ; Komplement bilden
        AND     (HL)            ; Schnittmenge bilden
        LD      (BC),A          ; 8 Elemente abspeichern
```

Soll festgestellt werden, ob eine Menge A Teilmenge einer Menge B ist, so kann man testen, ob der Schnitt von A und B mit A übereinstimmt. HL zeigt auf die Menge A, DE auf die Menge B; ist A Teilmenge von B, so soll das Z-Flag gesetzt werden:

```
        LD      A,(DE)          ; 8 Elemente holen
        AND     (HL)            ; Schnittmenge bilden
        CP      (HL)            ; 8 Elemente vergleichen
        JP      NZ,FERTIG       ; A nicht Teilmenge von B
        INC     HL              ; auf die naechsten
        INC     DE              ; 8 Elemente zeigen
        LD      A,(DE)          ; 8 Elemente holen
        AND     (HL)            ; Schnittmenge bilden
        CP      (HL)            ; 8 Elemente vergleichen
FERTIG: :                       ; gemeinsame Fortsetzungsstelle
        :
```

Für die restlichen Operationen bleibt nichts anderes übrig, als die einzelnen Elemente der Menge, das heißt die Elemente des Bit-Feldes zu bearbeiten. Für das Hinzufügen eines Elements muß das entsprechende Bit gesetzt werden; für das Entfernen muß es gelöscht werden; um festzustellen, ob ein bestimmtes Element in der Menge enthalten ist, wird das entsprechende Bit getestet. Alle drei Techniken wurden bereits im Abschnitt 7.3.2 genau beschrieben.

Zur Berechnung der Kardinalität einer Menge werden die von ihr belegten Bytes rotiert und die herausgeschobenen Bits, die den Wert 1 tragen, gezählt. HL zeigt auf die Menge, A enthält anschließend die Kardinalität:

```
        XOR     A               ; Akkumulator loeschen
        LD      C,2             ; Anzahl der Bytes
BYTES:  LD      B,8             ; Anzahl der Bits pro Byte
BITS:   RRC     (HL)            ; Indikator herausschieben
        ADC     A,0             ; und aufaddieren
        DJNZ    BITS            ; alle Bits einbeziehen
        INC     HL              ; auf die naechsten
                                ; 8 Elemente zeigen
        DEC     C               ; restliche Anzahl von Bytes
        JP      NZ,BYTES        ; alle Bytes einbeziehen
```

Will man auf jedem der Elemente eine Aktion durchführen, so wird ähnlich vorgegangen. Zunächst wird wieder das entsprechende Bit herausgeschoben, anschließend getestet und dann die Aktion durchgeführt, wenn es gesetzt ist. Als Beispiel wird die Summe der Elemente einer Menge von Zahlen gebildet:

```
        XOR     A               ; Akkumulator loeschen
        LD      E,A             ; naechstes Element der Menge
        LD      C,2             ; Anzahl der Bytes
BYTES:  LD      B,8             ; Anzahl der Bits pro Byte
BITS:   RRC     (HL)            ; Element herausschieben
        JP      NC,WEITER       ; Element nicht in der Menge
        ADD     A,E             ; Element aufaddieren
WEITER: INC     E               ; naechstes Element der Menge
        DJNZ    BITS            ; alle Bits einbeziehen
        INC     HL              ; auf die naechsten
                                ; 8 Elemente zeigen
        DEC     C               ; restliche Anzahl von Bytes
        JP      NZ,BYTES        ; alle Bytes einbeziehen
```

7.7 Verbunde

Ein Verbund (engl. record) ist eine Zusammenfassung von Daten (möglicherweise verschiedenen Typs) zu einem Datensatz. Die Anzahl der Komponenten eines Verbunds und ihre Typen sind durch die Definition der Datenstruktur festgelegt. Anders als bei Feldern kann man die Adressen der Komponenten eines Verbunds nicht durch eine einfache Funktion angeben.

Die einfachste Form eines Verbunds ist der ungepackte Verbund. Dabei beginnen alle Komponenten – unabhängig von ihrem tatsächlichen Speicherbedarf – an einer Byte-Grenze. Es wird dadurch möglicherweise Speicherplatz verschwendet (wenn nämlich eine Komponente eine nicht durch 8 teilbare Anzahl von Bits benötigt). Die Komponenten sind meist, wenn sie nicht sowieso eine durch 8 teilbare Anzahl von Bits belegen, links (seltener rechts) mit Null-Bits aufgefüllt.

Die Adresse des ersten belegten Bytes eines Verbunds nennt man seine Basisadresse; die Adresse einer einzelnen Komponente errechnet sich als Summe der Basisadresse und der jeweiligen Relativadresse der Komponente. Ungepackte Verbunde lassen sich leicht manipulieren.

Ein gepackter Verbund belegt genau soviel Speicherplatz, wie alle einzelnen Komponenten zusammen. Die Komponenten beginnen nicht unbedingt an Byte-Grenzen. Bei der Adressierung sind deshalb Byte- und Bit-Adressen zu berücksichtigen, was die Bearbeitung kompliziert werden läßt. Gepackte Verbunde verwendet man häufig, wenn die Komponenten Bits oder Nibbles sind.

Bei Verbunden mit Varianten (engl. variant records) sind die Typen einzelner Komponenten von sogenannten Diskriminatoren abhängig. Die Typen der Komponenten gehen damit aus dem Speicherinhalt des Verbunds hervor. Verbunde mit Varianten können als gepackte oder ungepackte Verbunde auftreten.

7.7.1 Ungepackte Verbunde

Als Beispiel für einen ungepackten Verbund sei im folgenden eine Karteikarte eines Unternehmens betrachtet. Die Komponenten sollen folgendermaßen beschaffen sein:

Vorname	20 Zeichen
Name	20 Zeichen
Geburtsdatum	6 Zeichen
Monatsverdienst	13 Bits
Abteilung	3 Bits

Der Monatsverdienst benötigt zwar nur 13 Bits, da aber ein ungepackter Verbund aufgebaut werden soll, belegt er 2 volle Bytes. Ebenso werden für die 3 Bits der Komponente Abteilung ein volles Byte verwendet. Eine Speicherdefinition eines uninitialisierten Verbunds obigen Typs hat damit folgende Form:

```
VORNAM: DEFS    20      ; Vorname
NAME:   DEFS    20      ; Name
GEBURT: DEFS    6       ; Geburtsdatum
VERDIE: DEFS    2       ; Monatsverdienst
ABTEIL: DEFS    1       ; Abteilung
```

Ein Beispiel für einen initialisierten Verbund obigen Typs wäre:

```
VORNAM: DEFM    'Otto                ' ; Vorname
NAME:   DEFM    'Huber               ' ; Name
GEBURT: DEFM    '260448'               ; Geburtsdatum
VERDIE: DEFW    3264                   ; Monatsverdienst
ABTEIL: DEFB    3                      ; Abteilung
```

Soll dieser Verbund bearbeitet werden, so könnte man die indirekte Adressierung mittels eines der Registerpaare BC, DE, HL benutzen. Übergibt man beispielsweise die Basisadresse im HL-Registerpaar und will den Monatsverdienst ins DE-Registerpaar holen, so würde die Adressierroutine lauten:

```
        LD      DE,46           ; Relativadresse des
                                ; Monatsverdienstes
        ADD     HL,DE           ; Adresse des
                                ; Monatsverdienstes
        LD      E,(HL)          ; Monatsverdienst
        INC     HL              ; ins DE-Registerpaar
        LD      D,(HL)          ; holen
```

Bereits an diesem kleinen Beispiel werden einige Mängel der Adressierung von Verbunden mittels der Registerpaare sichtbar:

Der Zeiger auf den Verbund wird – wenn man ihn nicht explizit sichert – durch die Adressierung verändert.

Zur Berechnung der Adresse einer Komponente werden arithmetische und/oder inkrementierende bzw. dekrementierende Befehle benötigt. Bei Verwendung arithmetischer Befehle wird neben dem Daten-Adreß-Register ein weiteres Registerpaar belegt.

Wird auf mehrere Komponenten zugegriffen, so hängt die Berechnung der Adresse einer Komponente von der zuvor ausgeführten Operation ab. Als Programmierer verliert man bei komplexen Vorgängen rasch den Überblick. Wird die Struktur des Verbunds geändert, so muß das Programm im Regelfall neu geschrieben werden.

Gemeinsame Verwendung eines Programmstücks für die Adressierung von Komponenten eines Verbunds durch mehrere Programme ist nur unter erschwerten Bedingungen zu realisieren.

Eine Abhilfe schaffen hier die Indexregister IX und IY. Ein Indexregister zeigt (während eines komplexen Adressierungsvorgangs) stets auf eine feste Adresse. (Meist ist dies die Basisadresse eines Verbunds.) Die Adressierung einer Komponente eines Verbunds geschieht dabei durch Angabe der Relativadresse der Komponente (relativ zum Wert des Indexregisters). Die Relativadresse muß im Bereich −80H bis +7FH liegen.

Obiges Beispiel würde mit Hilfe des Indexregisters IX folgendermaßen lauten, wenn IX zu Beginn auf den Anfang des Verbunds zeigt:

```
        LD      E,(IX+46)
        LD      D,(IX+47)
```

Das Programmstück ist entschieden kürzer als das zuvor mittels HL-Registerpaar programmierte. (Allerdings trügt hier der Schein etwas, denn der Objekt-Code des

zweiten Programms belegt immerhin auch 6 Bytes. Im Gegensatz dazu belegt die erste Variante 8 Bytes.)

Insbesondere wenn hintereinanderliegende Bytes eines Verbunds bearbeitet werden, kann die Adressierung durch Indexregister mehr Objekt-Code belegen als die Adressierung durch ein Registerpaar. Trotzdem ist die Adressierung durch Indexregister übersichtlicher, weniger fehleranfällig, konsequenter und gut modifizierbar. In der Adressierung mehrerer Komponenten sind niemals Abhängigkeiten der Adreßberechnung enthalten.

Die Indexregister IX und IY verhalten sich völlig gleich. Sie können in fast allen Befehlen, die das HL-Registerpaar benutzen, dieses ersetzen. Für den Prozessor wird diese Substitution durch ein vorangestelltes Byte (DDH für IX, FDH für IY) gekennzeichnet. Dadurch wächst natürlich der Umfang des Objekt-Codes. Bei Verwendung der Indexregister zur indirekten Adressierung der Daten kommt noch die Relativadresse hinzu, die ebenfalls ein Byte belegt.

Nun noch ein paar Betrachtungen zu zwei häufig vorkommenden Anwendungen von ungepackten Verbunden: Deskriptoren und Kontrollblöcke.

Deskriptoren sind bereits von den Feldern bekannt. Nur war an dieser Stelle noch nicht klar, daß es sich um Verbunde handelt. Eine typische Anwendung sind Deskriptoren für Zeichenketten-Variablen in BASIC-Interpretern. Es wird im folgenden davon ausgegangen, daß der Variablenname aus zwei Zeichen besteht und die Länge der Zeichenketten durch ein Byte angegeben wird. Der Deskriptor sieht dann meist so aus:

```
NAME:   DEFS    2       ; Variablenname
LAENGE: DEFS    1       ; Laenge der Zeichenkette
ADRESS: DEFS    2       ; Adresse des Textes
```

Es soll nun die Länge der Zeichenkette ins B-Register und die Anfangsadresse des Textes ins Registerpaar HL gebracht werden, falls der Variablenname mit einem im DE-Registerpaar gegebenen Variablennamen übereinstimmt. Die Basisadresse des Deskriptors steht dabei im IY-Register:

```
        LD      A,D             ; erstes Zeichen
        CP      (IY+00H)        ; des Namens testen
        JP      NZ,FERTIG       ; Name verschieden
        LD      A,E             ; zweites Zeichen
        CP      (IY+01H)        ; des Namens testen
        JP      NZ,FERTIG       ; Name verschieden
        LD      B,(IY+02H)      ; Laenge des Textes holen
        LD      L,(IY+03H)      ; Adresse des
        LD      H,(IY+04H)      ; Textes holen
FERTIG: :                       ; gemeinsame Fortsetzungsstelle
```

Anhand des Z-Flags läßt sich erkennen, ob der Name übereinstimmt.

Ein Kontrollblock besteht aus Informationen über den Zustand eines externen Gerätes sowie über die Ansprechbarkeit spezieller Treiber. Als Beispiel wird ein Drucker-Kontrollblock mit folgender Struktur betrachtet:

```
NAME:   DEFS    2       ; Geraete-Name, spezifiziert
                        ; das angeschlossene Geraet
ZLAENG: DEFS    1       ; Zeilenlaenge
ZPOSIT: DEFS    1       ; aktuelle Position des
                        ; Druckkopfes in der Zeile
SLAENG: DEFS    1       ; Seitenlaenge
SPOSIT: DEFS    1       ; aktuelle Position des
                        ; Druckkopfes auf der Seite
TREIBA: DEFS    2       ; Adresse des Treiber-Programms
```

Zeigt IX auf den Kontrollblock, so bringt folgendes Programm die Koordinaten des Druckkopfes auf dem Papier ins Registerpaar DE:

```
ZPOS    EQU     ZPOSIT-NAME     ; Relativadresse der
                                ; Position in der Zeile
SPOS    EQU     SPOSIT-NAME     ; Relativadresse der
                                ; Position auf der Seite
        LD      E,(IX+ZPOS)     ; Position in der Zeile
        LD      D,(IX+SPOS)     ; Position auf der Seite
```

Soll die Struktur des Kontrollblocks geändert werden, so genügt eine Korrektur der Größen ZPOS und SPOS, um das Programm an die Änderung anzupassen.

7.7.2 Gepackte Verbunde

In einem gepackten Verbund folgen alle Komponenten so dicht wie nur möglich aufeinander, um so wenig Speicherplatz wie möglich zu belegen. Die Karteikarte aus dem vorhergehenden Abschnitt wird als gepackter Verbund realisiert:

```
VORNAM: DEFS    20       ; Vorname
NAME:   DEFS    20       ; Name
GEBURT: DEFS    6        ; Geburtsdatum
VERABT: DEFS    2        ; Monatsverdienst und Abteilung
```

Aus der Struktur des Verbunds kann man nicht mehr erkennen, wo die Komponente Monatsverdienst endet und die Komponente Abteilung beginnt. Dies wird erst wieder aus den Algorithmen klar. Das folgende Programm erwartet die Nummer der Abteilung im Register B und den Monatsverdienst im Registerpaar HL. Es packt die beiden Angaben und legt sie im Verbund ab, dessen Basisadresse im IX-Register steht:

```
        RR      B                       ; Bit 2 wird
        RR      B                       ; zuerst
        RR      B                       ; gebraucht
        ADC     HL,HL                   ; Bit 2 von B nach HL bringen
        RL      B                       ; Bit 1 von
        ADC     HL,HL                   ; B nach HL bringen
        RL      B                       ; Bit 0 von
        ADC     HL,HL                   ; B nach HL bringen
        LD      (IX+VERABT-VORNAM),L    ; die beiden gepackten
        LD      (IX+VERABT-VORNAM+1),H  ; Komponenten abspeichern
```

Ein ganz typischer Fall liegt vor, wenn die Komponenten eines Verbunds Bits sind. Als Beispiel wären hier Verbunde zu nennen, deren erste Komponente vom Typ Bit angibt, ob ein bestimmter Grafikpunkt gesetzt ist oder nicht, und deren zweite Komponente vom Typ Bit angibt, ob der Zustand des Grafikpunktes geändert werden darf. (So etwas nennt man ein Attribut des Punktes.) Die folgende Routine löscht alle Grafikpunkte eines Feldes (die Anzahl der Bytes des Feldes im Registerpaar DE, IX zeigt auf den Anfang des Feldes), die nicht durch das Attribut geschützt sind. (Die Verwendung der Indexregister ist für diese Routine nicht optimal.):

```
TEST:   LD      A,D             ; restliche Anzahl von Bytes
        OR      E               ; auf Null testen
        JP      Z,FERTIG        ; ganzes Feld bearbeitet
        LD      B,4             ; Anzahl der Punkte pro Byte
LOESCH: RLC     (IX+00H)        ; Attribut holen
        JP      C,GESCH         ; geschuetzter Punkt
        RES     7,(IX+00H)      ; Punkt loeschen
GESCH:  RLC     (IX+00H)        ; Punkt rotieren
        DJNZ    LOESCH          ; 4 Punkte bearbeiten
        INC     IX              ; auf naechstes Byte zeigen
        DEC     DE              ; restliche Anzahl von
                                ; Bytes berechnen
        JP      TEST            ; ganzes Feld abarbeiten
FERTIG: :                       ; Fortsetzungsstelle
```

7.7.3 Verbunde mit Varianten

Bisher wurde angenommen, daß der Typ einer Komponente eines Verbundes stets derselbe ist. Es kann aber durchaus vorkommen, daß ein Verbund Daten darstellen soll, deren Typen wechseln. Auch die Zahl der Komponenten variiert mitunter. Man gibt deshalb in diesen Fällen zu den eigentlichen Komponenten zusätzliche Komponenten – Diskriminatoren – hinzu, die bestimmen, welche Form der Verbund nun wirklich haben soll.

Hier ein Beispiel zur Beschreibung geometrischer Figuren. Für ein Rechteck werden zwei Bestimmungsgrößen (Länge und Breite) benötigt, jedoch nur eine für einen Kreis (Radius). Als ungepackter Verbund könnte dies folgendermaßen aussehen:

```
SELEKT: DEFS    1       ; Diskriminator
                        ; 0 steht fuer Kreis
                        ; 1 steht fuer Rechteck
RADIUS:
LAENGE: DEFS    2       ; Radius des Kreises bzw.
                        ; Laenge des Rechtecks
BREITE: DEFS    2       ; Breite des Rechtecks,
                        ; unbenutzt bei Kreisen
```

Je nach Wert des Diskriminators enthält die erste Komponente den Radius eines

Kreises oder die Länge eines Rechtecks. Die zweite Komponente wird nur für Rechtecke benutzt. Der von einem Verbund belegte Speicherplatz ist in der Regel stets gleich groß, unabhängig von der intern verwendeten Struktur. Durch die Überlagerung der beiden alternativen Komponenten Radius und Länge wird Speicherplatz eingespart. Ist eine Überlagerung nicht wünschenswert, so verwendet man Verbunde ohne Varianten, in denen der Diskriminator eine normale Komponente darstellt:

```
KREIS:  DEFB    0       ; Diskriminator fuer Kreise
RADIUS: DEFW    1200    ; Radius des Kreises
LEER:   DEFS    2       ; unbenutzte Komponente

RECHTE: DEFB    1       ; Diskriminator fuer Rechteck
LAENGE: DEFW    940     ; Laenge des Rechtecks
BREITE: DEFW    385     ; Breite des Rechtecks
```

Eine mögliche Aufgabe für die Bearbeitung einer solchen Struktur könnte darin bestehen, die Fläche der Figur (angenähert) auszurechnen. Das entsprechende Programm bringt für einen Kreis dessen Radius in das Registerpaar BC und setzt das Z-Flag; hingegen werden für ein Rechteck Länge und Breite in die Registerpaar HL und DE gebracht sowie das Z-Flag zurückgesetzt. IY soll die Basis-Adresse des Verbunds enthalten:

```
        XOR     A               ; Testgroesse bereitstellen
        OR      (IY+00H)        ; Diskriminator testen
        JP      Z,KREIS         ; Verbund stellt Kreis dar
        LD      L,(IY+01H)      ; Laenge des
        LD      H,(IY+02H)      ; Rechtecks holen
        LD      E,(IY+03H)      ; Breite des
        LD      D,(IY+04H)      ; Rechtecks holen
        JP      FERTIG          ; Aufgabe geloest
KREIS:  LD      C,(IY+01H)      ; Radius des
        LD      B,(IY+02H)      ; Kreises holen
FERTIG: :                       ; gemeinsame Fortsetzungsstelle
```

Nun ein Beispiel für Varianten mit gleicher Anzahl von Komponenten, aber unterschiedlicher Komponententypen: Von der Tastatur sollen vier Eingabezeichen geholt und anschließend überprüft werden, ob diese eine Speicheradresse in hexadezimaler Schreibweise darstellen. Ist dies der Fall, so werden die vier Zeichen in die dargestellte Adresse konvertiert. Ansonsten werden die Zeichen unverändert abgelegt und später als Name einer Variablen interpretiert. Die angemessene Struktur für das Problem ist folgender Verbund mit Variante:

```
SELEKT: DEFS    1       ; Diskriminator
                        ; 0 steht fuer Adresse
                        ; 1 steht fuer Name
ADRESS:
NAME:   DEFS    4       ; Name einer Variablen belegt
                        ; vier Bytes, Adresse einer
                        ; Variablen belegt die ersten
                        ; beiden Bytes, folgende Bytes
                        ; sind ungenutzt
```

In Assemblerschreibweise läßt sich die Struktur komplizierter Verbunde nicht sinnvoll ausdrücken. Eine korrekte Interpretation der Datenstruktur ist deshalb nur aus den Algorithmen und den Kommentaren zu ersehen.

Verbunde mit Varianten kann man ebenfalls packen, um Speicherplatz zu sparen. Das kann insbesondere deshalb sinnvoll sein, weil der Diskriminator normalerweise nur wenige Bits benötigt.

7.8 Tabellen

In konventioneller Darstellungsweise ist eine Tabelle (engl. table) ein rechteckförmiges Schema von Werten, wobei die Werte innerhalb einer Spalte vom selben Typ sind, während die Werte innerhalb einer Zeile in einem logischen Zusammenhang stehen. Im Computerbereich wird eine Zeile einer Tabelle manchmal auch Datensatz genannt.

Tabellen sind ungemein mächtige Datenstrukturen. Relationale Datenbanken sind vollständig aus Tabellen aufgebaut, in denen jeder Datensatz einem Eintrag in der Datenbank entspricht.

7.8.1 Implementierung von Tabellen

Die logische Strukturierung einer Tabelle in Datensätze legt nahe, eine Tabelle als Feld von Verbunden zu implementieren. Jeder Verbund stellt damit einen Datensatz

dar. Durch Kombination der für Felder und Verbunde erlernten Techniken gelangt man zu den verschiedenen Formen von Tabellen.

Eine Tabelle fester Länge kann ohne Deskriptor aufgebaut werden. Dazu ein Beispiel: In einer Tabelle sollen Länge, Breite und Dicke von Holzplatten für eine Möbelfabrik gespeichert werden. Jedes Produkt hat eine Produktnummer, die durch eine 16-Bit-Zahl codiert wird. Länge, Breite und Dicke werden in Millimetern als ganzzahlige 16-Bit- bzw. 8-Bit-Größen angegeben. Jeder Datensatz hat damit folgende Struktur:

Artikelnummer	2 Bytes
Länge	2 Bytes
Breite	2 Bytes
Dicke	1 Byte

Die Tabelle könnte folgendermaßen aussehen:

```
PLATTE:
        DEFW    329     ; Produktnummer
        DEFW    1800    ; Laenge
        DEFW    450     ; Breite
        DEFB    12      ; Dicke

        DEFW    2391    ; Produktnummer
        DEFW    1260    ; Laenge
        DEFW    235     ; Breite
        DEFB    14      ; Dicke

        :
        :

        DEFW    4138    ; Produktnummer
        DEFW    435     ; Laenge
        DEFW    240     ; Breite
        DEFB    8       ; Dicke
```

Die Länge der Tabelle wird irgendwo in den Zugriffsalgorithmen versteckt.

Eine weitere Möglichkeit, ohne Tabellendeskriptor auszukommen, besteht darin, hinter den letzten Eintrag einen Kennwert zu setzen, der sich von allen an dieser Stelle möglichen zulässigen Werten für Datensätze unterscheidet. Der Kennwert kann den Platz eines ganzen Verbundes einnehmen, er kann aber auch kürzer sein, zum Beispiel ein Byte oder ein Bit. Das folgende Beispiel zeigt eine Tabelle von Namen, jeweils aus Vorname und Familienname bestehend, die durch ein Null-Byte abgeschlossen ist:

```
NAMEN:
        DEFM    'Hans    '
        DEFM    'Mueller '
        DEFM    'Klaus   '
        DEFM    'Schulze '
        DEFM    'Heinrich'
        DEFM    'Lehmann '
        DEFB     0                ; Ende-Markierung der Tabelle
```

Als letzte Möglichkeit steht schließlich die Verwendung eines Deskriptors zur Wahl. Dieser kann beispielsweise unmittelbar vor dem ersten Tabelleneintrag stehen. Folgende Tabelle gibt eine Folge von Meßpunkten durch Paare von Meßwerten an; der Deskriptor besteht aus der Anzahl der Tabelleneinträge:

```
PUNKTE: DEFB    5       ; Anzahl der Tabellenelemente

        DEFW    14756
        DEFW    19334

        DEFW    12390
        DEFW    9534

        DEFW    34679
        DEFW    22879

        DEFW    17998
        DEFW    6879

        DEFW    56782
        DEFW    31255
```

Es kann durchaus vorkommen, daß die Datensätze nur aus je einem Wert bestehen, zum Beispiel wenn Mengen durch Tabellen dargestellt werden sollen. In diesem Fall entartet die Tabelle zum gewöhnlichen Feld.

7.8.2 Indizierter Zugriff auf Tabellen

Die Feldstruktur einer Tabelle kann man explizit ausnutzen, um mittels eines Index auf einen bestimmten Tabelleneintrag zuzugreifen. Man darf die Elemente der Tabelle dann natürlich nicht beliebig anordnen, weil sonst der Zusammenhang zwischen Index und Tabellenelement verlorengeht. Außerdem muß der Index des gesuchten Tabellenelements bekannt sein.

Als Beispiel soll der Zugriff auf einen bestimmten Punkt der Meßreihe aus dem vorhergehenden Abschnitt betrachtet werden. Der Index soll im Akkumulator stehen und ab Null gezählt werden. Die erste Koordinate des Meßpunkts soll in das Registerpaar DE, die zweite ins Registerpaar BC gebracht werden. Das Registerpaar HL zeigt auf die Tabelle. Die Längeninformation im Deskriptor wird verwendet, um den Index auf seine Gültigkeit zu überprüfen:

```
CP      (HL)            ; Gueltigkeit des Index pruefen
JP      NC,FEHLER       ; ungueltiger Index
INC     HL              ; auf ersten Tabelleneintrag zeigen
EX      DE,HL           ; Basisadresse sichern
LD      H,0             ; Index zu
LD      L,A             ; Wort machen
ADD     HL,HL           ; Relativadresse
ADD     HL,HL           ; berechnen
ADD     HL,DE           ; Adresse des gesuchten Eintrags
LD      E,(HL)          ; erste
INC     HL              ; Koordinate
LD      D,(HL)          ; holen
INC     HL              ; auf zweite Koordinate zeigen
LD      C,(HL)          ; zweite
INC     HL              ; Koordinate
LD      B,(HL)          ; holen
```

Auch einen Stapel kann man als Tabelle auffassen. (In einem späteren Abschnitt wird näher auf das Thema Stapel eingegangen.) Das oberste Element ist dabei zweckmäßigerweise der letzte Tabelleneintrag – so wächst die Tabelle zu größeren Adressen hin. Im Deskriptor vermerkt man am besten den Stapel-Zeiger.

Wenn der Stapel Elemente vom Typ „Byte“ aufnimmt und der Stapel-Zeiger der Struktur unter der Adresse STAPEL abgespeichert ist (dies muß nicht unmittelbar vor dem ersten Tabelleneintrag sein), so realisieren folgende Unterprogramme die Operationen PUSH und POP. Der Akkumulator dient zur Aufnahme eines Stapel-Elements.

```
PUSH:   LD      HL,(STAPEL)     ; Stapel-Zeiger holen
        INC     HL              ; auf freien Speicherplatz zeigen
        LD      (HL),A          ; Element ablegen
        LD      (STAPEL),HL     ; Stapel-Zeiger abspeichern
        RET

POP:    LD      HL,(STAPEL)     ; Stapel-Zeiger holen
        LD      A,(HL)          ; Element entnehmen
        DEC     HL              ; auf oberstes Element
                                ; des Stapels zeigen
        LD      (STAPEL),HL     ; Stapel-Zeiger abspeichern
        RET
```

Auf Fehlerbehandlungen (Stapelüberlauf, Stapelunterlauf) und Retten des Registerpaares HL wurde hier verzichtet.

In ähnlicher Weise läßt sich auch ein Puffer, dessen Elemente nicht vom Typ „Byte“ sind, durch eine Tabelle realisieren. Die neu hinzukommenden Elemente werden vom Produzenten ans Ende der Tabelle angehängt. Der Konsument entnimmt Elemente vom Anfang der Tabelle. Dadurch freiwerdender Speicherplatz muß irgendwann durch Verschieben der restlichen Tabelleneinträge wieder nutzbar gemacht werden (dies kann zum Beispiel jedesmal nach dem Entfernen eines Elements geschehen).

7.8.3 Schlüssel-orientierter Zugriff auf Tabellen

Beim schlüssel-orientierten Zugriff auf eine Tabelle spielt die Reihenfolge der Elemente keine Rolle. Ein bestimmtes Element wird dadurch ausgewählt, daß für eine oder mehrere Komponenten des Verbunds Werte vorgegeben werden. In der ersten Tabelle aus Abschnitt 7.6.1 könnte zum Beispiel eine Produktnummer vorgegeben sein, oder eine bestimmte Kombination von Länge, Breite und Dicke.

Die Suchoperation kann fehlschlagen, wenn kein Element vorhanden ist, das die Vorgaben erfüllt. Andererseits kann es auch vorkommen, daß die Beschreibung auf mehrere Elemente der Tabelle zutrifft.

Allgemeiner kann man statt Werten auch Relationen zwischen bestimmten Komponenten vorgeben; und statt eines eindeutig bestimmten Eintrags kann man auch die Menge aller Einträge bestimmen, die der Beschreibung entsprechen. In einem Möbelkatalog könnten beispielsweise folgende Daten aufgelistet sein:

- Artikelname
- Artikelnummer
- Artikelbezeichnung
- Farbe
- Material
- Preis

Eine mögliche Suchoperation wäre dann: Liefere alle Artikelnummern von Einträgen mit der Bezeichnung „Couch", der Farbe schwarzbraun oder rostbraun, aus Rohleder, mit einem Preis nicht höher als 3.400,– DM.

Die Vorgehensweise einer solchen Suchoperation ist folgende: Man stellt sich einen Zeiger auf das erste Element der Tabelle bereit und prüft, ob diese die geforderten Eigenschaften hat. Wenn dies der Fall ist, so wird das Element (oder die benötigten Komponenten) aus der Tabelle kopiert. Anschließend wird in beiden Fällen (durch Addition der festen Länge eines Tabellenelements) der Zeiger auf das nächste Element der Tabelle fortgeschaltet, bis das Ende der Tabelle erreicht ist. Die Feldstruktur der Tabelle wird dabei nur zum Fortschalten der Basisadresse des Verbunds benutzt.

Hierzu ein Beispiel: Gegeben sei eine Tabelle, in der für eine Reihe von Personen drei Kenngrößen festgehalten werden.

Alter (in Jahren)	1 Byte
Gewicht (in kg)	1 Byte
Körpergröße (in cm)	1 Byte

Als Ende-Markierung für die Tabelle wurde ein Null-Byte gewählt, weil das Alter 0 nicht vorkommen kann. Die Tabelle sieht beispielsweise folgendermaßen aus:

```
PERSON:
        DEFB    26      ; Alter
        DEFB    93      ; Gewicht
        DEFB    188     ; Groesse

        DEFB    39      ; Alter
        DEFB    65      ; Gewicht
        DEFB    176     ; Groesse

         :
         :

        DEFB    19      ; Alter
        DEFB    80      ; Gewicht
        DEFB    182     ; Groesse

        DEFB    0       ; Ende-Markierung
```

Nun wird eine zweite Tabelle PERS2 mit gleicher Struktur aufgebaut, welche die Kenngrößen derjenigen Personen enthält, welche zwischen 18 und 35 Jahren (einschließlich) alt und kleiner als 175 cm sind:

```
        LD      IX,PERSON       ; Zeiger auf erste Tabelle
        LD      IY,PERS2        ; Zeiger auf zweite Tabelle
TEST:   LD      A,(IX+0)        ; Alter holen
        OR      A               ; auf Null-Byte testen
        JP      Z,FERTIG        ; zweite Tabelle komplett
        CP      18              ; Alter testen
        JP      C,WEITER        ; juenger als 18
        CP      36              ; Alter testen
        JP      NC,WEITER       ; aelter als 35
        LD      A,(IX+2)        ; Groesse holen
        CP      175             ; Groesse testen
        JP      NC,WEITER       ; nicht kleiner als 175
        LD      A,(IX+0)        ; Alter holen
        LD      (IY+0),A        ; Alter kopieren
        LD      A,(IX+1)        ; Gewicht holen
        LD      (IY+1),A        ; Gewicht kopieren
        LD      A,(IX+2)        ; Groesse holen
        LD      (IY+2),A        ; Groesse kopieren
WEITER: INC     IX              ; auf naechstes
        INC     IX              ; Element der ersten
        INC     IX              ; Tabelle zeigen
        INC     IY              ; auf naechstes
        INC     IY              ; Element der zweiten
```

```
        INC     IY                ; Tabelle zeigen
        JP      TEST              ; Rest der Tabelle bearbeiten
FERTIG: LD      (IY+0),0          ; Ende der zweiten
                                  ; Tabelle markieren
```

Das Programm ist allerdings nicht optimiert. Wer möchte, kann sich aber an einer optimierten Version versuchen!

Ein spezieller Fall liegt vor, wenn jeder Tabelleneintrag nur aus einer Komponente besteht. Dann liegt eine Folge von Elementen eines bestimmten Typs vor, die man als Menge interpretieren kann. Auch hierzu ein Beispiel: Es sollen Tabellen betrachtet werden, die Teilmengen der Menge von ASCII-Zeichen darstellen. Die Teilmengen sollen ungeordnet sein. Die Kardinalität der Teilmengen ist dem Deskriptor zu entnehmen. Zwei solcher Teilmengen wären:

```
MENGE1: DEFB    4       ; Kardinalitaet 4
        DEFB    'H'
        DEFB    '*'
        DEFB    'a'
        DEFB    'Z'
MENGE2: DEFB    3       ; Kardinalitaet 3
        DEFB    '+'
        DEFB    '-'
        DEFB    ' '
```

Hier exemplarisch einige Mengenoperationen auf dieser Struktur:

Mit Hilfe des Unterprogramms ELEM wird festgestellt, ob ein im Akkumulator stehendes Zeichen in der Menge enthalten ist, auf die das Registerpaar HL zeigt. Wenn ja, soll das Z-Flag gesetzt werden:

```
ELEM:   PUSH    BC                ; Registerinhalt sichern
        LD      C,(HL)            ; Tabellenlaenge beschaffen
        INC     HL                ; auf erstes Element
                                  ; der Menge zeigen
        INC     C                 ; Laenge auf
        DEC     C                 ; Null testen
        JP      Z,LEER            ; leere Menge
        LD      B,0               ; Laenge zu Wort machen
        CPIR                      ; Zeichen in Menge suchen
        POP     BC                ; Register restaurieren
        RET
LEER:   INC     C                 ; Z-Flag loeschen
        POP     BC                ; Register restaurieren
        RET
```

Das Hinzufügen eines Elements, das im Akkumulator steht, zu einer Menge, die durch das Registerpaar HL adressiert wird, läßt sich durch folgendes Unterprogramm realisieren:

```
HINEIN: PUSH    HL                ; Registerinhalt sichern
        CALL    ELEM              ; Feststellen, ob Zeichen
                                  ; bereits in der Menge
                                  ; enthalten ist
        JP      Z,FERTIG          ; Zeichen schon in der Menge
        LD      (HL),A            ; HL zeigt direkt hinter Menge,
                                  ; Zeichen hinzufuegen
        POP     HL                ; Register restaurieren
        INC     (HL)              ; Laengenangabe aktualisieren
        RET
FERTIG: POP     HL                ; Register restaurieren
        RET
```

Hier gilt dem (in diesem Fall erwünschten) Nebeneffekt von ELEM besondere Beachtung: Falls das gesuchte Zeichen noch nicht in der Menge enthalten ist, zeigt das Registerpaar HL nach Rückkehr aus ELEM auf das nächste Zeichen direkt hinter der Menge.

Als letztes soll die Vereinigung der beiden Mengen betrachtet werden, auf die das Registerpaar HL beziehungsweise DE zeigt. Die zweite Menge soll dabei in die erste Menge eingefügt werden:

```
VEREIN: PUSH    AF                ; Registerinhalte
        PUSH    BC                ; sichern
        PUSH    DE
        LD      A,(DE)            ; Kardinalitaet der zweiten
        LD      B,A               ; Menge holen
```

```
        INC     B               ; Schleife abweisend machen
        JP      VER3            ; in Schleife einspringen
VER2:   INC     DE              ; auf naechstes Element
                                ; der zweiten Menge zeigen
        LD      A,(DE)          ; Element holen
        CALL    HINEIN          ; Element zur ersten
                                ; Menge hinzufuegen
VER3:   DJNZ    VER2            ; gesamte Menge durchgehen
        POP     DE              ; alle Register
        POP     BC              ; restaurieren
        POP     AF
        RET
```

7.9 Übungsaufgaben (II)

Aufgabe zu Abschnitt 7.4.1

40. Jede Zeichenkette, in der kein Zeichen doppelt vorkommt, kann man als Auflistung einer Menge von Zeichen ansehen. Es soll ein Programm geschrieben werden, das für ungeordnete Repräsentationen von Zeichenmengen (7 Bit ASCII) die Operationen „Gleich", „Teilmenge von" und „Element von" realisiert. Wieviele Operationen werden mindestens und wieviele höchstens gebraucht? Wie verändern sich Mindest- und Höchstanzahl von Operationen bei geordneten Repräsentationen?

Aufgaben zu Abschnitt 7.4.2

41. Unter Verwendung von Inzidenzvektoren sollen die Operationen „Vereinigung", „Schnitt" und „Differenz" für die Menge der kleinen Buchstaben programmiert werden.

42. Für die Menge der ASCII-Zeichen (7 Bit) soll ein Programm geschrieben werden, das die symmetrische Differenz ermittelt. Auch hier sollen Inzidenzvektoren zum Einsatz kommen.

Aufgabe zu Abschnitt 7.5.1

43. Wie könnte ein Programm aussehen, das im gezeigten Drucker-Kontrollblock nach Ausgabe eines Zeichens die aktuelle Position korrigiert?

Aufgabe zu Abschnitt 7.5.2

44. Es soll ein gepackter Verbund vereinbart werden, der aus folgenden Komponenten besteht:

Sekunde	6 Bits
Minute	6 Bits
Stunde	5 Bits
Tag	5 Bits
Monat	4 Bits
Jahr	11 Bits
Wochentag	3 Bits

Für den Wochentag soll folgende Codierung verwendet werden: 0 = Montag, 1 = Dienstag, ..., 6 = Sonntag. Nun soll ein Programm geschrieben werden, das den Inhalt des Verbunds um eine Sekunde fortschaltet und im dementsprechend Minuten, Stunden usw. aktualisiert.

Aufgabe zu Abschnitt 7.5.3

45. Ein Verbund ist zu definieren, der wahlweise ein Dreieck, ein Quadrat, ein Rechteck oder einen Kreis beschreibt.

Aufgaben zu Abschnitt 7.6.1

46. Eine Menge von echt positiven ganzzahligen Raumkoordinaten soll durch eine Tabelle dargestellt werden. Folgende Koordinaten sollen benutzt werden:

12	44	21
16	39	55
22	117	98
3	39	44
16	57	83
32	61	9

47. Die Buchstabenmenge G, J, E, f, n, R, t, Z soll als Tabelle dargestellt werden.

Aufgabe zu Abschnitt 7.6.2

48. Bei einem Quiz erzielten die Kandidaten folgende Punkte:

Huber	12
Meier	28
Schulz	21
Gruber	18
Kaiser	14
Weiss	21

Diese Informationen sollen als indizierte Tabelle abgespeichert werden. Anschließend ist ein Unterprogramm zu schreiben, das zu vorgegebenem Index den Anfangsbuchstaben des Namens und die erreichte Punktzahl liefert.

Aufgabe zu Abschnitt 7.6.3

49. Folgende Liste von Paaren (Vorname, Alter) soll als Tabelle definiert werden:

Petra	25
Hans	28
Otto	27
Hanna	29
Klaus	22
Inge	27
Heinz	27
Claudia	26
Peter	22

Es ist ein Unterprogramm zu entwickeln, das zu vorgegebenem Alter die Tabelle derjenigen Personen erstellt, die jünger sind.

Teil 8 – Komplexe Programmstrukturen

von Frank Dachselt

In diesem Teil sollen, wie die Überschrift bereits verrät, komplexe Programmstrukturen betrachtet werden. Daß Programm- und Datenstrukturen nicht unabhängig voneinander existieren, ist sicher schon in den vorangegangenen Teilen deutlich geworden. Komplexe Datenstrukturen bedingen mit Sicherheit komplexe Programme. Und wer umgekehrt ein umfangreiches Programm entwirft, wird meist nicht umhinkommen, die anfallenden Daten in komplexen Strukturen zu verwalten. Im folgenden wird deshalb zunächst auch eine Datenstruktur betrachtet, die sehr eng mit bestimmten Programmiertechniken verbunden ist und in den Beispielen vorangegangener Teile (unbewußt) verwendet wurde.

8.1 Der Stack

8.1.1 Das LIFO-Prinzip

Ein Stack ist eine Datenstruktur, die ähnlich einem eindimensionalen Feld aus Elementen gleicher Größe besteht, wobei aber die Größe dieses Feldes (Anzahl der Feldelemente) veränderlich ist. Deshalb besitzt ein Stack neben einer feststehenden Grenze – der Stackbasis – auch eine variable Grenze. Die Position dieser variablen Grenze wird in einem speziellen Register oder einer speziellen Speicherzelle – dem Stackpointer – abgelegt.

Der Zugriff auf die Elemente des Stacks erfolgt stets an seiner variablen Grenze, also an der Stelle, auf die der Stackpointer zeigt. Wird ein Wert in den Stack geschrieben (man sagt auch „auf dem Stack abgelegt“), dann gelangt dieser stets in die nächste freie – bisher also noch nicht benutzte – Position unmittelbar an der variablen Grenze. Dem Stack als eindimensionales Feld wird somit ein neues Element hinzugefügt. Der Stackpointer wird dabei um die Länge dieses neuen Elements verändert, so daß der nächste Schreibzugriff auf die folgende freie Position erfolgt. Beim Lesen vom Werten vom Stack wird dieser Vorgang sinngemäß umgekehrt. Der Lesezugriff liefert stets den Inhalt des Elements, das sich unmittelbar an der variablen Grenze befindet. Dabei wird der Stackpointer um die Länge dieses Elements zurückgestellt, so daß ein weiterer Lesezugriff auf das vorhergehende Element zugreift.

Werden die Werte A_1, A_2 und A_3 in dieser Reihenfolge auf dem Stack abgelegt, dann

liefern drei aufeinanderfolgende Lesezugriffe die Werte A_3, A_2 und A_1 – also in umgekehrter Reihenfolge – zurück. Genau dieses Verhalten hat zu weiteren Bezeichnungen für den Stack geführt. Der Stack arbeitet nach dem sogenannten LIFO-Prinzip (Last in – First out), bei dem also der zuletzt hineingeschriebene Datenwert zuerst ausgelesen wird. Gebräuchlich sind auch die Bezeichnungen Stapelspeicher oder Kellerspeicher, die zudem eine bildliche Vorstellung von der Funktion des Stacks liefern. Zu letzterem läßt sich auch das auf der linken Seite von Bild 12 gezeigte Gerät heranziehen: Hier werden auf kleine Zettel geschriebene Notizen abgelegt, indem sie nacheinander auf diesen „Stack“ geschoben werden. Soll eine dieser Notizen wieder gelesen werden, müssen zuvor alle darüberliegenden entfernt – also gelesen – werden.

Bei der Realisierung eines Stacks gibt es einige Variationsmöglichkeiten, die aber seine prinzipielle Funktion nicht beeinflussen:

- Der Stack kann von höheren zu niederen oder von niederen zu höheren Adressen aufgebaut werden. Bei Schreibzugriffen wird der Stackpointer dann jeweils um eine Elementelänge erniedrigt bzw. erhöht, bei Lesezugriffen entsprechend umgekehrt.
- Der Stackpointer kann entweder auf das zuletzt abgelegte Element oder bereits auf die nächste freie Position zeigen. Bei einem Schreibzugriff muß der Stackpointer im ersten Fall vor dem eigentlichen Schreiben der Datenwerte verändert werden, im zweiten Fall danach. Beim Lesezugriffen kehrt sich diese Reihenfolge entsprechend um.

8.1.2 Systemstack und Z80-Stackoperationen

8.1.2.1 Funktionsweise der Stackoperationen

Der Z80 stellt dem Programmierer bereits einen Stack zur Verfügung, der im Zusammenhang mit bestimmten Befehlsgruppen verwendet wird. Dazu existiert im Registersatz des Z80 ein spezielles Register: der Stackpointer SP (siehe auch Abschnitt 3.1.1). Der Stack selbst befindet sich in einem Teil des Hauptspeichers, der im Prinzip frei wählbar ist. Durch die automatische Verwaltung des Stackpointers wird viel Programmieraufwand gespart und zudem eine zügige Programmabarbeitung erreicht. Dieser Stack ist bei Z80-Systemen stets vorhanden und in Funktion, da ohne ihn viele Befehle und Funktionen nicht verwendet werden können. Dieser Stack trägt deshalb auch den Namen *Systemstack*.

Der Systemstack des Z80 ist wortorganisiert, d.h. seine Elemente sind jeweils 16 Bit

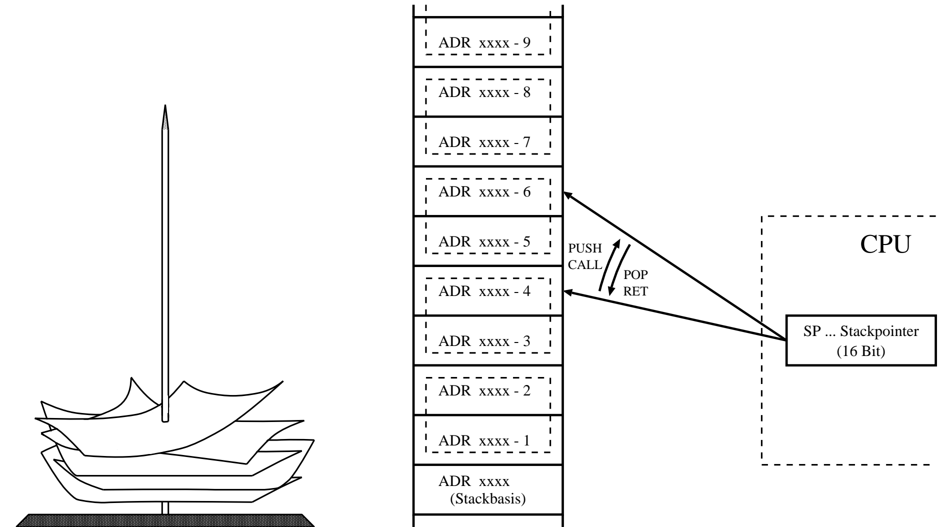


Bild 12: Zur Funktionsweise des (System-)Stacks

bzw. zwei Byte groß. Damit findet in jedem Element der Inhalt eines Doppelregister Platz. Der Stack wird von höheren zu niederen Adressen hin aufgebaut. Der Stackpointer zeigt dabei stets auf das zuletzt auf dem Stack abgelegte Element, genauer gesagt auf das niederwertige Byte des zuletzt abgelegten Wortes.

Wie wird nun der Systemstack in Programmen verwendet? Dazu existieren zunächst zwei Befehlsgruppen, mit denen jeweils der Inhalt eines der Doppelregister AF, BC, DE, HL, IX oder IY auf dem Stack abgelegt wird bzw. der oberste Eintrag vom Stack in eines dieser Doppelregister gebracht wird. Diese Befehlsgruppen sind die PUSH- und POP-Befehle:

```
PUSH    AF      ; Inhalt des Doppelregisters AF auf dem
                ; Stack ablegen
PUSH    BC      ; dito fuer BC
PUSH    DE      ; dito fuer DE
PUSH    HL      ; dito fuer HL
PUSH    IX      ; dito fuer IX
PUSH    IY      ; dito fuer IY
```

```
POP     IY      ; obersten Stackeintrag nach IY bringen
POP     IX      ; dito fuer IX
POP     HL      ; dito fuer HL
POP     DE      ; dito fuer DE
POP     BC      ; dito fuer BC
POP     AF      ; dito fuer AF
```

AF bezeichnet dabei ein Doppelregister, das sich aus den jeweils 8 Bit breiten Registern A(Akkumulator) und F (Flagregister) zusammensetzt. Dieses Doppelregister existiert nur innerhalb der beiden genannten Befehlsgruppen.

Entsprechend der oben beschriebene Funktionsweise des Stacks lassen sich die PUSH- und POP-Befehle in folgende Elementarschritte zerlegen:

PUSH nn (nn = AF, BC, DE, HL, IX, IY)

- Erniedrige den Inhalt des Stackpointers SP um 1;
- Bringe den Inhalt des höherwertigen Bytes von nn in die durch SP adressierte Speicherzelle (höherweriges Byte auf der höherwertigen Adresse ablegen);
- Erniedrige den Inhalt des Stackpointers SP um 1;
- Bringe den Inhalt des niederwertigen Bytes von nn in die durch SP adressierte Speicherzelle (niederwertiges Byte auf der niederwertigen Adresse ablegen);

POP nn

- Bringe den Inhalt der durch SP adressierte Speicherzelle in das niederwertige Byte von nn;
- Erhöhe den Inhalt des Stackpointers SP um 1;
- Bringe den Inhalt der durch SP adressierte Speicherzelle in das höherwertige Byte von nn;
- Erhöhe den Inhalt des Stackpointers SP um 1;

Die zweite Befehlsgruppe, die mit dem Stack zusammenarbeitet, bilden die Unterprogrammbefehle CALL und RET. Wie bereits im Abschnitt 6.5 beschrieben, muß bei einem Unterprogrammaufruf mittels CALL-Befehl die Adresse, an der die Programmabarbeitung nach dem RET-Befehl fortgesetzt wird, zwischengespeichert werden. Die Zwischenspeicherung dieser Rückkehradresse geschieht auf dem Stack: Bei einem CALL wird der Inhalt des Programmzählers PC, der zu diesem Zeitpunkt bereits auf den Folgebefehl zeigt, zuerst auf dem Stack abgelegt. Erst danach wird der Programmzähler mit der Adresse des aufzurufenden Unterprogramms geladen. Der RET-Befehl am Ende des Unterprogramms braucht dann nur noch diese Rückkehradresse vom Stack zu holen und sie in den Programmzähler zu laden. Die Funktionsweise der Unterprogrammbefehle liese sich auch mit PUSH- und POP-Befehlen und der Benutzung des Registerpaars HL nachbilden:

```
  ; Nachbildung des CALL-Befehls

          LD      HL,RADR ; Rueckkehradresse zuerst nach HL
          PUSH    HL      ; und dann auf dem Stack ablegen
          JP      UPADR   ; normaler Sprung ins Unterprogramm
RADR:


  ; Nachbildung des RET-Befehls

          POP     HL      ; Rueckkehradresse vom Stack holen
          JP      (HL)    ; indirekten Sprung zur Rueckkehradresse
```

Die Nachbildung bedingter Unterprogrammaufrufe und Rückkehrbefehle (siehe Tabelle auf Seite 67) wäre noch ein erheblich größerer Aufwand erforderlich. Abgesehen davon, daß kein zusätzliches Registerpaar benötigt wird, laufen bei den Unterprogrammbefehlen also bereits recht komplexe Vorgänge ab, was die Anwendung der Unterprogrammtechnik für den Assemblerprogrammierer nahezu ebenso bequem wie in einer Hochsprache macht.

8.1.2.2 Stackoperationen in einfachen Programmstrukturen

Bei der Verwendung des Stacks ist es außerordentlich wichtig, auf die Symmetrie der Stackoperationen zu achten. Nur so wird erreicht, daß ein POP-Befehl auch tatsächlich den gewünschten Wert vom Stack zurückließt und ein RET-Befehl die richtige Rückkehradresse vom Stack holt. In diesem Sinne ist auch die oben angegebene Aufstellung der PUSH- und POP-Befehle zu verstehen. Bedingt durch das FIFO-Prinzip werden die Registerinhalte im Normalfall in umgekehrter Reihenfolge, in der sie auf dem Stack abgelegt wurden, wieder zurückgeholt. Eine Ausnahme bildet lediglich der Fall, wenn die Registerinhalte bei dieser Gelegenheit gezielt vertauscht werden sollen. Letzteres erfordert aber bereits erhöhte Aufmerksamkeit vom

Programmierer und eine gute Dokumentation im Quelltext, um spätere Mißverständnisse zu vermeiden.

Aus dem gleichen Grund ist bei Unterprogrammen darauf zu achten, daß alle innerhalb eines Unterprogramms auf dem Stack abgelegten Registerinhalte vor dem RET-Befehl wieder vom Stack geholt werden. Besteht das Unterprogramm aus einer einfachen Befehlssequenz, dann ist das einfach zu realisieren: Es müssen ebenso viele POP- wie zuvor PUSH-Befehle enthalten sein.

Schwieriger wird es, wenn das Unterprogramm komplexere Strukturen wie Verzweigungen und Schleifen enthält. Bei Verzweigungen muß man die Stackeinträge beachten, die aus dem vorangegangen Programmteil „mitgebracht“ werden. Diese Stacktiefe wird jedem der möglichen Zweige übergeben und ist von diesen entsprechend zu verarbeiten. Im folgenden Beispiel ist dazu hinter jedem PUSH- und POP-Befehl die momentan erreichte Stacktiefe relativ zum Beginn des Unterprogramms mit dem Zeichen „*“ angegeben (bei PUSH-Befehlen nach und bei POP-Befehlen vor der Ausführung):

```
        PUSH    AF      ; Registerpaare AF und HL       *
        PUSH    HL      ; dem Stack ablegen             **
        ...
        JR      ADR1
        POP     HL      ;                               **
        ...
        POP     AF      ; AF vom Stack holen            *
        RET

ADR1:   PUSH    BC                                      ***
        ...
        POP     BC                                      ***
        POP     HL                                      **
        POP     AF                                      *
        RET
```

Nach der Verzweigung zur Marke ADR1 beträgt die momentane Stacktiefe bereits 2, so daß auch am Ende dieses Zweiges noch die beiden zugehörigen POP-Befehle ausgeführt werden müssen, bevor der Rücksprung erfolgt.

Das Gegenstück zu Programmverzweigungen, bei denen die Stacktiefe an die abgehenden Zweige weitergereicht wird, sind Zusammenführungen von Programmsequenzen. Auch hier übernimmt der abgehende Zweig jeweils die Stacktiefe des vorangegangenen Programmstücks. Damit die Programmabarbeitung im abgehenden Zweig konsistent bleibt, muß sichergestellt sein, daß jeder ankommende Zweig den Zusammenführungspunkt mit derselben Stacktiefe erreicht. Ist das bei einem der ankommenden Zweige nicht der Fall, dann muß diese Stacktiefe durch geeignete PUSH- oder POP-Befehle erzeugt werden, bevor die Zusammenführung erfolgt.

```
PS1:    PUSH    AF      ; Programmsequenz 1             *
        PUSH    BC                                      **
        ...
        POP     BC                                      **
        POP     AF                                      *
        PUSH    HL      ; !!!                           *
        JR      FADR
        ...

PS2:    PUSH    HL      ; Programmsequenz 2             *
        ...

FADR:   ...             ; gemeinsamer Fortsetzungspunkt
        ...             ; mit Stacktiefe 1
        POP     HL                                      *
        ...
```

Die Programmsequenz 1 sollte ursprünglich einmal mit der Stacktiefe 0 (relativ zu ihrem Anfang) beendet werden. Am Zusammenführungspunkt (Marke FADR) mit der Sequenz 2 wird aber durch diese die Stacktiefe 1 vorgegeben (PUSH HL). Deshalb wird hier am Ende der Sequenz 1 der Befehl PUSH HL eingefügt, bevor der Sprung zur Marke FADR erfolgt, um die Sequenz ein mit der geforderten Stacktiefe 1 zu verlassen und den Registerinhalt HL am Ende wiederherstellen zu können.

Das gleiche Ergebnis läßt sich auch anders erreichen:

```
PS1:    PUSH    AF      ; Programmsequenz 1              *
        PUSH    BC                                       **
        ...
        POP     BC                                       **
        POP     AF                                       *
        JR      FADR
        ...

PS2:    PUSH    HL      ; Programmsequenz 2              *
        ...

        POP     HL      ;!!!                             *
FADR:   PUSH    HL      ;!!!                             *
        ...             ; gemeinsamer Fortsetzungspunkt
        ...             ; mit Stacktiefe 1
        POP     HL                                       *
        ...
```

Hier wird die Sequenz 2 so erweitert, daß sie vor den Zusammenführungspunkt die Stacktiefe 0 erreicht (POP HL). Damit kann auch die Sequenz 1 mit der Stacktiefe 0 direkt zur Marke FADR springen. Hier muß dann aber zunächst wieder das Registerpaar HL auf den Stack gebracht werden, damit ab der Marke FADR die gleichen Verhältnisse wie im ersten Fall herrschen.

Besondere Aufmerksamkeit erfordern Stackoperationen „in der Nähe“ von Schleifen. Hier treten Verzweigung und Zusammenführung von Programmabläufen in kombinierter Form auf. Es ist daher notwendig, daß der Schleifenkörper in die Symmetrie der Stackoperationen mit einbezogen wird. Das heißt konkret, daß die zueinandergehörenden Stackoperationen entweder vollständig außerhalb oder vollständig innerhalb des Schleifenkörpers liegen müssen. Die beiden folgenden Beispiele sollen das verdeutlichen. Es handelt sich dabei um eine Zählschleife mit 135 Durchläufen und dem Register B als Schleifenzähler, um den bequemen DJNZ-Befehl verwenden zu können.

Im ersten Fall soll der Inhalt des Registers B vor der Schleife auch nach Ausführung der Schleife wieder zur Verfügung stehen. Bevor der Schleifenzähler B initialisiert wird, muß deshalb das Register B (in Form des Doppelregisters BC) auf den Stack gerettet werden, um es nach der Abarbeitung der Schleife wiederherstellen zu können:

```
        ...
        PUSH    BC      ; BC retten
        LD      B,135   ; Schleifenzaehler initialisieren
LOOP:   ...             ; Schleifenkoerper
        ...
        DJNZ    LOOP    ; Schleifensteuerung
        POP     BC      ; BC wiederherstellen
        ...
```

Im zweiten Fall soll dagegen der Inhalt des Registers B nach dem Abarbeiten der Schleife ohne Bedeutung sein. Vielmehr wird das Register B jetzt im Schleifenkörper noch für andere Zwecke benötigt, die den Wert des Schleifenzählers zerstören. Deshalb wird der Schleifenkörper durch zwei Stackoperationen „geklammert“, so daß vor dem DJNZ-Befehl wieder der tatsächliche Wert des Schleifenzählers im Register B steht:

```
        ...
        LD      B,135   ; Schleifenzaehler initialisieren
LOOP:   PUSH    BC      ; BC retten
        ...             ; Schleifenkoerper
        ...
        POP     BC      ; BC wiederherstellen
        DJNZ    LOOP    ; Schleifensteuerung
        ...
```

Die Verwendung des Stacks zur kurzzeitigen Zwischenspeicherung von Registerinhalten ist also eine recht bequeme Sache, die allerdings mit etwas erkauft werden muß: nämlich mit Abarbeitungszeit. Bedingt durch die notwendigen Zugriffe auf den Hauptspeicher ist die Abarbeitungszeit der PUSH- und POP-Befehle mit 11 bzw. 10 Takten (für die Standardregister, siehe Anhang B.2) relativ lang. Günstiger ist es deshalb, wenn die Zwischenspeicherung ohne Speicherzugriffe auskommt. Das ist immer dann möglich, wenn nur ein spezielles Register zwischengespeichert werden muß und andere Register in der CPU noch zur Verfügung stehen. In solchen Fällen erfolgt die Zwischenspeicherung günstigerweise mittels einfacher Ladebefehle in einem anderen Register. In diesem Sinne soll das vorhergehende Beispiel noch einmal betrachtet werden: Der Schleifenzähler B muß wieder innerhalb der Schleife gerettet

werden, allerdings wird das Register C im Schleifenkörper nicht gebraucht und auch sein Inhalt nach Abarbeitung der Schleife ist ohne Bedeutung:

```
        ...
        LD      B,135   ; Schleifenzaehler initialisieren
LOOP:   LD      C,B     ; B im Register C retten
        ...             ; Schleifenkoerper
        ...
        LD      B,C     ; Schleifenzaehler B wiederherstellen
        DJNZ    LOOP    ; Schleifensteuerung
        ...
```

Gegenüber den PUSH- und POP-Befehlen benötigt der LD-Befehl nur 4 Takte zur Abarbeitung. Pro Schleifendurchlauf werden also 13 Takte gespart, bei 135 Durchläufen also immerhin 1755 Takte, was im D001 bereits etwa 1 ms entspricht. Laufzeitoptimierung lohnt sich also besonders innerhalb von Schleifen und anderen oft benutzten Programmabschnitten. Die Länge des Programmcodes ist übrigens in beiden Fällen gleich, da PUSH/POP- und LD-Befehle jeweils 1-Byte-Befehle sind.

Außerdem ist der Stackbereich eine wichtige und daher auch stets knappe Systemressource, deren Nutzung möglichst sparsam erfolgen sollte. Auch aus diesem Grund ist es oft sinnvoll, Alternativen zu nutzen.

8.1.2.3 Stackoperationen in Unterprogrammen

Ein typischer Anwendungsfall von Stackoperationen sind Unterprogramme. Immer dann, wenn über einzelne CPU-Register keine Ergebnisse an den aufrufenden Programmteil zurückgegeben werden, wird oft gefordert, daß das Unterprogramm diese Register aus der Sicht des aufrufenden Programms nicht verändert. Dann kann ein solches Unterprogramm an beliebigen Stellen aufgerufen werden und es ist stets sichergestellt, daß es – außer seiner eigentlichen Funktion – keinen Einfluß auf übergeordneten Programmablauf hat. Das folgende Beispiel zeigt, wie mit den Stackoperationen ein „Rahmen“ um das eigentliche Unterprogramm gelegt wird, um die genannte Forderung zu erfüllen:

```
; Rettung und Wiederherstellung aller Standard-Register
; im Unterprogramm
UP:     PUSH    AF
        PUSH    BC
        PUSH    DE
        PUSH    HL
        ...             ; das eigentliche Unterprogramm
        ...
        POP     HL
        POP     DE
        POP     BC
        POP     AF
        RET
```

Registerpaare, die im Unterprogramm unverändert bleiben, müssen nicht zwischengespeichert und die zugehörigen Stackoperationen können aus diesem Rahmen entfernt werden. Registerpaare, die der Ergebnisübergabe an das aufrufende Programm dienen, dürfen natürlich nicht auf den Stack gelegt werden.

Der Programmierer ist aber nicht immer in der Lage, die verwendeten Unterprogramme in diesem Sinne seinen Erfordernissen anzupassen. Das ist zum Beispiel bei der Verwendung der CAOS- und BDOS-Systemroutinen oder von Unterprogrammbibliotheken der Fall. Das nachträgliche Einfügen von Stackoperationen ist hier nicht möglich. Um dennoch den beschriebenen Effekt zu erzielen, werden die Stackoperationen im aufrufenden Programm vor bzw. nach dem CALL-Befehl angeordnet:

```
; Unterprogramm-Aufruf mit Rettung aller Standard-Register
; im Hauptprogramm
        PUSH    AF
        PUSH    BC
        PUSH    DE
        PUSH    HL

        CALL    UP

        POP     HL
        POP     DE
        POP     BC
        POP     AF
```

Bezüglich der tatsächlich zwischenzuspeichernden Registerpaare gilt das oben gesagte. Man beachte hier wie im ersten Fall die Symmetrie der Stackoperationen untereinander und gegenüber dem Unterprogrammaufruf.

An dieser Stelle zahlt sich eine gute Dokumentation der Unterprogramme aus, wie sie im Abschnitt 6.5 bereits beschrieben wurde. Aus den Angaben über veränderte Register (VR) und Parameterrückgabe (PA) kann abgelesen werden, welche Registerpaare evtl. zwischengespeichert werden müssen. Die Dokumentation der CAOS-Unterprogramme im D001-Systemhandbuch ist in dieser Hinsicht positiv zu erwähnen, da sie diese Informationen vollständig enthält. Anders sieht es dagegen bei den BDOS-Routinen im D004 aus. Um eventuelle Funktionserweiterungen und Neuimplementierungen nicht zu behindern, wird auf die Angabe veränderter oder nicht veränderter Register verzichtet. Die Dokumentation beschränkt sich auf die zur Parameterübergabe verwendeten Register. Hier muß deshalb stets vom ungünstigsten Fall ausgegangen werden, d.h. auch Register, die nicht explizit zur Parameterrückgabe ausgewiesen sind, müssen als veränderlich angesehen werden.

8.1.3 Lokale Stacks

Wie bereits angedeutet, gehört zum Umgang mit dem Stack eine gewisse Sparsamkeit, da er – wie alle Systemressourcen – nur in begrenztem Umfang zur Verfügung steht. Für den Stack wird im Hauptspeicher ein bestimmter Bereich reserviert. Das übernimmt zunächst einmal das Betriebssystem, indem es den Stackpointer entsprechend initialisiert. Wo dieser Bereich im Hauptspeicher eingerichtet wird, ist für die Funktion des Stacks ohne Bedeutung. Wichtig ist nur, daß er ständig zur Verfügung steht (z.B. darf dieser Speicherbereich nicht ohne weiteres weggeschaltet oder mit einem Schreibschutz versehen werden, wie das beim Schalten von Speicherblöcken und Modulen im D001 möglich ist).

Das D001-Systemhandbuch auf Seite 62 gibt an, wo sich der Systemstack im Grundgerät normalerweise befindet: im Bereich von 140H bis 1C4H, wobei der Stack wie beschrieben von der höheren Adresse zu niederen hin aufgebaut wird. Wenn ein Programm vom CAOS-Prompt aus oder selbststartend aufgerufen wird, dann steht dieser Bereich – abzüglich eines Eintrags, der bereits vom Betriebssystem belegt ist (Rückkehraddresse für den abschließenden RET-Befehl) – diesem Programm zunächst zur Verfügung. Es gibt aber noch weitere Einschrankungen, die durch Interruptroutinen (siehe späteren Kursteil) verursacht werden. Die maximale Stackbelastung durch Interruptroutinen ist keine feststehende Größe; sie hängt vielmehr von Systemzustand und eventuell zusätzlich laufenden Hintergrundprozessen ab. Ohne den weiteren Kursteilen vorzugreifen, sei an dieser Stelle der Wert von 10 Stackeinträgen, also 20 Bytes, genannt, die für den Interruptbetrieb im D001 mindestens bereitstehen sollten.

Damit bleiben also (1C2H − 140H) − 20 = 110 Bytes bzw. 55 Stackeinträge, die einem Programm im D001 mit einiger Sicherheit zur Verfügung stehen. Es empfiehlt sich aber im Interesse eines stabilen Systems, diese Grenze nie ganz auszunutzen, sondern schon vorher nach Alternativen zu suchen. Ein weiteres Problem besteht nämlich darin, die maximale Stacktiefe eines Programms zu bestimmen. Bei größeren Programmen dürfte das praktisch unmöglich sein, denn zu diesem Zweck sind alle möglichen Pfade bei der Programmabarbeitung – einschließlich der aufgerufenen Systemunterprogramme – zu untersuchen und deren Stacktiefe zu dokumentieren. An dieser Stelle muß man sich daher oft mit Abschätzungen begnügen, die zudem auch etwas Programmiererfahrung bedürfen. Ein Hilfe bietet hier das Systemhandbuch des Grundgerätes, in dem nämlich für die CAOS-Systemroutinen die jeweilige Stacktiefe (die Anzahl der Stackeinträge; der notwendige Platz in Bytes ist doppelt so groß) angegeben ist – im übrigen ein weiterer Punkt, der zu einer guten Dokumentation von Unterprogrammen gehört.

Die meisten CAOS-Programme sind aber nicht so komplex, daß sie an die Grenzen des vom Betriebssystem vorgegebenen Systemstacks stoßen. Sollte dies jedoch trotz sparsamer Verwendung des Stacks der Fall sein, dann muß für die Laufzeit dieses Programms ein neuer Stack – ein sogenannter *lokaler Stack* – eingerichtet werden. Dazu ist zunächst ein geeigneter Speicherbereich zu finden, der die oben genannten Forderungen erfüllt. Im Grundgerät sollte man sich dafür auf den RAM0 beschränken. Der RAM4, der zwar prinzipiell ebenfalls geeignet wäre, sollte nicht verwendet werden, da dadurch unnötige Systembeschränkungen entstehen (mehr dazu in einem der folgenden Kursteile). Um einen zusammenhängenden Code- und Datenbereich im RAM0 und RAM4 zu erhalten, ist es günstig, den lokalen Stack im unteren Adreßbereich des RAM0 anzulegen, so daß er sich direkt überhalb der dortigen CAOS-Arbeitszellen anschließt.

Das folgende Beispiel zeigt, wie ein lokaler Stack für maximal 128 Einträge (256 Bytes) im Bereich 200H ... 2FFH angelegt wird:

```
LSTACK  EQU     300H - 2        ; lokale Stackbasis definieren

START:  LD      (LSTACK),SP     ; alten Stackpointer merken
        LD      SP,LSTACK       ; SP auf lokalen Stack setzen
```

Zunächst wird die Marke LSTACK definiert; im Beispiel erhält sie den Wert 2FEH. Die folgenden zwei Befehle sind unmittelbar am Programmstart einzufügen. In den obersten zwei Bytes des neuen Stackbereiches (Zellen 2FEH und 2FFH) wird der alte Wert des Stackpointers zwischengespeichert, um ihn später wiederherstellen zu können. Danach wird der Stackpointer auf den neuen Wert gesetzt. Da beim Z80-Systemstack der Stackpointer stets auf den letzten Eintrag zeigt, erfolgt der erste Eintrag in den neuen Stack in die Zellen 2FCH und 2FDH.

Ab der Adresse 300H kann sich nun der Code- oder Datenbereich (z.B. mittels geeigneter ORG-Anweisung) anschließen. Am Ende der Programms wird unmittelbar vor dem Rücksprung ins CAOS der ursprüngliche Stack reaktiviert, damit der RET-Befehl die richtige Rückkehradresse findet:

```
        LD      SP,(LSTACK)     ; alten SP wiederherstellen
        RET                     ; Ruecksprung zum CAOS
```

Für das CP/M-System im D004 gelten die Aussagen zur Verwendung des Systemstacks in ähnlicher Weise. Erschwerend kommt hier hinzu, daß der Speicherbereich, in dem sich der Systemstack befindet, sowie dessen Größe nirgends dokumentiert ist. Weiterhin ist die Stacktiefe der BDOS-Routinen unbekannt. Das ist auch nicht anders zu erwarten, da es CP/M-Systeme mit unterschiedliche Implementierungen und für verschiedene Hardwareausstattungen gibt und damit auch die Stackdaten nicht einheitlich sind. Aus diesem Grund sollte man sich unter CP/M schon bei mittleren Programmenlängen für einen lokalen Stack entscheiden.

Der beste Platz für den lokalen Stack in einem CP/M-Programm ist das Ende des TPA. Von dort aus wird dieser zu niederen Adressen hin aufgebaut, während sich Programm- und Datenbereiche am TPA-Anfang befinden (genauere Darstellungen über den Aufbau und die Abläufe von Programmen unter CP/M sind in den folgenden Kursteilen zu finden). Das folgende Beispiel zeigt, wie ein solcher Stack einzurichten ist:

```
BDOS_AD EQU     6

START:  LD      HL,(BDOS_AD)    ; BDOS-Beginn = TPA-Ende
        LD      L,0             ; evtl. Seitenausrichtung
        LD      SP,HL           ; SP auf lokalen Stack setzen
```

Das TPA-Ende wird aus der Systemzelle 6 ermittelt, die die Einsprungadresse ins BDOS enthält. Diese Adresse kann direkt zur Initialisierung des Stackpointers verwendet werden. Wird zuvor das niederwertige Byte auf Null gesetzt (LD L,0), dann erreicht man, daß der Stack an einer „runden“ Adresse (Seitenausrichtung) beginnt, das ist aber nicht unbedingt notwendig. Auf eine Zwischenspeicherung des alten Wertes des Stackpointers kann hier verzichtet werden, da CP/M-Programme anstelle eines RET-Befehls auch mit einem Warmstart des Systems beendet werden können:

```
        JP      0               ; Sprung zum CP/M-Warmstart
```

Dieser Warmstart sorgt unter anderem auch für eine Neuinitialisierung des Systemstacks und ist bei Verwendung eines lokalen Stacks stets anzuwenden.

Teil 9 – Assembler unter CP/M

von Mario Leubner

In den vorangegangenen Teilen wurden die Grundlagen der Assemblerprogrammierung und die Handhabung von EDAS vermittelt. Mit dem D004 steht uns jedoch ein zweites Betriebssystem zur Verfügung, das zu CP/M kompatibel ist. Unabhängig davon ob MicroDOS, ML-DOS oder das Z-System tatsächlich genutzt wird, werde ich in diesem Beitrag stellvertretend von CP/M sprechen. Die Aussagen gelten für alle Systeme gleichermaßen.

Doch warum soll man von EDAS umsteigen? Warum soll man sich überhaupt mit verschiedenen Assemblern beschäftigen? Nun für den Anfang ist EDAS sicher gut geeignet sich mit der Assemblersprache vertraut zu machen. EDAS enthält den EDitor und den ASsembler in einem Programm und läßt sich über ein CAOS-Menü steuern. Das wird zunächst als angenehm empfunden, bindet jedoch auch an den zur Verfügung stehenden Funktionsumfang. Für die Erzeugung von Programmen der CAOS-Betriebsart ist EDAS bestens geeignet, da die erzeugten Programme sofort im richtigen Format *.KCC entstehen. Bei umfangreichen Programmen wird man jedoch irgendwann an die Leistungsgrenzen von EDAS stoßen, die unter EDAS 1.6 maximal zur Verfügung stehenden 47,5 K Speicher müssen für den Quelltext und die Markentabelle ausreichen. Man kann zwar den Quelltext in mehrere Teile zerlegen und nacheinander übersetzen, doch die Übersichtlichkeit geht dann schnell verloren.

Warum also nicht einmal Ausschau halten, was auf dem KC-System noch alles so möglich ist? Unter CAOS ist außer EDAS kein weiterer Assembler bekannt. CP/M bietet dagegen ein recht breites Angebot in dieser Richtung. Es existieren verschiedene Assembler und dazugehörige Tools, einige für die Syntax des i8080. Wir erinnern uns, daß unser Prozessor ein U880 oder Z80 ist. CP/M wurde ja ursprünglich für den i8080 geschrieben, der Z80 ist dazu kompatibel hat aber einen größeren Befehlsumfang. Wir brauchen also einen Assembler der alle Z80-Befehle kennt. In diesem Programmierkurs werden wir uns auf einen Assembler beschränken, der allgemein zur Verfügung steht. Deshalb fiel meine Wahl auf ASM.COM, das ist die robotron-Version des Makro- Assemblers M80.COM von Microsoft. Dieser Assembler übersetzt Z80-Befehle, kann aber auch 8080-Befehle verarbeiten. Der dazugehörige Linker LINK.COM (robotron) bzw. L80.COM (Microsoft) ist jedoch nicht so flexibel wie der Linker LINK131.COM von Digital Research, deshalb nutzen wir diesen als Basis für die Beispiele in unserem Kurs.

9.1 Einleitung

Unter CP/M ist die Assemblerprogrammierung nicht mit einem einzigen Programm wie bei EDAS zu machen. Die Entwicklung eines Programmes läuft in mehreren Schritten ab. Bild 14 verdeutlicht diesen Ablauf.

Als Mindestanforderung benötigt man also:

- einen Editor, um den Quelltext einzugeben,
- einen Assembler zur Erzeugung des Zwischencodes,
- einen Linker bzw. Programmverbinder, um schließlich ein lauffähiges Programm daraus zu erzeugen.

Die Ergebnisse jedes Arbeitsschrittes werden in einer Datei gespeichert. Am Anfang steht der eingegebene Quelltext und am Ende das ausführbare Maschinenprogramm. Als Erweiterung bekommt jede Datei eine spezielle Kennung während der Dateiname in der Regel identisch ist. Mit dem Editor schreiben wir den Quelltext und speichern diesen als NAME.MAC auf Diskette. Im nächsten Arbeitsschritt liest der Assembler den Quelltext NAME.MAC von Diskette und erzeugt daraus einen Zwischencode, dieser wird als NAME.REL wieder auf der Diskette abgelegt. Zusätzlich können wir uns eine Druckdatei NAME.PRN erzeugen lassen, welche das komplette Listing einschließlich der erzeugten Maschinencodes enthält. Für weitere Zusatzprogramme kann außerdem noch die Cross-Referenz in eine Datei NAME.CRF geschrieben werden. Um aus dem Zwischencode NAME.REL ein lauffähiges Programm zu erzeugen, benötigen wir noch einen Programmverbinder. Zunächst ist die Notwendigkeit dieses zusätzlichen Schrittes vielleicht nicht sofort erkennbar. Für kleine Programme, die nur aus einem Modul bestehen, würde sich der Schritt erübrigen wenn der Assembler direkt Maschinencode erzeugen könnte. Die Technik mit dem Zwischencode ermöglicht es jedoch, verschiedene Module zu verbinden (daher auch der Name Programmverbinder, engl. Linker). Es besteht also die Möglichkeit, immer wiederkehrende Funktionen in eigene Module auszulagern, diese Module bis zum Zwischencode zu übersetzen und später in verschiedene andere Programme einzubinden. Der dritte Arbeitsschritt ruft also den Linker auf, welcher alle erforderlichen Zwischencodemodule zu einem lauffähigen Programm zusammensetzt. Als Ausgabe wird vom Linker standardmäßig eine Datei NAME.COM erzeugt, die unter CP/M direkt durch Eingabe des Kommandos NAME gestartet werden kann. Alternativ ist die Erzeugung weiterer Dateiformate wie NAME.PRL oder NAME.SPR möglich. Diese Dateien sind auf bestimmte Adressen übersetzt und enthalten zusätzlich noch Informationen die eine nachträgliche Umrechnung auf andere Adressen gestattet. Meist werden solche Dateien benutzt, wenn erst während der Laufzeit die Adresse bekannt ist, auf der die

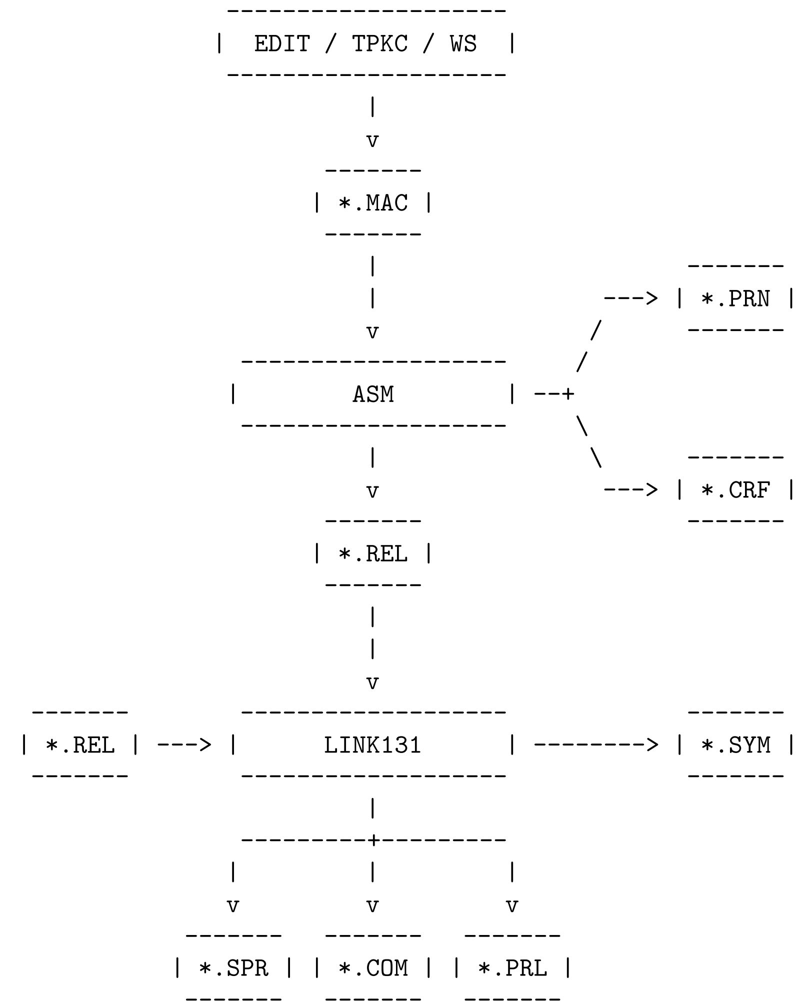


Bild 14: Zusammenwirken der verschiedenen Programme und Dateien beim Assembler unter CP/M

Programme abzuarbeiten sind. Ein Beispiel dafür sind die verschieblichen Treiber für das KC-System.

Weitere Werkzeuge der Assemblerprogrammierers sind:

- Verwaltungsprogramme für die Zwischencodemodule (LIB.COM)
- Debugger zum Testen von Maschinenprogrammen (DU.COM, ZSID.COM)
- Rückübersetzungsprogramme bzw. Reassembler (DISASS.COM, DZ.COM, RAZ80.COM)
- Cross-Referenz-Programme (CREF.COM)

9.2 Der Editor

9.2.1 Editierprogramme

Um einen Quelltext zu erzeugen, brauchen wir als erstes einen Editor, also ein Textverarbeitungsprogramm. Dieses ist nicht Bestandteil des Assembler-Programmpaketes, vielmehr kann jedes beliebige Textverarbeitungsprogramm eingesetzt werden, das in der Lage ist ASCII-Text zu erzeugen. Dazu gibt es auf dem KC 85 verschiedene Möglichkeiten. Stellvertretend ein paar Hinweise zu den gängigsten Programmen:

TPKC oder WS:
: Die zu Wordstar kompatible Textverarbeitung TPKC ist gut geeignet für die Erstellung von Assemblerquelltexten. Es stehen umfangreiche Funktionen zur Verfügung, eine Begrenzung des Textspeichers ist nur durch die Diskettenkapazität gegeben. Während man für normale Texte das Kommando „D“ benutzt, sollte man für Quelltexte das Kommando „N“ verwenden. Das hat den Vorteil, daß Tabulatorschritte als TAB-Steuerzeichen eingegeben werden können und man erreicht besser ein übersichtliches Aussehen des Programmes. Außerdem werden in diesem Modus keine „hohen“ Zeichen erzeugt was für manche Assemblerprogramme ein Problem darstellt.

EDIT80.COM:
: Zum robotron-Paket des Assemblers gehört neben ASM.COM und LINK.COM noch ein Editor mit dem Namen EDIT.COM (EDIT80 von Microsoft). EDIT80 ist ein „zeilen- und zeichenorientierter Textaufbereiter“. So umständlich wie dieser offizielle Name ist auch die Bedienung des Programmes. Der Vollständigkeit halber soll das Programm erwähnt werden. Wer gern mit Zeileneditoren arbeitet, kann ja EDIT für die Erzeugung des Quelltextes benutzen. Eine Beschreibung der Kommandos existiert als Textdatei EDIT80.DOK bzw. EDIT80.HLP.

ZDE.COM:
: Dieses Programm für das Z-System stellt eine kleine Nachbildung von Wordstar dar. Es ist schneller als TPKC und besitzt nicht den vollen Funktionsumfang. Als Grenze der Textgröße ist der zur Verfügung stehende TPA-Bereich ausschlaggebend. Für kleine Quelltexte (jedoch immerhin bis etwa 40 K) eignet sich ZDE hervorragend. Man sollte auch hier den Nicht-Dokument-Modus wählen, das erreicht man durch die Angabe der Option /N oder /A beim Aufruf des Editors bzw. entsprechender Installation. Wer sich mit den TPKC-Kommandos auskennt, wird sich auch schnell mit ZDE anfreunden. Voraussetzung: man hat das Z-System.

WordPro:
: Natürlich könnte man auch WP5 oder WP6 zur Erstellung der Quelltexte benutzen. Aber dazu muß man sich im Betriebssystem CAOS befinden – nach jedem Arbeitsschritt muß die Betriebsart gewechselt, also mit JUMP FC neu gebootet werden. Die Dateien von WordPro5 und WordPro6 sind bedingt ASCII-kompatibel, man sollte keine Druckersteuerzeichen verwenden und Umlaute sowie Sonderzeichen nur in Kommentaren einsetzen. Dennoch würde ich WordPro nur denjenigen empfehlen, die sich absolut nicht mit einem Textverarbeitungsprogramm von CP/M anfreunden können.

Im weiteren Verlauf des Beitrages werde ich davon ausgehen, daß mit TPKC gearbeitet wird. Die anderen Textverarbeitungsprogramme bleiben unberücksichtigt.

9.2.2 Das erste Programm unter CP/M: HALLO

Um nun ein erstes Gefühl mit dem Editor und Assembler unter CP/M zu bekommen, schlage ich vor, das erste Beispielprogramm aus Abschnitt 4.7 zu wiederholen. Die Aufgabe besteht also darin, die Zeichenkette „Hallo KC-Club !“ auf dem Bildschirm auszugeben. Zunächst solltet Ihr Euch vergewissern, daß alle benötigten Programme auf dem aktuellen Laufwerk (bzw. im Suchpfad) vorhanden sind: TPKC.COM, TPHT.OVR, TPOVLY0.OVR, ASM.COM, LINK.COM. Starten wir zunächst die Textverarbeitung durch Eingabe von:

```
A>TPKC
```

Nach ein paar Versionsanzeigen zum Terminal und Drucker erscheint das Anfangsmenü von TPKC. Wir wollen eine Programmdatei erstellen und drücken die Taste „N“. Nun werden wir nach dem Dateinamen gefragt, geben wir also ein HALLO.MAC ein – die Erweiterung .MAC sorgt dafür, daß der Assembler die Datei als Quelltext erkennt. Haben wir die Namenseingabe mit Enter bestätigt, dann erscheint für einen Moment die Anzeige „Neue Datei“ und ein leerer Bildschirm steht zur Eingabe bereit. Existiert bereits eine Datei HALLO.MAC auf der Diskette, dann wird diese geladen und angezeigt. Auf diese Weise ist das Ändern möglich, falls man eine Korrektur am Quelltext vornehmen muß. Beginnen wir also mit der Eingabe der ersten Programmzeilen. Wie bereits im Beispiel aus Punkt 4.7 beginnen wir mit ein paar Kommentaren:

```
;*****************************************
; Ausgabe von "Hallo KC-Club !" unter CP/M
;
; 01.02.1999
;
; Autor: Mario Leubner
;*****************************************
```

Unter EDAS mußten wir uns an dieser Stelle Gedanken machen, wo das Programm später abgearbeitet wird. CP/M-Programme laufen standardmäßig auf der Adresse 100H, da aber bis zur Erzeugung des Zwischencodes noch keine feste Adresse vergeben wird, ist kein ORG-Befehl erforderlich.

Nun kommen wir zum eigentlichen Programm. Bei unserem CAOS-Programm mußten wir zunächst ein Menüwort definieren, das ist bei CP/M nicht nötig. Hier werden die Programme mit dem Namen aufgerufen, der als Dateiname mit dem Typ .COM auf der Diskette steht. Um das Programm durch die Eingabe des Kommandos HALLO zu starten, ist also eine Datei HALLO.COM zu erzeugen. Dazu später mehr, wenn wir den Assembler und Linker aufrufen.

Unter CAOS hatten wir den Programmverteiler 1 mit der Funktion 23H (OSTR) aufgerufen um die Zeichenkette auszugeben. Unter CP/M gibt es eine ähnliche Systemschnittstelle, den BDOS-Aufruf auf der Adresse 0005H. Beschrieben sind die Funktionen mit allen Parametern im Handbuch für den Programmierer. Zur Ausgabe einer Zeichenkette benötigen wir die Funktion 9. Die Funktionsnummer ist generell im Register C zu übergeben, Registerpaar DE enthält weitere Parameter: in unserem Fall ist das die Adresse der Zeichenkette. Die Zeichenkette endet nicht wie unter CAOS mit einer Null, sondern mit dem Dollarzeichen. Damit ist unser Programm fertig und kann mit einem RET-Befehl beendet werden. Bei der Eingabe der Befehle erweist es sich als günstig, alle Marken linksbündig, die Befehle aber ebenfalls untereinander zu schreiben. Besonders einfach erreicht man das durch Verwendung der Tabulatortaste (Control-I oder Shift+Leertaste). Schreiben wir im Quelltext also folgende Programmzeilen:

```
BDOS    EQU     0005H   ; Vereinbarung der Systemschnittstelle

        LD      C,9     ; Funktionsnummer in Register C
        LD      DE,TEXT ; Adresse der Zeichenkette
        CALL    BDOS    ; Aufruf der BDOS-Funktion
        RET             ; Programmende

TEXT:   DB      'Hallo KC-Club !$'
        END
```

Zunächst wird das Symbol BDOS definiert und mit dem Wert 0005H belegt. Später kann man im Programm statt CALL 5 auch den Befehl CALL BDOS schreiben, damit weiß man sofort, daß es sich um den Systemaufruf handelt. Vorher müssen wir aber noch dafür sorgen, daß die benötigten Register mit den richtigen Werten belegt sind. Register C wird mit dem Wert 9 geladen und DE mit der Adresse der Zeichenkette. Hier bediene ich mich wiederum eines Symbols, das im Programm erst später definiert wird. In der letzten Programmzeile steht diese Marke TEXT, gefolgt von einem Doppelpunkt. Dieser Doppelpunkt ist unter CP/M (im Gegensatz zu Edas) dringend erforderlich. Wenn er vergessen wird, erhält man eine Fehlermeldung. Die Zeichenkette selbst folgt als Definition mit der Pseudooperation DB, dabei an das abschließende Dollarzeichen denken! Ein häufiger Fehler ist das Vergessen dieses Zeichens. Der Assembler und Linker erkennt den Fehler nicht und das scheinbar fehlerfrei übersetzte Programm funktioniert nicht. Der letzte Befehl eines jeden Programme lautet END. Unter Edas 1.6 ist dieser Befehl optional, bei CP/M muß er geschrieben werden. Ist soweit alles eingetippt, kann der Quelltext abgespeichert und die Textverarbeitung beendet werden (Kommando: Control-K-X). Zur Kontrolle lassen wir uns das Directory anzeigen, es sollte die Datei HALLO.MAC enthalten.

9.3 Der Assembler ASM.COM

Wenn das Assemblerprogramm vollständig entworfen ist, kann es übersetzt werden. Der Assembler übersetzt die Anweisungen der Quelldatei einschließlich der Makroerweiterungen und Wiederholungspseudooperationen. Das Ergebnis ist ein verschieblicher Objektcode, der mit dem Programmverbinder gebunden und geladen werden kann. Der verschiebliche Objektcode kann in eine Datei vom Typ .REL geschrieben werden. Diese ist nicht direkt ausführbar, sondern erst nach der Behandlung durch den Programmverbinder.

Der Assembler übersetzt die Quelldatei in zwei Pässen: Pass 1 berechnet wieviel Code zu erzeugen ist, bildet eine Symboltabelle und erweitert die Makroanweisungen. Während Pass 2 wird dann der verschiebliche Code erzeugt und ausgegeben.

9.3.1 Überblick

ASM.COM unterstützt 2 Assemblersprachen: Das sind 8080-Mnemonik und Z80-Mnemonik, in unserem Kurs werden wir uns auf den leistungsfähigeren Z80-Modus beschränken.

Bei ASM.COM handelt es sich um einen Makroassembler: Der Programmierer schreibt einen Block mit einer Anzahl Befehlen, dieser Block erhält einen Namen. Ein solcher Block wird Makro- Definition genannt und erzeugt noch keinen eigenen Code. Später schreibt man im Quelltext nur den Makro-Namen und eventuell erforderliche Parameter hinzu, dies ist der Makro-Aufruf. Während der Übersetzung des Quelltextes werden die Makro-Aufrufe durch die Befehle aus der Makro-Definition ersetzt. Dieser Vorgang nennt sich Makro-Expansion oder Makro-Erweiterung. Zu den Makrooperationen zählen auch Wiederholungsbefehle. Makros können geschachtelt werden, in einem Makro kann also ein weiterer Makro aufgerufen werden. Die Schachtelungstiefe von Makros ist nur durch den Speicherplatz beschränkt.

ASM.COM unterstützt außerdem bedingte Assemblierung. Der Programmierer kann eine Bedingung bestimmen unter welcher ein Teil des Programmes assembliert wird oder nicht. Bedingungen können bis zu 255-mal geschachtelt werden.

9.3.2 Aufbau der Quelldatei

Der Aufbau eines Assemblerquelltextes unterscheidet sich nicht wesentlich von einem EDAS-Quelltext. In jeder Zeile steht eine Anweisung, die in die bekannten Felder unterteilt sind:

Symbol Operation Argument Kommentar

Alle Felder sind optional, auch Leerzeilen sind möglich. Die Anweisungen beginnen in einer beliebigen Spalte. Zwischen den einzelnen Feldern muß mindestens ein Leerzeichen oder Tabulator stehen, zur besseren Lesbarkeit können mehrere eingefügt werden.

Symbol: Dieses Feld kann eines der drei Symboltypen enthalten:

- Marke
- Public
- External

gefolgt von einem Doppelpunkt, außer in SET-, EQU- oder MAKRO-Anweisungen.

Ein Symbol darf aus den folgenden Zeichen bestehen:

A...Z a...z 0...9 $. ? @ _

Es darf jedoch nicht mit einer Ziffer oder dem Unterstreichen-Zeichen beginnen.

Symbole dürfen keine Registernamen sein.

Operation: Dieses Feld enthält einen Operationscode, eine Pseudooperation, einen Makro-Namen oder einen Ausdruck.

Argument: Dieses Feld enthält Ausdrücke, Variable, Registernamen, Operanden oder Operatoren.

Kommentar: Dieses Feld enthält Kommentare, die immer mit einem Semikolon beginnen.

9.3.2.1 Public und External

Neben den bereits von EDAS bekannten „normalen“ Marken, kann ASM mit zwei weiteren Symboltypen arbeiten. Ein Public-Symbol wird wie eine Marke definiert, der

Unterschied besteht darin daß ein solches Symbol auch von anderen Programmteilen genutzt werden kann. Ein Symbol wird zum Public durch die Verwendung von 2 Doppelpunkten:

```
MARKE:: RET
```

oder einer der Pseudooperationen PUBLIC, ENTRY oder GLOBAL:

```
        PUBLIC  MARKE

MARKE:  RET
```

Das Resultat beider Varianten ist identisch. Um ein solches Symbol in einem anderen Programmteil zu verwenden, muß es in diesem Teil als externes Symbol definiert werden. Auch dafür existieren zwei gleichwertige Varianten. Entweder man schreibt zwei Doppelkreuze nach dem Symbolnamen:

```
        CALL    MARKE##
```

oder benutzt eine der Pseudooperationen EXT, EXTRN oder EXTERNAL

```
        EXT     MARKE
        CALL    MARKE
```

Die Wertzuweisung der externen Symbole erfolgt erst während der Programmverbindung mit dem Linker. Wurde ein benutztes externes Symbol in keinem der geladenen Module als Public definiert, dann erhält man beim Linken eine Fehlermeldung.

9.3.2.2 Speichersegmente (PC-Modus)

Im Gegensatz zu EDAS kann ASM zwischen 4 Speichersegmenten unterscheiden. Der Wert eines Symbols wird gemäß seinem PC-Modus (Speicherplatzzuordnungszählermodus) bestimmt. Mittels Pseudooperationen wird festgelegt, in welches Segment der erzeugte Code geladen wird. Die 4 Segmente sind:

ASEG	absolutes Segment
CSEG	coderelatives Segment (Standardeinstellung)
DSEG	datenrelatives Segment
COMMON	commonrelatives Segment

Absoluter Modus:
: Dieser Modus erzeugt nichtverschieblichen Code. Der Programmierer legt mittels ORG-Anweisung die Adresse fest, auf der der Programmblock ausgeführt wird. Dieser Modus ist vergleichbar mit der Arbeitsweise von EDAS.

Coderelativer Modus:
: Der coderelative (programmrelative) Modus erzeugt Code für einen verschieblichen Programmteil. Das ist die Standardeinstellung von ASM.COM und Voraussetzung für das Verbinden mehrerer Module zu einem Programm. Die Adreßzuweisung erfolgt erst beim Linken.

Datenrelativer Modus:
: Dieser Modus erzeugt verschieblichen Code für Datenbereiche. Eine Unterscheidung zwischen coderelativen und datenrelativen Modus ist erforderlich wenn der Programmcode im ROM abgearbeitet werden soll und die Datenbereiche im RAM liegen. Für eine übersichtliche Programmierung empfiehlt sich auch sonst eine Trennung der Bereiche.

Commonrelativer Modus:
: In diesem Modus erzeugter Code wird in einen gemeinsamen Datenbereich geladen. So kann von unterschiedlichen Modulen auf die gleichen Datenspeicherplätze zugegriffen werden.

9.3.3 Operationscodes

Operationscodes sind mnemonische Namen für die CPU-Befehle (Maschinenanweisungen). Die zulässigen Befehle werden nicht vom verwendeten Assemblerprogramm bestimmt, sondern von der CPU. An dieser Stelle müssen wir nichts Neues lernen. Die tabellarischen Übersichten aus Abschnitt 3.3 gelten also auch für ASM.COM.

9.3.3.1 Vergleich ASM – EDAS

Dennoch sollten wir uns die Syntax der Befehle etwas genauer anschauen, denn der Teufel steckt mitunter im Detail. Die folgenden Informationen sind besonders hilfreich, wenn es darum geht, vorhandene EDAS-Quellen für ASM.COM aufzubereiten. Für gleiche CPU-Befehle sind mitunter abweichende Schreibweisen anzuwenden, die nachstehende Tabelle zeigt die Unterschiede:

EDAS			ASM
EX AF	oder	EX AF,AF’	EX AF,AF’
OUT n,A	oder	OUT (n),A	OUT (n),A
IN A,n	oder	IN A,(n)	IN A,(n)
ADD x	oder	ADD A,x	ADD A,x
ADC x	oder	ADC A,x	ADC A,x
SUB x	oder	SUB A,x	SUB x
SBC x	oder	SBC A,x	SBC A,x
AND x	oder	AND A,x	AND x
XOR x	oder	XOR A,x	XOR x
OR x	oder	OR A,x	OR x
CP x	oder	CP A,x	CP x

Das x steht dabei in den logischen und arithmetischen Befehlen wahlweise für einen Registernamen oder einen Direktoperanden. Wie zu erkennen ist, unterstützt EDAS alle Formen wie sie unter ASM gefordert werden. EDAS ist aber etwas toleranter. Außerdem sind unter ASM keine illegalen Z80-Befehle zugelassen. Bei der Übernahme von EDAS-Quellen für ASM sollte weiterhin beachtet werden, daß nach Marken ein Doppelpunkt geschrieben werden muß. Das in EDAS für den Programmzähler als Alternative zu $ zugelassene # existiert bei ASM auch nicht. Statt (HL) kann sowohl bei EDAS als auch ASM der „Registername“ M benutzt werden. Diese Schreibweise wird jedoch nicht von allen Assemblern unterstützt und sollte unter CP/M möglichst vermieden werden.

9.3.4 Operanden und Operatoren

Die Argumente von Operationscodes und Pseudooperationen werden gewöhnlich als Ausdrücke bezeichnet, da sie mathematischen Ausdrücken ähneln. Ausdrücke können einen oder mehrere Operanden beinhalten. Wenn ein Ausdruck mehrere Operanden hat, sind diese durch Operatoren verbunden. Gültige Operanden sind Zahlen, Zeichen bzw. Zeichenketten oder Symbole.

Zahlen: Mit ASM kann in unterschiedlichen Zahlensystemen gearbeitet werden. Standardmäßig ist das Dezimalsystem (Basis 10) eingestellt. Diese Basis kann von 2 (binär) bis 16 (hexadezimal) ausgewählt werden. Unabhängig von der gerade eingestellten Zahlenbasis kann eine Zahl wie folgt dargestellt werden:

nnnnB	binär (falls Basis kleiner als 12 ist)
nnnnD	dezimal (falls Basis kleiner als 14 ist)
nnnnO	oktal
nnnnQ	oktal
nnnnH	hexadezimal
X’nnnn’	hexadezimal

Zeichen(ketten):
Eine Zeichenkette besteht aus null oder mehr Zeichen und wird mit Anführungszeichen oder Apostrophe begrenzt. Die Begrenzer können als Zeichen verwendet werden, wenn sie für jedes Vorkommen doppelt geschrieben werden. Läßt sich die Zeichenkette in 2 Byte darstellen, können damit weitere Verknüpfungen erfolgen. In diesem Fall spricht man von einer Zeichenkonstanten. Beispiele:

’A’	ist eine Zeichenkette
“A“	ist die gleiche Zeichenkette
’A’+1	ist eine Zeichenkonstante mit dem Wert 42H
’AB’	ist eine Zeichenkette
’AB’+1	ist eine Zeichenkonstante mit dem Wert 4143H
“Text$“	ist eine Zeichenkette

Symbole: Ein Symbol kann als Operand verwendet werden. Das Symbol wird dabei ausgewertet und der Wert für das Symbol eingesetzt. Ein spezielles Symbol gibt die Adresse des Speicherzuordnungszählers wieder: $

Rang	Operator	Funktion
(1)	NUL	ergibt wahr, wenn das Argument (ein Parameter) Null ist
(1)	TYPE	gibt ein Byte mit der Eigenschaft des Arguments zurück, Bit 0,1 geben den Modus an: 0000 0000 absolut 0000 0001 coderelativ 0000 0010 datenrelativ 0000 0011 commonrelativ Bit 7 ist gesetzt, wenn der Ausdruck ein EXTERNAL enthält. Bit 5 ist gesetzt, wenn der Ausdruck lokal definiert wurde.
(2)	LOW	Isolieren der niederwertigen 8 Bit eines 16-Bit-Wertes
(2)	HIGH	Isolieren der höherwertigen 8 Bit eines 16-Bit-Wertes
(3)	*	Multiplikation
(3)	/	Division
(3)	MOD	Modulo (Restdivision)
(3)	SHR	Rechtsverschiebung um die Anzahl der angegebenen Bit-Positionen
(3)	SHL	Linksverschiebung um die Anzahl der angegebenen Bit-Positionen
(4)	−	Vorzeichen Minus – zeigt an, daß der folgende Wert negativ ist
(5)	+	Addition
(5)	−	Subtraktion
(6)	EQ	gleich
(6)	NE	ungleich
(6)	LT	kleiner als
(6)	LE	kleiner oder gleich
(6)	GT	größer als
(6)	GE	größer oder gleich
(7)	NOT	bitweise Negation des folgenden Operanden
(8)	AND	logisches UND
(9)	OR	logisches ODER
(9)	XOR	exclusives ODER

9.3.4.1 Liste der Operatoren

Der Assembler ASM.COM erlaubt sowohl arithmetische als auch logische Operatoren. Für logische Operationen gilt dabei die folgende Zuordnung: falsch gleich Null und wahr ungleich Null. Alle Operatoren außer +, -, * und / müssen von ihren Operanden durch wenigstens ein Leerzeichen getrennt werden. Die Reihenfolge der Abarbeitung von Ausdrücken mit mehreren Operatoren wird von deren Rangordnung bestimmt. Diese kann unter Verwendung runder Klammern um den Teil des Ausdrucks, der eine höhere Rangordnung erhalten soll, verändert werden. In der nebenstehenden Tabelle ist die Rangordnung in Klammern angegeben, eine kleinere Zahl bedeutet eine höhere Rangordnung.

9.3.5 Pseudooperationen

Im Gegensatz zu den 10 Pseudooperationen von EDAS besitzt ASM immerhin 75 dieser Operationen. Beginnend bei einfachen Einzelfunktions-Pseudooperationen bis hin zur Makroprogrammierung ergibt sich ein großes Anwendungsgebiet. Dabei ist es nicht wichtig, von Anfang an alle Befehle zu kennen. Für die meisten Anwendungen reichen eine Handvoll Pseudooperationen aus. Man sollte aber in etwa einen Überblick haben, welche Möglichkeiten es gibt, um bei Bedarf im Handbuch oder der HELP-Datei nachzulesen. Dem Anfänger bleibt es also selbst überlassen, dieses Kapitel durchzuarbeiten, es nur zu überfliegen oder gleich mit dem nächsten Kapitel fortzufahren. Auf alle Fälle kann man bei Bedarf auf diese Seiten zurückblättern. Die Beschreibung der Pseudooperationen wurde aus der ASM.HLP unkommentiert übernommen.

9.3.5.1 Daten- und Symboldefinition

DEFB exp[, exp[, ...]]
DEFM exp[, exp[, ...]]
DS exp[, exp[, ...]]

> Definiert Datenbytes, bei DEFB sind nur Ausdrücke zugelassen, die einzelne Bytes erzeugen, bei DEFM Zeichenketten und bei DB beides.

DC exp

> Definiert eine Zeichenkette, wobei im letzten Zeichen das höchstwertige Bit gesetzt wird.

DW exp[, exp[, ...]]
DEFW exp[, exp[, ...]]
Definiert Datenworte, das niederwertige Byte zuerst.

DS exp[, value]
DEFS exp[, value]
Reserviert Speicherplatz, soll der Speicher mit einem Wert initialisiert werden, ist value anzugeben.

label EQU exp
Weist der Marke einen festen Wert zu.

label ASET exp
label DEFL exp
Weist der Marke einen Wert zu, der später wieder geändert werden kann.

EXT label[, label[, ...]]
EXTRN label[, label[, ...]]
EXTERNAL label[, label[, ...]]
BYTE EXT label[, label[, ...]]
BYTE EXTRN label[, label[, ...]]
BYTE EXTERNAL label[, label[, ...]]
Erklärt die Namen als externals, d.h. sie werden in einem anderen Modul definiert.

ENTRY label[, label[, ...]]
GLOBAL label[, label[, ...]]
PUBLIC label[, label[, ...]]
Erklärt die Namen als allgemein gültig (public), d.h. sie können auch in anderen Modulen benutzt werden.

9.3.5.2 PC-Modus

ASEG
Absolutes Segment auf Adresse 0 oder direkt nach dem letzten ASEG.

ORG exp
Wert des Speicherplatzzuordnungszählers ändern.

CSEG
Coderelatives Segment (Programm) einstellen.

DSEG
Der folgende Code wird in das Datensegment geschrieben.

COMMON /blockname/
Legt einen gemeinsamen Datenbereich für alle COMMON-Blöcke an.

.PHASE exp
Der folgende absolute Code wird für die Ausführung auf der angegebenen (absoluten) Adresse erzeugt.

.DPHASE
Beendet die durch .PHASE eingeleitete Verschiebung, zu jedem .PHASE muß ein abschließendes .DEPHASE existieren. Der Code innerhalb eines .PHASE-Blockes wird als absolut erzeugt.

9.3.5.3 Dateibezogene Operationen

NAME (“string“)
Definiert den Namen für ein Modul, nur die ersten 6 Zeichen werden ausgewertet. Ein Modulname kann auch mit der Pseudooperation TITLE definiert werden. Wenn keine der beiden Pseudooperationen NAME oder TITLE vorhanden sind, wird der Modulname aus dem Namen der Quelldatei erzeugt.

INCLUDE filename[.typ]
$INCLUDE filename[.typ]
MACLIB filename[.typ]
Alle drei Versionen der INCLUDE-Anweisung sind sinnverwandt. Während der Assemblierung fügen sie Quellcode aus einer anderen Quelldatei ein. Der Dateiname ist in Großbuchstaben (!) zu schreiben, zusätzlich kann eine Laufwerksangabe erfolgen. Der Standard-Dateityp .MAC wird verwendet, falls nichts anderes angegeben wurde.

END [exp]
Die End-Anweisung bezeichnet das Programmende. Als optionale Angabe kann dem Programmverbinder die Startadresse des Programms mitgeteilt werden.

9.3.5.4 Listensteuerung

Die von ASM wahlweise Möglichkeit zur Ausgabe einer Druckliste kann mit den folgenden Pseudooperationen beeinflußt werden. Diese Befehle haben keinen Einfluß auf den zu erzeugenden Code!

TITLE string
: Bezeichnet eine Kopfzeile, die im Listing auf jede Seite als erste Zeile gesetzt wird. Wenn keine NAME-Anweisung geschrieben wurde, werden die ersten 6 Zeichen des Titels als Modulname verwendet.

SUBTTL string
$TITLE string
: Bezeichnet einen Untertitel, der in jedem Seitenkopf in die Zeile unter den Titel gedruckt werden soll.

*EJECT [length]
$EJECT
PAGE [length]
: Veranlaßt den Assembler, die Ausgabe mit einer neuen Seite zu beginnen. Der optionale Wert length bestimmt die Seitenlänge der neuen Seite in Zeilen und muß im Bereich zwischen 10 und 255 liegen, die Standardzeilenzahl ist 50. *EJECT muß in der Spalte 1 beginnen!

Die folgenden Pseudooperationen schalten zwischen verschiedenen Zuständen um, es ist jeweils nur der zuletzt geschriebene Befehl wirksam:

.LIST Protokollieren aller Zeilen mit ihrem Code (Standard)

.XLIST Unterdrücken des Protokollierens

.SFCOND
: unterdrückt „falsche“ Bedingungen im Listing

.LFCOND
: protokolliert „falsche“ Bedingungen

.TFCOND
: kippt die aktuelle Einstellung des /X-Schalters in der Kommandozeile um (unabhängig von .SFCOND und .LFCOND)

/X und .TFCOND	−>	„falsche“ Bedingungen drucken
kein /X und .TFCOND	−>	„falsche“ Bedingungen unterdrücken

.LALL protokolliert Makroanweisungen mit allen Zeilen

.SALL unterdrückt Makroanweisungen mit allen Zeilen

.XALL protokolliert nur Zeilen, die Code erzeugen (Standard)

.CREF Erzeugen einer Cross-Referenz (Standard)

.XCREF Unterdrücken der Cross-Referenz

9.3.5.5 Makro-Operationen

ASM.COM unterstützt Makros. Die Makrofähigkeit erlaubt, Blöcke von Anweisungen zu schreiben, die wiederholt verwendet werden können, ohne daß sie erneut geschrieben werden müssen. Diese Anweisungsblöcke beginnen entweder mit einer Makrodefinition oder einer Wiederholungspseudooperation und enden mit der Pseudooperation ENDM. Die folgenden Pseudooperationen werden von ASM.COM unterstützt:

MACRO	Makrodefinition
REPT	Wiederholung
IRP	undefinierte Wiederholung
IRPC	undefinierte Wiederholung mit Zeichen
ENDM	Makro-Ende
EXITM	vorzeitiges Makro-Ende
LOCAL	Symbol gilt nur innerhalb des Makroblocks

Makro-Definition und Aufruf:

```
label   MACRO   [dummy[,dummy...]]
        :
        :
        ENDM
```

Der Anweisungsblock von MACRO bis ENDM bildet die Makrodefinition. Nachdem der Makro definiert worden ist, kann label zum Aufrufen des Makros verwendet werden. Die Makrodefinition muß vor dem ersten Makroaufruf stehen! Jeder symbolische

Parameter dummy wird bei der Verwendung des Makros durch den echten Parameter ersetzt.

```
label   [parameter[,parameter...]]
```

Diese Anweisung ruft den Makro label auf. Die Parameter ersetzen 1:1 die im Makro enthaltenen dummys.

ENDM Damit wird dem Assembler mitgeteilt, daß der Makro- oder Wiederholungsblock zu Ende ist. Jede Pseudooperation MACRO, REPT, IRP und IRPC muß mit einem zugehörigen ENDM abgeschlossen werden.

EXITM Wenn ein Makro- oder Wiederholungsblock verlassen werden soll, bevor die Erweiterung vollständig ist, wird die Pseudooperation EXITM verwendet. Meistens wird EXITM in Verbindung mit einer bedingten Assemblierung verwendet. Wenn ein EXITM übersetzt wird, werden alle folgenden Anweisungen bis zu ENDM nicht generiert.

LOCAL dummy[,dummy...]
Die Pseudooperation LOCAL wird nur innerhalb eines Makrodefinitionsblockes verwendet. Wenn LOCAL ausgeführt wird, legt der Assembler ein einmaliges Symbol für jedes dummy an und ersetzt dieses Symbol für jedes Auftreten von dummy in diesem Anweisungsblock. Auf diese Weise vermeidet man Mehrfachdefinitionen bei mehrfacher Verwendung der Makros. Die vom Assembler angelegten Symbole reichen von ..0001 bis ..FFFF. Der Anwender sollte Symbole der Form ..nnnn vermeiden! Die LOCAL-Anweisung muß allen anderen Anweisungstypen in der Makrodefinition vorausgehen.

9.3.5.6 Wiederholungspseudooperationen

Wiederholungspseudooperationen sind eine spezielle Form der Makroprogrammierung. Hier gibt es jedoch keine Unterscheidung in Definition und Aufruf, die Wiederholung wird sofort ausgeführt. Wie bei den Makros müssen die Wiederholungsblöcke jeweils mit dem Befehl ENDM abgeschlossen werden.

```
[label] REPT    exp
        :
        ENDM
```

Wiederholen des Anweisungsblockes entsprechend des Wertes von exp.

```
[label] IRP     dummy,<parameter_list>
        :
        ENDM
```

Der Anweisungsblock wird für jeden Parameter wiederholt. Die Parameter können Symbole, Zeichenketten, Zahlen oder Zeichenkonstanten sein. Die Parameter müssen in spitze Klammern eingeschlossen werden.

```
[label] IRPC    dummy,string
        :
        ENDM
```

Der Anweisungsblock wird für jedes Zeichen der Zeichenkette wiederholt. Es wird dummy jeweils durch ein Zeichen der Zeichenkette ersetzt.

9.3.5.7 Bedingte Assemblierung

Die Pseudooperationen zur bedingten Assemblierung ermöglichen dem Anwender, Codeblöcke zu entwerfen, die spezielle Bedingungen testen und entsprechend vorgehen. Alle Bedingungen haben folgendes Format:

```
IFxxxx [argument]               COND [argument]
 :                               :
 :                               :
[ELSE                           [ELSE
 :                               :
 : ]                             : ]
ENDIF                           ENDC
```

Zu jedem IF/COND muß ein zugehöriges ENDIF/ENDC die Bedingung abschließen. Der Assembler bewertet Bedingungen mit „wahr“ (Wert ungleich Null) oder mit „falsch“ (Wert gleich Null). Der Code im Bedingungsblock wird übersetzt, wenn der Wert der Bedingung entspricht. Andererseits ignoriert der Assembler den Bedingungsblock oder übersetzt den ELSE-Teil.

IF exp
IFT exp
COND exp
: Der folgende Code wird übersetzt, wenn exp „wahr“ ist, also nicht Null ergibt.

IFE exp
IFF exp
: Der folgende Code wird übersetzt, wenn exp „falsch“ ist, also Null ergibt.

IF1
: Die folgenden Anweisungen werden nur im Pass 1 übersetzt.

IF2
: Die folgenden Anweisungen werden nur im Pass 2 übersetzt.

IFDEF symbol
: Die folgenden Anweisungen werden übersetzt, wenn das Symbol definiert ist oder als EXTERNAL erklärt wurde.

IFNDEF symbol
: Die folgenden Anweisungen werden übersetzt, wenn das Symbol nicht definiert ist und nicht als EXTERNAL erklärt wurde.

IFB <string>
: Die folgenden Anweisungen werden übersetzt, wenn der String leer ist (zum Testen symbolischer Parameter erforderlich).

IFNB <string>
: Die folgenden Anweisungen werden übersetzt, wenn der String nicht leer ist (zum Testen symbolischer Parameter erforderlich).

IFIDN <string1>,<string2>
: Die folgenden Anweisungen werden übersetzt wenn die beiden Zeichenketten identisch sind.

IFDIF <string1>,<string2>
: Die folgenden Anweisungen werden übersetzt wenn die beiden Zeichenketten verschieden sind.

ELSE
: Umkehrung des IF-Status, um alternativen Code zu erzeugen. ELSE kann mit jeder beliebigen Bedingung verwendet werden, je Bedingung ist nur ein ELSE zulässig.

ENDIF
ENDC
: Abschluß eines Bedingungsblockes. ENDIF muß ein zugehöriges IF haben und ENDC ein zugehöriges COND.

9.3.5.8 Verschiedenes

ASM.COM übersetzt normalerweise Z80-Mnemoniks, als Alternative kann aber auch in einen Modus gewechselt werden, der 8080- Mnemoniks übersetzt. Solange im Z80-Modus gearbeitet wird, muß also keiner der folgenden Befehle benutzt werden:

.Z80 Übersetzung von Z80-Befehlscodes (Standard)

.8080 Übersetzung von 8080-Befehlscodes

.RADIX exp
: Setzt die aktuelle Zahlenbasis, der dezimal anzugebende Wert kann im Bereich von 2 bis 16 gewählt werden. Die Standardeinstellung ist 10, also das Dezimalsystem.

.REQUEST dateiname[,dateiname...]
: Forderung an den Programmverbinder, die in der Liste enthaltenen Dateien nach undefinierten externen Symbolen zu durchsuchen. Diese Pseudooperation funktioniert leider in der robotron-Version ASM.COM nicht korrekt und sollte nach Möglichkeit nicht verwendet werden. Die zu durchsuchenden Dateien können dem Programmverbinder auch später noch mitgeteilt werden.

.PRINTX delimiter text delimiter
: Der zwischen den Begrenzerzeichen stehende Text wird während der Übersetzung auf dem Bildschirm angezeigt.

.COMMENT delimiter text delimiter
: Der zwischen den Begrenzerzeichen stehende Text wird als Kommentar gewertet. Für lange Kommentare kann so das Semikolon entfallen, das sonst in jeder einzelnen Zeile stehen müßte.

9.3.5.9 Spezielle Zeichen

;	leitet einen Kommentar ein
;;	Kommentar in einem Makroblock, nicht Teil der Erweiterung
::	zwei Doppelpunkte definieren eine Marke als Public-Symbol
##	zwei Doppelkreuze erklären ein Symbol als Extern
$	aktuelle Programmadresse
&	verbindet Text und/oder Symbole in Makros
!	zeigt im Makroargument an, daß das folgende Zeichen ein Literal ist
%	Konvertierung eines symbolischen Parameters im Makroaufruf (meist für .PRINTX verwendet)

Zu beachten ist bei der Konvertierung von EDAS-Quellen, daß das Zeichen % bei EDAS für die MODULO-Division genutzt wird. Bei ASM.COM muß dafür der Operator MOD verwendet werden, während das Zeichen % eine Sonderfunktion innerhalb von Makros hat.

9.3.6 Aufrufen des Assemblers

Nach dieser doch recht trockenen theoretischen Kost, kommen wir zu einer praktischen Angelegenheit – dem Aufruf des Assemblers. Doch auch hier geht es nicht ganz ohne Vorkenntnisse. Also noch ein wenig Grundlagen bevor wir mit dem ersten Beispielprogramm diese zur Anwendung bringen. Der Assembler wird vom CP/M-Prompt aufgerufen durch die Eingabe von:

```
A>ASM [Object][,[List]]=Source[/Switch]
```

Die Kommandozeile des Assemblers setzt sich aus 4 Feldern zusammen. ASM kann auch ohne Argumente aufgerufen werden. Dann wird nur der Assembler geladen und gibt einen Stern (*) aus, das ist das Bereitschaftszeichen, daß der Assembler weitere Kommandos erwartet. Wenn das ASM-Kommando direkt in der CP/M-Kommandozeile steht, kehrt der Assembler nach Abarbeitung nicht mit * zurück sondern gibt die Steuerung zurück an CP/M. Bei Eingabe der Kommandos am Prompt von ASM besteht zusätzlich die Möglichkeit, weitere Übersetzungen auszuführen ohne daß der Assembler neu geladen werden muß. Um von der Eingabebereitschaft des Assemblers (*) zu CP/M zurückzukehren ist die BRK-Taste (^C) zu drücken.

Source: Um einen Quelltext zu übersetzen, muß wenigstens das Gleichheitszeichen und der Name der Quelldatei angegeben werden. Der Dateityp kann weggelassen werden, wenn *.MAC verwendet wurde. Das einfachste Kommando lautet also:

```
A>ASM =HALLO
```

Dieses Kommando veranlaßt den Assembler, die Quelldatei HALLO.MAC zu übersetzen und das Ergebnis in einer verschieblichen Objectcodedatei HALLO.REL abzulegen.

Object: Die Eintragung object in der Kommandozeile ist optional. Wie wir gesehen haben wird auch ohne diese Angabe eine Objectcodedatei erzeugt. Object muß angegeben werden wenn der Name der Objectcodedatei nicht mit dem Namen der Quelldatei übereinstimmt. Soll keine Objectcodedatei ausgegeben werden, dann muß das Kommando mit dem Komma der Listdatei beginnen:

```
A>ASM ,=HALLO
```

erzeugt demnach keine Ausgabedatei. Diese Syntax wird benutzt um zu überprüfen ob der Quelltext fehlerfrei ist. Weil keine Dateien ausgegeben werden müssen, ist der Assemblerlauf schneller beendet.

List: Die Eintragung List in der Kommandozeile ist immer optional. Eine Listdatei wird nur erzeugt, wenn die Eintragung vorhanden ist, das Komma vor dem Dateinamen ist zu schreiben. Der Assembler ergänzt den Dateityp .PRN falls nichts anderes angegeben wurde. Das Kommando

```
A>ASM ,HALLO=HALLO
```

übersetzt HALLO.MAC und erzeugt nur die Druckdatei HALLO.PRN. Wird zusätzlich die Ausgebe der Objectdatei gewünscht, muß dies so angegeben werden:

```
A>ASM HALLO,HALLO=HALLO
```

9.3.6.1 Schalter

Die Abarbeitung des Assemblers kann über zusätzliche Funktionen beeinflußt werden. Dafür sind Schalter im Kommando anzugeben. Das sind Buchstaben, vor denen ein Schrägstrich steht. Es kann eine beliebige Anzahl von Schaltern angegeben werden, aber vor jedem Schalter muß ein Schrägstrich stehen. Die folgende Tabelle enthält die möglichen Schalter und deren Bedeutung:

Schalter	Bedeutung
/H	Adressen in der Ausgabedatei hexadezimal drucken (Standard)
/O	Adressen in der Ausgabedatei oktal drucken
/R	erzwingt die Erzeugung einer Objectdatei *.REL
/L	erzwingt die Erzeugung einer Druckdatei *.PRN
/C	erzwingt die Erzeugung einer Cross-Referenzdatei *.CRF
/Z	Zilog-Opcodes (Z80) übersetzen (Standard)
/I	Intel-Opcodes (8080) übersetzen
/P	Jedes /P legt einen extra Stack von 256 Byte an. /P sollte verwendet werden wenn Stack-Überlauf-Fehler gemeldet werden.
/X	Unterdrücken der Liste falscher Bedingungen, siehe .TFCOND
/M	initialisiert den Blockdatenbereich, der mit DS definiert wurde auf 00H, anderenfalls ist der Inhalt unbestimmt.

9.3.6.2 Fehlermeldungen und Fehlercodes

Während der Übersetzung auftretende Fehler werden entweder als Meldung oder Fehlercode am Bildschirm angezeigt sowie in Spalte 1 der Druckdatei gedruckt. Die Fehlercodes bestehen jeweils aus einem Buchstaben mit folgender Bedeutung:

A = Argument Error
: Das zu der Pseudooperation gehörige Argument befindet sich nicht im korrekten Format oder es liegt außerhalb des zugelassenen Bereiches.

C = Conditional nesting error
: Fehler in der Bedingungsschachtelung: ELSE ohne IF, ENDIF ohne IF, zweimal ELSE für ein IF, ENDC ohne COND.

D = Double Defined Symbol
: Bezugnahme auf ein Symbol, das mehrfach definiert wurde.

E = External error
: Verwendung eines externen Symbols, das in diesem Zusammenhang nicht erlaubt ist.

M = Multiply Defined Symbol
: Definition eines Symbols, das bereits definiert wurde.

N = Number error
: Fehler in einer Zahl, meist falsche Schreibweise oder ein falsches Zeichen.

O = Bad opcode or objectionable syntax
: Falscher Operationscode oder nicht einwandfreie Syntax: SET, EQU oder MACRO ohne einen Namen, falsche Syntax in einem Operationscode oder falsche Syntax in einem Ausdruck (nichtpaarige runde Klammern, Anführungszeichen, aufeinanderfolgende Operatoren usw.)

P = Phase error
: Phasenfehler: Der Wert einer Marke oder eines EQU-Namens ist im Pass 2 anders als im Pass 1.

Q = Questionable
: Fragwürdig: meistens ist eine Zeile nicht ordentlich abgeschlossen oder eine nicht zugelassene Befehlsform. Dies ist eine Warnung.

R = Relocation
: Verschiebung: Nichterlaubte Anwendung einer Verschiebung in einem Ausdruck (z.B. abs-rel). Daten-, Code- und Common- Bereiche sind verschieblich.

U = Undefined symbol
: Ein Symbol, auf das Bezug genommen wird, ist nicht definiert. Für einige Pseudooperationen wird im Pass 1 ein V-Fehler und im Pass 2 ein U-Fehler erzeugt.

V = Value error
: Wertfehler: Eine Pseudooperation (z.B. .RADIX, .PAGE) hat im Pass 1 einen undefinierten Wert, obwohl er im Pass 1 bekannt sein müßte.

Neben diesen Fehlercodes, die sich direkt auf eine Befehlszeile des Quelltextes beziehen, gibt es noch Fehlermeldungen die im Klartext (englisch) angezeigt werden:

„?File not found“
: Die zu übersetzende Quelldatei wurde nicht gefunden.

„No end statement“
: Es ist keine END-Anweisung vorhanden oder sie nicht durchlaufen infolge eines falsch/nicht abgeschlossenen Makros.

„Unterminated conditional“
: Mindestens eine Bedingung (IF/COND) ist am Ende der Datei nicht abgeschlossen worden.

„Unterminated REPT/IRP/IRPC/MACRO“
: Mindestens ein Block ist nicht mit ENDM abgeschlossen worden.

„Symbol table full“
: Bei der Bildung der Symboltabelle durch den Assembler ist der verfügbare Speicher erschöpft. Hinweis: Kommentare in Makros sollten mit 2 Semikolons (;;) geschrieben werden.

„Stack overflow, try more P switches“
: Der Stack reicht nicht aus, das Kommando sollte mit mehr /P-Schaltern wiederholt werden.

„xx Fatal error(s), yy warning(s)“
: Die Anzahl der schweren Fehler und Warnungen, wird am Ende jeder Übersetzung angezeigt. Wurde kein Fehler erkannt, erscheint: „No Fatal error(s)“.

9.3.6.3 Übersetzung des Beispielprogramms

Damit haben wir alle Grundlagen für die Übersetzung unseres Quelltextes HALLO.MAC erlernt. Zunächst also ein Test ob der Quelltext fehlerfrei ist:

```
A>ASM ,=HALLO

No Fatal Error(s)

A>_
```

Zeigt Euer Bildschirm nach Eingabe dieses Kommandos noch Fehler an, dann solltet Ihr mit dem Editor nochmal den Quelltext HALLO.MAC betrachten und den Fehler beseitigen. Ansonsten erzeugen wir die Objectcodedatei durch Eingabe von:

```
A>ASM =HALLO
```

Jetzt lassen wir uns das Directory anzeigen und sollten die Datei HALLO.REL vorfinden.

9.4 Der Linker LINK.COM

Im Gegensatz zu EDAS ist der vom Assembler ASM.COM erzeugte Objectcode noch nicht von der CPU ausführbar. Es ist noch ein weiterer Arbeitsschritt notwendig: Das Verbinden der Programme, engl. linken – daher der Name “Linker“. Was zunächst als überflüssig und umständlich erscheint, hat aber auch Vorteile wie wir später sehen werden. Die vom Assembler erzeugten Dateien *.REL sind noch nicht an eine feste Adresse gebunden, sind also verschieblich (natürlich nur wenn man nicht im absoluten Modus ASEG gearbeitet hat). Der Linker kann mehrere dieser REL-Dateien zu einem Programm verbinden. Wenn man die immer wiederkehrenden Abläufe jeweils in ein eigenes Modul schreibt, dann braucht man diese Module im Quelltext des zu erzeugenden Programmes nicht mehr zu schreiben. Das Zusammensetzen erfolgt erst beim Linken. Ein weiterer Vorteil: Jedes dieser Module muß nur einmal getestet werden und wenn es funktioniert, braucht man sich um die Funktion keine Gedanken mehr machen. Nur eine saubere Dokumentation der Ein- und Ausgangsparameter ist unbedingt erforderlich. Außerdem lassen sich mehrere REL-Dateien zu einer Datei zusammenfassen, das nennt man dann eine Bibliothek. Der Linker durchsucht solche Bibliotheksdateien und schließt die erforderlichen Module in die Ausgabedatei ein.

Der von uns verwendete Linker von “Digital Research“ heißt LINK131.COM (im weiteren Verlauf nur LINK genannt). Mit LINK werden verschiebliche Objektmodule *.REL zu einer ausführbaren Datei *.COM verbunden. Verschiebliche Dateien können externe Bezüge und Eintrittspunkte (Public-Variablen) enthalten. Verschiebliche Dateien können auch externe Bezüge zu anderen Modulen in der Datei-Bibliothek enthalten.

9.4.1 Aufruf von LINK

LINK wird aus der CP/M-Kommandozeile aufgerufen, wobei folgende Angaben erforderlich sind:

```
LINK [Comdatei[Optionen]=]Reldatei[Optionen][,...]
```

Unbedingt ist der Name der Reladtei anzugeben, die das Hauptprogramm darstellt. Der einfachste Aufruf lautet also:

```
LINK HALLO
```

Dabei wird HALLO.REL geladen und die Datei HALLO.COM erzeugt. Das ist genau der Fall für unser kleines Beispielprogramm, das ja bereits als HALLO.REL vorliegt. Gebt also diese Kommandozeile ein und verfolgt am Bildschirm was passiert:

```
A>LINK HALLO
LINK 1.31

ABSOLUTE     0000
CODE SIZE    0019 (0100-0118)
DATA SIZE    0000
COMMON SIZE  0000
USE FACTOR     00

A>_
```

Zunächst meldet sich LINK mit der Versionsnummer, dann dauert es einen Moment bis die Datei HALLO.REL geladen ist. Diese wird bearbeitet und als COM-Datei ausgegeben, dabei zeigt uns LINK noch an welche Adressen belegt wurden. Hier finden wir die 4 bekannten Bereiche absolut, coderelativ, datenrelativ und commonrelativ wieder. Da wir im Quelltext keinen Modus angegeben hatten, arbeitete der Assembler im Standardmodus, also coderelativ. Und genau in diesem Modus wurden die Daten erzeugt. Hinter CODE SIZE steht 0019, das bedeutet daß 19h Bytes erzeugt wurden. In Klammern steht dann noch der belegte Speicherbereich von 100H bis 118H. Wurden andere Speichersegmente belegt, dann erscheinen hinter diesen ähnliche Meldungen.

Schauen wir uns nun im Directory an was sich getan hat. Es müßten folgende Dateien vorhanden sein:

HALLO.MAC   HALLO.REL   HALLO.COM   HALLO.SYM

In der vom Linker erzeugten Datei HALLO.SYM werden die externen Bezüge und öffentlichen Symbole in einer Symboltabelle zusammengefaßt. In unserem Beispiel steht nichts in der Datei, da wir keine solchen Symbole verwendet haben. Wichtiger ist die Datei HALLO.COM, die wir ja eigentlich erzeugen wollten. Nun gilt es zu kontrollieren, wer sein erstes Assemblerprogramm unter CP/M zum Laufen bringt. Also starten wir es:

```
A>HALLO
Hallo KC-Klub !
A>_
```

Wenn alles geklappt hat, dann zeigt euer Bildschirm diese Meldung und ihr dürft euch dazu gratulieren! Wenn etwas anderes angezeigt wird oder sich der Laufwekssprompt nicht wieder meldet, dann beginnt der schwierigste Teil der Assemblerprogrammierung: Die Fehlersuche. Aber nicht gleich verzweifeln, die Dateien mit allen Zwischenschritten liegen ja als Diskettendatei vor. Wenn gar nichts mehr geht, dann hilft auch mal die RESET-Taste und ein JUMP FC bringt das System schon wieder zum Laufen. Im Fehlerfall muß man aber immer wieder beim Quelltext beginnen:

EDIT – > ASM – > LINK – > Test?

Zurück zur Theorie, wir waren bei der Kommandozeile von LINK und hatten aus einer REL-Datei eine gleichnamige COM-Datei erzeugt. Soll die COM-Datei einen anderen Namen erhalten, dann ist dieser als erstes und danach ein Gleichheitszeichen anzugeben. Es kann auch ein anderes Laufwerk mit angegeben werden, zum Beispiel:

```
A>LINK B:HALLO=HALLO
```

Die erzeugte Ausgabedatei HALLO.COM wird nach Laufwerk B: geschrieben. Sollen mehrere REL-Dateien verbunden werden, dann sind diese mit Komma getrennt aufzuführen:

```
A>LINK HALLO1=HALLO,TEST,HILFE
```

In diesem Fall würden die Dateien HALLO.REL, TEST.REL und HILFE.REL verbunden und als ausführbare Datei HALLO1.COM auf die Diskette geschrieben.

9.4.2 Optionen

Die Ausführung von LINK kann durch Verwendung von Optionen gesteuert werden. Diese LINK Optionen werden in eckigen Klammern hinter der Dateibezeichnung angegeben. Mehrere Angaben werden durch Kommas getrennt. Optionen können sowohl hinter dem Namen der COM-Datei als auch hinter den REL-Dateien geschrieben werden. Folgende Optionen existieren:

A — Zusatzspeicher; reduziert den Pufferbereich und schreibt temporäre Daten auf Diskette

B — Linkt ein BIOS für ein gebanktes CP/M-System.

1. Das Datensegment wird auf Seitengrenze ausgerichtet.
2. Legt die Länge des Codesegments im Header ab.
3. Erzeugt wird standardmässig der Dateityp SPR.

Dhhhh — Adresse Datensegment; setzt die Speicheradresse für den Common- und Datenbereich

Gn — Go; setzt die Startadresse auf die Marke n

Lhhhh — Load; ändert die Ladeadresse des Moduls auf hhhh. Standard ist 0100H

NL — Keine Ausgabe der Symboltabelle auf Console.

NR — Keine Datei der Symboltabelle.

OC — Ausgabe ist Kommandodatei .COM (Standard).

OP — Ausgabe einer .PRL Datei (page relocatable)

OR — Ausgabe einer .RSP Datei (resident system process)

OS — Ausgabe einer .SPR Datei (system page relocatable)

Phhhh — Programmbeginn; ändert den Standardwert für den Programmbeginn auf hhhh. Standard ist 0100H.

$Cd — Ziel der Console-Meldungen. d kann sein: X (Console), Y (Drucker), oder Z (ohne Ausgabe). Standard ist X.

$Id — Quelle der dazwischenliegenden Dateien; d ist ein Laufwerk A-P. Standard ist das aktuelle Laufwerk.

$Ld — Quelle der Bibliotheks-Dateien; d ist ein Laufwerk A-P. Standard ist das aktuelle Laufwerk.

$Od — Ziel einer Objektdatei; d kann Z oder ein Diskettenlaufwerk A-P sein. Standard ist die Ausgabe auf das gleiche Laufwerk, auf dem sich auch die erste Datei im LINK-Kommando befindet.

$Sd — Ziel der Symboldatei; d kann Y oder Z oder ein Diskettenlaufwerk A-P sein. Standard ist die Ausgabe auf das gleiche Laufwerk, auf dem sich auch die erste Datei im LINK-Kommando befindet.

Wie man sieht, besitzt auch LINK eine Menge an zusätzlichen Optionen, die man am Anfang nicht unbedingt wissen muß. LINK ist so voreingestellt, daß für die "normale" Arbeit nichts zusätzlich angegeben werden muß. Die Ein- und Ausgabedateien befinden sich normalerweise auf dem aktuellen Laufwerk und als Ausgabedatei wird standardmäßig eine COM-Datei erzeugt. Wem es stört, daß LINK eine Symboltabelle NAME.SYM erzeugt, der sollte sich vielleicht als erstes die Option NR einprägen. Wichtig ist bei der Angabe der Optionen, daß diese in ECKIGE Klammern zu schreiben sind. Falls der KC auf deutschen Zeichensatz eingestellt ist, entspricht dies dem großen Ä und dem großen Ü. Dem Linker ist egal, wie die Zeichen auf dem Bildschirm aussehen, es muß nur der richtige Zeichencode angegeben werden.

9.5 Softwarebibliotheken

Eine neue Qualität erfährt die Assemblerprogrammierung durch die Verwendung von Softwarebibliotheken. Immer wiederkehrende Programmteile, wie zum Beispiel die Ausgabe einer Zeichenkette auf den Bildschirm oder Drucker oder das formatierte Anzeigen einer Zahl, werden in solchen Bibliotheken gesammelt und können bei Bedarf dem Assemblerprogramm hinzugefügt werden. Im Assemblerquelltext werden die aus den Softwarebibliotheken zu verwendenden Routinen als EXTERN definiert. Die Zusammenführung des eigentlichen Assemblerprogrammes mit den Routinen aus der Softwarebibliothek erfolgt erst beim Linken.

Es gibt verschiedene fertige Softwarebibliotheken, die von erfahrenen Assemblerprogrammierern zusammengestellt, sorgfältig getestet und ständig weiterentwickelt wurden. Durch die Nutzung solcher Routinen wird die Assemblerprogrammierung teilweise fast so einfach wie eine höhere Programmiersprache. Wichtigste Voraussetzung ist jedoch eine genaue Dokumentation zu den einzelnen Routinen einer Softwarebibliothek. Bekannt sein muß der Name einer Funktion, welche Register als Eingangsparameter dienen, die Ausgangsparameter und welche Register von der Routine verändert werden. Zusätzlich kann noch die benötigte Stacktiefe und die Codegröße angegeben

sein. Diese Informationen sind jedoch nicht ganz so wichtig wie die zuerst genannten und deshalb oft nicht verfügbar.

9.5.1 Verwaltung mit LIB.COM

LIB.COM ist die SCPX-Version (robotron) des Microsoft-Programms LIB80.COM und ist ein Dienstprogramm zum Bilden und Verwalten von Bibliotheken aus in Assemblersprache geschriebenen Programmen. Diese Programme (im folgenden Moduln genannt) werden zu einer Bibliothek zusammengefaßt und können (auch einzeln) als Unterprogramme für andere Assemblerprogramme oder als Laufzeitbibliothek für Compiler höherer Programmiersprachen (z.B. BASIC oder FORTRAN) verwendet werden. Der Vorteil der Bildung von Bibliotheken liegt unter anderem darin, daß alle für die Ausführung eines Programms benötigten Routinen vom Programmverbinder LINK eingebunden werden. Dazu ist nur der Bibliotheksname, gefolgt von einem /S in der LINK-Kommandozeile anzugeben. Immer wiederkehrende Funktionen müssen so nur einmal erzeugt werden und können einfach in verschiedene Programme eingebunden werden.

9.5.1.1 Aufruf des Bibliothekars

Der Bibliothekar LIB.COM wird aufgerufen durch das Kommando

```
LIB
```

und meldet sich mit einem Stern als Bereitschaftszeichen. An dieser Stelle können nun verschiedene LIB-Kommandos eingegeben werden. Es ist auch möglich, das LIB-Kommando direkt in der CP/M-Kommandozeile anzuhängen. Dann führt LIB dieses Kommando direkt aus und kehrt – falls kein Fehler aufgetreten ist – zur CP/M-Kommandoebene zurück. LIB-Kommandos bestehen aus einem wahlfreien Zielfeld, einem Quellfeld und einem wahlfreien Schalter-Feld. Das Format ist:

```
ziel=quelle/schalter
```

Im Zielfeld erfolgt die Angabe des Namens der zu bildenden Bibliothek. Der Standarddateityp ist .REL und sollte nicht anders angegeben werden, da LINK dies sonst nicht akzeptiert. Das Zielfeld entfällt, wenn keine Bibliothek gebildet wird, z.B. wenn nur der Inhalt einer vorhandenen Bibliothek gelistet wird.

Die Angabe des Quellfeldes ist notwendig. Quellfeld-Eintragungen bezeichnen Dateinamen oder Moduln, sie beziehen sich auf REL-Dateien. Folgende Syntaxmöglichkeiten existieren:

PROG1
: bezeichnet die Datei PROG1.REL bzw. alle darin enthaltenen Moduln

PROG2<UP1>
: bezeichnet das Modul UP1 aus der Datei Bibliothek PROG2.REL

PROG2<UP1,UP2>
: bezeichnet die beiden Moduln UP1 und UP2 aus der Bibliothek PROG2.REL

PROG3<..UP3>
: bezeichnet alle Moduln aus PROG3.REL vom ersten Modul bis einschließlich UP3

PROG3<UP4..>
: bezeichnet alle Moduln aus PROG3.REL von UP4 bis zum letzten Modul

PROG3<UP2..UP4>
: bezeichnet alle Moduln aus PROG3.REL von UP2 bis UP4, diese eingeschlossen

Statt der direkten Angabe des Modulnamen kann eine relative Kette angegeben werden. Dabei wird der Abstand zum benannten Modul verwendet, als eine ganze Zahl im Bereich von 1 bis 255. Dazu ein Beispiel: In der Bibliothek PROG1.REL sind folgende Moduln in der angegebenen Reihenfolge enthalten: UP1, UP2, UP3, UP4, UP5, SUM. Dann bezeichnet:

PROG1<UP2+2>	den Modul UP4
PROG1<UP5-3>	den Modul UP2
PROG1<UP1+1..SUM-1>	alle Moduln außer UP1 und SUM selbst

Das Schalterfeld im LIB-Kommando bewirkt zusätzliche Funktionen. Die Schalter werden entweder in eine extra Kommandozeile geschrieben oder einem Quellfeld angehangen. Folgende Schalter sind möglich:

/E
: schreibt eine neu gebildete Bibliothek auf Diskette und kehrt zu CP/M zurück. Wurde keine Bibliothek gebildet oder eine vorhandene geändert, sollte LIB mit ^C (BRK) beendet werden.

/R hat die gleiche Wirkung wie /E, jedoch ohne Rückkehr zu CP/M. /R wird verwendet, wenn unmittelbar im Anschluß die nächste Bibliothek bearbeitet werden soll.

/L erzeugt eine Liste der Moduln einer angegebenen Datei und der in ihnen enthaltenen Symboldefinitionen, außerdem wird für jedes Modul die Codegröße aufgelistet.

/U listet alle Symbole, die bei einem einfachen Durchlauf der Bibliothek undefiniert bleiben.

/C setzt LIB zurück, alle bisherigen Kommandos werden ignoriert.

/O setzt den Anzeigemodus auf Oktalbasis

/H setzt den Anzeigemodus auf Hexadezimalbasis zurück (Standard)

9.5.1.2 Anwendungsbeispiele zu LIB

Die zwei wichtigsten Anwendungsfälle sind das Listen einer vorhandenen Bibliothek und das Bilden einer neuen Bibliothek.

```
A>LIB
*MATHLIB/U
*MATHLIB/L
.
.
(Liste der Symbole in MATHLIB.REL)
.
.
*^C
A>_
```

In diesem Beispiel erfolgt der Aufruf des Bibliothekars durch das CP/M-Kommando LIB. MATHLIB/U bewirkt, daß alle undefinierten Symbole aufgelistet werden – falls an dieser Stelle nichts angezeigt wird, sind keine vorhanden. MATHLIB/L listet alle Namen und Symboldefinitionen der in MATHLIB.REL enthaltenen Moduln auf. Schließlich wurde LIB mit ^C verlassen. Soll nur eine Bibliothek gelistet werden, reicht auch die Angabe in der CP/M-Kommandozeile:

```
A>LIB MATHLIB/L
.
.
(Liste der Symbole in MATHLIB.REL)
.
.
A>_
```

In diesem Fall würde LIB.COM geladen, anschließend die Namen und Symboldefinitionen aller Moduln aus MATHLIB.REL gelistet und zur Kommandoebene von CP/M zurückgekehrt.

Das nächste Beispiel soll die Bildung einer mathematischen Bibliothek MATHLIB.REL aus den vorhandenen Moduln SIN.REL, COS.REL, TAN.REL, COTAN.REL und EXP.REL verdeutlichen:

```
A>LIB
*MATHLIB=SIN,COS,TAN,COTAN
*EXP
*/E
A>_
```

Hier wird LIB.COM wiederum nur geladen, um weitere Kommandos am Prompt entgegenzunehmen. Das erste LIB-Kommando definiert den Namen der zu bildenden Bibliothek mit MATHLIB.REL. Daran schließen sich gleich die Namen der ersten vier aufzunehmenden Dateien an. In der nächsten Zeile steht nur EXP, das ist der Name eines Moduls das ebenfalls zu MATHLIB gehören soll. Es könnte auch in der vorhergehenden Zeile mit angegeben werden. Die Moduln werden in der Reihenfolge aufgenommen wie sie angegeben werden. Schließlich bewirkt /E daß die Ausgabe der Datei MATHLIB.REL und das Verlassen von LIB. Um diese Aktion in einer CP/M-Kommandozeile zu absolvieren müßte man schreiben:

```
A>LIB MATHLIB=SIN,COS,TAN,COTAN,EXP/E
```

Angenommen man möchte in der vorhandenen Bibliothek MATHLIB.REL das Modul TAN durch eine neuere Version ersetzen und gleichzeitig ein neues Modul SQR aufnehmen, ohne die gesamte Bibliothek neu zu erzeugen. Dazu eignet sich folgendes Kommando:

```
A>LIB MATHLIB=MATHLIB<..TAN-1>,TAN,MATHLIB<TAN+1..>,SQR/E
```

MATHLIB.REL wird hierdurch neu erzeugt aus allen Moduln der bisherigen MATHLIB.REL bis einschließlich des Moduls das sich unmittelbar vor TAN befindet. Dem schließt sich das neue Modul TAN.REL an, gefolgt von dem Rest aus MATHLIB.REL und dem neuen Modul SQR.REL. Auf die gleiche Art und Weise lassen sich so auch Moduln aus der Bibliothek wieder entfernen. Mit den richtigen Kommandos läßt sich so eine Bibliothek recht komfortabel pflegen und erweitern.

9.5.2 Standardbibliotheken: SYSLIB & Co

Für CP/M existieren mehrere Sammlungen fertiger Bibliotheken mit Programmcode für die verschiedensten Anwendungsfälle. Die wichtigste ist wohl die SYSLIB. Bereits in den 80er Jahren hat sich Richard Conn bemüht, Routinen für wiederkehrende Arbeiten zu sammeln und zu dokumentieren. Im Laufe der Jahre ist die SYSLIB gewachsen, die Routinen wurden überarbeitet und optimiert, so daß sie heute eine solide Basis darstellen. Da die SYSLIB aus Amerika kommt, ist die Dokumentation leider in englisch – aber das ist eigentlich der einzige Nachteil.

Die Beschreibung aller Routinen der SYSLIB würde den Rahmen dieses Kurses sprengen. Die Folgende kurze Übersicht soll nur die Themen aufzeigen, die in der SYSLIB enthalten sind. Es lohnt sich für fast jedes Programm, zunächst zu überprüfen welche der vorhandenen Funktionen genutzt werden können:

- Directory Manipulation
- Zeichenorientierte Dateiein-/-ausgabe
- allgemeine Dateizugriffe
- Zugriff auf LBR-Dateien
- Aufbereitung von Dateinamen, FCB-Initialisierung
- Manipulation von Laufwerk, Userbereich, DMA-Adressen
- Umwandlung von Zahlen
- Ausgabe von Zahlen
- Eingabe von Zeichenketten
- Ausgabe von Zeichenketten und Dateinamen
- Zeichenorientierte Ein-/Ausgabe
- Fallunterscheidung, Sprungtabellen, berechnete Sprünge
- Umwandlung Groß-/Kleinschreibung, Zeichentests
- Mathematische Funktionen
- Berechnung von CRC-Prüfsummen
- Zufallszahlengenerator
- Vergleich von Zeichenketten und Zahlen
- Speicherverwaltung
- Sortierung von Feldern

Darauf aufbauend existieren weitere Bibliotheken zu speziellen Randgebieten und Anwendungsfällen. Die folgende Liste zeigt die Anwendungsgebiete und die aktuellen Versionen der verfügbaren Bibliotheken. Zu jeder Bibliothek gibt es entsprechende HELP-Dateien.

allgemeine Routinen für CP/M:
    SYSLIB.REL V4.5 10 Aug 92 (C) 1989-92 by Alpha Systems Corp.

Unterstützung für ZCPR-Umgebungen::
    Z3LIB.REL V4.5 29 Aug 92, (C) 1989,90,91 by Alpha Systems Corp.

Standardisierter Terminalzugriff bei ZCPR-Umgebungen::
    VLIB.REL V4.5 29 Aug 92 (C) 1989,90,91 by Alpha Systems Corp.

Routinen für Systeme mit DateStamper(tm)::
    DSLIB.REL V4.4 30 Jun 91 (C) 1988-91 by H.F.Bower

Zusatzfunktionen für Systeme mit ZSDOS/ZDDOS::
    ZSLIB.REL V3.4 8 October 91 portions (c) 1989,90,91 Carson Wilson

Spezielle Routinen für den KC85::
    KCLIB.REL V1.0 22.08.98 (C) 1998 by ML-Soft.

Hinweise: Die Bibliotheken existieren meist in zwei Varianten, eine im Format M-REL und eine weitere im SLR-Format. Die letztere erkennt man am angehängten S im Dateinamen (also SYSLIBS.REL statt SYSLIB.REL). Für die in unserem Kurs verwendeten Programme ASM.COM, LINK131.COM und LIB.COM eignet sich nur die Variante im Format M-REL! Zur Erhöhung der Lesbarkeit werden in den Bibliotheken und deren Beschreibungen Symbolnamen mit bis zu 8 Zeichen verwendet. Das Format M-REL bzw. die damit arbeitenden Programme können aber externe Symbole nur bis maximal 6 gültige Zeichen verarbeiten. Aus diesem Grund sind alle Symbolnamen in diesen Versionen auf 6 Zeichen abgeschnitten, die Symbole wurden aber so gewählt, daß es keine doppelten Symbole gibt. Im Assembler-Quelltext muß dies in der gleichen Art und Weise durchgeführt werden, dann findet der Programmverbinder auch sicher die zugehörigen Programmteile.

9.5.3 Spezialroutinen: KCLIB

Ausgehend vom Gedanken der SYSLIB habe ich inzwischen begonnen, allgemein nutzbare Routinen für den KC85 zu sammeln und in einer gesonderten Bibliothek zusammenzustellen. Die meisten dieser Routinen sind speziell auf den KC85 und seine Möglichkeiten zugeschnitten. Ich werde aber auch andere Moduln aufnehmen, die in der SYSLIB nicht vorhanden sind. Die KCLIB befindet sich noch in der Aufbauphase und wird je nach Bedarf weiterentwickelt und erweitert.

Die Beschreibung der Funktionen ist in einer HELP-Datei zusammengefaßt, die parallel mit der Entwicklung der KCLIB aktualisiert wird – natürlich in deutscher Sprache! Für jede Funktion sind die Eingabeparameter (PE), Rückgabeparameter (PA), die veränderten Register (VR) und die Systemanforderungen (SYSTEM) beschrieben. Die KCLIB nutzt teilweise Funktionen der SYSLIB. Beim Linken muß deshalb erst die KCLIB und danach die SYSLIB durchsucht werden, ansonsten kann der Linker unter Umständen nicht alle externen Bezüge auflösen.

Folgende Routinen sind momentan verfügbar:

KCTEST	Test auf KC-System
ZASVS	Bestimmung der ZAS-Version
READ1	1 Byte im D001 lesen
WRITE1	1 Byte im D001 schreiben
RD256	256 Byte im D001 lesen
KCCOPY	Speichertransfer D004 -> D001
KCREAD	Speichertransfer D001 -> D004
GOSUB	Unterprogramm im KC aufrufen
MUL32	Multiplikation 16Bit*16Bit
DIV32	Division 32Bit/16Bit
RELOC	PRL-Format relozieren
KCLVER	Version der KCLIB testen

Am Beispiel der ersten Routine KCTEST soll gezeigt werden, wie die Funktionen der KCLIB genutzt werden. In der HELP-Datei stehen die folgenden Angaben:

```
Name: KCTEST
PE:   -
PA:   CY = 1 wenn kein KC-System vorliegt
VR:   AF,BC,DE,HL
SYSTEM:    -

FUNKTION: testet die KC-Kennung im BIOS und gibt im Fehlerfall eine
          Meldung auf dem Bildschirm aus. Für Programme, die nur
          auf einem KC85 laufen, empfiehlt sich der Aufruf dieser
          Funktion.

BEISPIEL:

          EXT     KCTEST          ; ext. Funktion definieren

  START:  CALL    KCTEST          ; KC-System?
          RET     C               ; nein, Programm beenden
```

Angenommen es soll ein CP/M-Programm für einen KC85 geschrieben werden, das den Grafikmodus des KC-Systems nutzt. Diese Grafikbefehle werden als ESC-Sequenzen zur Ausgabe gebracht. Andere CP/M-Rechner werden diese Sequenzen nicht verstehen. Daher empfiehlt sich ein Test, ob die richtige Hardware vorliegt. Wie dieser Test aussieht, darum braucht man sich nicht zu kümmern, denn die Funktion „KCTEST“ wurde sorgfältig ausgewählt und berücksichtigt auch, daß es inzwischen verschiedene CP/M-Betriebssysteme auf dem KC85 gibt. Werden von dem zu erstellenden Programm spezielle Anforderung an ZAS-Funktionen gestellt, dann empfiehlt sich der Aufruf der Routine ZASVS, welche die Versionsnummer der gerade laufenden ZAS-Version meldet.

9.6 BDOS & BIOS

Eine der wichtigsten Informationen für die Erstellung von Assemblerprogrammen ist neben der Handhabung der Programme Assembler, Linker usw. die Kenntnis der Softwareschnittstellen. Unter CAOS sind es die CAOS-Unterprogramme, welche über Programmverteiler aufgerufen werden. Etwas ähnliches existiert auch bei CP/M. Da das System in mehrere Bestandteile gegliedert ist, besitzt jeder Teil seine definierten Eintrittspunkte. BIOS und BDOS sind die residenten Systembestandteile, die auch von anderen Programmen benutzt werden dürfen.

Das BIOS (Basic Input Output System) liefert alle Treiber zur Ein- und Ausgabe, also die direkten Schnittstellen zu Tastatur, Bildschirm, Drucker, Diskettenlaufwerk

usw. Das BIOS liegt im oberen Speicherbereich des 64K-Adreßraumes und beginnt mit einer Sprungtabelle. Diese Sprungtabelle hat folgenden Aufbau:

Sprung	Nr.	Funktion
JP BOOT	0	BIOS-Kaltstart
JP WBOOT	1	BIOS-Warmstart
JP CONST	2	Konsolenstatus abfragen
JP CONIN	3	Konsoleneingabe (Tastatur)
JP CONOUT	4	Konsolenausgabe (Bildschirm)
JP LIST	5	Druckerausgabe
JP PUNCH	6	Zusatzausgabe
JP READER	7	Zusatzeingabe
JP HOME	8	Spur 0 einstellen
JP SELDSK	9	Laufwerk auswählen
JP SETTRK	10	Spur auswählen
JP SETSEC	11	Sektor auswählen
JP READ	12	Sektor lesen
JP WRITE	13	Sektor schreiben
JP LISTST	14	Status Druckerausgabe
JP SECTRAN	15	Sektornummer umrechnen
JP RTCIO	16	Treiber für RTC-Uhr (nur im ZBIOS des KC!)

Die Funktionen von 0 bis 15 sind in allen CP/M 2.2 kompatiblen Systemen gleich. Danach können sich weitere systemabhängige Aufrufe anschließen, wie im ZBIOS des KC85 der Aufruf des Uhrentreibers. Der Aufruf der Zeichenein- bzw. -ausgabe über die BIOS-Schnittstelle ist sehr schnell und wird daher oft genutzt. Der Zugriff auf Datenträger ist dagegen nur auf der niederen Ebene möglich, hier kann nach der Auswahl von Laufwerk, Spur, Sektor und DMA-Adresse lediglich ein 128 Byte großer Sektor gelesen oder geschrieben werden. Das BIOS kennt also keine Datenstrukturen wie das Directory oder Dateien.

Wie findet man nun die BIOS-Sprungtabelle im CP/M-System? Da jedes BIOS eine andere Größe haben kann, existiert keine feste Adresse. Während des Bootvorganges wird aber auf der Adresse 0000H ein Sprung zur Funktion WBOOT eingetragen. Das ist die zweite (!) Sprungadresse in der Sprungtabelle. Daraus läßt sich berechnen, wo die Sprungtabelle beginnt und mit dem entsprechenden Offset (3 Byte für jeden Sprungbefehl) die Adresse der anderen Funktionen ermitteln. Oder einfacher: Verwendung der SYSLIB-Routine mit dem Namen BIOS. Vor dem Aufruf der Funktion gibt man einfach im Register A die Nummer der BIOS-Funktion an, den Rest übernimmt die Routine. Als Übergabeparameter wird von den BIOS-Funktionen das Registerpaar BC benutzt, die Rückgabeparameter stehen in den Registern A bzw. HL zur Verfügung. Es muß davon ausgegangen werden, daß alle anderen Registerinhalte (mit Ausnahme der Indexregister IX und IY sowie der Zweitregister) zerstört werden. Für weitere Informationen empfiehlt sich das Lesen der Programmierhandbücher des jeweiligen Betriebssystems.

Die zweite und wichtigere Schnittstelle stellt das BDOS dar. Das BDOS baut auf die BIOS-Funktionen auf, das heißt alle über das BIOS erreichbaren Funktionen sind im wesentlichen auch über das BDOS nutzbar. Das BDOS (Basic Disc Operation System) stellt nun die Verwaltung der Datenträger bereit und nutzt dazu selbst die BIOS-Funktionen. Während im BIOS nur Spuren und Sektoren verwaltet werden, kennt das BDOS die Verzeichnisstruktur und kann auf Dateien zugreifen. Die BDOS-Funktionen zur Dateiarbeit stellen den wesentlichsten Teil des BDOS dar.

Für den Assemblerprogrammierer stellt das BDOS einen Sprungverteiler mit einer Reihe an definierten Funktionen bereit. Der Sprungverteiler wird über einen Aufruf auf die Adresse 0005H angesprungen. Im Register C steht dabei die Funktionsnummer und in DE bzw. E ein eventuell erforderlicher Eingangsparameter. Rückgabewerte stehen in den Registern A oder HL. Je nach CP/M-Version steht eine unterschiedliche Anzahl an Funktionen zur Verfügung. Das macht die Anwendung auf den ersten Blick kompliziert. Doch die grundlegenden Funktionen mit den Nummern 0 bis 36 sind in allen Systemen gleich. Werden nur diese Funktionen genutzt, dann dürfte das Programm auf verschiedenen Rechnern ohne Anpassung laufen. Eine besondere Bedeutung hat in dem Zusammenhang die BDOS-Funktion 12, damit fragt man die CP/M-Versionsnummer ab. Mit deren Hilfe kann man dann entscheiden, ob weitere Funktionen aufgerufen werden können oder nicht. Ein Vergleich der Handbücher zu KC-MicroDOS 2.6 und ZSDOS macht die Unterschiede bereits deutlich. Anhand der Versionsnummer (2.2 bei ZSDOS bzw. 2.6 bei MicroDOS) läßt sich am KC-System jedoch feststellen, welche BDOS-Funktionen zur Verfügung stehen.

Die BDOS-Funktionen sind im jeweiligen Handbuch für den Programmierer genau

dokumentiert. Als Beispiel soll die Funktion 9 dienen, mit deren Hilfe eine Zeichenkette auf dem Bildschirm ausgegeben werden kann. Im Register DE ist die Adresse der Zeichenkette zu übergeben, diese muß mit dem Dollarzeichen '$' abgeschlossen sein. Die Ausgabe einer Zeichenkette könnte demnach wie folgt programmiert werden:

```
BDOS    EQU     5        ; Definition der BDOS-Schnittstelle
        ...
        LD      DE,TEXT  ; Beginn der Zeichenkette
        LD      C,9      ; Nummer der BDOS-Funktion
        CALL    BDOS     ; Aufruf der BDOS-Funktion
        ...
TEXT:   DB      'angezeigte Zeichenkette$'
        ...
```

Auch an dieser Stelle der Hinweis, daß in der SYSLIB eine gleichnamige Funktion BDOS existiert. Ein- und Ausgabeparameter sind identisch mit dem normalen BDOS-Aufruf, allerdings werden zusätzlich die Register BC und DE gerettet. Das ist oft günstig, wenn in DE die FCB-Adressen für weitere Diskettenfunktionen benötigt werden. In BC stehen oft Zählervariablen, die so nicht extra gerettet werden müssen.

Das BDOS liegt unmittelbar vor dem BIOS, also ebenfalls im oberen Bereich des Z80-Arbeitsspeichers. Davor liegt noch der CCP-Kommandoprozessor von CP/M, der meist eine Größe von 2K (800H) hat. Da der Kommandoprozessor aber nur für die Eingabe der Kommandozeile und das Laden von Programmen benötigt wird, kann dieser Systembestandteil vom Anwenderprogramm überschrieben werden. Bei der Rückkehr zum Betriebssystem mittels Warmstart, also einen Sprung zur Adresse 0, dann wird der Kommandoprozessor nachgeladen. Um festzustellen, wie weit ein Anwenderprogramm den Speicher benutzen darf, bietet sich die Adresse 6 an, also der BDOS-Einsprung. Alles was davor liegt, ist im Prinzip frei.

MicroDOS macht bei der Systemaufteilung einen Unterschied. Die Bestandteile CCP, BDOS und BIOS sind derart verknüpft, sodaß eine Trennung nicht möglich ist. Der Einsprung zum BDOS wurde deshalb bis vor den Kommandoprozessor vorverlegt, wodurch dieser nicht mehr überschrieben wird und ein Nachladen bei einem Warmstart nicht mehr erforderlich ist.

Die folgende Tabelle zeigt den prinzipiellen Aufbau der Speichersegmente eines CP/M-Systems (speziell im D004):

```
---------------------    FFFFh
:   1K Koppel-RAM   :
---------------------    FC00h
:       BIOS        :
---------------------
:       BDOS        :
---------------------
:       CCP         :
---------------------
:                   :  Anwenderstack
:       TPA         :        :
:                   :  Anwenderprogramm
---------------------    0100h
:    Systempage     :
---------------------    0000h
```

Der Bereich zwischen 0 und 100h dient als Datenbereich für Systemzwecke, er enthält die Einsprungstellen zu BIOS und BDOS, wird zur Ablage von 2 Dateisteuerblöcken und eines DMA-Bereiches benutzt. Als feste Adresse läßt sich nur die Anfangsadresse eines Anwenderprogrammes, also 100h angeben. Im KC-System liegt am oberen Ende des Speicherbereiches der Koppel-RAM, daran schließt sich mit fallenden Adressen BIOS, BDOS und CCP an. Je nachdem wie groß diese Systemteile sind, bleibt der Rest des Speichers für Anwenderproramme nutzbar. Im KC-System stehen etwa 50K TPA-Speicher zur Verfügung, das entspricht einem Bereich von 100H bis C900H (MicroDOS). Bei ML-DOS kann es mehr sein, das Z-System reduziert den TPA je nach Ausbau wieder um einige Byte.

Für das Anwenderprogramm ist es wichtig, die maximal nutzbare Speicheradresse zu ermitteln. Oft wird dort direkt der Anwenderstack angelegt, während das Programm und die Datenbereiche von 100h aufwärts liegen. Bei umfangreichen Programmen sollte man zudem testen, ob genügend Speicherplatz für die Ablage von Daten zur Verfügung steht. Ein versehentliches Überschreiben von Teilen des BDOS oder BIOS hat mit Sicherheit einen Systemabsturz zur Folge und sollte unbedingt vermieden werden! Anders als beim CAOS, welches in EPROM's gebrannt ist, liegt das Betriebssystem CP/M im RAM und ist hardwaremäßig nicht vor Schreibzugriffen geschützt.

9.7 Weitere Hilfsmittel

Neben Assembler und Linker gibt es weitere Programme, die je nach Anforderung und Aufgabe eine weitere Hilfe für die Entwicklung von Assemblerprogrammen bieten. Als wichtigsten Vertreter sehe ich Debugger, also Programme zum Testen und zur Fehlersuche. Nicht jedes Assemblerprogramm läuft auf Anhieb fehlerfrei. Da ist es mitunter günstig das Programm schrittweise abzuarbeiten, dabei Registerinhalte und Speicherzellen zu kontrollieren und gezielt zu manipulieren. Und genau diese Aufgabe erledigen Debugger. Die bekanntesten Vertreter aus der CP/M-Welt sind wohl SID.COM für den i8080-Prozessor und der für den Z80 weiterentwickelte ZSID.COM. Der U880-Debugger DU.COM von robotron ist zum ZSID kompatibel.

Eine weitere Aufgabe besteht in der Anpassung fremder Programme an den eigenen Rechner. Sei es um Bildschirmsteuerzeichen anzupassen oder englische Texte in's deutsche zu konvertieren. Nicht immer sind die erforderlichen Stellen im Programm bekannt bzw. die Quelltexte vorhanden. Dann kann man sich eines Reassemblers bedienen, um vom Programm oder Teilen davon einen Quelltext zu erzeugen. Reassembler gibt es sehr viele mit großen Unterschieden in Bedienung und Komfort. Als typische Vertreter soll RAZ80 und DazzleStar vorgestellt werden.

9.7.1 Debugger DU.COM

Der Debugger (Programmtester) ist ein Hilfsmittel zur Fehlersuche in Programmen, zu deren Optimierung und Erprobung. Er erlaubt das schrittweise Abarbeiten, den Test einzelner Unterprogramme sowie die Protokollierung der Programmschritte bzw. deren Speicherinhalte. Es gibt verschiedene Möglichkeiten für den Aufruf des Debuggers:

(1) DU
(2) DU ?
(3) DU p.COM
(4) DU p.COM p.SYM
(5) DU p.HEX
(6) DU p.UTL

Variante (1) ruft nur den Debugger auf. Er steht nach seinem Aufruf im Speicher ab Adresse 0100H, der Bereich oberhalb von DU ist für die Modifizierung von Speicherzellen bis BDOS frei.

Variante (2) zeigt zusätzlich die Adressen an, es wird keine Datei geladen.

Variante (3) ruft zuerst den Debugger auf. Nach seinem Aufruf verschiebt er sich bis zur Untergrenze des BDOS, d.h. er steht dann genau unterhalb von BDOS. Der Bereich ab 0100H ist jetzt für das zu testende Programm frei, das mit p.COM angegeben wird. Anstatt einer COM-Datei kann auch eine beliebige andere Datei geladen werden, deshalb ist der Dateityp stets anzugeben.

Variante (4) lädt zusätzlich eine Symboltabelle p.SYM.

Variante (5) ist vergleichbar mit Variante (3), allerdings wird eine Datei im Intel-HEX-Code geladen.

Variante (6) lädt und startet DU-Zusatzfunktionen, diese müssen als Datei *.UTL vorliegen.

Der Debugger meldet sich nach dem Start mit

```
DU 1520 (SCPX) V1.0
U880-Debugger
SYMBOLS
NEXT PC   END
nnnn pppp eeee
#_
```

und erwartet eine Kommandoeingabe. Die Anzeige „SYSMOLS“ entfällt, wenn keine Symboltabelle geladen wurde. nnnn ist die Adresse des nächsten freien Speichers nach dem zu testenden Programm, pppp der initialisierte Befehlszählerstand und eeee die letzte freie Adresse. Der Debugger kann zu jeder Zeit mit ^C (BRK) verlassen werden.

Jedes DU-Kommando besteht aus dem Kommandotyp (ein Buchstabe) und optionalen Spezifikationen, die durch Leerzeichen oder Komma voneinander getrennt sind. Die erste Variablen steht ohne Leerzeichen unmittelbar nach dem jeweiligen Kommandobuchstaben. Die Zahlenbasis in der Adresseingabe ist hexadezimal. Es muß bei Angabe von Daten als erstes Zeichen immer eine Ziffer stehen, Zahlen die mit einem Buchstaben beginnen, ist eine „0“ voranzustellen. Folgende Kommandos enthält DU:

Kommando	Form	Wirkung
A		Speicherinhalt ändern:
	A<adr>	ab Adresse <adr>
	A	ab aktueller Position
	-A	wird von DU überschrieben
C	C<adr>	Abarbeiten eines Unterprogramms, das ab
		der Adresse <adr> steht, bis zu
		einem RET-Befehl
D		HEX/ASCII-Dump von Speicherinhalten:
	D	Dump ab der aktuellen Adresse
	D<adr>,<adr>	Dump von <adr> bis <adr>
	D<f>	Dump ab aktueller Adresse bis
		Endadresse <f>
F	F<adr>,<f>,<d>	Speicher füllen ab <adr> bis <adr>
		mit Byte <d>
G	G	Echtzeitabarbeitung ab aktueller Adresse:
	G<p>	ab Adresse <p>
	G<p>,<a>	ab Adresse <p> mit Haltepunkt bei
		Adresse <a>
	G<p>,<a>,<b>	ab Adresse <p> mit Haltepunkt bei
		Adresse <a> und <b>
H		Rechnen mit Hexadezimalzahlen:
	H<a>,<b>	Summe/Differenz der Zahlen <a> und <b>
	H<a>	Liefert symbolisch Adresse und den
		Dezimalwert von <a>
	H	Gibt die Liste der Adressen jedes
		Elements der Symboltabelle aus (wenn
		vorhanden)
I	I<c1>,<c2>..	Initialisiert den Filesteuerblock
		(FCB) neu mit Dateibezeichnung

Kommando	Form	Wirkung
L		Gibt Assemblermnemonic aus:
	L	ab aktueller Adresse
	L<adr>	ab Adresse <adr>
	L<adr>,<f>	von Adresse <adr> bis <f>
	-L...	Unterdrückt Marken, wenn vorhanden
M	M<adr>,<f>,<d>	Verschiebt Speicherinhalte von Adresse
		<adr> bis <f> zur Anfangsadresse <d>
P		Setzen und Anzeigen eines Durchlauf-
		zählers, max. 8 Zähler sind erlaubt:
	P	Anzeigen
	P<adr>	Zähler auf die Adresse <adr> setzen,
		Zähler startet mit 1
	P<adr>,<d>	Zähler auf die Adresse <adr> setzen,
		Zähler startet mit Wert von <c>
	-P	Löscht alle Zähler
	-P<adr>	Löscht nur den Zähler auf Adresse <adr>
R	R	Lädt die mit I bezeichnete Datei ab 0100H
	R<d>	Lädt die mit I bezeichnete Datei ab
		0100H+<d>
S	S<adr>	Ändert Speicherzelle der Adresse <adr>
		im Byteformat (8-Bit)
	SW<adr>	Ändert Speicherzelle der Adresse <adr>
		im Adressformat (16-Bit)
T		Potokolliert schrittweises Abarbeiten:
	T	einen Schritt
	T<n>	<n> Schritte
	T<n>,<c>	<n> Schritte, ruft bei jedem Schritt
		Zusatzfunktion <c> auf
	-T...	ohne Symbole und Marken
	TW...	ohne Zusatzfunktion
	-TW...	ohne Symbole und Zusatzfunktion

Kommando	Form	Wirkung
U	U	Wie T aber ohne Zwischenschritte
	-U...	zeigt nur CPU-Register vor dem Befehl an
	UW...	Wie TW, ohne Zwischenschritte
	-UW...	ohne Passpunkte
X	X	Anzeigen der CPU-Registerinhalte
	X<f>	Anzeige der Flags
	X<r>	Registerinhalte ändern
		Register: A, B, C, D, E, H, L, X, Y, P, A',B',C', D',E',H',L'

9.7.2 Reassembler RAZ80.COM

RAZ80.COM von „Microbyte & Bitresearch" ist ein Reassembler, der automatisch aus einem vorhandenen Programmcode den Quelltext erzeugt. Die letzte mir bekannte Version ist die V3.2, die Dokumentation bzw. daraus entstandene HELP-Datei ist allerdings V3.0 – geringfügige Unterschiede sind also nicht auszuschließen. RAZ80 sollte jedermann, der es benötigt, zur Verfügung gestellt werden, ohne eine jegliche Gebühr zu erheben. Kommerzielle Vermarktung zwecks Bereicherung ist verboten!

RAZ80 wurde für Z80-Programme unter CP/M 2.x geschrieben, die mit dem Macro-Assembler MACRO80 von MicroSoft weiterverarbeitet werden sollen. RAZ80 ist ein Zwei-Pass-Reassembler, der im Pass 1 Marken generiert und im Pass 2 Quellcode erzeugt und diesen auf Bildschirm, Drucker oder in eine Datei ausgibt. Das zu reassemblierende Programm kann mit Optionen in verschiedene Segmente aufgeteilt werden, um Code von Text, Konstanten (Bytes, Words), Datenbereichen und Markentabellen zu unterscheiden.

Wird nur RAZ80 eingegeben, so erscheint eine Liste aller Optionen als Hilfetext, falls Ihr eine Option nicht exakt wißt und dieses Papier zu weit vom Rechner entfernt ist...

```
A>raz80
Z80 - Reassembler V3.2  (C) 1986 by MicroByte & BitResearch
use:  RAZ80 [d:]infile[.ext] [option[s]]
 output:        infile.RAZ on default drive
 options:
  s:xxxx        start address (def.0100)
  o:xxxx        address of first byte (def.0100)
  e:xxxx        end address (def.EOF)
  a:xxxx        db ascii segment
  b:xxxx        db byte segment
  c:xxxx        code segment (def.)
  d:xxxx        ds segment
  m:xxxx        dw label segment
  n:xxxx        dw word segment
  f             options are in infile.RDF
  w or y        labelnames are in infile.SYM
  v             predefined CP/M-labels
  l             list pass2 on console
  p[:nn]        list pass2 on printer [page length]
  t             opcode comment in output
  u             upper case output
  x / i         hexdump / bindump
 xxxx   =       16-bit-hex-address
```

Um RAZ80 zur Arbeit zu bewegen, ist als erstes der Name der zu reassemblierenden Datei, evtl. gefolgt von Optionen anzugeben:

```
RAZ80 [d:]infile[.ext] [option[s]]
```

Das Inputfile kann sich auf einem beliebigen Laufwerk befinden, das Outputfile wird auf dem aktuellen Laufwerk abgespeichert. Das Outputfile bekommt den selben Dateinamen wie das Inputfile, jedoch mit dem Dateityp RAZ.

Optionen:

Mit den Optionen läßt sich die Arbeitsweise von RAZ80 steuern. Die Funktion der einzelnen Optionen ist der Hilfeseite bzw. HELP-Datei RAZ80.HLP zu entnehmen. Es lassen sich zum Beispiel die Anfangs- und Endadresse einschränken um nur Teile des Programmes zu analysieren (Optionen S und E). Interessant für die Reassemblierung von CAOS-Programmen erscheint die Option O mit der eine andere als die Standardadresse 100H als Programmbegin angegeben werden kann. Da CAOS-Programme aber einen 128-Byte Vorblock besitzen, ist als Adresse 80H weniger anzugeben.

Weitere Optionen dienen der Steuerung der Ausgabeform und der Einteilung in Code- oder Datenbereiche. Zunächst ist aber meist nicht bekannt, wo welche Bereiche liegen.

So wird man einen ersten Lauf ohne zusätzliche Optionen starten und nach Bearbeitung des entstandenen Quelltextes weitere Optionen an das Kommando anhängen. Werden es zu viele Optionen, lassen sich diese auch in einer ASCII-Steuerdatei ablegen, welche den selben Dateinamen mit Typ RDF erhält. Als einzige Option gibt man dann „f“ an um die weiteren Optionen aus der Datei zu lesen.

Als Zusammenfassung läßt sich sagen, mit RAZ80 ist schnell ein Programmcode zu einem Quelltext reassembliert. Das Ergebnis ist recht ansehnlich und läßt sich gut weiterverarbeiten, wenn nicht allzuviele unbekannte Datenbereiche und Zeichenketten enthalten sind. Die Angabe der Optionen zur Programmsteuerung ist etwas umständlich. Zur Suche der Datenbereiche gibt es keine Unterstützung des Programmes.

9.7.3 Reassembler DazzleStar

Der zweite Vertreter unter den Reassemblern arbeitet nach einem ganz anderen Prinzip. Wie der Name schon vermuten läßt, gibt es gewisse Gemeinsamkeiten zum Textprogramm WordStar. Wird Dazzlestar ohne Parameter aufgerufen, gibt es auch hier eine Anleitung. Diese umfaßt jedoch mehrere Seiten und ist sehr ausführlich. Normalerweise ruft man Dazzlestar jedoch mit dem Programmname des zu reassemblierenden Programmes als Parameter auf. Danach befindet man sich wie bei WordStar direkt im Quelltext und kann sich mit den Kursortasten darin bewegen. Durch Steuerzeichen lassen sich jetzt direkt an der Stelle wo man sich im Programm befindet, Marken definieren, Programm- und Datenbereiche unterscheiden, Sprungtabellen definieren usw.

DazzleStar bietet umfangreiche Kommandos, die eine gewisse Einarbeitung erforderlich machen. Wer bereits mit WordStar (bzw. TPKC) vertraut ist, dem stellt die Bedienung von DazzleStar jedoch keine größeren Probleme dar. Als günstig erweist sich, daß man schon bei der Erarbeitung des Reassemblertextes den Marken „richtige“ Namen geben kann. Außerdem lassen sich sofort Kommentare einfügen. Die Unterscheidung von Programm-, Daten- und Textsequenzen wird erleichtert, da der gerade bearbeitete Programmteil sowohl im OP-Code wie auch als HEX- und ASCII-Feld angezeigt wird.

Ist ein Programm noch nicht vollständig bearbeitet, kann mit dem Kommando ^KS oder ^KX der aktuelle Bearbeitungsstand in eine Datei gespeichert werden um später an dieser Stelle fortzusetzen. Diese Datei wird dann einfach mittels ^KR wieder eingelesen. Ist die Bearbeitung beendet, dann kann das Ergebnis entweder auf den Drucker, oder als Datei ausgegeben werden. Die erzeugte Datei *.MAC ist bestens geeignet, um mit einem Assembler übersetzt zu werden. Dennoch wird jedes Reassemblerlisting nicht so gut sein wie ein Originalquelltext mit den entsprechenden Kommentaren. Eine weitere Analyse und nachträgliche Bearbeitung mit weiteren Kommentaren wird immer nötig sein!

DazzleStar eignet sich jedoch hervorragend, um bestimmte Programmstellen aufzuspüren. Man kann Unterprogrammaufrufe und Sprungbefehle direkt verfolgen, Datenbereiche markieren und findet so schnell die für den jeweiligen Anwendungsfall interessierenden Stellen im Programm. Etwas Erfahrung und Einarbeitung in DazzleStar sollte man aber dennoch voraussetzen.

9.8 Übungsaufgaben

Das Programm HALLO.MAC ist so umzustellen, daß die Ausgabe der Zeichenkette auf den Drucker statt auf dem Bildschirm erfolgt. Dazu ist die BDOS-Funktion 5 zu verwenden, die ein Zeichen zum Drucker sendet. Hinweis: BDOS-Funktionen verändern alle Register. Wiederbenötigte Register sind also vor dem Aufruf der BDOS-Funktion zu retten und anschließend wiederherzustellen!

1. Man verwende eine Programmschleife, die Zeichen für Zeichen ausgibt, bis das Ende der Zeichenkette erkannt wird. (HALLO1.MAC)
2. Man schreibe ein Unterprogramm, das mit den gleichen Parametern wie in HALLO.MAC aufgerufen wird. (HALLO2.MAC)
3. Die Druckerausgabe wurde in ein eigenes Modul ausgelagert, welches als Quelltext DRUCK.MAC sowie als verschiebliches Modul DRUCK.REL vorliegt. Man schreibe HALLO.MAC so um, daß dieses Modul als externe Funktion benutzt wird. (HALLO3.MAC)
4. Wie könnte ein Kommando mit dem Bibliotheksprogramm LIB.COM aussehen, das aus der in Abschnitt 9.5.1.2 erzeugten Bibliothek MATHLIB.RAL das Modul COS entfernt?
5. Welches Kommando ist geeignet, um aus der Bibliothek KCLIB.REL die im Modul KCTEST enthaltenen Symbole aufzulisten? Wieviel Programmcode belegt dieses Modul?

Anhang A – Lösungen zu den Übungsaufgaben

A.1 Lösungen für Teil 2

Aufgabe 1:

Umwandlung der Zahl 23 in das Dualsystem nach der Potenz-Methode (8-Bit-Darstellung → Begin mit $i = 7$):

i	Rest von Z	Vergleichswert 2^i	$2^i \leq Z$?	Binärziffer b_i
7	23	128	nein	0
6	23	64	nein	0
5	23	32	nein	0
4	23	16	ja	1
3	7	8	nein	0
2	7	4	ja	1
1	3	2	ja	1
0	1	1	ja	1
				0 0 0 1 0 1 1 1

Umwandlung der Zahl 23 = 0001 0111 B in die Hexadezimal- und Oktaldarstellung

Hexadezimalzahl		1	7
↑		↑	↑
Dualzahl		0001	0111
↕			
Dualzahl	000	010	111
↓	↓	↓	↓
Oktalzahl	0	2	7

Ergebnis: 23 = 0001 0111 B = 17 H = 27 O

Umwandlung der Zahl 78 in das Dualsystem nach der Potenz-Methode (8-Bit-Darstellung → Begin mit $i = 7$):

i	Rest von Z	Vergleichswert 2^i	$2^i \leq Z$?	Binärziffer b_i
7	78	128	nein	0
6	78	64	ja	1
5	14	32	nein	0
4	14	16	nein	0
3	14	8	ja	1
2	6	4	ja	1
1	2	2	ja	1
0	0	1	nein	0
				0 1 0 0 1 1 1 0

Umwandlung der Zahl 78 = 0100 1110 B in die Hexadezimal- und Oktaldarstellung

Hexadezimalzahl		4	E
↑		↑	↑
Dualzahl		0100	1110
↕			
Dualzahl	001	001	110
↓	↓	↓	↓
Oktalzahl	1	1	6

Ergebnis: 78 = 0100 1110 B = 4E H = 116 O

Umwandlung der Zahl 127 in das Dualsystem nach der Quotient-Methode (8-Bit-Darstellung → führende Ziffern mit Null auffüllen):

i	Division $Z/2$	Quotient	Rest ⟶ Binärziffer b_i
0	127/2	63	1
1	63/2	31	1
2	31/2	15	1
3	15/2	7	1
4	7/2	3	1
5	3/2	1	1
6	1/2	0	1
			1 1 1 1 1 1 1

Ergebnis: 127 = 0111 1111 B = 7F H = 177 O

Umwandlung der Zahl 234 in das Dualsystem nach der Quotient-Methode (8-Bit-Darstellung → führende Ziffern mit Null auffüllen):

i	Division $Z/2$	Quotient	Rest ⟶ Binärziffer b_i
0	234/2	117	0
1	117/2	58	1
2	58/2	29	0
3	29/2	14	1
4	14/2	7	0
5	7/2	3	1
6	3/2	1	1
7	1/2	0	1
			1 1 1 0 1 0 1 0

Ergebnis: 234 = 1110 1010 B = EA H = 352 O

***Aufgabe 2*:**

Umwandlung der Dualzahl 1010 1001 B in das Dezimalsystem:

$$\begin{aligned}1010\,1001\,\mathrm{B} &= 1\cdot 2^7 + 0\cdot 2^6 + 1\cdot 2^5 + 0\cdot 2^4 + 1\cdot 2^3 + 0\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0\\ &= 1\cdot 128 + 0\cdot 64 + 1\cdot 32 + 0\cdot 16 + 1\cdot 8 + 0\cdot 4 + 0\cdot 2 + 1\cdot 1\\ &= 169\end{aligned}$$

Umwandlung der Hexadezimalzahl 3A H in das Dezimalsystem:

$$\begin{aligned}3\mathrm{A\,H} &= 3\cdot 16^1 + 10\cdot 16^0\\ &= 3\cdot 16 + 10\cdot 1\\ &= 58\end{aligned}$$

Umwandlung der Oktalzahl 127 O in das Dezimalsystem:

$$\begin{aligned}127\,\mathrm{O} &= 1\cdot 8^2 + 2\cdot 8^1 + 7\cdot 8^0\\ &= 1\cdot 64 + 2\cdot 8 + 7\cdot 1\\ &= 87\end{aligned}$$

***Aufgabe 3*:**

positive Dualzahl	0001 1011 = 27	0111 1011 = 123
⇓		
Einerkomplement	1110 0100	1000 0100
⇓		
Zweierkomplement	1110 0101 = −27	1000 0101 = −123

***Aufgabe 4*:**

Dezimalzahl	33	−47
8-Bit-Darstellung	0010 0001	1101 0001
12-Bit-Darstellung	0000 0010 0001	1111 1101 0001

Werden bei Zahlen in Zweierkomplement-Darstellung führende Bits hinzugefügt, dann werden diese Bits so gesetzt, wie das MSB der ursprünglichen Darstellungsbreite, bei positiven Zahlen also auf Null und bei negativen Zahlen auf Eins.

***Aufgabe 5*:**

33 + 67 = 100

	0010 0001
+	0100 0011
Überträge	0 0000 0110
Ergebnis	0110 0100

74 + (−27) = 47

	0100 1010
+	1110 0101
Überträge	1 1000 0000
Ergebnis	0010 1111

(−47) + 55 = 13

	1101 0111
+	0011 0111
Überträge	1 1110 1110
Ergebnis	0000 1110

(−13) + (−38) = (−51)

	1111 0011
+	1101 1010
Überträge	1 1110 0100
Ergebnis	1100 1101

In allen vier Beispielen sind *carry-in* und *carry-out* an der Position des Vorzeichenbits – die ersten beiden Bits in der Übertragszeile – gleich. Das bedeutet, daß alle vier Ergebnisse gültig sind, sich also mit der hier verwendeten 8-Bit-Darstellung bestimmen lassen.

A.2 Lösungen für Teil 5

Aufgabe 1:

Das Programm könnte etwa so aussehen:

```
LD      A,41H           ; Wert 41H in Akkumulator laden
ADD     A,20H           ; 20H addieren
LD      A,3
ADD     A,48
LD      A,254Q
ADD     A,221Q
LD      A,1110110B
ADD     A,11010010B
```

Der erste Operand muß zuvor in den Akkumulator gebracht werden, um die anschließende Addition ausführen zu können.

Aufgabe 2:

Das Programm für die Subtraktionen sieht ähnlich aus:

```
LD      A,68H           ; Wert in den Akkumulator laden
SUB     20H             ; Inhalt des Akkumulators um 20H
                        ; verringern
LD      A,57
SUB     48
LD      A,231Q
SUB     116Q
LD      A,110001B       ; fuehrende Nullen sind nicht
                        ; erforderlich
SUB     10100001B
```

Nachdem der erste Operand in den Akkumulator geladen wurde, kann die Subtraktion ausgeführt werden. Achtung: SUB-Befehl ohne Angabe des Registers A!

Aufgabe 3:

Der erste Operand eines LD-Befehls gibt stets das Ziel der Operation an. Es muß also ein Register oder eine Speicherzelle (Variable) sein.

Aufgabe 4:

Es wird jeweils das gleiche Programm benötigt, nämlich:

```
NEG             ; Inhalt des Akkumulators negieren
```

Zuvor muß selbstverständlich der entsprechende Wert in den Akkumulator geladen werden (z.B. LD A,80H). Die Wirkung des NEG-Befehls für die angegebenen Zahlen kann der folgenden Tabelle entnommen werden:

Wert	Negation	Z-Flag	S-Flag	C-Flag	P-Flag
80H	80H	0	1	1	1
0	0	1	0	0	0
164Q	214Q	0	1	1	0
10111001B	01000111B	0	0	1	0

Aufgabe 5:

- Das N-Flag (Addition/Subtraktion) wird abhängig vom Befehl gesetzt oder zurückgesetzt; es hängt niemals von den Operanden ab.
- Ist das Z-Flag (Null) gesetzt, so ist das Ergebnis der Operation gleich Null, also nicht negativ; damit ist das S-Flag (Vorzeichen) zurückgesetzt.
- Beim ADD- und beim SUB-Befehl sind C-Flag (Übertrag), H-Flag (Halbbyte-Übertrag) und S-Flag (Vorzeichen) voneinander unabhängig. Der Status des P/V-Flags (Überlauf) hängt von den Flags C (Übertrag), S (Vorzeichen) und N (Addition/Subtraktion) ab.
- Beim NEG-Befehl ist das C-Flag (Überlauf) genau dann gelöscht, wenn das Z-Flag (Null) gesetzt; H-Flag (Halbbyte-Übertrag) und S-Flag (Vorzeichen) sind dagegen voneinander unabhängig. Das P/V-Flag (Überlauf) hängt vom S-Flag (Vorzeichen) und vom H-Flag (Halbbyte-Übertrag) ab.

Aufgabe 6:

Der Text lautet: *Gut gemacht!*

Hinter dem Text folgt noch ein Wagenrücklauf (CR) und ein Zeilenvorschub (LF).

Aufgabe 7:

Die ASCII-Codierung des Satzes (mit einem Wagenrücklauf und einem Zeilenvorschub am Ende) lautet:

44H 65H 72H 20H 41H 73H 73H 65H 6DH 62H 6CH 65H 72H 6BH 75H 72H 73H 20H 64H 65H 73H 20H 4BH 43H 2DH 43H 6CH 75H 62H 73H 20H 69H 73H 74H 20H 53H 70H 69H 74H 7AH 65H 21H 0DH 0AH

Aufgabe 8:

Das Programm ist recht einfach und lautet beispielsweise:

```
        SUB     'A'-1   ; Grossbuchstaben in Ordnungszahl umwandeln
                        ; ord('A') = 1, ord('B') = 2, ...
```

Hier lag das Problem hauptsächlich darin, den Zusammenhang zwischen ASCII-Code und Hex-Zahlen möglichst effektiv in den Assemblerquelltext umzusetzen.

Aufgabe 9:

So könnte das Programm aussehen:

```
        ADD     A,'a'-1 ; Ordnungszahl in Kleinbuchstaben umwandeln
                        ; ord('a') = 1, ord('b') = 2, ...
```

Aufgabe 10:

Lösung:

```
        LD      BC,22131        ; Registerpaar BC mit 22131 laden
        LD      D,B             ; Inhalt des Registerpaares BC in
        LD      E,C             ; das Registerpaar DE bringen
```

Aufgabe 11:

Wir verwenden am besten das Registerpaar HL, denn somit ergibt sich ein kürzerer Maschinencode:

```
        LD      HL,(4256H)      ; Das ab Adresse 4256H stehende
        LD      (567BH),HL      ; Wort ab Adresse 567BH ablegen
```

Aufgabe 12:

Die Deklarationen lauten:

```
BYTE:   DEFB    45H     ; Byte-Variable mit 45H initialisieren
PLUS:   DEFB    '+'     ; Zeichen-Variable mit '+' initialisieren
WORT:   DEFB    1700    ; Wort-Variable mit 1700 initialisieren
NAKLAR: DEFM    'KC-Club? Was sonst!'   ; Zeichenketten-Variable
                                        ; initialisieren
```

Dabei belegen:

BYTE	1 Byte
PLUS	1 Byte
WORT	2 Bytes
NAKLAR	19 Bytes

Aufgabe 13:

Die Deklarationen lauten:

```
VAR1:   DEFS    1       ; Speicherplatz fuer ein Byte reservieren
ADR1:   DEFS    2       ; Speicherplatz fuer ein Wort reservieren
LGINT:  DEFS    4       ; Speicherplatz fuer 32-Bit-Zahl reservieren
STR7:   DEFS    7       ; Speicherplatz fuer Zeichenkette mit
                        ; 7 Zeichen reservieren
PUFFER: DEFS    128     ; Speicherplatz fuer Puffer mit 128 Bytes
                        ; reservieren
```

Aufgabe 14:

Vergleiche hierzu auch MULT10:

```
OP1:    DEFS    1       ; Speicherplatz fuer Operand (8-Bit-Zahl)
ERG:    DEFS    2       ; Speicherplatz fuer Ergebnis (16-Bit-Zahl)
MULT12: LD      A,(OP1) ; Operand laden
        LD      L,A
        LD      H,0     ; Operand zu 16-Bit-Zahl erweitern
        LD      D,H
```

```
         LD      E,L     ; Kopie erstellen
         ADD     HL,HL   ; Operand verdoppeln
         ADD     HL,DE   ; Operand verdreifachen
         ADD     HL,HL   ; Operand versechsfachen
         ADD     HL,HL   ; Operand verzwoelffachen
         LD      (ERG),HL        ; Ergebnis abspeichern
```

Aufgabe 15:

Anhand der gezeigten 32-Bit-Addition ließ sich dieses Programm sofort herleiten (oder nicht?):

```
OP1:     DEFS    8               ; 64-Bit-Speicherplatz fuer ersten
OP2:     DEFS    8               ; und zweiten Operanden reservieren
ERG:     DEFS    8               ; Ergebnisspeicherplatz (64 Bit)
ADD64:   LD      HL,(OP1)        ; niederwertige 16 Bits
         LD      DE,(OP2)        ; der Operanden holen
         ADD     HL,DE           ; addieren
         LD      (ERG),HL        ; und ablegen
         LD      HL,(OP1+2)      ; die naechsten 16 Bits
         LD      DE,(OP2+2)      ; der Operanden holen
         ADC     HL,DE           ; mit Uebertrag addieren
         LD      (ERG+2),HL      ; und ablegen
         LD      HL,(OP1+4)      ; die naechsten 16 Bits
         LD      DE,(OP2+4)      ; der Operanden holen
         ADC     HL,DE           ; mit Uebertrag addieren
         LD      (ERG+4),HL      ; und ablegen
         LD      HL,(OP1+6)      ; die hoechstwertigen 16 Bits
         LD      DE,(OP2+6)      ; der Operanden holen
         ADC     HL,DE           ; mit Uebertrag addieren
         LD      (ERG+6),HL      ; und ablegen
```

Aufgabe 16:

Dieses Programm läuft wie bei der Addition ab, wichtig war hier nur, das C-Flag (Übertrag) richtig zu setzen:

```
OP1:     DEFS    8               ; 64-Bit-Speicherplatz fuer ersten
OP2:     DEFS    8               ; und zweiten Operanden reservieren
ERG:     DEFS    8               ; Speicherplatz fuer Ergebnis
SUB64:   SCF                     ; C-Flag setzen
         CCF                     ; und loeschen
         LD      HL,(OP1)        ; niederwertige 16 Bits
         LD      DE,(OP2)        ; der Operanden holen
         SBC     HL,DE           ; subtrahieren
         LD      (ERG),HL        ; und ablegen
         ...
         ...                     ; weiter wie oben
```

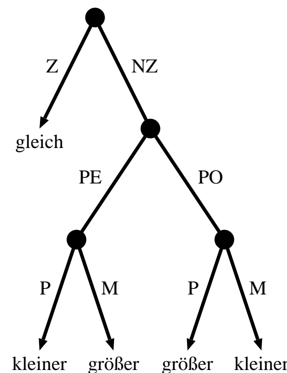
A.3 Lösungen für Teil 6

Die folgenden Programmausschnitte zeigen einige Lösungsvorschläge zu den gestellten Aufgaben und demonstrieren zugleich einige Optimierungsvarianten. Trotzdem ist in vielen Fällen eine weitere Optimierung durchaus möglich, insbesondere was die Verkürzung des resultierenden Objektcodes betrifft. Die vollständigen und reichlich dokumentierten Quelltexte sind im Archiv ASM06L.PMA zu finden.

Aufgabe 1:

Der Kürze wegen zeigt der folgende Quelltext nur den Ausschnitt des Programms COMPZ, in dem der Vergleich und die bedingen Spruenge liegen. Da es sich um kurze Sprungdistanzen handelt, verwendet man vorzugsweise relative Sprünge (JR-Befehle). Die Auswertung des P/V- und des Sign-Flags ist aber nur mittels absoluter Sprünge (JP-Befehle) möglich (siehe Tabelle auf Seite 60), so daß auch diese Verwendung finden müssen.

```
; Vergleich und bedingte Spruenge
      CP  B

      JR  Z,GL     ; Z -> gleich

      JP  PE,UEB   ; Ueberlauf -> PE

      JP  P,GR     ; P und PO -> groesser

      JR  KL       ; M und PO -> kleiner

UEB:  JP  P,KL     ; P und PE -> kleiner

      JR  GR       ; M und PE -> groesser
```



Der letzten Befehl dieser Sequenz kann hier eigentlich noch entfallen, da der Programmteil für das Vergleichsergebnis „größer“ unmittelbar folgt. Sieht man davon einmal ab, ist die gezeigte Sequenz bereits die Lösung mit den wenigsten Sprungbefehlen und damit auch die speicherplatzgünstigste. Zum besseren Verständnis ist auch der zugehörige Entscheidungsbaum abgebildet (siehe Abschnitt 8.3).

Aufgabe 2:

Im folgenden werden wiederum nur die Initialisierung und die Steuerung der Schleifen dargestellt.

LINEW1/LINEW2: Zeichnen einer waagerechten Linie.

Gegenüber dem Beispielprogramm LINE.ASM (Zeichnen einer senkrechten Linie, Archiv ASM06.PMA) sind hier zunächst nur wenige Änderungen notwendig. Anstelle von L (Spalte) wird das Register H (Zeile) auf einen festen Wert (16) gesetzt und das Register L in der Schleife mit dem jeweils aktuellen Wert des Schleifenzählers B geladen. Der Vergleichswert für den Schleifenabbruch ist jetzt 40.

```
        LD      B,0             ; Startwert entspricht 1. Spalte
        LD      H,16            ; 17. Zeile ist fest
LOOP:
        LD      L,B             ; Spalte = B und Zeile = 16
        LD      (0B7A0H),HL     ; als Cursorpos. uebergeben

        ...                     ; Zeichen "H" ausgeben

        INC     B               ; Schleifensteuerung
        LD      A,B
        CP      40              ; Spalte 39 ist letzte Spalte
        JR      C,LOOP
```

Da im Register L wird stets der Wert des Schleifenzählers B benötigt wird, läßt sich das Programm noch etwas vereinfachen, indem das Register L gleich als Schleifenzähler verwendet wird. Dabei werden alle B durch L ersetzt und der Befehl LD L,B entfernt. Das ergibt dann einen um ein Byte kürzeren Objektcode und – was mindestens genauso wertvoll ist – spart ein Register.

```
        LD      L,0             ; Startwert entspricht 1. Spalte
        LD      H,16            ; 17. Zeile ist fest
LOOP:
        LD      (0B7A0H),HL     ; Zeile = L und Zeile = 16
                                ; als Cursorpos. uebergeben

        ...                     ; Zeichen "H" ausgeben

        INC     L               ; naechste Spalte
        LD      A,L             ; Schleifensteuerung
        CP      40              ; Spalte 39 ist letzte Spalte
        JR      C,LOOP
```

DIAG1: Zeichnen einer Diagonalen von links oben nach rechts unten

Hier stimmen Zeilen- und Spaltenwert stets überein, wobei dieser Wert von 0 bis 39 läuft (damit entsteht zugegebenermaßen keine richtige Diagonale, sondern nur eine 45-Grad-Gerade, aber wer möchte, kann sich ja mal an einer echten Diagonale versuchen). Das Register L wird als Schleifenzähler verwendet und im Schleifenkörper zusätzlich in das Register H kopiert, bevor HL als Cursorposition übergeben wird.

```
                                ; erste Cursorposition ist (0,0)
        LD      L,0             ; Spalte
LOOP:
        LD      H,L             ; Zeile = Spalte
        LD      (0B7A0H),HL     ; als Cursorpos. uebergeben

        ...                     ; Zeichen "H" ausgeben

        INC     L               ; naechste Spalte
        LD      A,L             ; Schleifensteuerung
        CP      32              ; Spalte 39 ist letzte Spalte
        JR      C,LOOP
```

DIAG21/DIAG22: Zeichnen einer Diagonalen von links unten nach rechts oben

Im Gegensatz zu DIAG1 sind bei dieser Diagonalen die Werte für Zeile und Spalte nicht gleich. An dieser Stelle soll deshalb ein von der Curserposition unabhängiger Schleifenzähler (Register B) verwendet werden. Die Curserposition im Register HL wird ausgehend vom Startwert (H = 31, L = 0) mittels Incrementier- und Decrementierbefehlen in jedem Schleifendurchlauf weitergesetzt.

```
        LD      B,0             ; Startwert des Schleifenzaehlers

                                ; erste Cursorposition ist (31,0)
        LD      H,31            ; Zeile
        LD      L,0             ; Spalte
LOOP:
        LD      (0B7A0H),HL     ; HL als Cursorpos. uebergeben

        ...                     ; Zeichen "H" ausgeben

                                ; naechste Cursorposition
        DEC     H               ; Zeile  - 1
        INC     L               ; Spalte + 1

        INC     B               ; Schleifensteuerung
        LD      A,B
        CP      32
        JR      C,LOOP
```

Da der Wert des Schleifenzählers in B für die aktuelle Cursorposition nicht von Bedeutung ist, kann statt der aufwärtzählenden eine abwärtszählende Schleife verwendet, die durch den DJNZ-Befehl günstiger zu realisieren ist.

```
        LD      B,32            ; Startwert fuer 32 Durchlaeufe

                                ; erste Cursorposition ist (31,0)
        LD      H,31            ; Zeile
        LD      L,0             ; Spalte
LOOP:
        LD      (0B7A0H),HL     ; HL als Cursorpos. uebergeben

        ...                     ; Zeichen "H" ausgeben

                                ; naechste Cursorposition
        DEC     H               ; Zeile  - 1
        INC     L               ; Spalte + 1

        DJNZ    LOOP            ; Schleifensteuerung
```

Bei gleicher Funktion besitzt die zweite gegenüber der ersten Variante einen um 4 Byte kürzeren Objektcode!

FLAECHE1/FLAECHE2: Zeichnen einer zentrierten Flaeche

Ein solches zwei-dimensionales Gebilde läßt sich am besten mit zwei verschachtelten Schleifen erzeugen. Die erste Variante verwendet dazu zwei aufwärtzählende Schleifenzähler (Register B und C) und die unabhängig davon in den Registerpaar HL stehende Curserposition. Die Fläche wird zeilenweise gezeichnet.

```
; aeussere Schleife mit B zaehlt die gezeichneten Zeilen
        LD      B,0
        LD      H,6             ; Startzeile
LOOPA:
; innere Schleife mit C zaehlt die gezeichneten Spalten je Zeile
        LD      C,0
        LD      L,7             ; Startspalte
LOOPI:
        LD      (0B7A0H),HL     ; HL als Cursorpos. uebergeben

        ...                     ; Zeichen "H" ausgeben

; naechste Cursorposition in der inneren Schleife
        INC     L               ; Spalte + 1

        INC     C               ; Steuerung der inneren Schleife
        LD      A,C
        CP      25              ; 25 Spalten schon errreicht?
        JR      C,LOOPI

; wenn ja, dann naechste Position in der aeusseren Schleife
        INC     H               ; Zeile + 1

        INC     B               ; Steuerung der aeuss. Schleife
        LD      A,B
        CP      20
        JR      C,LOOPA
```

Auch das läßt sich noch optimieren, wenn man die Schleifen mit abwärtszählenden Schleifenzählern im Register B realisiert. Dabei muß während der Abarbeitung der inneren Schleife der Wert des Zählers der äußeren Schleife zwischengespeichert werden, wozu z.B. das Register C geeignet ist.

```
; aeussere Schleife
        LD      B,20            ; Startwert fuer 20 Durchlaeufe
        LD      H,6             ; Startzeile
LOOPA:
; innere Schleife
        LD      C,B             ; "aeusseres" B in C merken
        LD      B,25            ; Startwert fuer 25 Durchlaeufe
        LD      L,7             ; Startspalte
LOOPI:
        LD      (0B7A0H),HL     ; HL als Cursorpos. uebergeben

        ...                     ; Zeichen "H" ausgeben

; naechste Cursorposition in der inneren Schleife
        INC     L               ; Spalte + 1

        DJNZ    LOOPI           ; Steuerung der inneren Schleife

; naechste Cursorposition in der aeusseren Schleife
        INC     H               ; Zeile + 1

        LD      B,C             ; gemerktes B zurueckholen
        DJNZ    LOOPA           ; Steuerung der aeuss. Schleife
```

Der Objektcode der zweiten Variante ist damit um 6 Bytes kürzer als der der ersten.

Aufgabe 3:

Um sicherzustellen, daß unter CAOS genug Parameter an ein Programm übergeben wurden, überprüft man am besten sofort nach dem Programmstart den Wert in der Zelle 0B781H. Der Befehl LD A,(0B781H) ist unmittelbar nach dem Programmstart nicht unbedingt erforderlich, da die Parameteranzahl zusätzlich auch im Akkumulator an das Programm übergeben wird.

```
PMIN    EQU     4               ; Mindestanzahl an Parametern

START:  LD      A,(0B781H)      ; A = tatsaechliche Parameteranzahl
        CP      PMIN            ; Ist A gross genug?
        JR      C,FEHLER        ; nein, dann Fehler

WEITER: ...                     ; eigentliches Programm

FEHLER: ...                     ; Ausgabe einer Fehlermeldung
        RET                     ; und das Programm verlassen
```

Anhang B – Tabellen und Übersichten

B.1 ASCII-Code

B.1.1 Tabelle des US-ASCII-Codes

HEX	ASCII	HEX	ASCII	HEX	ASCII	HEX	ASCII
00H	NUL	20H	SP	40H	@	60H	‘
01H	SOH	21H	!	41H	A	61H	a
02H	STX	22H	“	42H	B	62H	b
03H	ETX	23H	#	43H	C	63H	c
04H	EOT	24H	$	44H	D	64H	d
05H	ENQ	25H	%	45H	E	65H	e
06H	ACK	26H	&	46H	F	66H	f
07H	BEL	27H	’	47H	G	67H	g
08H	BS	28H	(	48H	H	68H	h
09H	HT	29H	)	49H	I	69H	i
0AH	LF	2AH	*	4AH	J	6AH	j
0BH	VT	2BH	+	4BH	K	6BH	k
0CH	FF	2CH	,	4CH	L	6CH	l
0DH	CR	2DH	–	4DH	M	6DH	m
0EH	SO	2EH	.	4EH	N	6EH	n
0FH	SI	2FH	/	4FH	O	6FH	o
10H	DLE	30H	0	50H	P	70H	p
11H	DC1	31H	1	51H	Q	71H	q
12H	DC2	32H	2	52H	R	72H	r
13H	DC3	33H	3	53H	S	73H	s
14H	DC4	34H	4	54H	T	74H	t
15H	NAK	35H	5	55H	U	75H	u
16H	SYN	36H	6	56H	V	76H	v
17H	ETB	37H	7	57H	W	77H	w
18H	CAN	38H	8	58H	X	78H	x
19H	EM	39H	9	59H	Y	79H	y
1AH	SUB	3AH	:	5AH	Z	7AH	z
1BH	ESC	3BH	;	5BH	[	7BH	{
1CH	FS	3CH	<	5CH	\	7CH	|
1DH	GS	3DH	=	5DH	]	7DH	}
1EH	RS	3EH	>	5EH	ˆ	7EH	˜
1FH	US	3FH	?	5FH	_	7FH	DEL

Die Zeichen mit den Codierungen 00H ... 1FH und 7FH sind Steuerzeichen.

B.1.2 Funktion wichtiger Steuerzeichen

Zeichen	Bedeutung	Funktion
BEL	bell	Klingel oder akustisches Signal (Summer, Piepser) ertönen lassen
BS	backspace	auf vorhergehendes Zeichen zurückpositionieren
HT	horizontal tab	auf nächste Tabulatorposition der Zeile springen
LF	line feed	Zeilenvorschub (eine Zeile tiefer gehen)
FF	form feed	Seitenvorschub (auf den Anfang der nächsten Seite gehen)
CR	carriage return	Wagenrücklauf (auf den Anfang derselben Zeile gehen)

B.1.3 Nationale und internationale Varianten des ASCII-Codes

	23H	24H	40H	5BH	5CH	5DH	5EH	60H	7BH	7CH	7DH	7EH
US-ASCII	#	$	@	[	\	]	ˆ	`	{	|	}	˜
Deutsch	#	$	§	Ä	Ö	Ü	ˆ	`	ä	ö	ü	ß
Schwedisch	§	¤	È	Ä	Ö	Å	ˆ	è	ä	ö	å	ü
Dänisch	#	¤	È	Æ	Ø	Å	Ü	è	æ	ø	å	ü
Britisch	£	$	@	[	\	]	ˆ	`	{	|	}	˜
Französisch	£	$	à	°	ç	§	ˆ	`	é	ù	è	¨
International	#	¤	@	[	\	]	ˆ	`	{	|	}	¯

B.2 Flagbeeinflussung der Z80-Befehle

Befehl		Takte	Flags					
			C	Z	P/V	S	N	H
ADC	A,(HL)	7	X	X	V	X	0	X
ADC	A,(Ix+d)	19	X	X	V	X	0	X
ADC	A,r	4	X	X	V	X	0	X
ADC	A,n	7	X	X	V	X	0	X
ADC	HL,rr	15	X	X	V	X	0	?
ADD	A,(HL)	7	X	X	V	X	0	X
ADD	A,(Ix+d)	19	X	X	V	X	0	X
ADD	A,r	4	X	X	V	X	0	X
ADD	A,n	7	X	X	V	X	0	X
ADD	HL,rr	15	X	.	.	.	0	?
ADD	Ix,rr	15	X	.	.	.	0	?
ADD	Ix,Ix	15	X	.	.	.	0	?
AND	(HL)	7	0	X	P	X	0	1
AND	(Ix+d)	19	0	X	P	X	0	1
AND	r	4	0	X	P	X	0	1
AND	n	7	0	X	P	X	0	1
BIT	b,(HL)	12	.	X	?	?	0	1
BIT	b,(Ix+d)	20	.	X	?	?	0	1
BIT	b,r	8	.	X	?	?	0	1
CALL	nn	17	.	.	.	.	.	.
CALL	cc,nn	10/17	.	.	.	.	.	.
CCF		4	X	.	.	.	0	?
CP	(HL)	7	X	X	V	X	1	X
CP	(Ix+d)	19	X	X	V	X	1	X
CP	r	4	X	X	V	X	1	X
CP	n	7	X	X	V	X	1	X
CPD		16	.	X	X	X	1	?
CPDR		16/21	.	X	X	X	1	?

Befehl		Takte	Flags					
			C	Z	P/V	S	N	H
CPI		16	.	X	X	X	1	?
CPIR		16/21	.	X	X	X	1	?
CPL		4	.	.	.	.	1	1
DAA		4	X	X	P	X	.	X
DEC	(HL)	11	.	X	V	X	1	X
DEC	(Ix+d)	23	.	X	V	X	1	X
DEC	r	4	.	X	V	X	1	X
DEC	rr	6	.	.	.	.	.	.
DEC	Ix	10	.	.	.	.	.	.
DI		4	.	.	.	.	.	.
DJNZ	e	8/13	.	X	V	X	1	X
EI		4	.	.	.	.	.	.
EX	(SP),HL	19	.	.	.	.	.	.
EX	(SP),Ix	23	.	.	.	.	.	.
EX	AF,AF’	4	.	.	.	.	.	.
EX	DE,HL	4	.	.	.	.	.	.
EXX		4	.	.	.	.	.	.
HALT		8	.	.	.	.	.	.
IM	0-2	8	.	.	.	.	.	.
IN	r,(C)	12	.	X	P	X	0	0
IN	A,(n)	11	.	X	P	X	0	0
INC	(HL)	11	.	X	V	X	0	X
INC	(Ix+d)	23	.	X	V	X	0	X

Befehl		Takte	Flags					
			C	Z	P/V	S	N	H
INC	r	4	.	X	V	X	0	X
INC	rr	6	.	.	.	.	.	.
INC	Ix	10	.	.	.	.	.	.
IND		16	.	X	?	?	1	?
INDR		16/21	.	X	?	?	1	?
INI		16	.	X	?	?	1	?
INIR		16/21	.	X	?	?	1	?
JP	nn	10	.	.	.	.	.	.
JP	(HL)	4	.	.	.	.	.	.
JP	(Ix)	8	.	.	.	.	.	.
JP	cc,nn	10	.	.	.	.	.	.
JR	e	12	.	.	.	.	.	.
JR	cc,e	7/12	.	.	.	.	.	.
LD	(rr),A	7	.	.	.	.	.	.
LD	(HL),r	7	.	.	.	.	.	.
LD	(HL),n	10	.	.	.	.	.	.
LD	(Ix+d),r	19	.	.	.	.	.	.
LD	(Ix+d),n	19	.	.	.	.	.	.
LD	(nn),A	13	.	.	.	.	.	.
LD	(nn),rr	20	.	.	.	.	.	.
LD	(nn),HL	16	.	.	.	.	.	.
LD	(nn),Ix	20	.	.	.	.	.	.
LD	(nn),SP	20	.	.	.	.	.	.
LD	A,(rr)	7	.	.	.	.	.	.
LD	r,(HL)	7	.	.	.	.	.	.
LD	r,(Ix+d)	19	.	.	.	.	.	.
LD	A,(nn)	13	.	.	.	.	.	.
LD	r,r	4	.	.	.	.	.	.
LD	r,n	7	.	.	.	.	.	.
LD	rr,(nn)	20	.	.	.	.	.	.

Befehl		Takte	Flags					
			C	Z	P/V	S	N	H
LD	HL,(nn)	16	.	.	.	.	.	.
LD	Ix,(nn)	20	.	.	.	.	.	.
LD	rr,nn	10	.	.	.	.	.	.
LD	Ix,nn	14	.	.	.	.	.	.
LD	SP,(nn)	20	.	.	.	.	.	.
LD	SP,HL	6	.	.	.	.	.	.
LD	SP,Ix	10	.	.	.	.	.	.
LD	SP,nn	10	.	.	.	.	.	.
LDD		16	.	?	X	?	0	0
LDDR		16/21	.	?	0	?	0	0
LDI		16	.	?	X	?	0	0
LDIR		16/21	.	?	X	?	0	0
NEG		8	X	X	V	X	1	X
NOP		4	.	.	.	.	.	.
OR	(HL)	7	0	X	P	X	0	0
OR	(Ix+d)	19	0	X	P	X	0	0
OR	r	4	0	X	P	X	0	0
OR	n	7	0	X	P	X	0	0
OTDR		16/21	.	1	?	?	1	?
OTIR		16/21	.	1	?	?	1	?
OUT	(C),r	12	.	.	.	.	.	.
OUT	(n),A	11	.	.	.	.	.	.
OUTD		16	.	X	?	?	1	?
OUTI		16	.	X	?	?	1	?
POP	rr	10	.	.	.	.	.	.
POP	Ix	14	.	.	.	.	.	.
PUSH	rr	11	.	.	.	.	.	.
PUSH	Ix	15	.	.	.	.	.	.

Befehl		Takte	Flags					
			C	Z	P/V	S	N	H
RES	b,(HL)	15	.	.	.	.	.	.
RES	b,(Ix+d)	23	.	.	.	.	.	.
RES	b,r	8	.	.	.	.	.	.
RET		10	.	.	.	.	.	.
RET	cc	5/11	.	.	.	.	.	.
RETI		14	.	.	.	.	.	.
RETN		14	.	.	.	.	.	.
RL	(HL)	15	X	X	P	X	0	0
RL	(Ix+d)	23	X	X	P	X	0	0
RL	r	8	X	X	P	X	0	0
RLA		4	X	.	.	.	0	0
RLC	(HL)	15	X	X	P	X	0	0
RLC	(Ix+d)	23	X	X	P	X	0	0
RLC	r	8	X	X	P	X	0	0
RLCA		4	X	X	P	X	0	0
RLD		18	.	X	P	X	0	0
RR	(HL)	15	X	X	P	X	0	0
RR	(Ix+d)	23	X	X	P	X	0	0
RR	r	8	X	X	P	X	0	0
RRA		4	X	.	.	.	0	0
RRC	(HL)	15	X	X	P	X	0	0
RRC	(Ix+d)	23	X	X	P	X	0	0
RRC	r	8	X	X	P	X	0	0
RRCA		4	X	X	P	X	0	0
RRD		18	.	.	.	.	.	.
RST	n	11	?	?	?	?	?	?

Befehl		Takte	Flags					
			C	Z	P/V	S	N	H
SBC	A,n	7	X	X	V	X	1	X
SBC	A,(HL)	7	X	X	V	X	1	X
SBC	A,(Ix+d)	19	X	X	V	X	1	X
SBC	A,r	4	X	X	V	X	1	X
SBC	HL,rr	15	X	X	V	X	1	X
SCF		4	1	.	.	.	0	0
SET	b,(HL)	15	.	.	.	.	.	.
SET	b,(Ix+d)	23	.	.	.	.	.	.
SET	b,r	8	.	.	.	.	.	.
SLA	(HL)	15	X	X	P	X	0	0
SLA	(Ix+d)	23	X	X	P	X	0	0
SLA	r	8	X	X	P	X	0	0
SRA	(HL)	15	X	X	P	X	0	0
SRA	(Ix+d)	23	X	X	P	X	0	0
SRA	r	8	X	X	P	X	0	0
SLR	(HL)	15	X	X	P	X	0	0
SLR	(Ix+d)	23	X	X	P	X	0	0
SLR	r	8	X	X	P	X	0	0
SUB	(HL)	7	X	X	V	X	1	X
SUB	(Ix+d)	19	X	X	V	X	1	X
SUB	r	4	X	X	V	X	1	X
SUB	n	7	X	X	V	X	1	X
XOR	(HL)	7	0	X	P	X	0	0
XOR	(Ix+d)	19	0	X	P	X	0	0
XOR	r	4	0	X	P	X	0	0
XOR	n	7	0	X	P	X	0	0

Legende zur Tabelle der Flagbeeinflussung

Operanden:

r	=	8-Bit-Register (A, B, C, D, E, H, I, L, R)
rr	=	16-Bit-Register (AF, BC, DE, HL, IX, IY, SP)
Ix	=	IX oder IY
n	=	8-Bit-Zahl
nn	=	16-Bit-Zahl
b	=	Bitnummer (0...7)
d	=	Offset für IX oder IY
e	=	Offset des Sprungzieles
cc	=	Flagbedingung (C, NC, Z, NZ, PO, PE, P, M)

Flags:

C	=	Carry (C, NC)
Z	=	Zero (Z, NZ)
P/V	=	Parity/Overflow (PO, PE)
S	=	Sign (P, M)
N	=	Addition/Subtraktion
H	=	Half-Carry
.	=	Zustand des Flags wird nicht verändert
1	=	Flag ist gesetzt
0	=	Flag ist rückgesetzt
X	=	Flag ist entsprechend der durchgeführten Operation beeinflußt
P	=	P/V-Flag zeigt Parität
V	=	P/V-Flag zeigt Überlauf
?	=	Zustand des Flags ist nicht definiert oder unbekannt

Takte:

Diese Spalte gibt die Ausführungszeit des Befehls in Maschinentaktzyklen an.