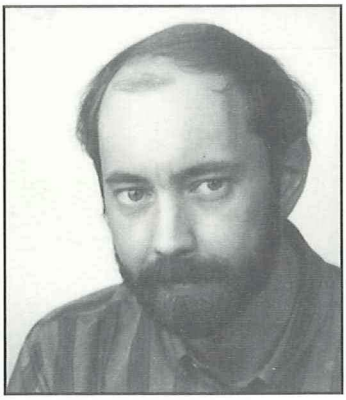
Markt&Technik

Gerd Möllmann
Michael Bauer

Programmieren in Assembler mit Top-Ass plus für den C128

Komplettpaket zur Erlernung der Maschinensprache des C128 (im 128er-Modus). Lehrbuch der Assemblersprache des 6502. Mit umfangreichem Tabellenteil zum Nachschlagen der Opcodes, Token und Befehle.

Auf Diskette:
Das Assembler-Entwicklungspaket Top-Ass inklusive Makroassembler, Monitor, Reassembler und Debugger.



Programmieren in Assembler mit Top-Ass plus für den C128

GERD BERNHARD MÖLLMANN, geboren am 3.1.1957 in Borken/Westfalen, legte 1975 das Abitur am Gymnasium seiner Heimatstadt ab. Das anschließende Studium der Mathematik, mit Schwerpunkt auf dem numerischen Zweig, führte zwangsläufig zu ersten Programmieraufgaben – zunächst auf Großrechnern in verschiedenen Hochsprachen. 1985 fand dann der Schritt in eine freiberufliche Tätigkeit statt. Seither sind verschiedene Softwareentwicklungen und Veröffentlichungen entstanden, weitere werden folgen.

MICHAEL BAUER, geboren am 11.2.1946 in Gmunden. Nach langjähriger Tätigkeit auf dem Sektor der Qualitätssicherung in der Industrie machte er sein Computerhobby Anfang 1985 zum Beruf. Heute beschäftigt er sich hauptberuflich mit der professionellen EDV als Systemmanager.

Neben diesem Buch hat der Autor bereits eine Reihe von Beiträgen in verschiedenen Zeitschriften zu diesem Thema veröffentlicht.

Das vorliegende Set aus Buch plus Diskette beinhaltet alles, was der Einsteiger in die Maschinensprache benötigt: ein professionelles Programmpaket zur Programmierung des C128 in Assembler, dazu eine Darstellung der Assemblersprache selbst, eine Einführung in die 6502-Maschinensprache und last not least ausführliche Programmbeschreibungen der Top-Ass-Programme mit Hintergrundinformationen. Abgerundet wird das Handbuch durch einen umfangreichen Tabellenteil, der in alphabetischer Form alle wichtigen Befehle, Tokens und Opcodes enthält.

Das Assembler-Entwicklungssystem Top-Ass-Plus für den Commodore 128 besteht aus sechs leistungsfähigen Modulen wie

- Makroassembler,
- Editor zum Assembler,
- Linker zum Assembler,
- Loader zum Assembler,
- Monitor und Debugger mit Breakpoints sowie einem
- Reassembler.

Hardware-Anforderungen:
Commodore 128 D oder Commodore 128 mit Floppy 1541, 1570 oder 1571, eventuell Drucker.

ISB N 3-89090-416-5

Markt&Technik

4 001057 904163

DM 59,-
sFr 54,30
öS 460,20

Gerd Möllmann
Michael Bauer

Programmieren in Assembler mit Top-Ass plus für den C128

Komplettpaket zur Erlernung der Maschinensprache des C128 (im 128er-Modus).
Lehrbuch der Assemblersprache des 6502.
Mit umfangreichem Tabellenteil zum Nachschlagen der Opcodes, Token und Befehle.

Markt&Technik Verlag AG

CIP-Kurztitelaufnahme der Deutschen Bibliothek

Möllmann, Gerd:
Programmieren in Assembler mit Top-Ass plus für den C128 : Komplettpaket zur Erlernung d. Maschinensprache d. C128 (im 128er-Modus) ; Lehrbuch d. Assemblersprache d. 6502 ; mit umfangreichem Tabellenteil zum Nachschlagen d. opcodes, token u. Befehle / Gerd Möllmann. – Haar bei München : Markt-und-Technik-Verlag, 1987. – & 1 Diskette
ISBN 3-89090-416-5

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
90 89 88 87

ISBN 3-89090-416-5

Einbandgestaltung: Grafikdesign Heinz Rauner
Druck: Jantsch, Günzburg
Printed in Germany

Inhalt

Vorwort

Seit Ende 1985 das Top-Ass-Assembler-Entwicklungspaket erschienen ist, haben mich als Autor der Top-Ass-Programme viele Anrufe und Briefe von Programmierern erreicht, die darüber klagten, daß ihnen der Einstieg in die Maschinenprogrammierung auf diesem Rechner unnötig schwergemacht werde. Nirgends seien ausreichend Informationen zu finden, die auf die Bedürfnisse von Einsteigern zugeschnitten seien, die erhältlichen Informationen seien zu sehr verstreut, so daß man sie sich mühsam zusammensuchen müsse, und überhaupt sei zuwenig über die Programmierung des C-128 auf Assemblerebene zu finden.

Für alle diese Programmierer ist das vorliegende Set aus Buch plus Diskette gedacht: Ein professionelles Programmpaket zur Programmierung in Assembler, dazu eine Darstellung der Assemblersprache selbst, eine Einführung in die Programmierung speziell des C-128 und last not least ausführliche Programmbeschreibungen der Top-Ass-Programme mit Hintergrundinformationen. Diese Kombination sichert Ihnen nach meiner festen Überzeugung den Einstieg in die Maschinensprache des C-128, wenn Sie als Einsatz etwas Geduld und Durchhaltevermögen aufbringen. Maschinenspracheprogrammierung ist für den Anfänger schwierig, wenn Sie sie aber einmal beherrschen - und das sollten Sie nach Durcharbeiten dieses Buches -, steht Ihnen ein Programmierfeld zur Verfügung, von dem Basic-Programmierer nur träumen können.

Als Co-Autor dieses Projekts zeichnet Michael Bauer aus München verantwortlich. Michael Bauer hat das Top-Ass-System durch einen Reassembler komplettiert, den Sie auf der beiliegenden Diskette zusammen mit den übrigen Top-Ass-Programmen der Version 2.0 finden können. Als Programmierer, der schon auf dem PET in Maschinensprache aktiv war, ist er der geeignete Mann, den Aufbau des Prozessors und seine Programmiersprache zu erläutern.

Zum Schluß möchte ich noch Christine Baumann danken, »meiner« Lektorin beim Markt & Technik Verlag, wenn ich so sagen darf. Ohne ihre Unterstützung und Anregung wäre auch dieses Buch nicht möglich gewesen.

Gerd Möllmann

Kapitel 1

6502-Assemblersprache

Maschinensprache-Programmierung - der direkte Zugriff auf alle Möglichkeiten, die ein Mikroprozessor bietet - ist kein Buch mit sieben Siegeln und keine Geheimwissenschaft. Mit etwas Geduld, Ausdauer und Freude ist diese Hürde zu nehmen. Der Preis, der winkt, sind schnelle und effiziente Programme.

Jeder Mikroprozessor hat seine eigene Maschinensprache. Dieses Buch behandelt das Assemblerpaket TOP-ASS für den Commodore C128.

Da in diesem Computer eine CPU 8502 arbeitet, wollen wir uns mit der Maschinensprache dieses Prozessors befassen. Der 8502 wird von der Commodore-Tochter MOS Technology, die auch den 6502 entwickelt hat, gefertigt. Alle Prozessoren dieser Familie verstehen die gleichen Befehle, mit einer Ausnahme: Der Prozessor 65C02 ist aufwärtskompatibel, d. h., er stellt noch einige Befehle zusätzlich zur Verfügung. Sonst unterscheiden sich die Prozessoren dieser Familie nur durch unterschiedliche Hardwareausführungen voneinander, z. B. durch einen verringerten Adreßraum oder durch zusätzliche I/O-Ports, wie bei unserem 8502.

Wo soll die Programmiersprache Assembler eingeordnet werden und welcher Unterschied besteht zur Maschinensprache?

Die Beantwortung dieser Frage zeigt schon einige Eigenschaften der verschiedenen Sprachen auf. Der Prozessor, oft auch CPU (Central Processing Unit) genannt, ist kein Sprachgenie; er versteht nur Maschinensprache, die nichts anderes als eine Abfolge von elektrischen Impulsen auf Leitungen ist. Diese Impulse lassen sich als Zahlen im Dualsystem (ein Zahlensystem, das nur 2 Ziffern, 0 und 1, kennt) beschreiben. Ein Programm in diesem System schreiben zu müssen, ist für jeden normalen Sterblichen eine Strafe: Die Rettung aus dieser Fron heißt Assembler! In Assembler werden die Befehle des Prozessors in Mnemonics (dieser Zungenbrecher stammt aus dem Griechischen [Mneme = Erinnerung] und bedeutet: Die Kunst, das Gedächtnis durch Hilfen zu stärken) geschrieben. Assembler ist die Sprache der mnemonischen Abkürzungen.

Die Operanden werden dagegen im Sedezimalsystem (Zahlensystem mit 16 Ziffern) oder Hexadezimalsystem, wie es nicht ganz korrekt auch noch genannt wird, angegeben. Manche Assembler, z. B. TOP-ASS, erlauben es, die Operanden auch dezimal und binär einzugeben; üblich ist in Assembler aber das Sedezimalsystem.

Maschinensprache	Assembler	Beschreibung
10101101 10110000 01001010	LDA $4AB0	Lade den Akkumulator mit dem Inhalt der Speicherzelle Nummer $4AB0 = 19120

Der Assembler, meistens selbst ein in Assembler geschriebenes Programm, übersetzt (= assembliert) den Quelltext (im Englischen als Sourcecode bezeichnet) in Maschinensprache, die dann unsere CPU versteht. Dabei wird eine Anweisung im Quelltext 1:1 in eine Anweisung im Objektkode (= Maschinenkode) assembliert; es ist also eine sehr maschinennahe Sprache.

Höhere Programmiersprachen wie BASIC oder PASCAL benötigen einen Interpreter oder Compiler, um von der CPU verstanden zu werden. Der Interpreter (= Übersetzer) führt bei jedem neuen Programmablauf das Programm direkt, also Befehl für Befehl, aus und übersetzt es dabei, während der Compiler (Englisch: to compile = zusammenstellen) das gesamte Programm einmalig in einen Objektkode übersetzt, der dann beim Start zur Ausführung gelangt.

1.1 Warum eigentlich Assembler?

Eine provozierende Frage, wie es scheint! Doch so abwegig ist sie nicht. Von der Problematik des Programmierens in Assembler (Englisch: to assemble = montieren) ist immer wieder zu lesen und zu hören. Einfacher soll es allemal in einer höheren Sprache wie z. B. in BASIC 7.0 im C128 gehen. Doch so schlimm ist es nicht, wie im Laufe dieses Buches zu sehen sein wird, da TOP-ASS eine beinahe genauso komfortable Programmierumgebung bietet, wie sie die gewohnte BASIC-Umgebung darstellt.

Warum also in Assembler programmieren?

Neben der Neugierde, etwas Neues zu erlernen, gibt es eine Reihe gewichtiger Gründe, die für die Programmierung in Assembler sprechen, und natürlich auch eine Menge an Gründen, die dagegen sprechen. In Bild 1.1 sind die Merkmale der Assemblersprache denen höherer Sprachen gegenübergestellt.

Merkmal	Assembler	Höhere Programmiersprachen	
		Interpreter	Compiler
Geschwindigkeit	extrem schnell	langsam	schnell
Effizienz	hoch	schlecht	mittel
Speicherausnutzung	ausgezeichnet	schlecht	gut
Transportabilität	schlecht	gut	gut
Programmierumgebung	mit TOP-ASS gut	sehr gut	gut
Fehlersuche	mit TOP-ASS gut	sehr gut	gut
Hardwarekenntnisse	unbedingt nötig	nicht nötig	nicht nötig
Hardwareeinbindung	sehr gut	bedingt	bedingt
Programmdokumentation	aufwendig	normal	normal

Bild 1.1 Gegenüberstellung der Merkmale von Sprachen

Wenn man diese Gegenüberstellung auswertet, stellt man fest, daß Assembler zur Bewältigung einiger Aufgabenstellungen besonders gut geeignet ist:

- Bei der Lösung von Problemen, die schnell erledigt werden müssen (Real-Time-Processing = Echtzeitverarbeitung).
- Wenn der verfügbare Speicherplatz knapp ist.
- Wenn das zu lösende Problem mehr I/O-Kontrolle (Englisch: Input/Output = Ein- und Ausgabe) als reine Berechnungen erfordert.
- Wenn nur eine begrenzte Anzahl von Daten zu verarbeiten ist.
- Wenn das Programm vom Umfang her überschaubar ist.
- Wenn dem BASIC-Interpreter im C128 durch Erweiterung seines Befehlssatzes auf die Sprünge geholfen werden soll.

1.2 Bits & Bytes, Teil 1: Grundlagen und Zahlensysteme

Unser Prozessor arbeitet mit elektrischen Strömen. Er kennt dabei nur 2 Zustände: Entweder es fließt ein Strom oder nicht. In der Mathematik findet dies seine Entsprechung im Dualzahlensystem.

1.2.1 Das Dual- oder Binärsystem

Das Dual- oder Binärsystem arbeitet mit zwei Ziffern, die unsere beiden Zustände repräsentieren können. Es bietet sich an, die Abläufe in unserem Prozessor mathematisch mit dem Dualsystem zu beschreiben. Genau dies wird auch gemacht. Aufbauend auf der Arbeit des britischen Mathematikers George Boole (1815 - 1864), der das erste System der Algebra der Logik (Boolesche Algebra) schuf, gelang es, die logischen Vorgänge in digitalen Schaltwerken (unser Prozessor ist nichts anderes) zu beschreiben und zu berechnen.

Die beiden Zustände, die auf den Leitungen herrschen können, können folgenderweise definiert werden:

Dual-zahl	Spannung V	Strom fließt	Elektrischer Pegel	Logischer Wahrheitswert
0	0 - 0.8	nein	Low	falsch
1	2.4 - 5	ja	High	wahr

Ich habe die englischen Pegelbezeichnungen angegeben, die ich im weiteren nur mit den Abkürzungen Lo und Hi verwenden werde. Die logischen Wahrheitswerte werden in der Booleschen Algebra verwendet.

Das Bit

Die kleinste Einheit des Dualsystems, das auch als Binärsystem bezeichnet wird, ist das Bit; es kann den Wert 0 oder 1 besitzen.

Ein Mikroprozessor »denkt« in Bits, da die Speicherzellen genauso wie die Informationsleitungen im Computer, die Busse, nur 2 verschiedene Zustände kennen.

Im Dualsystem kann nach den gleichen Regeln gerechnet werden wie in unserem tagtäglich gebrauchten Dezimalsystem, dessen Rechenregeln uns in »Fleisch und

Blut« übergegangen sind. Wir rechnen auch, ohne uns große Gedanken darüber zu machen, noch in einigen weiteren Zahlensystemen - z. B. in einem System mit der Basis 24 oder gar 60: Es ist die Zeitrechnung mit 24 Stunden und 60 Minuten.

1.2.2 Das Sedezimal- oder Hexadezimalsystem

Aus wievielen Ziffern besteht eine Dualzahl mit 16 Bit? Richtig, aus 16 Ziffern, wie z. B. die Dualzahl 1111111111111111; sie entspricht der Zahl 65535 im Dezimalsystem. Ich hoffe, ich habe mich bei den Einsern nicht verzählt! Sie sehen, für einen normalen Sterblichen scheint das Dualsystem nicht besonders gut geeignet zu sein, dafür ist es unserem Mikroprozessor auf den Leib geschneidert. Vielleicht gibt es eine Möglichkeit, diese Ziffernschlangen etwas zu kürzen, um das Arbeiten mit ihnen zu erleichtern. Da ich oben den dezimalen Wert der »Einserschlange« angegeben habe, muß anscheinend eine Rechenregel, ein Algorithmus (benannt nach dem Namen des arabischen Schriftstellers Ibu Musa al-khowarizmi; um 825 n. Chr.) existieren, mit dem man Zahlen aus einem System in ein anderes umrechnen kann. Diese Algorithmen werden wir weiter unten kennenlernen. Fürs erste genügt es, einige Zahlen in verschiedenen Systemen mit ihren Werten gegenüberzustellen:

System Basis	Binär 2	Quartär 4	Oktal 8	Dezimal 10	Sedezimal 16
	0	0	0	0	0
	1	1	1	1	1
	10	2	2	2	2
	11	3	3	3	3
	100	10	4	4	4
	101	11	5	5	5
	110	12	6	6	6
	111	13	7	7	7
	1000	20	10	8	8
	1001	21	11	9	9
	1010	22	12	10	A
	1011	23	13	11	B
	1100	30	14	12	C
	1101	31	15	13	D
	1110	32	16	14	E
	1111	33	17	15	F
	10000	40	20	16	10

Vergleicht man Zahlen gleichen Wertes in ihrer Länge, so fällt auf, daß für eine 4-Bit-Dualzahl im Dezimalsystem 2 Stellen, im Sedezimalsystem aber nur eine Stelle benötigt wird. Sie vermuten jetzt richtig: Anstelle des Dualsystems verwenden wir in Zukunft nur mehr das Sedezimalsystem. Doch keine Angst, die Buchstaben A bis F sehen zwar in einer Zahl etwas merkwürdig aus, aber man gewöhnt sich sehr schnell daran. Die Buchstaben werden benötigt, da wir nur 10 Ziffernzeichen haben, für das Sedezimalsystem aber 16 benötigen. Es ist eine Übereinkunft, für die fehlenden 6 Ziffernzeichen die ersten 6 Buchstaben des Alphabetes zu benutzen. So ergibt sich für die Dezimalzahl 43981 die ansprechende Buchstabenfolge ABCD im Sedezimalsystem.

Da wir gerade von Vereinbarungen sprechen - es gibt noch eine weitere: Eine Zahl, geschrieben im Dualsystem, wird mit einem vorangestellten Prozentzeichen gekennzeichnet, oder wüßten Sie sonst, welchen Wert die Zahl 1010 hat? Sedezimale Zahlen werden mit einem vorangestellten Dollarzeichen kenntlich gemacht, Zahlen in Dezimal werden ohne Kennzeichnung benutzt oder mit dem vorangestellten amerikanischen Gitterkreuz versehen.

Nach dieser Definition können Sie sofort sagen, die Zahl 1010 ist aus dem Dezimalsystem und hat den Wert Eintausendundzehn.

Das Nibble, das Byte und das Wort

Wissen Sie schon, was ein Nibble, oder gar, was ein Byte ist? Wenn ja, dann können Sie den Rest dieses Abschnitts überspringen, wenn nein:

Ein Nibble besteht aus 4 Bit, entspricht also exakt einer Ziffer im Sedezimalsystem, das Byte besteht aus 2 Nibble, also aus 8 Bit.

$$8\ Bit = 2\ Nibble = 1\ Byte$$

Das Wort (Englisch: Word) besteht aus 2 Byte, ist also 16 Bit lang. Die Adressen in unserem Prozessor werden in einem Wort angegeben. Das Kilobyte besteht *nicht* aus 1000 Byte, sondern aus

$$2^{10}\ Byte = 1024\ Byte = 1\ KByte.$$

1.2.3 Buchstaben und der ASCII-Code

Unser Prozessor soll nicht nur Zahlen verarbeiten, sondern auch andere Zeichen wie z. B. Buchstaben. Nun, er kann es nicht. Buchstaben und Sonderzeichen kann er nicht verstehen, er verarbeitet wirklich nur Dualzahlen. Wenn wir ihn aber überlisten können und die Buchstaben kodieren, was passiert dann? Solange wir nur Zahlen zur Kodierung verwenden, im Bereich des erlaubten Zahlenbereichs bleiben, eine

Kodier- und Dekodiervorschrift entwickeln und es zusätzlich verstehen, passende Ein- und Ausgabegeräte zu entwickeln, müßte es eigentlich funktionieren.

Der erlaubte Zahlenbereich liegt bei unserem Prozessor, da er 8-Bit-Zahlen verarbeitet, im Bereich von $00 bis $FF (0 bis 255). Wir haben also die Möglichkeit, mit 8 Bit $2^8 = 256$ Zeichen zu verschlüsseln.

Stellen wir uns als nächstes einen Kodier- und Dekodierschlüssel zusammen: Teilen wir dem Buchstaben »A« die Zahl 65 zu, so ist »B« dann 66, ..., »Z« ist 90. Mit diesen Zahlen können wir ohne Probleme mit unserer CPU arbeiten. Ein standardisiertes Kodierschema ist der ASCII-Code. ASCII ist die Abkürzung für American Standard Code for Information Interchange (= Amerikanischer Standardkode zum Informationsaustausch). Er benötigt nur 7 Bit. Unser C128 verwendet einen vom Standard abweichenden Code für die interne Speicherung und einen weiteren Kode für die Darstellung der Zeichen auf dem Bildschirm.

Die benötigte Ein- und Ausgabeeinheit stellt uns der C128 zur Verfügung.

1.3 Der 6502-Mikroprozessor

Die CPU 8502 gehört zur Klasse der 8-Bit-Prozessoren, d. h., sie besitzt einen 8-Bit breiten Datenbus und ein 16-Bit breites Adressenregister. Zu dieser Klasse gehören neben der 6500-Familie noch die 6800-, 8080- und die Z80-Familien. Diese Familien unterscheiden sich von unserem Prozessor durch eine andere Architektur und eine andere Maschinensprache.

In Bild 1.2 ist die Anschlußbelegung und in Bild 1.3 das Blockdiagramm der internen Prozessorarchitektur des 8502 zu finden. Sie ist identisch mit der Architektur der CPU 6502 - bis auf die zusätzlichen Ein-/Ausgabe-Leitungen und das Fehlen einiger Steuerleitungen.

Takteingang	1	40	RES	
RDY	2	39	R/W	
IRQ	3	38	D0	8-Bit-
NMI	4	37	D1	Datenbus
AEC	5	36	D2	
+5V	6	35	D3	
16-Bit- A0	7	34	D4	
Adreßbus A1	8	33	D5	
A2	9	32	D6	
A3	10	31	D7	
A4	11	30	P0	7-Bit-I/O-
A5	12	29	P1	Port
A6	13	28	P2	
A7	14	27	P3	
A8	15	26	P4	
A9	16	25	P5	
A10	17	24	P6	
A11	18	23	A15	
A12	19	22	A14	
A13	20	21	Masse	

Bild 1.2 Pinbelegung der CPU 8502

Der 8502 kann mit 16 Adreßleitungen 2^{16} = 64 KByte = 65536 Speicherzellen direkt ansprechen. Um, wie im C128, einen größeren Speicher verwalten zu können, haben sich die Entwickler des C128 etwas einfallen lassen. Sie gaben dem 8502 einen weiteren Baustein zur Unterstützung bei, eine MMU (Englisch: Memory Management Unit = Speicherverwaltungseinheit). Mit diesem Kunstgriff kann im C128 mit mehr als den normalen 64-kByte-Speicher gearbeitet werden. Die MMU verwaltet die Datenzugriffe auf die einzelnen Speicherbänke. Eine Bank im C128 ist 64 kByte groß.

Der Adreßbus ist unidirektional, d. h., der Prozessor kann den Bus nur beschreiben und nicht lesen. Um eine Speicherzelle ansprechen zu können, gibt der Prozessor ihre Adresse auf den Bus auf.

Der Datenbus besteht aus 8 Leitungen, auf denen ein Datum bidirektional, also in beiden Richtungen, vom Prozessor zum Speicher als auch vom Speicher zum Prozessor gesandt werden kann.

Um die Abläufe zu steuern, besitzt die CPU noch einen Steuerbus, um z. B. der angesprochenen Speicherzelle mitzuteilen, ob sie gelesen oder beschrieben werden soll. Um auf externe Ereignisse reagieren zu können, gibt es im Steuerbus noch die Interrupt-Leitungen (Englisch: Unterbrechung).

Die CPU 8502 kennt 56 Befehle und 13 Adressierungsarten, sie kann mit einer Taktfrequenz von 1 oder 2 MHz betrieben werden, und sie gestattet einen direkten Speicherzugriff (Englisch: Direct Memory Access = DMA). Als Besonderheit besitzt die Familie der 6500-Prozessoren eine Pipeline-Architektur (Englisch: Rohrleitung).

Durch diese Architektur kann die CPU einen neuen Befehl aus dem Speicher holen und dekodieren, während sie noch mit der Abarbeitung des vorhergegangenen Befehls beschäftigt ist. Damit ist die Verarbeitungsgeschwindigkeit bei gleichen Taktfrequenzen größer als bei den Prozessoren der anderen Familien (Ausnahme: Die 6800-Familie, die gleich schnell ist).

Auf den folgenden Seiten wollen wir anhand der Bilder 1.3 und 1.5 die interne Struktur des 8502 näher betrachten.

1.3.1 Die arithmetisch-logische Einheit und der Akkumulator

In der Mitte des Bildes ist die arithmetisch-logische Einheit (= ALU) zu erkennen. Sie ist das Rechenzentrum unseres Prozessors. In ihr werden in Zusammenarbeit mit dem Akkumulator, kurz Akku genannt, alle arithmetischen und logischen Operationen durchgeführt.

Die ALU besitzt 2 Eingänge, einer davon ist mit dem Akku verbunden, und einen Ausgang. Der Akku, ein 8-Bit breites Register nimmt das Ergebnis einer Operation auf. Da nur 8-Bit-Zahlen verarbeitet werden können, müssen Operationen mit 16-Bit-Zahlen sequentiell (= nacheinander) durchgeführt werden.

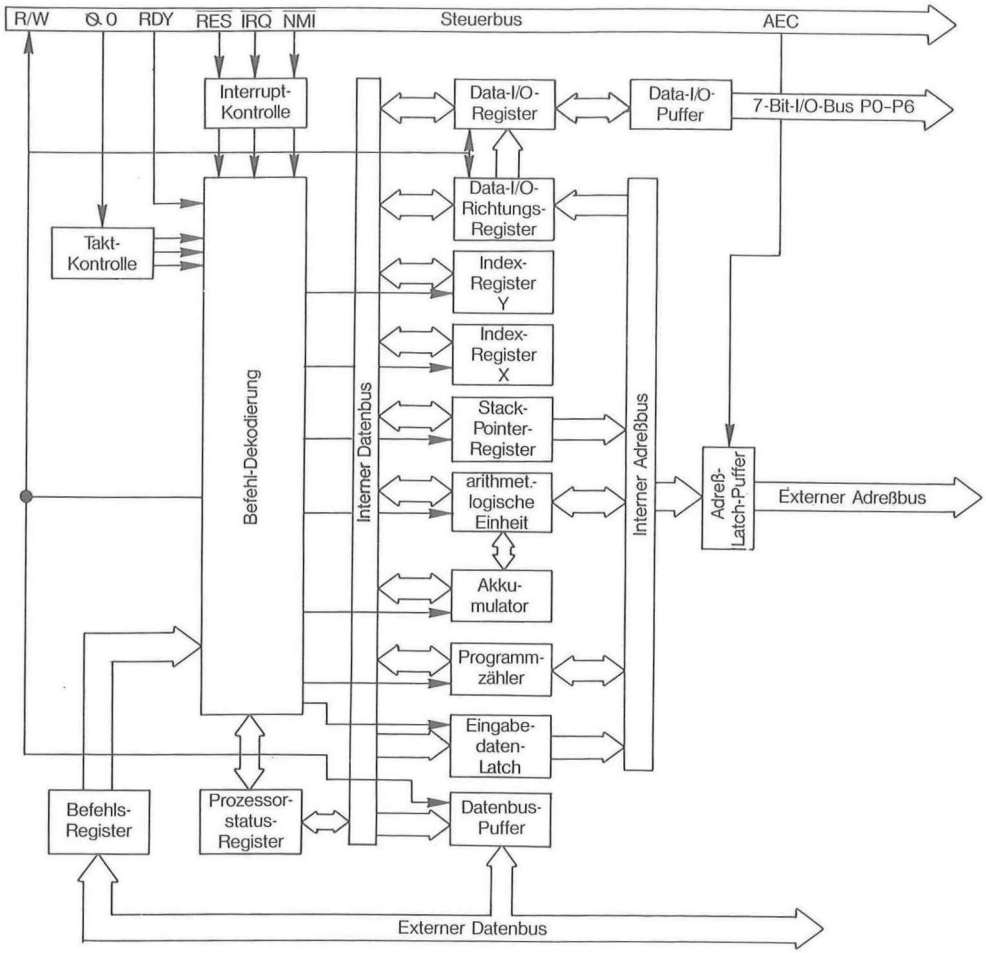


Bild 1.3 Blockdiagramm der CPU 8502

1.3.2 Das Stackpointer-Register

Über der ALU sehen wir das Stackpointer-Register. Es beinhaltet einen 8-Bit breiten Zeiger (Englisch: Pointer), der auf den Stack (Englisch: Stapel) zeigt. Der Stack ist ein besonderer Adreßbereich, der mit einer Länge von 256 Byte in der Page 1 (Englisch: Seite) liegt. Das ist der Adreßbereich von dezimal 256 bis 511 ($0100 bis $01FF). Wenn man die Adressen betrachtet, stellt man fest, daß der Zeiger 16 Bit groß sein müßte, um den Stack korrekt ansprechen zu können. Da der Stackbereich

aber fest definiert ist, er also immer an der gleichen Stelle liegt, kann der Prozessor automatisch das höherwertige Adreßbyte, das immer $01 ist, beisteuern.

Schritt	Operation	Stackpointer	Adresse	Stackinhalt
1	keine	$FF	$01FF	undefiniert
2	Schieben	$FE	$01FF $01FE	Datum 1 undefiniert
3	Schieben	$FD	$01FF $01FE $01FD	Datum 1 Datum 2 undefiniert
4	Holen	$FE	$01FF S01FE	Datum 1 undefiniert
usw.				

Bild 1.4 Nachführen des Stackpointers

Der Stack wird vom Prozessor als besonderer Speicher für wichtige Daten, z. B. Rücksprungadressen von Subroutinen benutzt. Der Zeiger im Stackpointer-Register zeigt immer auf die nächste freie Adresse auf dem Stack. Er wird mit jedem Schieben von Daten auf den Stack oder Lesen von Daten vom Stack automatisch vom Prozessor mitgeführt. Das Datum, das als letztes auf den Stack geschoben wurde, wird als erstes wieder vom Stack geholt. Diese Organisation eines Speichers wird als LIFO-Struktur (Englisch: Last in, First out = als letzter drauf, als erster wieder runter) bezeichnet.

Auf eine Eigenschaft des Stacks der 6500-Familie muß gesondert hingewiesen werden: Der Stack wird abwärts, also von der höchsten Adresse nach unten zur niedrigsten, mit Daten gefüllt. Bild 1.4 sollte die Art und Weise des Zugriffs auf den Stack deutlich machen. Der Begriff »undefiniert« im Bild bedeutet nur, daß der Inhalt dieser Speicherzelle belanglos ist, da ein erneuter Zugriff auf diese Zelle nur schreibend erfolgen kann und dadurch der alte Inhalt überschrieben wird.

Der Stackpointer kann aus einem Programm heraus geändert werden.

1.3.3 Die Zeropage

Den Begriff Zeropage (Englisch: Zero = Null = Seite 0) werden Sie im Blockdiagramm vergeblich suchen. Es handelt sich wie beim Stack um einen besonderen Speicherbereich, auf den jedoch kein Pointer zeigt.

Der gesamte Adreßraum mit 64 kByte läßt sich in 256 Seiten mit je 256 Byte unterteilen. Die Seite 0 von $0000 bis $00FF und die Seite 1, der Stack, werden vom Prozessor gesondert behandelt. In beiden Fällen steuert der Prozessor automatisch das höherwertige Adreßbyte bei; für die Zeropage ist es immer $00. Das Einzigartige an der Zeropage ist, daß der Prozessor eine Reihe von Befehlen mit einer speziellen Adressierungsart kennt, die kurz und sehr schnell in der Abarbeitung sind.

1.3.4 Der Programmzähler

Unterhalb des Akkus finden Sie den Programmzähler. Es ist ein 16-Bit breites Zählregister, das den gesamten natürlichen Adreßbereich des Prozessors überstreichen kann. Bei jedem Zugriff des Prozessors auf den Programmspeicher, um einen Befehl oder einen Operanden zu holen, wird der Programmzähler (Englisch: Program Counter = PC) automatisch nachgeführt. Auch der Programmzähler kann von einem Programm manipuliert werden.

1.3.5 Die Indexregister

Es wäre nicht möglich, mit nur einem Arbeitsregister, dem Akku, effiziente Programme zu schreiben. Deshalb besitzt die CPU noch zwei weitere Register, die X- und Y-Indexregister genannt werden. Beide sind 8 Bit breit und können zur Lösung verschiedener Aufgaben herangezogen werden. In den folgenden Abschnitten werden Sie weitere Informationen zu diesen beiden Registern finden.

1.3.6 Das Prozessorstatus-Register

Dieses Register ist ebenfalls 8 Bit breit. Jedes der Bits, bis auf das unbenutzte Bit 6, kann vom Prozessor gesetzt (Englisch: to set) oder gelöscht (Englisch: to reset oder: to clear) werden. Es signalisiert damit einen ganz bestimmten Zustand, der in einem Programm abgefragt werden kann. Die Bits funktionieren wie Fahnen (Englisch: Flag), die aufgezogen oder wieder eingeholt werden können. Im Bild 1.5 sind die Flags in das Prozessorstatusregister eingezeichnet. Auf die Verwendungs-

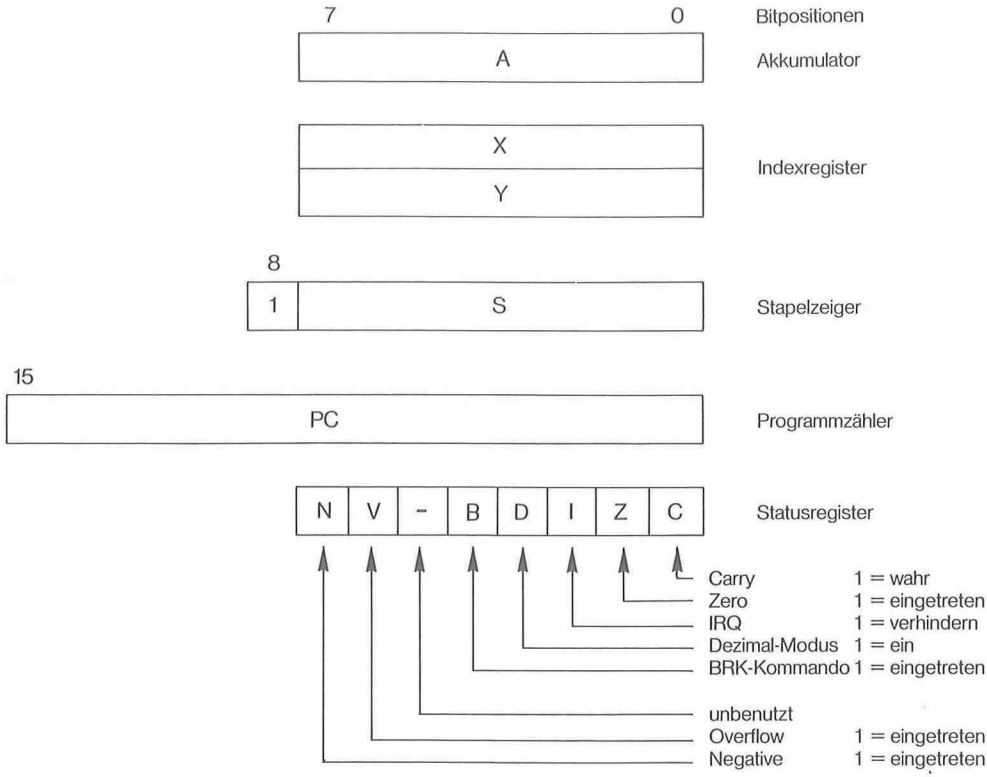


Bild 1.5 Die 8502-Register

möglichkeiten, die dieses Register mit seinen Flags dem Programmierer bietet, werden wir etwas später noch genauer eingehen.

1.3.7 Das Befehlsregister und die Befehlsdekodierung

Im Befehlsregister wird der Befehl, der aus dem Speicher geholt worden ist, zwischengespeichert. Im nächsten Schritt wird er dekodiert und abgearbeitet.

1.3.8 Die Taktkontrolle

Das Zusammenleben in einer Gemeinschaft kann nur dann gut funktionieren, wenn alle Mitglieder ein Gefühl für Takt im Umgang miteinander entwickeln. Ein Computersystem besteht ebenfalls aus vielen verschiedenen Mitgliedern, die miteinander kommunizieren (= Informationen austauschen) müssen. Die gemeinsame Basis, zu welchem Zeitpunkt etwas geschehen muß, wird durch den Systemtakt kontrolliert. Auch die CPU muß sich an dieses Protokoll halten. Mit dem Takt wird die CPU und die gesamte Peripherie synchronisiert. Damit ist sichergestellt, daß das spezifizierte Zeitprotokoll der integrierten Schaltkreise eingehalten wird und die Übertragung und Verarbeitung von Informationen (= Daten) auf den Bussen und in den ICs (Englisch: Integrated Circuits = Integrierte Schaltkreise) zu definierten Zeiten erfolgt.

1.3.9 Steuerbus und die Interruptlogik

Unter einem Bus oder einem Bussystem werden elektrische Leitungen, die gleiche oder ähnliche Signale transportieren, zusammengefaßt. Über den Steuerbus regelt die CPU den Verkehr mit der Umwelt. Er besteht, soweit es die CPU 8502 betrifft, aus 7 Leitungen. Sie haben folgende Funktionen:

- R/W: Mit dieser Leitung legt die CPU fest, ob sie Daten über den Datenbus lesen (Read) oder schreiben (Write) möchte. Ein High (= 1) bedeutet Lesen, ein Low (= 0) Schreiben.
- PHI0: Ist der Eingang des schon oben beschriebenen Systemtaktes.
- RDY: Gelangt über diesen Eingang ein High-Signal an die CPU, so stoppt sie nach Abarbeitung des letzten Befehls ihre Funktionen, bis die Leitung wieder auf Low liegt. Damit können langsame Speicherbausteine mit der CPU synchronisiert werden. RDY steht für READY (Englisch: fertig).
- RES: Ein Impuls von definierter Dauer auf der RESET-Leitung (Englisch: Zurücksetzen) setzt die CPU in einen definierten Anfangszustand.
- IRQ: Über diese Leitung kann die CPU gezwungen werden, ihre gerade laufende Tätigkeit zu unterbrechen und eine andere Arbeit einzuschieben. Der IRQ (Englisch: Interrupt Request = Aufforderung zur Unterbrechung) kann durch das IRQ-Flag im Prozessorstatusregister verhindert werden.
- NMI: Im Gegensatz zum IRQ kann der NMI (Englisch: Non maskable Interrupt = nicht maskierbare Unterbrechungsanforderung) *nicht* untersagt werden. Die CPU muß dieser Aufforderung in jedem Fall Folge leisten.
- AEC: Über den AEC-Eingang (Englisch: Address Enable Control = Adreßbus gültig machen) kann mit einem Low-Signal der Adreßbus von der CPU

getrennt werden, um von außen einen direkten Zugriff auf die Speicher zu ermöglichen.

Die Interruptlogik wertet die 3 möglichen Interruptaufforderungssignale RES, IRQ und NMI aus und übermittelt die Anforderung dann an die Befehlsdekodierung.

1.3.10 Der Adreßbus

Der 16-Bit breite Adreßbus ist unidirektional, d. h., die CPU kann den Adreßbus nicht lesen, sondern nur auf ihn schreiben. Es können 65536 Speicherzellen (auch die I/O-Bausteine werden als Speicherzellen adressiert) eindeutig angesprochen werden. Die 65536 Speicherzellen werden über die Adressen 0-65535 festgelegt.

1.3.11 Der Datenbus

Nach Ausgabe der Adresse und der R/W-Steuerung kann über den 8 Bit breiten, bidirektionalen Datenbus das Datum entweder aus der angesprochenen Speicherzelle gelesen oder in sie hineingeschrieben werden.

1.3.12 Die Data-I/O-Register

Die CPU 8502 verfügt, wie schon oben erwähnt, über einen speziellen I/O-Port. Es sind 8 Leitungen P0 bis P7 herausgeführt. Dieser Port ist bidirektional und gestattet einen direkten Datentransfer von und zur Peripherie. Die Datenflußrichtung kann im Data-I/O-Richtungsregister für jede Leitung getrennt festgelegt werden. Die Daten selbst werden in das Data-I/O-Register geschrieben bzw. aus ihm gelesen. Das Datenrichtungsregister liegt an der Adresse $0000 und das Datenregister bei $0001.

1.3.13 Die Unterschiede zur CPU 6502

Im Unterschied zum 8502 besitzt die 6502 keine I/O-Ports und keinen AEC-Eingang, dafür aber noch einige weitere Steuerleitungen und einen eingebauten Taktgenerator. Die zusätzlichen Steuerleitungen haben folgende Namen und Funktion:

- PHI01: Mit diesen Taktleitungen synchronisiert die CPU
- PHI02 selbst das gesamte System.
- SYNC: Mit einem Signal auf dieser Leitung zeigt die CPU, daß sie den nächsten Befehl zur Dekodierung aus dem Speicher holt.

- SO: Durch ein Signal auf dieser Leitung kann das Überlaufflag (Englisch: Set Overflow Flag = SO) im Prozessorstatusregister gesetzt werden.

Die bisher noch nicht angesprochenen Funktionsblöcke dienen entweder zur Zwischenspeicherung von Daten oder es handelt sich um Treiber, die die Ausgangsleistung der Signale auf den Leitungen erhöhen.

Weitere Informationen können den einschlägigen Hardwarebüchern und den Datenblättern entnommen werden, auf die im ausführlichen Literaturverzeichnis (Kapitel 1.23) hingewiesen wird.

1.4 Vom Problem zum Flußdiagramm und Struktogramm

Nachdem wir die Assemblersprache gegenüber den höheren Programmiersprachen abgegrenzt und uns einige Grundlagen in Zahlensystemen und Prozessorarchitektur erarbeitet haben, benötigen wir noch ein Gerüst für die Programmierung, bevor es an die ersten Befehle in Assembler geht.

1.4.1 Die Problemstellung

Die Reihenfolge, die zur Erstellung eines Programmes führt, ist eigentlich immer die gleiche:

Es beginnt mit einem Problem. Um das Problem lösen zu können, müssen Fragen gestellt und beantwortet werden:

- Wie läßt sich das Problem beschreiben?
- Ist die Beschreibung des Problems eindeutig?
- Welche Eingabegrößen sind bekannt?
- Welche Ausgabegrößen sind gesucht?
- Wie müssen diese Größen miteinander verknüpft werden, und gibt es dabei besondere Bedingungen zu berücksichtigen?

Ich habe mir angewöhnt, ein Problem zunächst mit meinen eigenen Worten schriftlich zu beschreiben, diesen Text auf Eindeutigkeit zu überprüfen und gegebenenfalls zu überarbeiten oder sogar komplett neu zu formulieren.

Im nächsten Schritt werden alle Eingabegrößen aufgelistet, benannt und auf ihren möglichen Bereich überprüft. Die Eingabe z. B. eines Wochentages kann nur ein String (Englisch: eine Folge von Zeichen) mit den Inhalten Montag, Dienstag, ... bis einschließlich Sonntag sein.

Jetzt ist es an der Zeit, die Verknüpfungs- oder Verarbeitungsvorschriften der Daten miteinander zu notieren und zu analysieren. Denkbare Vorschriften können z. B. mathematische Formeln sein. Unter besondere Bedingungen könnten z. B. die umsatzhöhen-abhängigen Berechnungen der Provision eines Verkäufers fallen.

Nach Beantwortung aller Fragen sollte anhand der Problemstellung noch einmal überprüft werden, ob das Problem mit den mittlerweile erarbeiteten Unterlagen auch wirklich vollständig gelöst ist.

1.4.2 Der Lösungsweg

Je genauer und detaillierter diese Liste erstellt wird, desto einfacher ist es dann, den Lösungsweg auszuarbeiten. In der Liste können Sie die drei Schritte jeder Datenverarbeitung erkennen:

- Eingabe
- Verarbeitung
- Ausgabe

Da die meisten Probleme komplex sind, ist es besser, sie in kleinere, überschaubare Teilprobleme zu zerlegen und für diese wieder die oben genannten Schritte zu vollziehen.

Bei der Lösung eines kleinen Problems bietet sich die Aufteilung des Programmes in die drei Module (Lateinisch: aus leicht austauschbaren, kleinen Teilen zusammengesetzte Einheit) Eingabe, Verarbeitung und Ausgabe an. Jedes dieser Module wird für sich selbst betrachtet und bearbeitet. Doch ein Punkt ist äußerst wichtig:

Ein Modul besitzt nur einen Eingang und nur einen Ausgang!

Damit soll ausgedrückt werden, daß nur an einer Stelle die Eingabedaten an das Modul übergeben werden und sie nur an einer Stelle nach der Bearbeitung wieder ausgegeben werden. Die Ein- und Ausgänge werden als Schnittstellen bezeichnet. Jedes Modul erwartet an seinem Eingang die Daten, die es bearbeiten soll. Damit sind wir beim zweiten wichtigen Punkt:

Die Datenformate und die Reihenfolge der Daten, die über die Schnittstellen an die Module übergeben werden, müssen exakt definiert sein und müssen immer eingehalten werden!

1.4.3 Die Benutzer-Schnittstelle

Benötigt Ihr Programm Daten, die interaktiv (Lateinisch: wechselweise Handlung) eingegeben werden müssen, oder es muß ein Mitmensch mit den Ergebnissen Ihres Programmes weiterarbeiten, so nehmen Sie auf den armen Benutzer Rücksicht!

Geben Sie ihm alle Informationen auf dem Bildschirm aus, die er benötigt, um die Eingabeaufforderungen zu verstehen. Geben Sie ihm die Möglichkeit, Tippfehler zu korrigieren. Prüfen Sie seine Eingaben und weisen Sie Fehleingaben mit einer aussagekräftigen Fehlermeldung zurück, damit sich das Programm nicht einfach verabschiedet oder gar die Ergebnisse der Verarbeitung schlicht und einfach falsch sind. Bauen Sie die Bildschirmeingabemasken übersichtlich auf. Die beiden besten Tips, die ich Ihnen dazu geben kann:

- Zeichnen Sie Ihre Bildschirmmasken vorher auf ein Blatt Papier auf!
- Bitten Sie einen Computerneuling, Ihr Programm zu testen. Sie werden an seinen Reaktionen sofort erkennen können, wo Sie noch Verbesserungen einbauen müssen.

1.4.4 Flußdiagramme und Struktogramme

Mittlerweile sind wir an einen Punkt gelangt, an dem die verbale Beschreibung der Problemlösung nicht mehr besonders gut zur Darstellung des Ablaufes geeignet ist. Besser eignet sich eine grafische Darstellung des Ablaufes der Problemlösung in Form eines Diagrammes.

Damit die grafischen Darstellungen von Programm- und Datenabläufen von jedermann verstanden werden können, hat der »Normenausschuß Informationsverarbeitung im Deutschen Institut für Normung e.V.« zwei Normen verabschiedet. Es sind dies die Norm »DIN 66001 Sinnbilder und ihre Anwendung« und die »DIN 66261 Sinnbilder für Struktogramme nach Nassi-Shneidermann«. In Bild 1.6a und 1.6b finden Sie eine Gegenüberstellung der Sinnbilder nach beiden Normen.

Beispiel:

Problembeschreibung:	Es ist ein Speicherbereich ab $1400 bis $143F mit $00 zu beschreiben.
Eingabegrößen:	Speicherbereich $1400 - $143f Füllbyte = $00
Ausgabegrößen:	keine
Verknüpfung:	keine
Lösung:	Es sind 64 Byte beginnend ab $1400 mit $00 zu beschreiben. Im Akku befindet sich das Füllbyte, das X-Register wird als Zähler und Indexregister benutzt. Da die Abfrage auf Null (Sie werden später sehen warum) am einfachsten zu realisieren ist, wird das X-Register von 64 abwärts gezählt.

Ablaufschema: Siehe Bild 1.7

Bemerkung: Die allgemeine Problemstellung lautet: Fülle einen Speicherbereich ab Startadresse bis Endadresse mit einem Füllbyte. Wenn die Problembeschreibung und -lösung so formuliert werden, gelangt man zu einem universell einsetzbaren Programm, ohne daß der Programmieraufwand nennenswert erhöht wird. Die allgemeinere Problemlösung ist prinzipiell der speziellen vorzuziehen.

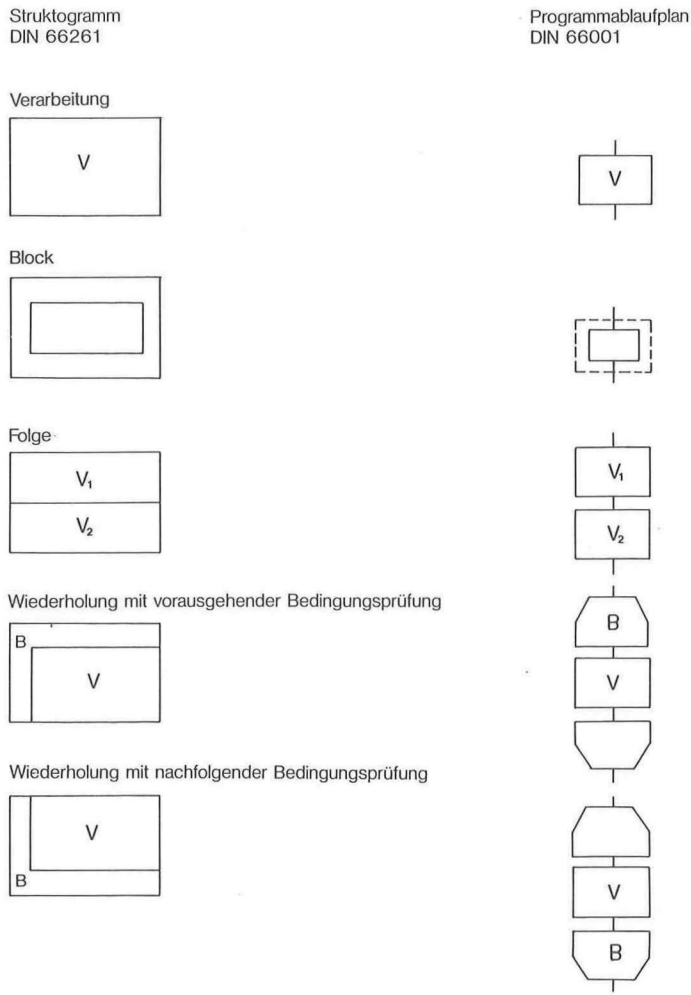


Bild 1.6a Gegenüberstellung der Sinnbilder

Geben Sie ihm alle Informationen auf dem Bildschirm aus, die er benötigt, um die Eingabeaufforderungen zu verstehen. Geben Sie ihm die Möglichkeit, Tippfehler zu korrigieren. Prüfen Sie seine Eingaben und weisen Sie Fehleingaben mit einer aussagekräftigen Fehlermeldung zurück, damit sich das Programm nicht einfach verabschiedet oder gar die Ergebnisse der Verarbeitung schlicht und einfach falsch sind. Bauen Sie die Bildschirmeingabemasken übersichtlich auf. Die beiden besten Tips, die ich Ihnen dazu geben kann:

- Zeichnen Sie Ihre Bildschirmmasken vorher auf ein Blatt Papier auf!
- Bitten Sie einen Computerneuling, Ihr Programm zu testen. Sie werden an seinen Reaktionen sofort erkennen können, wo Sie noch Verbesserungen einbauen müssen.

1.4.4 Flußdiagramme und Struktogramme

Mittlerweile sind wir an einen Punkt gelangt, an dem die verbale Beschreibung der Problemlösung nicht mehr besonders gut zur Darstellung des Ablaufes geeignet ist. Besser eignet sich eine grafische Darstellung des Ablaufes der Problemlösung in Form eines Diagrammes.

Damit die grafischen Darstellungen von Programm- und Datenabläufen von jedermann verstanden werden können, hat der »Normenausschuß Informationsverarbeitung im Deutschen Institut für Normung e.V.« zwei Normen verabschiedet. Es sind dies die Norm »DIN 66001 Sinnbilder und ihre Anwendung« und die »DIN 66261 Sinnbilder für Struktogramme nach Nassi-Shneidermann«. In Bild 1.6a und 1.6b finden Sie eine Gegenüberstellung der Sinnbilder nach beiden Normen.

Beispiel:

Problembeschreibung:	Es ist ein Speicherbereich ab $1400 bis $143F mit $00 zu beschreiben.
Eingabegrößen:	Speicherbereich $1400 - $143f Füllbyte = $00
Ausgabegrößen:	keine
Verknüpfung:	keine
Lösung:	Es sind 64 Byte beginnend ab $1400 mit $00 zu beschreiben. Im Akku befindet sich das Füllbyte, das X-Register wird als Zähler und Indexregister benutzt. Da die Abfrage auf Null (Sie werden später sehen warum) am einfachsten zu realisieren ist, wird das X-Register von 64 abwärts gezählt.

Ablaufschema: Siehe Bild 1.7

Bemerkung: Die allgemeine Problemstellung lautet: Fülle einen Speicherbereich ab Startadresse bis Endadresse mit einem Füllbyte. Wenn die Problembeschreibung und -lösung so formuliert werden, gelangt man zu einem universell einsetzbaren Programm, ohne daß der Programmieraufwand nennenswert erhöht wird. Die allgemeinere Problemlösung ist prinzipiell der speziellen vorzuziehen.

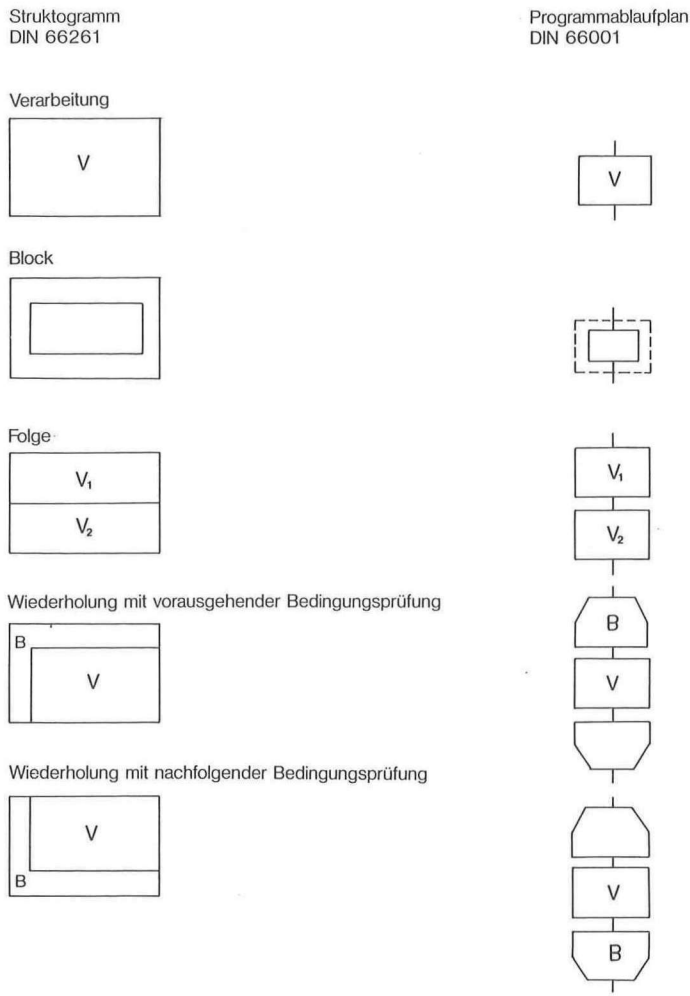
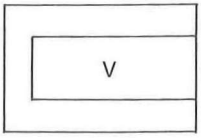


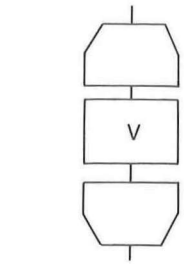
Bild 1.6a Gegenüberstellung der Sinnbilder

Struktogramm
DIN 66261

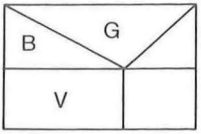
Programmablaufplan
DIN 66001

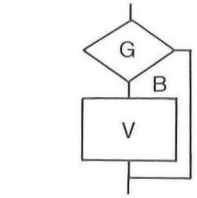
Wiederholung ohne Bedingungsprüfung



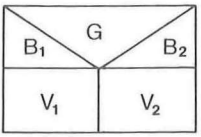


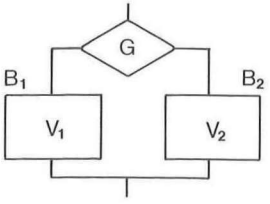
bedingte Verzweigung



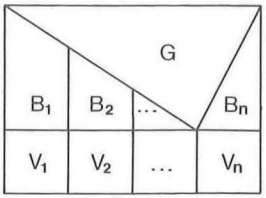


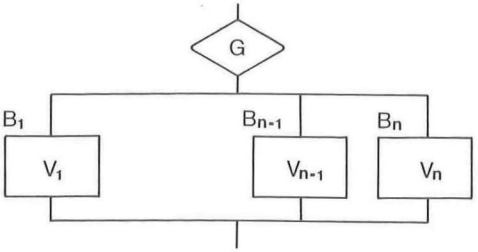
einfache Alternative





mehrfache Alternative





Abbruchanweisung

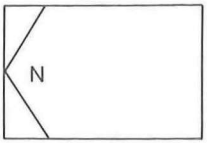


Bild 1.6b Gegenüberstellung der Sinnbilder

Die Frage, welche der beiden Normen man zur Erstellung der Ablaufpläne einsetzen soll, muß jeder Programmierer für sich selbst entscheiden. Ich persönlich ziehe die Sinnbilder nach Nassi-Shneidermann vor, da sie einen modularen, klar strukturierten Programmentwurf unterstützen. Modulare, strukturierte Programme zeichnen sich durch eine gute Lesbarkeit und eine ausgezeichnete Wartungsfreundlichkeit aus. Sie besitzen noch einen weiteren, nicht zu unterschätzenden Vorteil: Die Ersatzsymbole in der Norm sind für den Einsatz in einer Textverarbeitung konzipiert, d. h., man kann die Struktogramme ohne große Probleme mit seinem Textverarbeitungsprogramm und einem normalen Drucker erstellen.

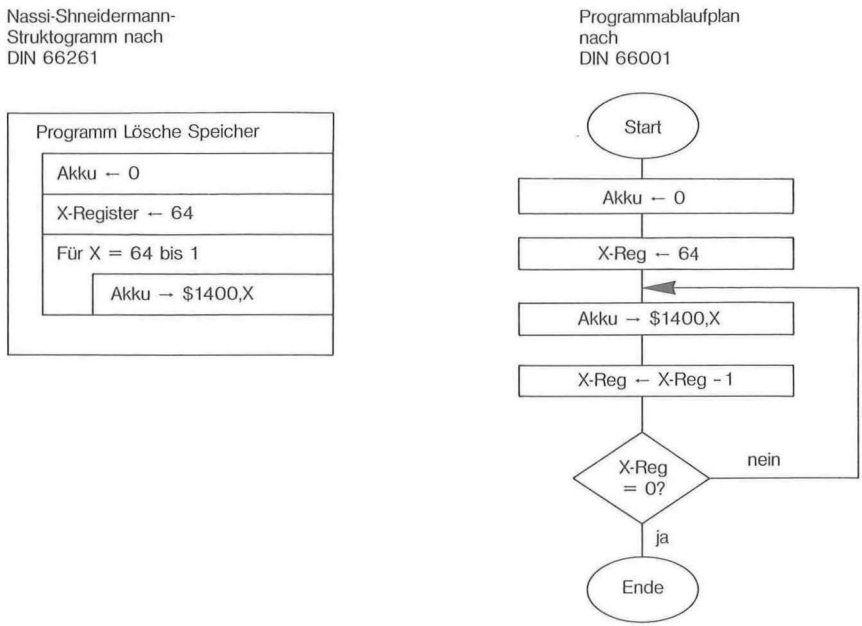


Bild 1.7 Programmablaufdiagramme nach DIN 66001 und DIN 66261 für das Programm »Lösche Speicherbereich«

1.4.5 Dokumentation

Eine ungeliebte Aufgabe, um die jeder Programmierer versucht, einen großen Bogen zu machen, ist die Dokumentation. Doch diese Arbeit ist sinnvoll. Spätestens nach einem Jahr, wenn man sein Programm erweitern oder umstellen möchte, stellt man fest, daß man sich doch nicht alles gemerkt hat. Die damalige Überzeugung, es geht auch ohne Dokumentation, da man das Programm ja selbst geschrieben hat, löst sich

in Frustration auf. Beim besten Willen fällt einem nicht mehr ein, warum das Programm an dieser bestimmten Stelle so umständlich programmiert wurde. Na, das wollen wir ändern, denn heute können wir es besser! Gesagt, getan - nur läuft das Programm jetzt nicht mehr.

Damit Sie nicht auch in diese Situation gelangen (ich muß gestehen, mir ist es einige Male passiert, bis ich daraus lernte), gewöhnen Sie sich an, die Dokumentation während aller Phasen der Programmerstellung, beginnend von der Problembeschreibung bis zu den Endtests, mitzuführen.

Was gehört zu einer Dokumentation?

- Die Problemstellung
- Die Problemlösung
- Die Programmlaufschemata (Flußdiagramme und Struktogramme)
- Die dokumentierten Programmlistings
- Der dokumentierte Testlauf
- Die Bedieneranleitung

Sie sehen, es ist nicht gerade wenig.

Im folgenden nur mehr einige Anmerkungen zu den Programmlistings. Allen meinen Programmen verpasse ich den Programmkopf in Bild 1.8. Abhängig von der Programmiersprache und der Art des Programmes können auch noch weitere Informationen aufgenommen werden. Es sind alle benötigten Informationen enthalten, um das Programm oder die Routine einwandfrei identifizieren und mit ihr arbeiten zu können. Für die 6500-Assembler leitet das Semikolon einen Kommentar ein, der bei der Assemblierung nicht berücksichtigt wird. Kommentieren Sie Ihre Programme reichlich, aber sinnvoll! Sinnlos wäre z. B. folgender Kommentar, da die Funktion eindeutig klar ist:

```
LDA #$03 ; Lade den Akku mit 3
```

Besser ist folgende Kommentierung, da sie mitteilt, wofür der Wert 3 steht:

```
LDA #$03 ; Lade Akku mit Anzahl Byte
```

Mit etwas Programmiererfahrung werden Sie Ihre eigenen Methoden der Dokumentation entwickeln, die Ihren Bedürfnissen angepaßt sind. Doch bitte ich Sie, die oben angeführten Regeln zu beherzigen. Sie werden manchem Ärger dadurch aus dem Wege gehen!

```
;TOP*****************************************************
; PROGRAMMNAME:          ..................................  *
; ===================================================== *
; BIBLIOTHEK:            ..................................  *
; PROGRAMMIERSPRACHE:    ..................................  *
; PROGRAMMIERER:         ..................................  *
; DATUM:                 ..................................  *
; GEAENDERT:             ..................................  *
; DATUM:                 ..................................  *
; PROGRAMMBESCHREIBUNG:  ..................................  *
;                        ..................................  *
;                        ..................................  *
; ===================================================== *
;                  BENUTZUNG DER REGISTER                  *
; AKKU:                  ..................................  *
; X-REGISTER:            ..................................  *
; Y-REGISTER:            ..................................  *
; ===================================================== *
;                     PARAMETER ÜBERGABE                   *
; NAME                      BESCHREIBUNG                   *
; ----------------------------------------------------- *
; ...................    ..................................  *
; ...................    ..................................  *
; ===================================================== *
;              FELDBESCHREIBUNG DER PARAMETER              *
; NAME     TYP      LAENGE     I/O   BESCHREIBUNG          *
; ----------------------------------------------------- *
; ........ ......   ...         .     ......................  *
; ........ ......   ...         .     ......................  *
; ===================================================== *
;                ERKLÄRUNG DER STEUERVARIABLEN             *
; ..................................................... *
; ..................................................... *
; ===================================================== *
;                     BENUETZTE DATEIEN                    *
; ..................................................... *
; ..................................................... *
;BOTTOM**************************************************
;
; DEKLARATIONEN
```

Bild 1.8 Beispiel eines Programmkopfes

1.5 Der Monitor und die Programmbeispiele

Eines der wichtigsten Hilfsmittel bei der Assemblerprogrammierung ist der Monitor.

1.5.1 Der Monitor

Der Monitor ist kein Bildschirm, wie man vielleicht annehmen könnte; doch kann man ihn mit einem Datensichtgerät vergleichen, da er Einblick in den Speicher und die Register gewährt. Er ist ein residentes Programm, das im Programmspeicher, den ROMs (Englisch: Read Only Memory = Nur-Lese-Speicher), des C128 steht. Residente Programme liegen immer im Speicher, im Gegensatz zu transienten Programmen, die erst von einem peripheren Datenspeicher in die RAMs (Englisch: Random Access Memory = Schreib-Lese-Speicher) geladen werden müssen. Aufgerufen wird der Monitor von BASIC aus mit dem Befehl

```
MONITOR    <RETURN>
```

Der Text in den spitzen Klammern bedeutet, daß Sie die Taste mit dieser Bezeichnung drücken sollen, in diesem Fall also die Taste RETURN.

Der Monitor wird ebenfalls aktiviert, wenn die CPU bei der Ausführung eines Programmes auf den Befehl *BRK*, ein Nullbyte, stößt. Dabei wird auf dem Bildschirm eine Statusanzeige sichtbar:

```
MONITOR
   PC      SR   AC   XR   YR   SP
; FB000    00   00   00   00   F9
```

Mit einem Blick kann man erkennen, in welcher Bank und bei welcher Adresse der Prozessor auf das BRK gestoßen ist. Den Inhalt des Prozessorstatusregisters SR muß man sich erst in eine binäre Zahl umrechnen, um den Zustand der Flags erkennen zu können. Unter den Bezeichnungen AC, XR und YR stehen die Inhalte der gleichnamigen Register. Der Inhalt SP ist der Pointer des Stackpointerregisters (siehe Kapitel 1.3.2). Diese Anzeige aller wichtigen prozessorbezogenen Daten kann beim Debugging (Englisch: Bug = Käfer, d. h. Entwanzen) eines Programmes ungemein helfen.

Der Monitor stellt eine Reihe von Befehlen zur Verfügung, mit denen Sie ein kleines Maschinenspracheprogramm eingeben, testen und korrigieren können. Es gibt Befehle zum Anzeigen und Modifizieren von Speicherzellen und der CPU-Register, Befehle zum Umgang mit Floppylaufwerken und Datasetten und einige Hilfsbefehle. Die Assembler-Programmierumgebung, die der Monitor zur Verfügung stellt, ist, verglichen mit der des TOP-ASS, primitiv, für unsere Beispielprogramme aber ausreichend.

Alle Befehle des Monitors sind im C128-Bedienungshandbuch »Anhang C: Maschinensprache-Monitor« ausführlich behandelt. Aus der Gesamtheit aller Befehle des Monitors werden wir folgende hauptsächlich einsetzen:

- A	Assemblieren	Eingabe und Testen der
- D	Disassemblieren	Beispielroutinen
- F	Fill	Zum Vorbereiten eines Speicherbereiches
- G	Go	Zum Starten des Programmes
- M	Memory	Zum Ansehen und Ändern von
- R	Register	Speicherbereichen und Registern
- X	Exit	Zur Rückkehr nach BASIC

Die Floppy-Befehle S (Save) und L (Load) benötigen Sie, wenn Sie die kleinen Beispielroutinen auf Diskette oder Tonband abspeichern wollen.

Bitte lesen Sie sich die Beschreibung der Monitor-Befehle im Bedienungshandbuch gut durch. Sie erleichtern sich damit das Durcharbeiten der nächsten Kapitel. Sie können sich dann auf das Erlernen der 6500-Assemblersprache allein konzentrieren.

Ein Befehl, der das Arbeiten mit unterschiedlichen Zahlensystemen stark erleichtert, ist von Commodore im Bedienungshandbuch leider unterschlagen worden. Die Symbole »$«, »+«, »&« und »%« dienen nicht nur, wie im Handbuch auf Seite C-3 beschrieben, zur Definition numerischer Werte in verschiedenen Zahlensystemen. Sie sind auch als Kommandos einsetzbar. Die Eingabe von

```
$0A <RETURN>
```

bewirkt die Ausgabe dieser Zahl in allen unterstützten Zahlensystemen:

```
$0A   Sedezimal
+10   Dezimal
&12   Oktal
%1010 Dual
```

Die oktale Zahlenangabe können wir vernachlässigen, da dieses System nicht üblich ist und aus diesem Grund von TOP-ASS in keiner Weise unterstützt wird. Der Vollständigkeit halber sei gesagt, daß oktale Zahlen auf der Basis 8 beruhen und die Ziffern 0-7 haben.

1.5.2 Eingabe der Programmbeispiele

In den nächsten Kapiteln werden die Befehle der CPU 8502 erklärt. Zu den einzelnen Abschnitten werden Sie kleine Routinen, die den Einsatz der Befehle verdeutlichen sollen, finden. Diese »Prográmmchen» werden, aus Platzmangel, ohne Programmkopf und meist ohne Dokumentation, aber mit Kommentar, wo nötig, abgedruckt.

Alle Beispielprogramme haben den gleichen Aufbau. Jede Zeile beginnt mit einer 5-stelligen sedezimalen Zahl, der Adresse (siehe Bild 1.9). Daran an schließt sich das Opcode-Byte (= Befehlsbyte) mit, je nach Befehl, 0 bis 2 Operandenbytes. Diese Bytes sind in der Reihenfolge, wie sie im Speicher stehen. Dabei kann gleich auf eine Besonderheit der Prozessoren der 6500-Familie hingewiesen werden:

Das niederwertige Byte einer Adresse im Operandenfeld steht vor dem höherwertigen Byte!

Beispiel: Die Adresse $4AB0 steht als Bytepaar $B0 $4A (in dieser Reihenfolge) im Speicher (siehe Bild 1.9).

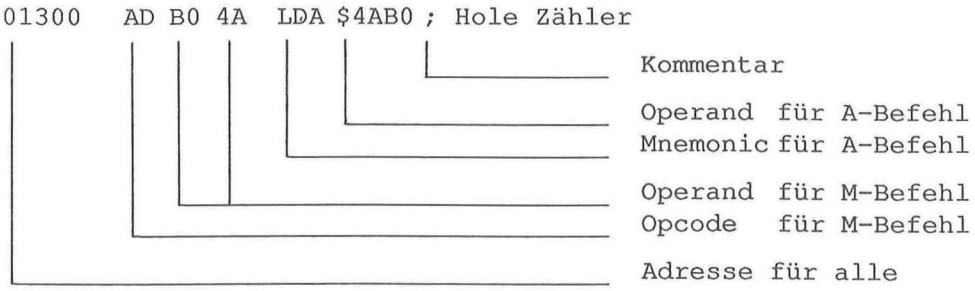


Bild 1.9 Format einer Programmzeile

Nach der Byteform des Befehls und dem Operanden folgt der Befehl nochmals, doch diesmal als Mnemonic, darauf wieder der Operand. Adreß-Operanden müssen hier korrekt eingegeben werden. Der Rest der Zeile ist dann, wenn vorhanden, der Kommentar.

Sie können die Programmzeilen auf zwei Arten eingeben:

- Mit dem *M-Befehl:*

  Tippen Sie folgendes ein: `M 01300` `<RETURN>`

  Wenn der Speicherbereich angezeigt wurde, können Sie mit den Cursorsteuertasten nach oben gehen und die Bytes mit den Werten, die unter »Opcode und Operand für den M-Befehl« stehen, überschreiben. Übernommen werden die Werte nach Drücken der Taste RETURN.

- Mit dem *A-Befehl:*

  Tippen Sie bitte ein: `A 01300 LDA $4AB0` `<RETURN>`

Die Zeile wird vom Monitor assembliert und in den Speicher geschrieben. Diese Form der Eingabe, so finde ich, ist für unsere Zwecke besser geeignet. Die nächste Zeile kann im Anschluß daran eingegeben werden.

Der Kommentar, den ich nur für Sie zur Erklärung angebe, darf nicht eingetippt werden, da Sie sonst eine Monitorfehlermeldung erhalten. Sie müssen die Zeile dann noch einmal, diesmal aber ohne Kommentar, eintippen.

Doch bevor Sie anfangen, ein neues Programm einzugeben, löschen Sie *immer* den Speicherbereich mit folgender Befehlsfolge:

```
F   01300   013FF   00  <RETURN>
```

Der Speicherbereich zwischen $1300 und $13FF in der Bank 0 wird mit $00 gefüllt. Damit haben Sie einen definierten Anfangszustand des Speichers für das Programm hergestellt.

Mit dem Befehl

```
D   01300                <RETURN>
```

werden die ersten 19 Zeilen Ihres Programmes disassembliert, also wieder in die Form zurückverwandelt, wie es hier im Buch steht - natürlich ohne Kommentare. Sie können damit auf einfache Art und Weise Tippfehler feststellen. Ich hoffe, daß der Druckfehlerteufel mit den Beispielprogrammen in den folgenden Kapiteln gnädig umgeht und sie ohne »Fehl und Tadel« läßt.

Tippen Sie bitte alle Beispiele an, und experimentieren Sie ungehemmt mit den Programmen. Das Schlimmste, was passieren kann, ist, daß der Rechner sich »aufhängt« und auf keinen Tastendruck mehr reagiert. Das kann geschehen, wenn z. B. Speicherinhalte in der Zeropage geändert werden; in einem solchen Fall hilft nur mehr der Druck auf die Reset-Taste. Sichere »Testgebiete« finden Sie in der Bank 0 im Bereich von $1300 bis $1BFF und in der Zeropage an den Adressen $FA bis $FF.

1.6 Die Adressierungsarten

Computer werden hauptsächlich zur Verarbeitung von Daten, zur Durchführung von Berechnungen oder zum Steuern von Anlagen eingesetzt.

Diese Vielzahl an möglichen Verwendungen bedingt, daß ein Mikroprozessor so beschaffen sein muß, daß er universell eingesetzt werden kann. Um diesen Forderungen gerecht werden zu können, gibt es zwei denkbare Lösungen:

- Man implementiert einen umfangreichen Befehlssatz und nimmt den Nachteil der Unüberschaubarkeit in Kauf oder

- man implementiert einen überschaubaren Befehlssatz, den man mit leistungsfähigen Adressierungen ausstattet.

Diesen zweiten Weg haben die Entwickler der 6500-Familie eingeschlagen. Die CPU 8502 hat einen mit 56 Befehlen kleinen Befehlssatz, der dafür aber mit 13 Adressierungsarten sehr mächtig ist. Die Art und Weise, wie ein Befehl ein Datum bearbeitet, ist bei gleichem Befehl immer gleich, doch die Quelle oder das Ziel des Datums ist entsprechend der gewählten Adressierungsart unterschiedlich. Die 13 Adressierungsarten für die CPU 8502 und die 3 weiteren, die die CPU 65C02 besitzt, sind:

- Implizite Adressierung
- Akkumulator-Adressierung
- Unmittelbare Adressierung
- Absolute Adressierung
- Zero-Page-Adressierung
- Relative Adressierung
- Indirekt-absolute Adressierung
- Absolut-X-indizierte Adressierung
- Absolut-Y-indizierte Adressierung
- Zeropage-X-indizierte Adressierung
- Zeropage-Y-indizierte Adressierung
- X-indiziert-indirekte Adressierung
- Indirekt-Y-indizierte Adressierung

Folgende Adressierungsarten stehen nur bei der CPU 65C02 zur Verfügung, die im C128 nicht eingebaut ist:

- Indirekt-X-indizierte Adressierung
- Relative-Test-Bit-Adressierung
- Set/Reset-Bit Adressierung

1.6.1 Implizite Adressierung

Kennzeichnung: `keine`
Beispiel:
```
DEX                 ; erniedrige X-Register um eins
```

Bei den Einbytebefehlen, die mit den Prozessorregistern arbeiten, liegt die implizite (Englisch: Implied) Adressierung vor. Implizit bedeutet soviel wie inbegriffen, mit einbezogen, d. h., der Operand ist durch den Befehl selbst festgelegt. Es gibt zwei Gruppen von Befehlen, die implizite Adressierung benutzen:

- Befehle, die nur an den Registern der CPU arbeiten. Es sind dies die Befehle CLC, CLD, CLI, CLV, DEX, DEY, INX, INY, NOP, SEC, SED, SEI, TAX, TSX, TXA, TXS und TYA. Sie benötigen zwei Taktzyklen zur Abarbeitung.

- Befehle, die auf den Speicher zugreifen, brauchen drei Taktzyklen. Es sind die Befehle BRK, PHA, PLA, PLP, RTI, RTS und die 65C02-spezifischen Befehle PHX, PHY, PLX und PLY.

1.6.2 Akkumulator-Adressierung

```
Kennzeichnung: A oder keine Kennzeichnung
Beispiel:      ASL A        ; schiebe den Akkuinhalt arithme-
                              tisch nach links
```

Die Akkumulator-Adressierung (Englisch: Accumulator) gehört zur impliziten Adressierungsart. Es sind deshalb Einbytebefehle, die mit dem Prozessorakkumulator arbeiten. Es gibt 4 Befehle mit dieser Adressierungsart: ASL, LSR, ROL und ROR. Die CPU 65C02 stellt zwei weitere Akkumulatorbefehle - DEC und INC - zur Verfügung.

Beim Assemblieren mit dem Monitor darf man keine Kennzeichnung verwenden.

```
Beispiel:      A 01300 ASL
```

1.6.3 Unmittelbare Adressierung

```
Kennzeichnung: #OPERAND
Beispiel:      LDA #$03     ; lade den Akku mit 3
```

Bei der unmittelbaren Adressierung (Englisch: Immediate) folgt auf den Opcode eine Konstante. Der Operand ist 1 Byte groß, da die CPU nur einen 8 Bit breiten Datenbus besitzt. Die Konstante kann in ein Register geladen werden oder geht als Operand in arithmetische oder logische Operationen ein.

Die Befehle ADC, AND, CMP, CPX, CPY, EOR, LDA, LDX, ORA und SBC verwenden diese Adressierungsart. Für die 65C02-CPU kommt der Befehl BIT hinzu.

1.6.4 Absolute Adressierung

```
Kennzeichnung: OPERAND
Beispiel:      LDA $1301    ; lade Akku mit dem Inhalt der
                              Speicherzelle $1301
```

Mit der absoluten Adressierung (Englisch: absolute) kann der gesamte natürliche Adreßraum der CPU überstrichen werden. Der Operand, der eine Adresse repräsentiert, besteht aus zwei Byte.

Diese Adressierungsart steht für folgende Befehle zur Verfügung: ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, EOR, INC, JMP, JSR, LDA, LDX, LDY, LSR, ORA, ROL, ROR, SBC, STA, STX und STY und, bei der CPU 65C02, STZ, TRB und TSB. Diese Adressierung gibt es für die meisten Befehle. Achtung: Das niederwertige Byte des Operanden steht an erster Stelle.

1.6.5 Zeropage-Adressierung

```
Kennzeichnung: OPERAND
Beispiel:      LDA $13      ; lade Akku mit dem Inhalt der
                              Speicherzelle $13
```

Die Zeropage-Adressierung ist eine Spielart der absoluten Adressierung. Der Operand besteht nur aus einem Byte. Da aber eine Adresse aus 2 Byte bestehen muß, generiert die CPU automatisch das höherwertige Byte, das immer $00 ist. Da die Befehle mit dieser Adressierungsart um 1 Byte kürzer sind als Befehle mit normaler absoluter Adressierung und sie zusätzlich noch sehr schnell sind (3 Taktzyklen), werden sie häufig eingesetzt.

Einsetzbar in der Zeropage-Adressierung sind die Befehle ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, EOR, INC, LDA, LDX, LDY, LSR, ORA, ROL, ROR, SBC, STA, STX und STY. Wieder bietet die CPU 65C02 weitere Befehle an: STZ, TRB und TSB.

1.6.6 Relative Adressierung

```
Kennzeichnung: OPERAND
Beispiel:      BCC $02      ; wenn das Carry-Flag gesetzt ist,
                              überspringe den folgenden 2-Byte-
                              Befehl
```

Die relative Adressierung erfordert ein Byte, das den Offset der angezielten Adresse vom aktuellen PC angibt. In einem Byte kann maximal ein Offset von $2^8 = 256$ Byte angegeben werden. Da ein Sprung aber in beiden Richtungen im Speicher möglich sein soll, bleibt demnach nur mehr eine Sprungweite von 128 Byte nach rückwärts und 127 Byte nach vorn übrig. Der Offset für einen Sprung rückwärts wird im 2er-Komplement angegeben. Darüber in den Kapiteln über die Sprungbefehle und Bits & Bytes, Teil 2, mehr. Sie brauchen diese Offsets weder bei der Eingabe mit dem Monitor, noch beim Arbeiten mit dem TOP-ASS zu berechnen, wenn Sie anstelle des Offsets die Adresse des Sprungzieles angeben. Die Umrechnung erledigen die beiden Programme für Sie.

Die Branchbefehle fragen die Flags des Prozessorstatusregisters ab. Je nach Befehl und Status des jeweiligen Flags wird zur angegebenen Adresse verzweigt oder nicht.

Die acht Branch-Befehle (Englisch: Verzweigung) sind die Befehle BCC, BCS, BEQ, BNE, BMI, BPL, BVC und BVS. Die CPU 65C02 bietet noch den Befehl BRA.

1.6.7 Indirekt-absolute Adressierung

```
Kennzeichnung:  (OPERAND)
Beispiel:       JMP ($3AB0) ; Springe zur Adresse, die in den
                              Speicherzellen $3AB1 und $3AB0
                              steht.
```

Die indirekt-absolute Adressierung ist in der 6500-Assemblersprache nur in einem Befehl, dem Sprungbefehl JMP, realisiert.

```
Beispiel:   Inhalt in $3AB0:   $D2
            Inhalt in $3AB1:   $FF
            Befehl:            JMP (3AB0)
```

Die Funktion ist einfach zu verstehen: Der Programmzähler wird mit der Adresse, deren niederwertiges Byte in $3AB0 und deren höherwertiges Byte in $3AB1 steht = $FFD2, geladen. Das bedeutet, daß ein Sprung zur Adresse $FFD2 durchgeführt wird. Mit dieser Adressierungsart besitzt man eine Möglichkeit, Sprungziele in einem Programm abhängig von Bedingungen zu setzen. Diese Adressierungsart ist beim Prozessor 65C02 für weitere Befehle realisiert worden: ADC, AND, CMP, EOR, LDA, ORA und STA.

1.6.8 Absolut-X-indizierte Adressierung

```
Kennzeichnung:  OPERAND,X
Beispiel:       LDA $3AB0,X ; Lade den Akku mit dem Inhalt der
                              Speicherzelle $3AB0 + X
```

Mit Einführung der absolut-indizierten Adressierungsart wird auch deutlich, warum die beiden Register X und Y Index-Register heißen. Der Inhalt des X-Registers stellt einen Index dar, der zum Operanden addiert wird. Diese Summe bildet dann erst die endgültige Adresse für den Befehl. Da es eine absolute Adressierungsart ist, belegt der Operand 2 Byte, es ist also zusammen mit dem Befehlsbyte ein 3-Byte-Befehl.

```
Beispiel:   Inhalt X-Register: $05
            Befehl:            LDA  $3AB0,X
```

Für die absolut-X-indizierte Adressierung stehen folgende Befehle zur Verfügung: ADC, AND, ASL, CMP, DEC, EOR, INC, LDA, LDY, ORA, ROL, ROR, SBC und STA. Zusätzlich gibt es bei der CPU 65C02 noch BIT und STZ.

Aus unverständlichen Gründen kann der Befehl STY mit dieser Adressierung nicht eingesetzt werden.

1.6.9 Absolut-Y-indizierte Adressierung

Kennzeichnung: `OPERAND,Y`
Beispiel:
```
LDA $3AB0,Y   ; Lade den Akku mit dem Inhalt der
                Speicherzelle $3AB0 + Y
```

Es gilt das gleiche wie bei der schon oben angesprochenen X-Register-absolut-indizierten Adressierung, nur daß hier der Inhalt des Y-Registers den Index darstellt.

Beispiel:
```
Inhalt Y-Register: $05
Befehl:            CMP $3AB0,Y
```

Vergleiche den Inhalt des Akkus mit dem Inhalt der Speicherzelle $3AB5.

Für die absolut-Y-indizierte Adressierung stehen folgende Befehle zur Verfügung: ADC, AND, CMP, EOR, LDA, LDX, ORA, SBC und STA.

Ähnlich STY bei absolut-X-indizierter Adressierung, kann STX nicht mit der absolut-Y-indizierten Adressierung verwendet werden, obwohl dies durchaus eine sinnvolle Kombination darstellen würde.

1.6.10 Zeropage-X-indizierte Adressierung

Kennzeichnung: `OPERAND,X`
Beispiel:
```
LDA $3A,X     ; Lade den Akku mit dem Inhalt der
                Speicherzelle $3A + X
```

Die Zeropage-indizierte Adressierung greift nur auf Speicherzellen in der Zeropage zu! Befehle mit dieser Adressierungsart sind 2-Byte-Befehle mit allen Vorteilen an Platzbedarf und Geschwindigkeit der Zeropage-Adressierung.

Beispiel:
```
Inhalt X-Register: $05
Befehl:            DEC $3A,X
```

Es wird der Inhalt der Speicherzelle $3F um 1 erniedrigt.

An Befehlen können für die 6500-CPUs ADC, AND, ASL, CMP, DEC, EOR, INC, LDA, LDY, ORA, ROL, ROR, SBC, STA und STY und für die CPU 65C02 noch BIT und STZ eingesetzt werden.

Frage: Wenn im X-Register $F0 steht, auf welche Speicherzelle wird mit dem Befehl LDA $3A,X zugegriffen?

Da $F0 + $3A = $012A ist, aber diese Adressierungsart nur den Zugriff auf die Zeropage gestattet, wird das höherwertige Adreßbyte $01 nicht berücksichtigt. Der Akku wird mit dem Inhalt der Speicherzelle $2A geladen. Dieses Verhalten der CPU muß bei allen Zeropage-indizierten Adressierungen berücksichtigt werden!

1.6.11 Zeropage-Y-indizierte Adressierung

Kennzeichnung: `OPERAND,Y`
Beispiel:
```
LDX $3A,Y   ; Lade das X-Register mit dem In-
              halt der Speicherzelle $3A + Y
```

Alles für die Zeropage-X-indizierte Adressierung Gesagte, trifft auch auf die Zeropage-Y-indizierte Adressierung zu.

Es gibt aber nur zwei Befehle, die diese Adressierung unterstützen: LDX und STX.

1.6.12 X-indiziert-indirekte Adressierung

Kennzeichnung: `(OPERAND,X)`
Beispiel:
```
STA ($3A,X); Speichere den Akku in die
             Speicherzelle, deren Adresse in
             den Zeropageadressen $3A + X und
             $3B + X steht.
```

Die indiziert-indirekte Adressierung kann man als Kombination der beiden Adressierungsarten indiziert und indirekt betrachten. In der Speicherzelle Operand + X steht das niederwertige, in der darauffolgenden das höherwertige Byte einer Adresse. Der Prozessor greift zur Abarbeitung des Befehls auf diese Zieladresse zu. Mit dieser Adressierungsart können Routinen, deren Adressen in Tabellen in der Zeropage stehen, aufgerufen werden. Die einzelne Routine wird durch den Index im X-Indexregister ausgewählt. Es sind 2-Byte-Befehle, wie die anderen Befehle mit Zeropage-Adressangaben.

Beispiel:
```
Inhalt X-Register: $04
Inhalt $3A:        ohne Bedeutung
Inhalt $3E:        $B0
Inhalt $3F:        $3A
Befehl:            ADC ($3A,X)
```

Es wird der Inhalt der Speicherzelle $3AB0 zum Akkuinhalt addiert. Es ist eine vorindizierte Adressierung, da die Speicherstelle, die die indirekte Adresse enthält, erst mit dem X-Register indiziert werden muß. Die Befehle ADC, AND, CMP, EOR, LDA, ORA, SBC und STA können mit dieser Adressierung eingesetzt werden.

1.6.13 Indirekt-Y-indizierte Adressierung

Kennzeichnung: `(OPERAND),Y`
Beispiel:

```
STA ($3A),Y    ; Hole die Adresse, die in den
                 Zeropageadressen $3A und $3B
                 steht. Erhöhe diese Adresse um
                 den Inhalt des Y-Registers und
                 speichere den Akkuinhalt in
                 die Speicherzelle mit dieser
                 Adresse
```

Die indirekt-indizierte Adressierung holt aus der Zeropage den Zeiger auf die Startadresse des gewünschten Speicherbereiches, erhöht ihn um den Index und führt den Befehl dann auf die jetzt adressierte Speicherzelle durch. Es wird also eine Nachindizierung durchgeführt.

Die Befehle mit dieser Adressierungsart gestatten den Zugriff auf Tabellen, deren indirekte Startadresse in der Zeropage steht. Auf das gewünschte Tabellenelement kann mit dem Index im Y-Register zugegriffen werden. Auch dies sind 2-Byte-Befehle.

Beispiel:

```
Inhalt Y-Register:  $04
Inhalt $3A:         $B0
Inhalt $3B:         $3A
Befehl:             STA ($3A),Y
```

Es wird der Inhalt des Akkus in der Speicherzelle $3AB4 gespeichert.

Die Befehle ADC, CMP, LDA, ORA und STA besitzen diese Adressierungsart.

1.6.14 Indirekt-X-indizierte Adressierung

Kennzeichnung: `(OPERAND),X`
Beispiel:

```
JMP ($3A),X; Hole die Adresse, die in den
             Zeropageadressen $3A und $3B
             steht. Erhöhe diese Adresse um
             den Inhalt des X-Registers und
             springe sie an.
```

Analog gilt alles für die indirekt-Y-indizierte Adressierung auch hier. Diese Adressierungsart steht aber nur für die CPU 65C02 zur Verfügung. Sie können sie in Programmen, die für die CPU 65C02 geschrieben sind, leider nur für einen Befehl einsetzen: Den JMP-Befehl.

1.6.15 Relative-Test-Bit-Adressierung

```
Kennzeichnung:  keine
Beispiel:       BBR 1,$3a,$03 ; Überspringe den folgenden 3-
                                Byte-Befehl, wenn Bit 1 im
                                Inhalt der Speicherzelle $3A
                                gelöscht ist.
```

Auch diese Branchbefehle stehen nur für die CPU 65C02 zur Verfügung.

Der Test erfolgt immer auf ein Bit eines Bytes in der Zeropage. Es sind 3-Byte-Befehle. Das erste Byte ist der Befehl, das zweite gibt die Adresse der zu testenden Zelle in der Zeropage an und das dritte Byte stellt den Sprungoffset zur Verfügung. Die Sprungweite hat den gleichen Bereich wie bei der relativen Adressierung.

Folgende Befehle setzen diese Adressierungsart ein: BBR und BBS.

1.6.16 Set/Reset-Bit Adressierung

```
Kennzeichnung:  keine
Beispiel:       RMB 7,$3a     ; Lösche Bit 7 in der Speicherzelle
                                $3A.
```

Diese Befehle stehen nur für die CPU 65C02 zur Verfügung.

Mit dieser Adressierung kann ein Bit eines Bytes in der Zeropage gesetzt oder gelöscht werden. Es sind 2-Byte-Befehle. Das erste Byte ist der Befehl, das zweite gibt die Adresse des zu bearbeitenden Bytes in der Zeropage an.

Es sind die beiden Befehle RMB und SMB, die diese Adressierungsart einsetzen.

1.7 Laden und Speichern

Mit die wichtigsten Befehle eines Prozessors sind seine Transportbefehle. Nachdem im C128 eine 8-Bit-CPU steckt, kann mit einem Befehl immer nur ein Byte transportiert werden. Dieses Byte kann aus einer Speicherzelle gelesen und zur CPU in eines der Register, den Akkumulator, das X- oder Y-Register, gebracht werden. Selbstverständlich kann ein Byte auch aus einem der Register in eine Speicherzelle geschrieben werden. Da alle Operationen durch die CPU ausgeführt werden, ist die Namensgebung dieser beiden Befehle einleuchtend:

Mnemonic	Funktion	Bemerkung
LDA	A <—Daten	Lade den Akkumulator
LDX	X <—Daten	Lade das Indexregister X
LDY	Y <—Daten	Lade das Indexregister Y
STA	Daten <—A	Speichere den Akkumulator
STX	Daten <—X	Speichere das Indexregister X
STY	Daten <—Y	Speichere das Indexregister Y

Bild 1.10 Mnemonics der Transportbefehle

Speichern (Englisch: STORE) eines Bytes in den Speicher und Laden (Englisch: LOAD) eines Bytes aus dem Speicher.

Da STORE und LOAD zu lang sind (ein Mnemonic der 6500-Assemblersprache ist 3 Zeichen lang), hat man sich auf die Abkürzungen LD für LOAD und ST für STORE geeinigt. Damit erkennbar ist, welches Register angesprochen ist, hängt man an diese Abkürzungen noch die Registerbezeichnung A, X oder Y an. In Bild 1.10 sind alle Mnemonics der Transportbefehle mit ihren Funktionen aufgelistet. Der Pfeil weist immer von der Quelle (Englisch: Source) auf das Ziel (Englisch: Target oder Destination).

So, jetzt geht es endlich los!

Befehl: LDA								Funktion: A <— Daten						
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y	
Hex			A9	AD	A5		*B2	BD	B9	B5		A1	B1	
Bytes			2	3	2		2	3	3	2		2	2	
Takte			2	4	3		5	*4	*4	4		6	*5	
Flags	N	V	B	D	I	Z	C	Lade den Akkumulator mit neuen Daten. Der alte Inhalt wird überschrieben.						
	X					X								

Befehl: LDX								Funktion: X <— Daten						
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y	
Hex			A2	AE	A6				BE		B6			
Bytes			2	3	2				3		2			
Takte			2	4	3				*4		4			
Flags	N	V	B	D	I	Z	C	Lade das Indexregister mit neuen Daten. Der alte Inhalt wird überschrieben.						
	X					X								

Befehl: LDY								Funktion: Y <— Daten						
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y	
Hex			A0	AC	A4			BC		B4				
Bytes			2	3	2			3		4				
Takte			2	4	3			*4		2				
Flags	N	V	B	D	I	Z	C	Lade das Indexregister mit neuen Daten. Der alte Inhalt wird überschrieben.						
	X					X								

Bild 1.11 Die LOAD-Befehle

Starten Sie bitte den Monitor, löschen Sie den Speicherbereich von $1300 bis $13FF (siehe Kapitel 1.5.2). Tippen Sie das kleine Testprogramm ein

```
01300 AD 01 13 LDA $1301   ;  Lade den Akku mit dem Wert der
                              Speicherzelle $1301 = $01
01303 8D 07 13 STA $1307   ;  und speichere den Akku in $1307
01306 00 BRK               ;  Beende das Programm durch Ab-
                              bruch
```

und starten Sie es durch Eingabe von

```
G 01300 <RETURN>
```

Bevor wir uns den Speicher ansehen, um festzustellen, was passiert ist, wollen wir im Trockenkurs probieren, das Programm zu analysieren:

- In der ersten Zeile wird der Akku mit dem Wert $01, dem Inhalt der Zelle $1301, geladen.
- Der Akkuinhalt wird in die Zelle $1307 geschrieben, die nun den Wert $01 enthalten müßte.
- Der Befehl BRK beendet das Programm.

Sehen wir uns also jetzt mit dem Befehl `M 01300 01307` den Speicher an. Die Zeile wird so aussehen:

```
>01300 AD 01 13 8D 07 13 00 01 00 00 00 00 00 00 00 00:
```

Der invers dargestellte Bereich der Zeile nach dem Doppelpunkt ist für uns im Moment noch nicht wichtig. In ihm werden die Bytes als Zeichenkodes (siehe Kapitel 1.2.3) interpretiert.

Sie sehen, in der Speicherzelle $1307 steht tatsächlich $01. Sollten Sie diesem Ergebnis nicht ganz trauen, so löschen Sie noch einmal den Speicherbereich, kontrollieren ihn mit dem M-Befehl, tippen das Programm neu ein, kontrollieren wieder mit dem M- oder mit dem D-Befehl und starten das Programm. Überzeugt? Wenn ja, probieren Sie das Programm mit den Befehlen für die Indexregister aus. Bitte achten Sie darauf, daß Sie bei all ihren Versuchen das Programm mit dem Befehl BRK oder $00 abschließen. Es ist ein spezieller Prozessorbefehl, der zu einem Programmabbruch führt, doch darüber später mehr.

Die oben geschilderte Vorgehensweise bei der Eingabe, dem Überprüfen und Modifizieren der Programme, sollten Sie bei den folgenden Beispielen ebenfalls anwenden. Ich werde nicht mehr gesondert darauf hinweisen.

Das war Ihr erstes, wenn auch noch einfaches Programm in 6502-Assembler! Es ist doch nicht schwierig – oder?

Befehl: STA				Funktion: M <— (A)									
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex				8D	85		*92	9D	99	95		81	91
Bytes				3	2		2	3	3	2		2	2
Takte				4	3		5	5	5	4		6	6

Flags	N	V	B	D	I	Z	C	Speichere den Akkuinhalt, der erhalten bleibt, die Speicherzelle wird überschrieben.

Befehl: STX				Funktion: M <— (X)									
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex				8E	86						96		
Bytes				3	2						2		
Takte				4	3						4		

Flags	N	V	B	D	I	Z	C	Speichere den X-Inhalt, der erhalten bleibt, die Speicherzelle wird überschrieben.

Befehl: STY				Funktion: M <— (Y)									
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex				8C	84					94			
Bytes				3	2					2			
Takte				4	3					4			

Flags	N	V	B	D	I	Z	C	Speichere den X-Inhalt, der erhalten bleibt, die Speicherzelle wird überschrieben.

Bild 1.12 Die STORE-Befehle

1.8 Erklärung der Befehlstabellen

Um Ihnen weitere Informationen über die verschiedenen Befehle zu geben, greife ich zur tabellarischen Darstellung. In den Tabellen finden Sie folgende Informationen:

- Die *mnemonische Form* des Befehls (Mnemonic)
- Die *Funktion* des Befehls. Der Pfeil zeigt auf das Ziel.
  Folgende Abkürzungen werden eingesetzt:
  A: Akkumulator
  X: X-Register
  Y: Y-Register
  M: Speicher (Englisch: Memory)
  M6: Bit 6 des Inhalts der Speicherzelle
  M7: Bit 7 des Inhalts der Speicherzelle
  S: Stackpointerregister
  PC: Programmzähler Lo- und Hi-Byte
  ADR: Adresse
  P: Prozessorstatusregister
  C: Carryflag im Prozessorstatusregister
  I: Interruptenableflag im Prozessorstatusregister
  D: Dezimalflag im Prozessorstatusregister
  V: Overflowflag im Prozessorstatusregister
  Die Klammer weist auf den Inhalt der Quelle bzw. des Ziels hin.
- *Adress.*: Die Adressierungsarten, die zur Verfügung stehen, wobei sie abgekürzt wiedergegeben sind. Die Reihenfolge entspricht der Reihenfolge, wie sie im Kapitel 1.6 besprochen worden sind.
- In *HEX* steht der sedezimale Code des Befehles. Ist die Spalte unter einer der Adressierungsarten mit 2 Byte ausgefüllt, so ist diese Adressierungsart für diesen Befehl vorhanden. Mit einem Stern markierte Opcodes stehen nur für die CPU 65C02 zur Verfügung.
- *Bytes* gibt die Anzahl der Bytes an, die der Befehl benötigt.
- *Takte* gibt die Maschinenzyklen an, die der Befehl benötigt. Ist die Zahl mit einem Asterisk (einem Stern: *) gekennzeichnet, so ist ein zusätzlicher Taktzyklus nötig, wenn eine Seite (Page) überschritten wird. Ist die Zahl mit einem # markiert, so ist ein zusätzlicher Zyklus bei einer Verzweigung und weitere zwei bei Pageüberschreitung nötig.
- *Flags:* Die Namen der Flags sind abgekürzt. Befindet sich unter einer Flagbezeichnung ein Eintrag, so beeinflußt dieser Befehl dieses Flag. Die Einträge bedeuten:

X: Flag wird vom Ergebnis der Operation geändert.
0: Das Flag wird auf Null gesetzt.
1: Das Flag wird auf 1 gesetzt.
6: Bit 6 des getesteten Bytes wird hineinkopiert.
7: Bit 7 des getesteten Bytes wird hineinkopiert.
**:* Bit wird gesetzt, bevor das Statusregister auf den Stapel geschoben wird.

Mit diesen Informationen dürften Sie keine Probleme mehr haben, die Befehlstabellen gewinnbringend einzusetzen.

Befehl: PHP					Funktion: Stack <— (P), S <— (S)-1								
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex Bytes Takte	08 1 3												

Flags	N	V	B	D	I	Z	C	Der Statusregisterinhalt wird auf den Stack gebracht, der Stackzeiger gesetzt. Der Registerinhalt bleibt.

Befehl: PLP					Funktion: P <— (Stack), S <— (S)+1								
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex Bytes Takte	28 1 4												

Flags	N	V	B	D	I	Z	C	Der Statusregisterinhalt wird mit dem Inhalt der Stackspitze geladen, der Stackzeiger neu gesetzt.
	X	X	X	X	X	X	X	

Bild 1.13 Die Statusregister-Befehle PHP und PLP

1.9 Das Prozessorstatusregister und die Flags

Wie im Kapitel 1.3.6 versprochen, werden wir jetzt etwas genauer auf dieses Register, seine 7 Flags und die Befehle zur Manipulation des Registers eingehen.

Bitte ziehen Sie das Bild 1.5 »Die 8502-Register« beim Durcharbeiten der nächsten Seiten zu Hilfe. Beginnen wir mit den Befehlen, die jeweils das gesamte Register betreffen:

1.9.1 Die Statusregisterbefehle PHP und PLP

Es gibt zwei Befehle, mit denen das gesamte Prozessorstatusregister angesprochen werden kann. Der erste ist der Befehl PHP (Englisch: Push Processor Status = schiebe den Prozessorstatus). Mit seiner Hilfe kann das gesamte Register auf den Stack gebracht werden. Eingesetzt wird dieser Befehl um den Prozessorstatus nach einer Operation, z. B. einem Vergleich, zu retten. Es können dann andere Operationen, die die Flags in diesem Register ändern würden, durchgeführt werden. Anschließend wird mit dem Befehl PLP (Englisch: Pull Processor Status = hole den Prozessorstatus) der vorherige Stand mit dem Vergleichsergebnis zur weiteren Verarbeitung wieder hergestellt. Der Stack dient hier als Sicherungsspeicher. Beide Befehle führen das Stackpointer-Register automatisch nach.

Im Bild 1.13 finden Sie die Ihnen mittlerweile schon bekannte Befehlstabelle mit allen wichtigen Informationen über beide Befehle.

1.9.2 Das Carry-Flag, CLC und SEC

Der Name Carry-Flag kommt aus dem Englischen und bedeutet soviel wie Übertragsanzeige. Das Carryflag zeigt einen Übertrag an, der bei einer Addition entstand, oder ein Borgen bei einer Subtraktion. Neben den beiden arithmetischen Operationen beeinflussen Schiebeoperationen (Englisch: Shift) und Vergleiche das Carryflag. Bei Schiebeoperationen wird direkt ein Bit aus dem Akkumulator in das Carryflag geschoben. Das Flag kann durch 2 Befehle (siehe das Kapitel über die Sprungbefehle) abgefragt werden.

Mit dem Befehl SEC (Englisch: Set Carry) kann das Flag gesetzt werden. Ein Bit setzen bedeutet, ihm den Wert 1 zuzuweisen. Mit dem Befehl CLC (Englisch: Clear Carry) kann es gelöscht werden, also auf den Wert Null gesetzt werden. Die Befehlstabelle finden Sie in Bild 1.14.

Folgende Befehle ändern das Übertragsflag: ADC, ASL, CMP, CPX, CPY, LSR, PLP, ROL, ROR, RTI, SBC und natürlich SEC und CLC.

Befehl: CLC				Funktion: C <— 0									
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex	18												
Bytes	1												
Takte	2												

Flags	N	V	B	D	I	Z	C	Das Übertragsflag wird auf 0 gesetzt.
							0	

Befehl: SEC				Funktion: C <— 1									
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex	38												
Bytes	1												
Takte	2												

Flags	N	V	B	D	I	Z	C	Das Übertragsflag wird auf 1 gesetzt.
							1	

Bild 1.14 Die Carryflag-Befehle CLC und SEC

1.9.3 Das Zero-Flag

Mit gesetztem Bit im Prozessorstatusregister signalisiert die CPU, daß das Ergebnis einer Operation Null ist oder ein Byte mit dem Wert $00 in eines der Register geladen wurde. Die Vergleichsbefehle führen intern eine Subtraktion durch, natürlich ohne die beiden zu vergleichenden Werte zu verändern. Das Zeroflag wird bei Gleichheit, denn dann ist das Ergebnis dieser internen Subtraktion gleich Null, ebenfalls gesetzt.

Es gibt keine direkten Befehle, dieses Flag zu setzen oder zu löschen. Mit einem kleinen Trick ist es aber möglich, auch wenn man dabei einen Registerinhalt (A, X oder Y) opfern muß:

Setzen: `LDA #$00 oder LDX #$00 oder LDY #$00`

Löschen: `LDA #$01 etc.`

Beeinflußt wird das Zeroflag durch die Befehle ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, DEX, DEY, EOR, INC, INX, INY, LDA, LDX, LDY, LSR, ORA, PLA, PLP, ROL, ROR, RTI, SBC, TAX, TSX, TXA und TYA. Folgende Befehle der CPU 65C02 ändern ebenfalls das Flag: PLX, PLY, TRB, TSB.

Die STORE-Befehle (STA, STX, STY) beeinflussen es also nicht, was Sie sich gut merken sollten.

1.9.4 Das Interrupt-Enable-Flag, CLI und SEI

Ist dieses Bit gesetzt, werden Unterbrechungsaufforderungen der Hardware (unter dem Begriff Hardware werden die Elektronik und die Mechanik eines Rechners zusammengefaßt) über die IRQ-Leitung (siehe Kapitel 1.3.9 »Der Steuerbus und die Interruptlogik«) ignoriert. Dieses Flag wird vom Prozessor automatisch bei einem Reset, z. B. nach dem Einschalten des Rechners, oder beim Abarbeiten einer früheren Interruptaufforderung gesetzt. Von Programmen aus kann das Bit mit dem Befehl SEI (Englisch: Set Interruptenable Flag) gesetzt und mit CLI (Englisch: Clear Interrupt Enable Flag) wieder gelöscht werden. Bild 1.15 beschreibt diese beiden Befehle. Folgende Befehle greifen auf das IRQ-Flag zu: BRK, CLI, PLP, RTI und SEI.

Das Interrupt-Enable-Flag kann nicht über BRANCH-Befehle abgefragt werden.

Befehl: CLI					Funktion: I <— 0								
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex Bytes Takte	58 1 2												

Flags	N	V	B	D	I	Z	C	Das Interruptflag wird gelöscht. Interrupts sind möglich.
					0			

Befehl: SEI					Funktion: I <— 1								
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex Bytes Takte	78 1 2												

Flags	N	V	B	D	I	Z	C	Das Interruptflag wird gesetzt. Interrupts sind unmöglich.
					1			

Bild 1.15 Die IRQ-Befehle CLI und SEI

1.9.5 Das Dezimal-Flag, CLD und SED

Dies ist das einzige Flag, das der Prozessor nicht selbst setzen kann; der Programmierer ist verantwortlich für den Zustand dieses Flags. Es dient dazu, dem Prozessor mitzuteilen, in welchem Zahlensystem er rechnen soll: Ist das Flag gelöscht, werden alle Rechnungen binär durchgeführt, ist es gesetzt, rechnet die CPU im Dezimalsystem mit BCD-Zahlen. BCD steht für Binary Coded Decimal, übersetzt: binär kodierte Dezimalzahl. Diese Zahlenkodierung werden wir im Kapitel »Bits & Bytes Teil 2« noch genauer kennenlernen.

Wichtig: Wenn Sie im Dezimal-Modus rechnen wollen, verbieten Sie mit SEI (siehe oben) Interrupts. Ansonsten springt die CPU in die Interruptroutine und berechnet dort alle Adressen nicht mehr binär sondern dezimal. Sie können sich sicherlich vorstellen, daß die Ergebnisse dann weit von den beabsichtigten entfernt liegen. Wundern Sie sich also bitte nicht, wenn der C128 sich verabschiedet. Üblicherweise

ist der erste Befehl, den die Prozessoren der 6500-Familie nach dem Einschalten abarbeiten, der Befehl CLD.

Die Befehlstabelle finden Sie in Bild 1.16.

Insgesamt sind es 4 Befehle, die dieses Flag verändern: CLD, PLP, RTI und SED

Befehl: CLD				Funktion: D <— 0									
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex Bytes Takte	D8 1 2												

Flags	N	V	B	D	I	Z	C	Das Dezimalflag wird auf Null gesetzt. ADC und SDC Befehle werden binär durchgeführt.
				0				

Befehl: SED				Funktion: D <— 1									
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex Bytes Takte	F8 1 2												

Flags	N	V	B	D	I	Z	C	Das Dezimalflag wird auf 1 gesetzt. ADC und SDC Befehle werden dezimal durchgeführt.
				1				

Bild 1.16 Die Dezimalflag-Befehle CLD und SED

1.9.6 Das Break-Flag

Der Prozessor setzt dieses Flag automatisch, wenn er auf einen Softwareinterrupt (Software sind alle Programme eines Rechners) durch den Befehl BRK (siehe Kapitel »Interrupts«) stößt. Mit Hilfe dieses Flags kann in der Interruptroutine zwischen einem Hardware- und einem Softwareinterrupt unterschieden werden. Es gibt keine direkten Befehle, um dieses Flag zu ändern. Mit folgendem Trick geht es aber:

```
PHP             ; Prozessorstatusregister auf Stack
                  bringen
PLA             ; Prozessorstatusregister vom Stack
                  in Akku
AND %11101111   ; BRK-Bit löschen
PHA             ; wieder auf Stack zurückbringen und
PLP             ; in das Prozessorstatusregister
                  bringen
```

Das Beispielprogramm ist eigentlich unsinnig, da es keinen Sinn macht, dieses Flag zu verändern. Es wird immer bei einem Software-Interrupt gesetzt und durch den Befehl RTI wieder gelöscht. Es soll nur als kleines Beispiel dienen. Haben Sie den Trick noch nicht auf Anhieb verstanden, so macht es nichts. Wie ein Bit gelöscht oder gesetzt wird, werden Sie später noch kennenlernen.

1.9.7 Das Überlaufflag und CLV

Das Überlaufflag (Englisch: Overflow Flag) dient zur Anzeige eines Überlaufs oder Unterlaufs, die bei einer Addition oder Subtraktion einer vorzeichenbehafteten Dualzahl aufgetreten sind. Genauso zeigt es an, daß wahrscheinlich das Ergebnis einer Operation mit Zahlen im Zweierkomplement falsch ist. Im Kapitel »Bits & Bytes, Teil 2« werden wir solche Rechnungen durchführen.

Auch der Befehl BIT ändert das Flag: er schreibt das Bit 6 des getesteten Bytes hinein.

In einem Programm kann das Overflowflag mit dem Befehl CLV (Englisch: Clear Overflow Flag) gelöscht werden. Das Flag kann bei den 6500-Prozessoren (nicht beim 8502) über die Steuerleitung SO (siehe Kapitel 1.3.13) von der Hardware gesetzt werden. Für die Befehlsbeschreibung siehe Bild 1.17. Beeinflußt wird das Flag durch die Befehle ADC, BIT, CLV, PLP, RTI und SBC.

Befehl: CLV				Funktion: V <— 0									
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex	B8												
Bytes	1												
Takte	2												

Flags	N	V	B	D	I	Z	C	Das Überlaufflag wird gelöscht.
		0						

Bild 1.17 Der Überlaufflag-Befehl CLV

1.9.8 Das Negativ-Flag

Das letzte noch nicht besprochene Flag heißt zwar Negativ-Flag (Englisch: Negative Flag); es ist aber bestimmt nicht negativ, daß es vorhanden ist. In dieses Flag wird das höchstwertige Bit eines Bytes hineinkopiert. Damit gibt das N-Flag das Vorzeichen einer vorzeichenbehafteten Dualzahl wieder: Ist das Negativ-Bit gesetzt, d. h., das Bit 7 des Bytes ist Eins, so ist die Dualzahl negativ. Durch diese Eigenschaft des N-Flags ist es sehr einfach, das Bit 7 eines Bytes in einem der Register zu prüfen. Auch hier gibt es keine direkten Befehle, um dieses Flag zu setzen oder zu löschen. Da aber die meisten Befehle zum Datentransport und zur Datenverarbeitung dieses Flag beeinflussen, ist es leicht, über eine der Register-Ladeanweisungen das Flag zu verändern:

```
Setzen:    LDA #$80
Löschen:   LDA #$00
```

Folgende Befehle der 6500-Familie wirken sich auf dieses Flag aus: ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, DEX, DEY, EOR, INC, INX, INY, LDA, LDX, LDY, LSR, ORA, PLA, PLP, ROL, ROL, ROR, RTI, SBC, TAX, TAY, TSX, TXA und TYA, dazu kommen noch die Befehle PLX, PLY, TRB und TSB bei Programmierung der CPU 65C02.

1.10 Die Register-Register-Befehle

Häufig muß ein Datum vom Akku in eines der Indexregister oder in umgekehrter Richtung geschoben werden. Sei es, um das Datum kurzzeitig zu retten, oder auch, um Akku und Indexregister auf einen identischen Wert zu initialisieren. Leider gibt es bei der 6500-Famile nur Befehle, die den Datentransport mit dem Akkumulator als Quelle oder als Ziel gestatten. Datentransport zwischen den beiden Indexregistern ist nur über den Akku oder einer Speicherzelle als Zwischenspeicher möglich.

Befehl: TAX					Funktion: X <— (A)								
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex	AA												
Bytes	1												
Takte	2												

Flags	N	V	B	I	D	Z	C	Kopiere Akkuinhalt zum X-Register. Akkuinhalt bleibt unverändert.
	X					X		

Befehl: TXA					Funktion: A <— (X)								
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex	8A												
Bytes	1												
Takte	2												

Flags	N	V	B	I	D	Z	C	Kopiere X-Registerinhalt in Akku. Der Registerinhalt bleibt unverändert.
	X					X		

Bild 1.18 Die Registerbefehle TAX und TXA

1.10.1 Die Akku-X-Register-Befehle TAX und TXA

Transfer Accu into X lautet die englische Beschreibung des Befehls TAX, übersetzt: Übertrage den Akkuinhalt in das X-Register. Dementsprechend heißt die englische Bezeichnung des Befehls TXA: Transfer X into Accumulator.

Der Inhalt des Ziels des Datentransports wird überschrieben, der Inhalt der Quelle bleibt unverändert. Bild 1.18 zeigt alle nötigen Informationen zu diesen beiden Befehlen.

1.10.2 Die Akku-Y-Register-Befehle TAY und TYA

Diese beiden Befehle entsprechen den Befehlen TAX und TYA in allen Einzelheiten, nur ist nicht mehr das Indexregister X das Ziel bzw. die Quelle des Datums, sondern das Y-Register.

In Bild 1.19 finden Sie alle nötigen Informationen zu diesen beiden Befehlen.

Befehl:	TAY				Funktion:	Y <— (A)							
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex	A8												
Bytes	1												
Takte	2												

Flags	N	V	B	I	D	Z	C	Kopiere Akkuinhalt zum Y-Register. Akkuinhalt bleibt unverändert.
	X					X		

Befehl:	TYA				Funktion:	A <— (Y)							
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex	98												
Bytes	1												
Takte	2												

Flags	N	V	B	I	D	Z	C	Kopiere Y-Registerinhalt in Akku. Der Registerinhalt bleibt unverändert.
	X					X		

Bild 1.19 Die Registerbefehle TAY und TYA

1.11 Inkrementieren und Dekrementieren

Inkrement (Lateinisch: Zuwachs einer Größe) und Dekrement (Lateinisch: Abnahme einer Größe) - zwei Begriffe - doch wofür stehen sie in der 6500-Assemblersprache?

Diese Frage wollen wir nun beantworten:

Wenn mit Hilfe der indizierten Adressierungsart auf eine Reihe von Werten, die hintereinander im Speicher stehen, zugegriffen werden soll, benötigt man einen Zähler, der den Index stellt. Wir wissen mittlerweile, daß die beiden Indexregister für diese Aufgabe besonders gut geeignet sind, ja speziell dafür vorhanden sind. Um nun den Index in einem dieser beiden Register auf einfache Weise um 1 erhöhen (= inkrementieren) oder erniedrigen (= dekrementieren) zu können, stellt uns die CPU verschiedene Befehle zur Verfügung:

1.11.1 Die Befehle INX, DEX, INY und DEY

Im Kapitel 1.4.4 haben wir eine kleine Problemlösung ausgearbeitet. Das beschriebene Problem eignet sich besonders gut, den Dekrementbefehl zu illustrieren. Bitte betrachten Sie noch einmal Bild 1.7, damit Sie den Ablauf unseres Programmbeispieles besser verstehen. Aus dem gleichen Grund sollten Sie die Bilder 1.20 und 1.21 genau studieren. Achten Sie bei den beiden Bildern auf die Auswirkungen, die die Befehle auf das Statusregister haben. Doch nun zu unserem Beispiel:

Befehl: INC					Funktion: M <— (M)+1									
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y	
Hex		*1A		EE	E6			FE		F6				
Bytes		1		3	2			3		2				
Takte		2		6	5			7		6				
Flags	N	V	B	I	D	Z	C	Erhöhe den Inhalt der Speicherzelle um eins.						
	X					X								

Befehl: INX					Funktion: X <— (X)+1									
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y	
Hex	E8													
Bytes	1													
Takte	2													
Flags	N	V	B	I	D	Z	C	Erhöhe den Inhalt des Registers um 1. Das Register kann als Zähler eingesetzt werden.						
	X					X								

Befehl: INY					Funktion: Y <— (Y)+1									
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y	
Hex	C8													
Bytes	1													
Takte	2													
Flags	N	V	B	I	D	Z	C	Erhöhe den Inhalt des Registers um 1. Das Register kann als Zähler eingesetzt werden.						
	X					X								

Bild 1.20 Die Inkrementierbefehle INX, INY und INC

```
01300  A9  00     LDA #$00     ;  Lade den Akku mit dem Lösch-
                                   byte
01302  A2  40     LDX #$40     ;  Lade das X-Register mit dem
                                   Index 64 (für 64 Speicher-
                                   zellen)
01304  9D  FF 13  STA $13FF,X  ;  Lösche das über den Index
                                   adressierte Byte im Spei-
                                   cher. Es sind die Adressen
                                   $1400-$143F
01307  CA         DEX          ;  erniedrige den Indexzähler
                                   um 1
01308  D0  FA     BNE $1304    ;  Ist der Index ungleich Null?
                                   Wenn ja, lösche nächste
                                   Speicherzelle
0130A  00         BRK          ;  Beende das Programm
```

Sollten Sie sich nicht mehr sicher sein, wie dieses Programm einzugeben ist, lesen Sie bitte Kapitel 1.8 nach. Bevor Sie das Programm starten, geben Sie bitte den Monitorbefehl

```
F 01400 014FF FF          <RETURN>
```

ein. Es werden alle Speicherzellen in dem angegebenen Adreßbereich mit $FF überschrieben. Sie können sonst nach Programmende mit dem M-Befehl nicht kontrollieren, ob auch wirklich 64 Byte gelöscht worden sind.

Einen Befehl, BNE, haben wir noch nicht besprochen. Er und eine Reihe weiterer Sprung- und Branchbefehle werden im Kapitel 1.13 besprochen. Für den Augenblick belassen wir es bei diesem Hinweis und setzen diesen Befehl einfach ein, ohne uns weitergehende Gedanken zu machen.

Wichtiger ist hier der Einsatz des X-Registers als Index und das Dekrementieren des Registers, um eine Reihe aufeinanderfolgender Speicherzellen ansprechen zu können. Wissen Sie noch, wie die hier benutzte Adressierung heißt?

Es ist die Absolut-X-indizierte Adressierung (siehe Kapitel 1.6.8).

Experimentieren Sie ruhig wieder etwas. Benützen Sie z. B. Y als Register oder setzen Sie eine andere Adressierungsart ein. Geben Sie Ihre neuen Programme ein und prüfen Sie, ob die Ergebnisse stimmen. Sie werden erkennen, daß selbst ein so einfaches Problem auf verschiedenen Wegen gelöst werden kann. Betrachten Sie Ihre Lösungen. Welche benötigt am wenigsten Speicherplatz, welche die wenigsten Taktzyklen? Mit diesen Untersuchungen werden Sie später in der Lage sein, die für ein Problem geeignetste Lösung zu finden und sie zu programmieren.

1.11.2 Die Befehle INC und DEC

Die Möglichkeit, Register zu inkrementieren oder zu dekrementieren ist eine feine Sache. Doch genauso häufig muß ein Byte in einer Speicherzelle um eins erhöht oder erniedrigt werden. Für diese Aufgaben stehen die Befehle INC (Englisch: Increment Memory) und DEC (Englisch: Decrement Memory) zur Verfügung. Es wird der Inhalt der adressierten Speicherzelle um 1 weiter- oder heruntergezählt. Mit diesen Befehlen können Zeiger (Englisch: Pointer) in der Zeropage geführt werden. Unser C128 besitzt eine Routine mit dem Namen CHRGET, die den Inkrementbefehl INC benützt. Bitte disassemblieren Sie den Bereich von $00380 bis $0038F mit dem Monitorbefehl

```
D 00380 0038C   <RETURN>
```

Sie werden auf Ihrem Bildschirm folgendes sehen (natürlich ohne die Kommentare):

```
00380  E6 3D      INC $3D       ; Erhöhe das Lo-Byte des Zeigers
                                  in den BASIC-Text um eins
00382  D0 02      BNE $0386     ; ist das Lo-Byte des Zeigers
00384  E6 3E      INC $3E         übergelaufen? Wenn ja, erhöhe
                                  auch das Hi-Byte des Zeigers
                                  um eins
00386  8D 01 FF   STA $FF01     ; schalte BASIC-Text-Bank ein
00389  A0 00      LDY #$00      ; Index = 0, damit das adres-
                                  sierte Byte in den Akku gela-
                                  den werden kann
0038B  B1 3D      LDA ($3D),Y ; Lade Zeichen aus dem BASIC-
                                  Text
0038D  8D 03 FF   STA $FF03     ; Blende die ROMs wieder ein
```

Die Funktionsweise der Routine sollte, bis auf die beiden STA-Befehle, die Bankumschaltungen durchführen, klar sein. Mit dieser Routine holt sich der C128 ein Zeichen aus dem BASIC-Programm. Der Name CHRGET steht für »Get a Character« (Englisch: Hole ein Zeichen).

Befehl: DEC								Funktion: M <— (M)-1						
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y	
Hex		*3A		CE	C6			DE		D6				
Bytes		1		3	2			3		2				
Takte		2		6	5			7		6				
Flags	N	V	B	I	D	Z	C	Erniedrige den Inhalt der Speicherzelle um eins.						
	X					X								

Befehl: DEX								Funktion: X <— (X)-1						
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y	
Hex	CA													
Bytes	1													
Takte	2													
Flags	N	V	B	I	D	Z	C	Erniedrige den Inhalt des Registers um 1. Das Register kann als Zähler eingesetzt werden.						
	X					X								

Befehl: DEY								Funktion: Y <— (Y)-1						
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y	
Hex	88													
Bytes	1													
Takte	2													
Flags	N	V	B	I	D	Z	C	Erniedrige den Inhalt des Registers um 1. Das Register kann als Zähler eingesetzt werden.						
	X					X								

Bild 1.21 Die Dekrementierbefehle DEX, DEY und DEC

1.12 Bits & Bytes, Teil 2: Binäres Rechnen

In diesem Kapitel wollen wir untersuchen, wie im Dualsystem gerechnet werden kann und wie sich logische Operationen auf die Bits einer Binärzahl auswirken. Rechnen im Sedezimalsystem ist, bedingt durch die Basis 16, nicht ganz so eingängig wie im Dualsystem. Erleichtern kann man es sich durch Umrechnung der sedezimalen Zahlen in Dualzahlen, auf die dann die Rechenoperation angewendet werden.

Die Algorithmen für die verschiedenen Rechenoperationen entsprechen exakt denen des Dezimal-Systems.

1.12.1 Die binäre Addition

Zum Verständnis ein kleines Beispiel im Dezimalsystem:

$$19 + 4 = 23$$

oder

$$\begin{array}{r} 19 \\ +\ 4 \\ \hline 1 \quad \text{<— Übertrag} \\ \hline 23 \end{array}$$

oder

$$\begin{array}{rcll} 9 + 4 & = & 3 & \text{und Übertrag 10} \\ 10 + \text{Übertrag} & = & 20 & \\ \hline & & 23 & \end{array}$$

Algorithmus 4: Die Addition

> Die Zahlen, die addiert werden sollen, werden untereinander geschrieben und spaltenweise von rechts nach links, unter Berücksichtigung etwaiger Überträge, zusammengezählt.

Im weiteren erarbeiten Sie sich bitte die entsprechenden Rechenregeln zur Vertiefung an einem Beispiel aus dem Dezimalsystem selbst.

Beispiel:

binär	dezimal	sedezimal	
0100 1111	79	4F	
+ 0011 1001	+ 57	+ 39	
1111 111	11	1	<— Übertrag
1000 1000	136	88	

Entsprechend dem Algorithmus 4 werden die Dualzahlen von rechts nach links spaltenweise addiert. Dabei entstehende Überträge werden bei der Addition der nächsten Spalte mit berücksichtigt. Den Übertrag haben wir mittlerweile schon unter dem Namen Carry kennengelernt. *Das Carryflag muß vor einer Addition gelöscht werden.* Tritt bei der Addition der beiden höchstwertigen Bits ein Übertrag auf, so nimmt das Carryflag diesen Übertrag auf. In diesem Fall ist das Carryflag gesetzt, ansonsten ist es gelöscht.

1.12.2 Die binäre Subtraktion

Bei der Subtraktion kann ein Borgen auftreten. Dieses Borgen muß bei der Berechnung der Differenz berücksichtigt werden.

Beispiel:

binär	dezimal	sedezimal	
0100 1100	76	4C	
- 0011 1001	- 57	- 39	
11 11	1		<— Borgen
0001 0011	19	13	

Algorithmus 5: Die Subtraktion

> Die Zahlen, die subtrahiert werden sollen, werden untereinander geschrieben und spaltenweise von rechts nach links unter Berücksichtigung etwaigen Borgens subtrahiert.

Das Carryflag muß vor der Subtraktion gesetzt werden, damit nach der Subtraktion erkannt werden kann, ob ein Borgen über die höchste Stelle hinaus erfolgte. In diesem Fall ist das Carryflag gelöscht.

1.12.3 Vorzeichenbehaftete Dualzahlen

In den Beispielen haben wir bis jetzt immer mit positiven Zahlen gearbeitet. Welche Möglichkeiten bestehen, eine Binärzahl als negativ zu kennzeichnen? Man hat sich darauf geeinigt, das höchstwertige Bit einer Zahl als Vorzeichenbit zu verwenden, wobei das gelöschte ein positives und das gesetzte Bit ein negatives Vorzeichen darstellt.

Solche Dualzahlen werden als vorzeichenbehaftete Binärzahlen (Englisch: Signed Binary) bezeichnet. Verwendet man 8-Bit, so ergibt sich ein Wertebereich von -128 bis +127. Genügt dieser Bereich nicht, so müssen eben mehr Bits zur Verfügung stehen, z. B. ein Word, d. h. 16 Bit. Es ergibt sich dann ein Bereich von -32768 bis +32767. Diesen Bereich kennen wir von den Integerzahlen in BASIC (das sind die Zahlenvariablen mit dem Kennzeichen %, z. B. AB%). Führen wir einmal eine Addition der Zahlen 19 und -16 als vorzeichenbehaftete Binärzahlen durch:

Beispiel:

+ 19	als signed-binary	:	0001 1001	
- 16	als signed-binary	:	1001 0110	
+ 3			1010 1111	= - 47

Dieses Ergebnis ist mit Sicherheit falsch. Es scheint also nicht ganz so einfach zu sein, mit diesen Zahlen richtig zu rechnen. Der Aufwand, der zur Korrektur der Ergebnisse in den Programmen betrieben werden muß, steht in keinem Verhältnis zum erzielten Gewinn. Auf der anderen Seite ist es eine bestechend elegante Lösung, statt mit Subtraktionen nur mehr mit Additionen arbeiten zu können.

Die Darstellung der positiven Zahlen ist in Ordnung, doch die der negativen Zahlen muß noch geändert werden, um auf das richtige Ergebnis zu kommen.

Die Umwandlung geschieht in zwei Schritten:

1.12.4 Das Einer-Komplement

Die negative Darstellung einer Zahl erhält man, indem man zu jedem Bit der positiven Zahl sein Komplement (Lateinisch: Ergänzung) niederschreibt. Jede 0 der positiven Zahl wird eine 1 und jede 1 eine 0.

Beispiel:

+ 19	als positive Binärzahl	:	0001 1001
- 19	als Einer-Komplement	:	1110 0110

Wie Sie sehen, wird die Forderung eingehalten, daß das höchstwertige Bit einer positiven Zahl gelöscht und einer negativen Zahl gesetzt sein muß. Versuchen wir noch einmal unsere Beispieladdition von oben:

```
+ 19                          :   0001 1001
- 16   als Einer-Komplement   :   1110 1001
-------------------------------------------
+  3                          (1)0000 0010    = + 2 mit Übertrag
```

Bis jetzt stimmt das Ergebnis immer noch nicht. Versuchen wir jetzt den zweiten Schritt unserer Zahlenumwandlung.

1.12.5 Das Zweier-Komplement

Die positiven Zahlen werden weiterhin als »Signed-Binary« wiedergegeben, die negativen als Einer-Komplement-Zahl, zu der noch eine Eins addiert wurde:

Beispiel:

```
+ 19   als positive Binärzahl   :   0001 1001
- 19   als Einer-Komplement     :   1110 0110
                                  + 0000 0001
                                  -----------
       Zweier-Komplement        :   1110 0111
```

Versuchen wir noch einmal unsere Testaddition:

```
- 16   als Einer-Komplement     :   1110 1001
                                  + 0000 0001
                                  -----------
       Zweier-Komplement        :   1110 1010

+ 19                            :   0001 1001
- 16   als Zweier-Komplement    :   1110 1010
-------------------------------------------
+  3                          (1)0000 0011    = + 3 mit Übertrag
```

Wenn wir den Übertrag ignorieren, stimmt diesmal unser Ergebnis. Versuchen Sie das Rechnen im Zweier-Komplement noch mit weiteren Zahlen. Überprüfen Sie dabei die Ergebnisse.

Sie werden feststellen, daß Sie fast immer richtige Ergebnisse mit richtigen Vorzeichen erhalten, wenn etwa entstehende Überträge unberücksichtigt bleiben. Doch hin und wieder stimmt es doch nicht? Betrachten wir dazu ein Beispiel:

```
+  65    0100 0001
+  64    0100 0000
-------------------
+ 129    1000 0001    =  - 127
```

positiv	2er Komplement	negativ	2er Komplement
		- 128	1000 0000
+ 127	0111 1111	- 127	1000 0001
+ 126	0111 1110	- 126	1000 0010
+ 125	0111 1101	- 125	1000 0011
+ 124	0111 1100	- 124	1000 0100
...		...	
+ 16	0001 0000	- 16	1111 0000
+ 15	0000 1111	- 15	1111 0001
+ 14	0000 1110	- 14	1111 0010
+ 13	0000 1101	- 13	1111 0011
+ 12	0000 1100	- 12	1111 0100
+ 11	0000 1011	- 11	1111 0101
+ 10	0000 1010	- 10	1111 0110
+ 9	0000 1001	- 9	1111 0111
+ 8	0000 1000	- 8	1111 1000
+ 7	0000 0111	- 7	1111 1001
+ 6	0000 0110	- 6	1111 1010
+ 5	0000 0101	- 5	1111 1011
+ 4	0000 0100	- 4	1111 1100
+ 3	0000 0011	- 3	1111 1101
+ 2	0000 0010	- 2	1111 1110
+ 1	0000 0001	- 1	1111 1111
0	0000 0000		

Bild 1.22 Zweier-Komplement-Zahlen im Bereich -128 bis +127

Hier tritt ein Überlauf von Bit 6 nach Bit 7 ein. Immer, wenn ein Überlauf von Bit 6 nach Bit 7 erzeugt wird und dabei das Vorzeichenbit geändert wird, ist das Ergebnis falsch. In folgenden Situationen entsteht ein Überlauf:

- Addition von großen positiven Zahlen
- Addition von betragsmäßig großen negativen Zahlen
- Subtraktion einer betragsmäßig großen negativen von einer großen positiven Zahl
- Subtraktion einer großen positiven von einer betragsmäßig großen negativen Zahl

Die Prozessoren der 6500-Familie besitzen im Prozessorstatusregister ein Überlaufflag. Es wird immer dann gesetzt, wenn das Ergebnis einer Addition oder Sub-

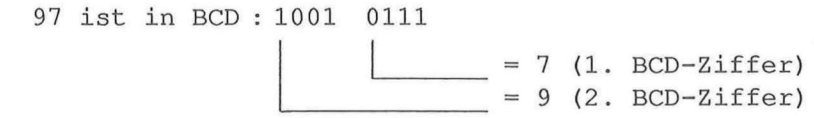
traktion mit Zweier-Komplement-Zahlen durch eine Über- oder Unterschreitung des maximal zulässigen Zahlenbereiches nicht korrekt ist.

1.12.6 BCD-Zahlen

BCD, die Abkürzung für »Binär kodierte Dezimalziffern« (Englisch: Binary Coded Decimal = BCD) ist eine spezielle Form, wie Dezimalzahlen gespeichert werden.

Wie wir wissen, kann der 8502-Prozessor bei gesetztem Dezimalflag im Prozessorstatusregister BCD-Zahlen verarbeiten. Um alle Ziffern von 0 bis 9 binär kodieren zu können, benötigt man 4 Bit. Es wäre natürlich eine riesige Platzverschwendung zur Darstellung einer Dezimalstelle jeweils ein Byte zu verwenden, da ja in einem Byte 2 Dezimalstellen untergebracht werden können. Man spricht dann von einer gepackten BCD-Kodierung.

Beispiel:



Bei Betrachtung des Bildes 1.23 fällt auf, daß 6 Bitkombinationen nicht erlaubt sind. Bei binären Additionen wird man Probleme bekommen, da die unerlaubten Bitkombinationen auftreten können.

In allen Fällen, wenn das Ergebnis größer als 9 ist, muß ein Übertrag in der nächsten BCD-Stelle erscheinen. Erreichen kann man das durch eine zusätzliche Addition von %0110 (= 6). Damit werden die verbotenen Kombinationen korrigiert und ein Übertrag in die nächste Stelle erzwungen. Doch Vorsicht, die 6 darf nur dann addiert werden, wenn das Ergebnis über 9 liegt. Bei unserem Prozessor muß das nicht mehr berücksichtigt werden, da er bei gesetztem Dezimalflag automatisch die Überträge korrigiert. Beim Experimentieren beachten Sie bitte den Hinweis in Kapitel 1.9.5.

Ziffer	BCD Kode
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	verboten
1011	verboten
1100	verboten
1101	verboten
1110	verboten
1111	verboten

Bild 1.23 4-Bit BCD-Kode

1.12.7 Die logischen Operationen

Die Schaltalgebra, mit der die Funktion eines digitalen Rechners (Englisch: Digit = Anzeigeneinheit bei Rechnern, entspricht hier einem Bit) beschrieben werden kann, ist ein Bereich der Booleschen Algebra. Die Ergebnisse logischer Operationen können nur entweder *wahr* (Englisch: True) oder *falsch* (Englisch: False) sein. Diese beiden möglichen Zustände lassen sich den binären Werten 1 und 0 zuordnen.

Man bezeichnet alle Verbindungen von Aussagen (Lateinisch: Junktionen) durch spezielle Kürzel (Junktoren) und definiert ihre Bedeutung durch eine Tabelle aller möglichen Kombinationen der Eingangswerte. Diese Tabellen werden Wahrheitstafeln genannt und legen die Ergebnisse der jeweiligen Verknüpfung eindeutig fest.

Die Prozessoren der 6500-Familie können neben den arithmetischen Befehlen auch logische Operationen durchführen. Im folgenden werden wir uns mit den drei logischen Grundverknüpfungen UND, ODER und NICHT und der daraus abgeleiteten Funktion EXOR, Exklusiv-ODER, befassen.

Logisches Zeichen	Zeichen der Schaltalgebra	Name	Bezeichnung
¬	Strich über dem Operanden	Negation	NICHT NOT
∧	•	Konjunktion	UND, AND
V	+	Disjunktion	ODER, OR
<=>	≡	Äquivalenz	Inklusiv-ODER
⊻	≢	Antivalenz	Exklusiv-ODER EXOR

Bild 1.24 Einige Verknüpfungen und ihre Symbole

1.12.7.1 Die UND-Verknüpfung

Die UND-Verknüpfung (Englisch: AND) ergibt nur dann das Ergebnis Wahr (= 1), wenn alle zu verknüpfenden Eingangswerte ebenfalls wahr sind.

Eingangs-Wert A	Eingangs-Wert B	Ergebnis UND
1	1	1
1	0	0
0	1	0
0	0	0

Bild 1.25 Wahrheitstafel der UND-Verknüpfung für 1-Bit-Werte

Beispiel:

	0111	7
UND	1001	9
	0001	1

Die UND-Verknüpfung wird beim Programmieren zum gezielten Löschen von Bits in einem Byte eingesetzt. In der Bit-Maske, die zur Verknüpfung herangezogen wird, müssen alle Bits, die gelöscht werden sollen, den Wert 0 haben, alle anderen 1.

Beispiel: Es soll das Bit 6 in einem Byte gelöscht werden:

Byte:		1110 0101
Maske:	UND	1011 1111
		1010 0101

Es wird tatsächlich nur das Bit 6 gelöscht, alle anderen bleiben unverändert. Dies kann analog bei den anderen Bits verwendet werden, z. B. bei einem BIT-Befehl.

1.12.7.2 Die ODER-Verknüpfung

Die ODER-Verknüpfung (Englisch: OR) ergibt immer dann das Ergebnis Wahr (= 1), wenn mindestens einer der zu verknüpfenden Eingangswerte ebenfalls wahr ist.

Eingangs-Wert A	Eingangs-Wert B	Ergebnis ODER
1	1	1
1	0	1
0	1	1
0	0	0

Bild 1.26 Wahrheitstafel der ODER-Verknüpfung für 1-Bit-Werte

Beispiel:

	0111	7
ODER	1001	9
	1111	15

Die ODER-Verknüpfung wird beim Programmieren zum gezielten Setzen von Bits in einem Byte eingesetzt. In der Bit-Maske, die zur Verknüpfung herangezogen wird, müssen alle Bits, die gesetzt werden sollen, den Wert 1 haben, alle anderen 0.

Beispiel: Es soll das Bit 6 in einem Byte gesetzt werden:

Byte:		1010 0101
Maske:	ODER	0100 0000
		1110 0101

Es wird tatsächlich Bit 6 gesetzt, alle anderen bleiben unverändert.

1.12.7.3 Die NICHT-Verknüpfung

Die Nicht-Funktion (Englisch: NOT) kehrt den Wahrheitswert einer Aussage um.

Eingangs-Wert	Ergebnis NOT
1	0
0	1

Bild 1.27 Wahrheitstafel der NICHT-Verknüpfung für 1-Bit-Werte

Diese Funktion ist eigentlich keine Verknüpfung, da sie sich nur auf eine Aussage bezieht, deren Wahrheitswert sie ins Gegenteil verkehrt. Zweimalige Anwendung der Negation auf eine Aussage (doppelte Verneinung) stellt wieder den ursprünglichen Wahrheitswert her.

Die NICHT-Verknüpfung wird beim Programmieren zur Invertierung einer Zahl benützt.

Beispiel: Die Zahl 0001 1001 soll invertiert werden:

Byte:	0001 1001
NICHT:	1110 0110

Es wird jedes gesetzte Bit gelöscht und jedes gelöschte wieder gesetzt. Erinnern Sie sich? Weiter oben hatten wir doch schon so etwas! Richtig, mit der logischen Operation NICHT können wir das Einer-Komplement einer Zahl erzeugen. Doch leider existiert im Befehlssatz unseres Prozessors der Befehl NOT nicht. Der nächste Abschnitt zeigt jedoch eine Ersatzlösung, die diesen Mangel beseitigt.

1.12.7.4 Die EXOR-Verknüpfung

Die Exklusiv-ODER-Verknüpfung ergibt immer dann das Ergebnis »wahr« (= 1), wenn beide zu verknüpfenden Eingangswerte unterschiedlich sind.

Eingangs-Wert A	Eingangs-Wert B	Ergebnis ODER
1	1	0
1	0	1
0	1	1
0	0	0

Bild 1.28 Wahrheitstafel der EXOR-Verknüpfung für 1-Bit-Werte

Beispiel:

```
        0111    7
EXOR    1001    9
-------------------
        1110   14
```

Die EXOR-Verknüpfung wird beim Programmieren zum Invertieren von Bits in einem Byte eingesetzt. In der Bit-Maske, die zur Verknüpfung herangezogen wird, müssen alle Bits gesetzt sein.

Beispiel: Es soll das Byte invertiert werden:

```
Byte:              1010 0101
Maske:    EXOR     1111 1111
-----------------------------
                   0101 1010
```

Mit der logischen Operation EXOR eines Bytes mit der Bit-Maske %11111111 können wir ebenfalls das Einer-Komplement einer Zahl erzeugen. Doch diesmal haben wir Glück, dieser Befehl ist im Befehlssatz vorhanden und heißt in der 6500-Assemblersprache EOR.

1.13 Die Sprung- und Branchbefehle

In den seltensten Fällen gelingt es, ein Problem so zu lösen, daß das resultierende Programm linear vom Start bis zum Ende, ohne Verzweigungen oder Schleifen, durchlaufen werden kann. Normalerweise benötigt man eine Reihe von Verzweigungen, Schleifen und Sprüngen. Selbst unser kleines Problem, daß wir in Kapitel 1.4.4 gestellt und in Kapitel 1.11.1 kodiert haben, benötigt eine Schleife, die durch einen bedingten Sprung, dem Befehl BNE, realisiert wird. Ohne Schleife wäre das Programm um ein Vielfaches länger, denn jede Speicherzelle müßte für sich selbst direkt durch den Befehl STA angesprochen werden. Die Prozessoren der 6500-Familie stellen drei Arten von Sprungbefehlen zur Verfügung:

- der bedingte Sprung (BRANCH-Befehle)
- der unbedingte Sprung (JMP) und
- der Subroutinenaufruf (JSR)

Den Subroutinenaufruf werden wir uns in einem späteren Kapitel zu Gemüte führen.

1.13.1 Der unbedingte Sprungbefehl JMP

JMP, die Abkürzung des englischen Wortes Jump = Sprung, zwingt den Prozessor beim Abarbeiten dieses Befehles, das Programm an einer anderen Stelle im Speicher fortzusetzen. Dabei wird die als Operand angegebene Adresse in das Programmzähler-Register geladen. Der nächste Befehl wird dann aus der Speicherzelle mit dieser neuen Adresse geholt. Da der JMP-Befehl 3 Byte lang ist, ist der Operand 16-Bit breit. Es sind also Sprünge im gesamten natürlichen Adreßraum der CPU möglich. Hier sparen wir uns ein Beispielprogramm, da die Wirkungsweise dieses Befehls eindeutig sein sollte.

Befehl:	JMP			Funktion:		PC <— ADR							
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex				4C			6C						*7C
Bytes				3			3						3
Takte				3			5						6

Flags	N	V	B	I	D	Z	C	Der Programmzähler wird mit der angegebenen Adresse geladen. Es wird ein Sprung dorthin durchgeführt.

Bild 1.29 Der unbedingte Sprungbefehl JMP

1.13.2 Die bedingten Sprünge

Die Bezeichnung dieser Sprünge verrät schon ihre besondere Eigenschaft: Sie werden nur ausgeführt, wenn eine Bedingung erfüllt ist. Im Englischen wird der bedingte Sprung als Branch (= Verzweigung) bezeichnet. Welche Bedingungen können wohl in einem binär rechnenden System abgefragt werden? Nun, z. B.:

- Ist ein Wert gleich Null?
- Ist ein Wert negativ?
- Ist das Ergebnis der 2er-Komplement Rechnung korrekt?

Denken Sie bei diesen Beispielen nicht auch sofort an das Prozessorstatusregister? Ja, seine Flags dienen als Bedingung für die bedingten Sprünge. Vier der Flags können für die bedingten Sprünge verwendet werden.

Da alle Branchbefehle 2-Byte-Befehle sind, wird das Sprungziel nur durch ein Byte definiert. Zur Adressierung des gesamten Adreßraumes reicht es natürlich nicht aus. Dieses Byte stellt daher einen Offset dar. Es können damit maximal 255 Byte übersprungen werden. Da die Sprünge aber nicht nur vorwärts, sondern auch rückwärts möglich sein sollen, ergibt sich ein endgültiger Bereich von 128 Byte rückwärts und 127 Byte vorwärts. Die Sprungweite wird also relativ zum Befehl angegeben. Dieses Displacement muß für die Rückwärtssprünge immer im 2er-Komplement angegeben werden. Benutzen Sie bei der Eingabe den Monitorbefehl A oder später TOP-ASS, geben Sie das Sprungziel als echte Adresse an. Beide Programme berechnen dann automatisch den notwendigen Offset. Geben Sie aber Ihre Programme als Bytefolgen mit dem Monitorbefehl M ein, müssen Sie tatsächlich den Offset selbst berechnen. Damit es etwas leichter für Sie wird, habe ich im Bild 1.30 die Branchoffsets für den gesamten möglichen Bereich zusammengestellt. Die oberste Zeile stellt das niederwertige, die 1. Spalte das höherwertige Nibble des Offsets dar. Sie wollen 21 Byte nach vorne springen. Suchen Sie die 21 in der 1. Tabelle auf, so finden Sie nach links das höherwertige Nibble 1 und nach oben das niederwertige Nibble 5: Der Offset ist also $15. Die Offsets $FE und $FF sind direkt auf den Branchbefehl oder den Offset gerichtet.

/	Offset für relative Vorwärtssprünge															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127

/	Offset für relative Rückwärtssprünge															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	128	127	126	125	124	123	122	121	120	119	118	117	116	115	114	113
9	112	111	110	109	108	107	106	105	104	103	102	101	100	99	98	97
A	96	95	94	93	92	91	90	89	88	87	86	85	84	83	82	81
B	80	79	78	77	76	75	74	73	72	71	70	69	68	67	66	65
C	64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49
D	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33
E	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17
F	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

Bild 1.30 Branch-Offsets

Sollen bedingte Sprünge über eine größere Distanz als erlaubt durchgeführt werden, kann man sich behelfen, indem die entgegengesetzte Bedingung als Branch benützt wird, um den JMP-Sprungbefehl auf die richtige Adresse zu überspringen. Klingt ziemlich kompliziert, nicht? Doch halb so wild, ein kleines Beispiel im Kapitel über die Befehle BEQ und BNE wird es verdeutlichen.

Doch beginnen wir nun die einzelnen Branchbefehle in der Reihenfolge, wie die Flags des Prozessorstatusregisters im Kapitel 1.9 behandelt wurden, zu besprechen.

Befehl: BCC						Funktion: Wenn C=0 ist, springe							
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex Bytes Takte						90 2 #2							

Flags	N	V	B	I	D	Z	C	Ist C=0, dann springe zur Adresse, deren Offset angegeben ist. Der Bereich ist von -128 bis +127 Byte.

Befehl: BCS						Funktion: Wenn C=1 ist, springe							
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex Bytes Takte						B0 2 #2							

Flags	N	V	B	I	D	Z	C	Ist C=1, dann springe zur Adresse, deren Offset angegeben ist. Der Bereich ist von -128 bis +127 Byte.

Bild 1.31 Die bedingten Sprungbefehle BCC und BCS

1.13.2.1 Die Branchbefehle BCC und BCS

Branch on Carry Clear und Branch on Carry Set, die beiden Branchbefehle, die das Übertragsflag als Bedingung verwenden.

Der Sprung wird beim Befehl BCC nur ausgeführt, wenn das Übertragsflag gelöscht ist, beim Befehl BCS nur dann, wenn es gesetzt ist. Es lassen sich also bedingte Sprünge immer nach Operationen, die das Carryflag beeinflussen, durchführen. Alle Befehle, die einen Einfluß auf dieses Flag haben, sind im Kapitel 1.9.2 aufgeführt.

Folgen BCC und BCS aufeinander, so wird der Befehl, der auf BCS folgt, nur dann abgearbeitet, wenn er selbst ein Sprungziel ist. Auf diese Weise ist es möglich, einen unbedingten Sprung durchzuführen. Worin liegt denn hier der Vorteil? Es werden doch 4 Byte anstelle von 3 Byte beim JMP verbraucht. Nun, alle relativen Sprünge beziehen sich auf eine Adresse als Ausgangspunkt. Egal, an welcher Adresse der Branchbefehl steht, wenn das Programm im Speicher verschoben wurde, der Sprung

führt immer zum gleichen Befehl, der als nächster, entsprechend der Programmlogik, abgearbeitet werden soll. Darin liegt der Vorteil. Programme, die nur relative Sprünge beeinhalten, sind adressen-unabhängig. Sie können in jedem beliebigen Adreßraum ablaufen.

1.13.2.2 Die Branchbefehle BEQ und BNE

Branch on Result equal to Zero, verzweige, falls das Ergebnis gleich Null ist und Branch on Result nöt equal to Zero, verzweige, wenn das Ergebnis ungleich Null ist.

Diese beiden Befehle fragen das Zeroflag im Prozessorstatusregister ab. Auch hier kann das Ergebniss aller Operationen, die das Zeroflag verändern, zur Steuerung der Programmlogik verwendet werden.

Hier das oben angekündigte Beispiel. Es ist kein ausgearbeitetes Programm, Sie können es eingeben, doch bitte starten Sie es nicht mit dem Befehl G.

```
01300 A5 FE     LDA $FE   ;  Hole das Flag eines Ereignisses
01302 F0 03     BEQ $1307 ;  Ist es Null gewesen?
                             Ja -> überspringe JMP
01304 4C 20 14  JMP $1420 ;  Wenn <> 0 dort weitermachen
01307 A2 00     LDX #$00  ;  Hier beginnt der Programmteil,
                             zur Behandlung des Falles Null
.....
.....
01420 A2 FF     LDX #$FF  ;  Hier beginnt der Programmteil,
                             zur Behandlung des Falles <> 0
```

Versuchen Sie bitte einmal, statt BEQ $1307 den Befehl BNE $1420 einzugeben. Es wäre ein Gewinn von 3 Byte, da der JMP-Befehl wegfällt. Doch leider, die maximale Sprungweite von 127 Byte ist überschritten, Sie bekommen bei der Eingabe eine Fehlermeldung.

Unser Programm in Kapitel 1.11.1 ist jetzt auch klar, mit BNE wird abgefragt, ob das Indexregister noch ungleich Null ist. Solange das der Fall ist, sind noch nicht alle 64 Byte gelöscht worden. Die Schleife ist dann noch nicht beendet.

Befehl: BEQ					Funktion: Wenn Z=1 ist, springe								
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex Bytes Takte						F0 2 #2							

Flags	N	V	B	I	D	Z	C	Ist Z=1, dann springe. Ergebnis ist 0. Der Sprungbereich reicht von -128 bis +127 Byte.

Befehl: BNE					Funktion: Wenn Z=0 ist, springe								
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex Bytes Takte						D0 2 #2							

Flags	N	V	B	I	D	Z	C	Ist Z==, dann springe. Ergebnis ist ungleich 0. Der Sprungbereich reicht von -128 bis +127 Byte.

Bild 1.32 Die bedingten Sprungbefehle BEQ und BNE

1.13.2.3 Die Branchbefehle BMI und BPL

Branch on Minus, verzweige, wenn negativ und Branch on Plus, verzweige, wenn positiv. Diese beiden Befehle können in zweifacher Hinsicht für bedingte Sprünge eingesetzt werden:

– Da das Bit 7 des gerade in ein Register geladenen Bytes in das Negativflag kopiert wird, besteht eine einfache Möglichkeit, dieses Bit abzufragen. Häufig legt man bei einem I/O-Port die Statusleitung an den Eingang 7, um ihren Zustand mit diesen beiden Befehlen schnell kontrollieren zu können.

Beispiel:

```
01300 AD 01 DD    LDA $DD01 ; Lade Datum, das am USERPORT
                              ansteht. Warte bis Bit 7
                              gesetzt ist.
01303 10 FB       BPL $1300 ; springe zurück, solange Bit 7
                              Null ist.
```

Auch dieses Beispiel ist kein vollständiges Programm. Sie könnten es z. B. zur Abfrage des Bit 7 des USERPORTs Ihres C128 benützen, wenn Sie ein Schalterinterface angeschlossen haben.

Befehl: BMI				Funktion: Wenn N=1 ist, springe										
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y	
Hex Bytes Takte						30 2 #2								
Flags	N	V	B	I	D	Z	C	Ist N=1 dann springe. Ergebnis ist negativ. Der Sprungbereich reicht von -128 bis +127 Byte.						

Befehl: BPL				Funktion: Wenn N=0 ist, springe										
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y	
Hex Bytes Takte						10 2 #2								
Flags	N	V	B	I	D	Z	C	Ist N=0 dann springe. Ergebnis ist positiv. Der Sprungbereich reicht von -128 bis +127 Byte.						

Bild 1.33 Die bedingten Sprungbefehle BMI und BPL

– Bei Addition und Subtraktion von 2er-Komplement-Zahlen können Sie auf diese Weise leicht feststellen, ob das Ergebnis positiv oder negativ ausgefallen ist. Doch dazu erst bei den Arithmetikbefehlen ein Beispiel.

1.13.2.4 Die bedingten Sprungbefehle BVC und BVS

Branch on Overflow Clear, verzweige, wenn kein Überlauf eingetreten ist und Branch on Overflow Set, verzweige, wenn ein Überlauf eingetreten ist. Diese beiden Befehle dienen bei der CPU 8502 nur zur Kontrolle der Ergebnisse von Rechenoperationen mit 2er-Komplement-Zahlen. Ist das Überlaufflag gesetzt, muß in eine Fehlerbehandlungsroutine, bzw. in eine Routine, die das Ergebnis korrigiert, gesprungen werden. Bei der CPU 6502 kann das Überlaufflag durch die Hardware über die Steuerleitung SO gesetzt werden. Dann können diese Befehle zur Abfrage eines Hardwareereignisses außerhalb der CPU dienen.

Befehl: BVC					Funktion: Wenn V=0 ist, springe									
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y	
Hex Bytes Takte						50 2 #2								
Flags	N	V	B	I	D	Z	C	Ist V=0, dann springe. Kein Überlauf eingetreten. Der Sprungbereich reicht von -128 bis +127 Byte.						

Befehl: BVS					Funktion: Wenn V=1 ist, springe									
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y	
Hex Bytes Takte						70 2 #2								
Flags	N	V	B	I	D	Z	C	Ist V=1, dann springe. Ein Überlauf ist eingetreten. Der Sprungbereich reicht von -128 bis +127 Byte.						

Bild 1.34 Die bedingten Sprungbefehle BVC und BVS

1.14 Vergleichen

Zu den häufigsten Aufgaben der Datenverarbeitung gehört das Vergleichen von Daten. Die Kenntnis, ob ein Datum kleiner, gleich oder größer als ein anderes ist, ist keineswegs eine triviale Angelegenheit. So sollen z. B. Namen in einer bestimmten Reihenfolge sortiert werden. Sortieren ist nichts anderes als ein Vergleichen von 2 Daten und ein evtl. notwendiges Vertauschen.

Für die Lösung dieser Problematik stellt uns die 6500-Assemblersprache drei Befehle

- CMP: Vergleiche mit Inhalt des Akkus
  Compare to Accumulator
- CPX: Vergleiche mit Inhalt des X-Registers
  Compare to Register X
- CPY: Vergleiche mit Inhalt des Y-Registers
  Compare to Register Y

zur Verfügung. Bei der Abarbeitung der Vergleichsbefehle führt die CPU eine Subtraktion durch. Das zu vergleichende Datum wird vom Registerinhalt intern abgezogen, ohne daß Registerinhalt oder Datum verändert werden. Entsprechend dem Ergebnis werden die Flags Z und C gesetzt:

- Zeroflag = 1, wenn beide Daten gleich sind
- Carryflag = 1, wenn der Inhalt des Registers größer ist

Das Negativflag verhält sich so unregelmäßig, daß es auf keinen Fall bei späteren Abfragen verwendet werden sollte.

Mit den Verzweigungsbefehlen kann im Anschluß an den Vergleich auf das Ergebnis reagiert werden. Im Bild 1.35 sind alle Möglichkeiten, wie die Verzweigungsbefehle eingesetzt werden können, gezeigt. Der Buchstabe R steht für den Inhalt eines der drei Register, der Buchstabe D für das Datum, das mit dem Registerinhalt verglichen werden soll. Die Pfeile weisen in Richtung des Daten- bzw. Programmflusses.

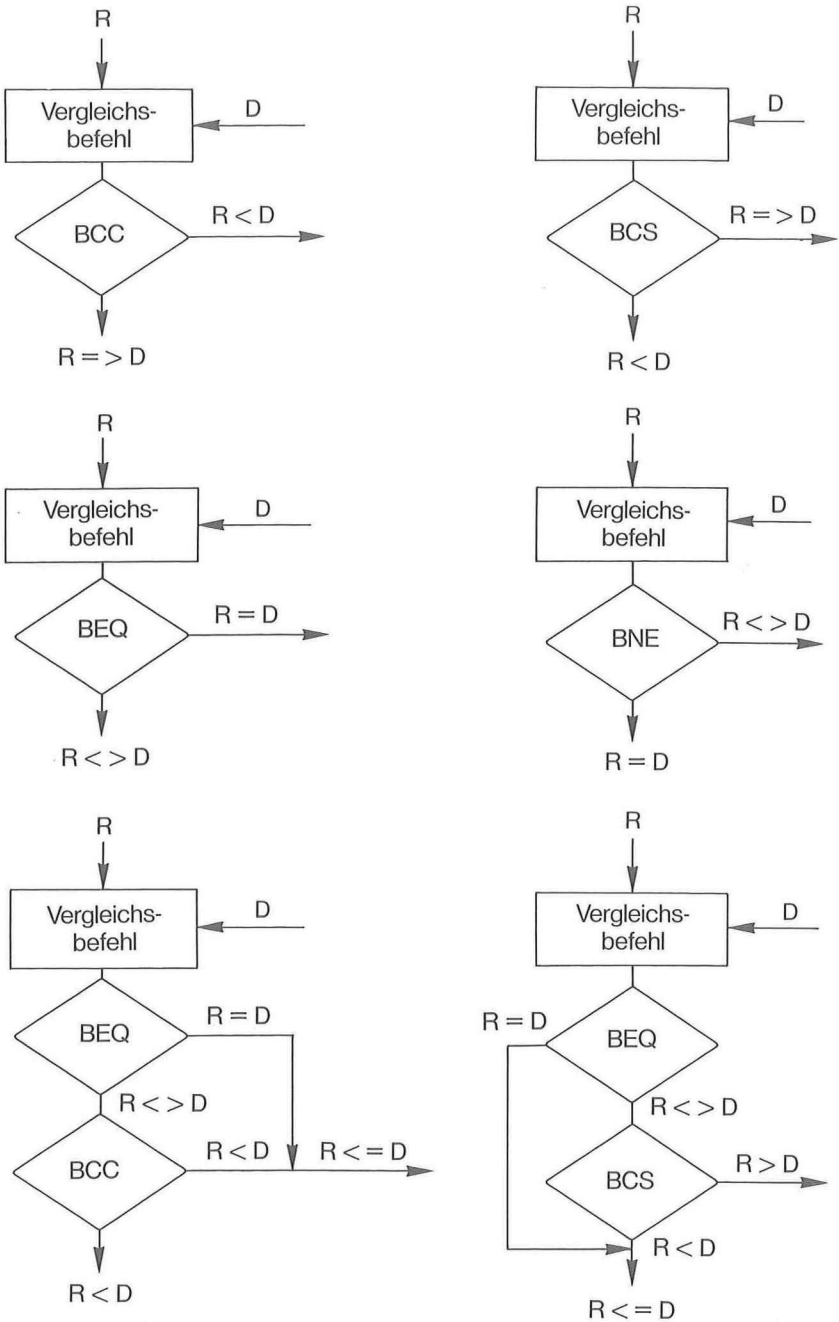


Bild 1.35 Einsatz der Vergleichsoperationen (R = Registerinhalt, D = Vergleichsdatum)

Wissen Sie jetzt, warum in unserem Beispielprogramm aus den Kapiteln 1.4.4 und 1.11 das Indexregister mit dem Wert 64 initialisiert wurde und nicht mit Null? Richtig, hätten wir in dem Programm das Indexregister inkrementiert, müßte ein zusätzliches CPX #$3F vor dem BNE-Befehl eingebaut werden. Dann müßten wir auch das STA $13FF,X in STA $1400,X ändern. Bitte versuchen Sie selbst, die Nuß zu knacken, warum diese Änderungen nötig geworden sind. Ändern Sie dann das Programm dahingehend, geben Sie es ein, starten und testen Sie es bitte.

Befehl: CMP			Funktion: N,Z,C <— (A)-Daten										
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex			C9	CD	C5		*D2	DD	D9	D5		C1	D1
Bytes			2	3	2		2	3	3	2		2	2
Takte			2	4	3		5	*4	*4	4		6	*5
Flags	N	V	B	I	D	Z	C	Das Vergleichsdatum wird vom Register abgezogen, die Flags N,Z und C entsprechend dem Ergebnis gesetzt.					
	X					X	X						

Befehl: CPX			Funktion: N,Z,C <— (X)-Daten										
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex			E0	EC	E4								
Bytes			2	3	2								
Takte			2	4	3								
Flags	N	V	B	I	D	Z	C	Das Vergleichsdatum wird vom Register abgezogen, die Flags N,Z und C entsprechend dem Ergebnis gesetzt.					
	X					X	X						

Befehl: CPY			Funktion: N,Z,C <— (Y)-Daten										
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex			C0	CC	C4								
Bytes			2	3	2								
Takte			2	4	3								
Flags	N	V	B	I	D	Z	C	Das Vergleichsdatum wird vom Register abgezogen, die Flags N,Z und C entsprechend dem Ergebnis gesetzt.					
	X					X	X						

Bild 1.36 Die Vergleichsbefehle CMP, CPX und CPY

Wir wollen diesmal kein neues Beispielprogramm entwickeln, sondern wieder ein bißchen im Betriebsystem unseres C128 stöbern. Nebenbei bemerkt, das Betriebssystem ist das Programm, das unseren Rechner zum »Ticken« bringt. Schauen Sie sich ruhig mit dem Monitor im Inneren unseres Rechners um. Das Betriebssystem ist verantwortlich für die Bedienung der Tastatur, des Bildschirms und der übrigen Peripherie. Ein weiterer Bestandteil unseres Computers ist der BASIC-Interpreter und der Monitor, den wir jetzt gleich aufrufen wollen.

Bitte disassemblieren Sie den Bereich von $00380 bis $00397 mit dem Monitorbefehl

```
D 00380 00397    <RETURN>
```

Sie werden auf Ihrem Bildschirm die aus Kapitel 1.11.2 schon bekannten Zeilen bis $0038C sehen. Uns interessieren jetzt aber die sich daran anschließenden ab $00390:

```
00380 E6 3D      INC   $3D      ;  Erhöhe das Lo-Byte des
                                   Zeigers in den BASIC-Text
                                   um eins
00382 D0 02      BNE   $0386    ;  Ist das Lo-Byte des Zei-
00384 E6 3E      INC   $3E      ;  gers übergelaufen? Wenn
                                   ja, erhöhe auch das Hi-
                                   Byte des Zeigers um eins
00386 8D 01 FF   STA   $FF01    ;  Schalte BASIC-Text-Bank
                                   ein
00389 A0 00      LDY   #$00     ;  Index = 0, damit das
                                   adressierte Byte in den
                                   Akku geladen werden kann
0038B B1 3D      LDA   ($3D),Y  ;  Lade Zeichen aus dem
                                   BASIC-Text
0038D 8D 03 FF   STA   $FF03    ;  Blende die ROMs wieder
                                   ein. Prüfe das Zeichen ob
                                   es eine Ziffer ist.
00390 C9 3A      CMP   #$3A     ;  Ist es größer/gleich dem
                                   ASCII »:«?
00392 B0 0A      BCS   $039E    ;  Ja -> springe

00394 C9 20      CMP   #$20     ;  Ist es ein Leerzeichen?
00396 F0 E8      BEQ   $0380    ;  Ja -> überlesen, nächstes
                                   Zeichen aus dem Text lesen
```

Die Routine CHRGET untersucht in dem Teil ab $00390 das aus dem BASIC-Text geholte Zeichen. Ist es ein Doppelpunkt, das Trennzeichen in einer BASIC-Zeile, so wird die Routine mit gesetztem Zeroflag verlassen, zusätzlich ist das Carryflag gesetzt. Damit wird signalisiert, daß das Zeichen auf keinen Fall eine Ziffer sein

kann. Mit den beiden nächsten Befehlen wird geprüft, ob das Zeichen ein Leerzeichen (Englisch: Blank oder Space) ist. Wenn ja, wird es überlesen und das nächste Zeichen geholt. Im Commodore-BASIC müssen die Befehle nicht wie in anderen BASIC-Dialekten durch ein Leerzeichen von den folgenden Zeichen getrennt sein. Vorhandene Leerzeichen werden von der CHRGET-Routine ignoriert. Das Stöbern im Betriebssystem ist gar nicht so schwierig. Sie werden im nächsten Abschnitt weitere Routinen aus dem ROM kennenlernen. Diese Routinen können Sie dann in eigenen Programmen verwenden. Doch genug jetzt über die Vergleichsbefehle.

Befehl: ADC			Funktion: A <— (A) + Daten + C										
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex			69	6D	65		*72	7D	79	75		61	71
Bytes			2	3	2		2	3	3	2		2	2
Takte			2	4	3		5	*4	*4	4		6	*5
Flags	N	V	B	D	I	Z	C	Addiere Datum mit dem Carry zum Akku. Das Ergebnis steht im Akku, der Übertrag im Carry.					
	X	X				X	X						

Befehl: SBC			Funktion: A <— (A) - Daten - C										
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex			E9	ED	E5		*F2	FD	F9	F5		E1	F1
Bytes			2	3	2		2	3	3	2		2	2
Takte			2	4	3		5	*4	*4	4		6	*5
Flags	N	V	B	D	I	Z	C	Subtrahiere Datum mit dem Carry vom Akku. Das Ergebnis steht im Akku, der Übertrag im Carry.					
	X	X				X	X						

Bild 1.37 Die arithmetischen Befehle ADC und SBC

1.15 Rechenbefehle

Das eigentliche Metier eines Computers ist das Rechnen; dafür ist er ursprünglich entwickelt worden. Im eigentlichen Sinn kann er auch nichts anderes. Selbst Textverarbeitung ist nichts anderes als mit Zahlen jonglieren, sind doch auch die Buchstaben als Zahlen kodiert. Das Rechnen mit Binärzahlen haben Sie im hoffentlich nicht zu trockenem Trockenkurs im Kapitel 1.12 schon kennengelernt. Jetzt ist es an der Zeit, diese Kenntnisse in ein Programm umzusetzen.

1.15.1 Die arithmetischen Befehle ADC und SBC

An Grundrechenarten stellt uns die 6500-Assemblersprache nur die Addition mit dem Befehl ADC (Englisch: Add with Carry = Addiere mit Übertrag) und die Subtraktion mit SBC (Englisch: Subtract with Carry = Subtrahiere mit Borgen) zur Verfügung. Beide Befehle können, je nach Zustand des Dezimalflags, im Prozessorstatusregister dual oder dezimal arbeiten. Bitte lesen Sie sich noch einmal die Kapitel 1.3.6 und 1.9.5 dazu durch.

Die Beispielprogramme sind jeweils für einen der beiden Befehle geschrieben. Für den anderen Befehl wäre nur z. B. ADC durch SBC und CLC durch SEC zu ersetzen.

Addition von 2 8-Bit-Zahlen

Addiere die Inhalte der Speicherzellen in $01360 und $01361. Schiebe das Ergebnis nach $01362
Geben Sie bitte in die beiden Speicherzellen mit dem M-Befehl Werte für die Summanden ein

```
01300 18          CLC           ;  Lösche Carryflag immer vor
                                    einer Addition
01301 AD 60 13    LDA   $1360   ;  Hole 1. Summand
01304 6D 61 13    ADC   $1361   ;  Addiere zum Akkuinhalt den
                                    2. Summanden
01307 8D 62 13    STA   $1362   ;  Lege die Summe ab
0130A 00                BRK     ;  Unterbreche das Programm
```

Experimentieren Sie mit verschiedenen Werten für die beiden Summanden in $01360 und $01361. Beachten Sie bitte nach dem BRK-Kommando den Inhalt des SR-Registers (siehe Kapitel 1.5) und da besonders das Carryflag.

Subtraktion von 2 16-Bit-Zahlen

Subtrahiere die beiden 16-Bit-Zahlen in $01360 und $01362
Schiebe die 16-Bit-Differenz nach $01364
Geben Sie bitte in die Speicherzellen für den Minuenden und den Subtrahend mit dem M-Befehl Werte ein.

```
01300 F8        SEC         ;  Setze Carryflag immer vor ei-
                                ner Subtraktion
01301 AD 60 13  LDA $1360   ;  Hole Lo-Byte des Minuenden
01304 ED 62 13  SBC $1362   ;  Subtrahiere vom Akkuinhalt
                                das Lo-Byte des Subtrahenden
01307 8D 64 13  STA $1364   ;  Lege Lo-Byte der Differenz ab
0130A AD 61 13  LDA $1361   ;  Hole Hi-Byte des Minuenden
0130D ED 63 13  SBC $1363   ;  Subtrahiere vom Akkuinhalt
                                das Hi-Byte des Subtrahenden
01310 8D 65 13  STA $1365   ;  Lege Hi-Byte der Differenz ab
01313 00        BRK         ;  Unterbreche das Programm
```

Experimentieren Sie auch hier mit verschiedenen Werten. Tauschen Sie den Befehl SEC durch CLC und die beiden SBCs durch ADC aus, haben Sie eine 16-Bit-Additionsroutine.

Zum Schluß noch eine 16-Bit-Addition für BCD-Zahlen. Bitte vergessen Sie nicht, die Befehle SEI vor dem SED und CLI nach dem CLD (Kapitel 1.9.4 und 1.9.5)!

Addition von 2 16-Bit-BCD-Zahlen

Addiere die beiden 16-Bit-BCD-Zahlen in $01360 und $01362
Schiebe die 16-Bit-Summe nach $01364
Geben Sie bitte in die Speicherzellen für die Summanden dem M-Befehl Werte ein.

```
01300 18        CLC         ; Lösche Carryflag
01301 AD 60 13  LDA $1360   ; Hole Lo-Byte des Summanden 1
01304 78        SEI         ; Verbiete Interrupt
01305 F8        SED         ; Dezimalmodus ein
01306 ED 62 13  ADC $1362   ; Addiere den Akkuinhalt zum Lo-
                              Byte des Summanden 2
01309 8D 64 13  STA $1364   ; Lege Lo-Byte der Summe ab
0130C AD 61 13  LDA $1361   ; Hole Hi-Byte des Summanden 1
0130F ED 63 13  SBC $1363   ; Addiere den Akkuinhalt zum Hi-
                              Byte des Summanden 2
01312 D8        CLD         ; Dezimalmodus wieder aus
01313 58        CLI         ; Erlaube wieder Interrupts
01314 8D 65 13  STA $1365   ; Lege Hi-Byte der Summe ab
01317 00        BRK         ; Unterbreche das Programm
```

Beim Probieren denken Sie bitte an den erlaubten Wertebereich der BCD-Zahlen und an die unerlaubten Bit-Kombinationen! Zur Kontrolle können Sie die Ergebnisse auf einem Blatt Papier nachrechnen.

1.15.2 Die logischen Befehle AND, EOR und ORA

Außer den oben besprochenen arithmetischen Operationen beherrscht die ALU noch drei logische Verknüpfungen. Sie werden mit den Assembler-Befehlen durchgeführt:

- AND: Führe eine bitweise UND-Verknüpfung zwischen dem Akkuinhalt und dem angegebenen Datum durch.
- ORA: Führe eine bitweise ODER-Verknüpfung zwischen dem Akkuinhalt und dem angegebenen Datum durch.
- EOR: Führe eine bitweise EXKLUSIV-ODER-Verknüpfung zwischen dem Akkuinhalt und dem angegebenen Datum durch.

Die beiden wichtigsten Einsatzgebiete der logischen Befehle, Bitmanipulation und Verknüpfungen von Wahrheitswerten, wurden mit Beispielen schon im Kapitel 1.12.7 intensiv besprochen. Das Beispiel aus Kapitel 12.7.2 »Setzen des Bit 6« soll stellvertretend für die anderen in 6500-Assembler kodiert werden:

```
01300 A9 A5     LDA #$A5    ; Byte in dem Bit 6 zu setzen ist
01302 09 40     ORA #$40    ; Bitmaske % 0100 0000
01304 8D 64 13  STA $1364   ; Speichere maskiertes Byte
01307 00        BRK         ; Beende Programm
```

Ich hoffe, Sie machen sich die Arbeit leicht und rechnen die Binärmaske aus der Dualzahl ins Sedezimalsystem mit Hilfe des Monitor-Befehls % um.

Befehl: AND			Funktion: A <— (A) und Daten										
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex			29	2D	25		*32	3D	39	35		21	31
Bytes			2	3	2		2	3	3	2		2	2
Takte			2	4	3		5	*4	*4	4		6	*5
Flags	N	V	B	I	D	Z	C	Bitweise UND-Verknüpfung zwischen Akkuinhalt und angegebenem Datum					
	X					X							

Befehl: EOR			Funktion: A <— (A) exklusiv-oder Daten										
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex			49	4D	45		*52	5D	59	55		41	51
Bytes			2	3	2		2	3	3	2		2	2
Takte			2	4	3		5	*4	*4	4		6	*5
Flags	N	V	B	I	D	Z	C	Bitweise EXOR-Verknüpfung zwischen Akkuinhalt und angegebenem Datum					
	X					X							

Befehl: ORA			Funktion: A <— (A) oder Daten										
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex			09	0D	05		*12	1D	19	15		01	11
Bytes			2	3	2		2	3	3	2		2	2
Takte			2	4	3		5	*4	*4	4		6	*5
Flags	N	V	B	I	D	Z	C	Bitweise ODER-Verknüpfung zwischen Akkuinhalt und angegebenem Datum					
	X					X							

Bild 1.38 Die logischen Befehle AND, EOR und ORA

1.16 Bitschiebeoperationen

Neben den arithmetischen und logischen Befehlen kennt unsere CPU noch eine weitere Gruppe von Befehlen, mit denen Daten verarbeitet werden können. Es sind die Bitschiebebefehle

- ASL: Schiebe Byte arithmetisch bitweise nach links
- LSR: Schiebe Byte logisch bitweise nach rechts
- ROL: Rotiere Byte um ein Bit nach links und
- ROR: Rotiere Byte um ein Bit nach rechts

Wie leicht erkennbar ist, sind es zwei unterschiedliche Gruppen: Die Schiebe- (Englisch: Shift) und die Rotieroperationen (Englisch: Rotate). In beiden Fällen werden die Bits in einem Byte um eine Stelle nach links oder rechts verschoben. Beim Rotieren gelangt das am Ende hinausrotierte Bit über das Carrybit wieder an das andere Ende des Bytes. Beim Schieben geht es verloren und in die freie Stelle am anderen Ende des Bytes wird eine Null hineingeschoben. Doch verständlicher als viele Worte wird Bild 1.39 diesen Sachverhalt erklären:

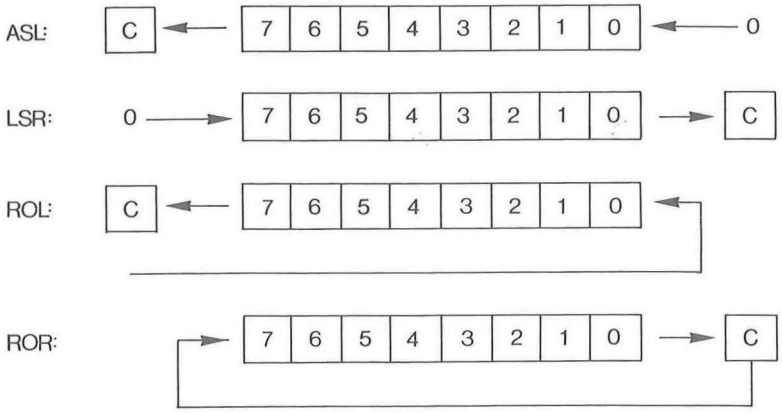


Bild 1.39 Funktion der Bitschiebe- und Rotierbefehle

Während die logischen Operationen nur bestimmte Bits in einem Byte ändern, wirken die Bitschiebe- und Rotierbefehle auf alle Bits in einem Byte.

Befehl: ASL				Funktion: C <— 7 6 5 4 3 2 1 0 <— 0									
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex		0A		0E	06			1E		16			
Bytes		1		3	2			3		2			
Takte		2		6	5			7		6			
Flags	N	V	B	I	D	Z	C	Schiebe Byte eine Bitstelle nach links. In Bit 0 wird eine Null, in das Carry Bit 7					
	X					X	X	geschoben.					

Befehl: LSR				Funktion: 0 —> 7 6 5 4 3 2 1 0 —> C									
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex		4A		4E	46			5E		56			
Bytes		1		3	2			3		2			
Takte		2		6	5			7		6			
Flags	N	V	B	I	D	Z	C	Schiebe Byte eine Bitstelle nach rechts. In Bit 7 wird eine Null, in das Carry Bit 0					
	0					X	X	geschoben.					

Bild 1.40 Die Schiebebefehle ASL und LSR

Die Befehle können zum bitweisen Testen eines Bytes oder zur Multiplikation einer 8-Bit-Zahl mit 2, bzw. Division durch 2, verwendet werden.

Als erstes Beispiel wollen wir eine kleine Routine zur Umwandlung eines Bytes in eine Folge von »0« und »1« schreiben. Die Aufgabenstellung ist einfach: Wenn ein Bit gesetzt ist, soll auf dem Bildschirm eine 1, wenn es gelöscht ist, eine 0 erscheinen. Das Byte ist 8-Bit lang, deshalb muß die Untersuchungsschleife auch 8mal durchlaufen werden. Beginnen wollen wir mit dem höchstwertigen Bit, damit die Einsen und Nullen in der richtigen Reihenfolge auf dem Schirm erscheinen. Wir können demnach die Befehle ASL oder ROL verwenden. Da das Byte nach 8 Rotieranweisungen wieder so ist, wie es beim Beginn war, wollen wir den ROL-Befehl benützen. Durch ASL hätten wir am Schluß ein vollständig gelöschtes Byte. Das Byte selbst soll sich in Speicherzelle $01330 befinden.

Für die Ausgabe auf dem Bildschirm wollen wir eine Routine aus dem Betriebssystem einsetzen. In der Adresse $FFD2 steht ein Sprungvektor, der auf diese

Routine mit dem Namen BSOUT zeigt. Als Vektoren werden Zeiger auf Programmroutinen bezeichnet.

```
01300 A0 0D      LDA #$0D     ; Führe einen Zeilenvorschub
01302 20 D2 FF   JSR $FFD2    ; aus
01305 A2 08      LDX #$08     ; X als Zähler für 8 Bit
01307 2E 30 13   ROL $1330    ; Rotiere Bit ins Carrybit
0130A B0 04      BCS $1310    ; Wenn es gesetzt ist, springe
0130C A9 30      LDA #$30     ; ASCII-Code für das Zeichen 0
0130E D0 02      BNE $1312    ; Springe immer, da Akku
01310 A9 31      LDA #$31     ; ASCII-Code für das Zeichen 1
01312 20 D2 FF   JSR $FFD2    ; Springe zur Routine um das
                               Zeichen im Akku auf dem
                               Bildschirm auszugeben
01315 CA         DEX          ; 8 Bit gemacht?
01316 D0 EF      BNE $1307    ; Nein -> nächstes Bit
01318 A9 0D      LDA #$0D     ; Fertig -> führe noch einen
0131A 20 D2 FF   JSR $FFD2    ; Zeilenvorschub aus
0131D 00         BRK          ; Beende Programm
```

Probieren Sie bitte dieses Programm aus, und testen Sie es mit verschiedenen Werten in der Speicherzelle $01330. Leider hat sich die Voraussage, daß das Byte am Ende der Routine wieder den gleichen Wert wie am Anfang besitzen wird, nicht erfüllt. Sollten Sie nicht dahinterkommen, warum es so ist, warten Sie bitte auf die Auflösung im nächsten Kapitel.

In unserem zweiten Prográmmchen wollen wir die 8-Bit-Zahl in der Speicherzelle $01330 durch 2 dividieren:

```
01300 4E 30 13   LSR $1330    ; Dividiere durch 2
01303 00         BRK          ; Das war es schon
```

Ein extrem kurzes Programm. Auch wenn es so kurz ist, testen Sie es mit verschiedenen Werten. Ändern Sie das Programm um, damit eine Multiplikation mit 2 durchgeführt wird und versuchen Sie zu erklären, warum es funktioniert. Eine ganze Reihe von Aufgaben, die hier auf Sie warten. Ich bin sicher, wenn Sie die Lösungen haben, können Sie auch alle Möglichkeiten, die diese Befehle in sich bergen, in Ihren eigenen Programmen einsetzen.

Befehl: ROL					Funktion: C <— 7 6 5 4 3 2 1 0 <— C								
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex		2A		2E	26			3E		36			
Bytes		1		3	2			3		2			
Takte		2		6	5			7		6			
Flags	N	V	B	I	D	Z	C	Rotiere Byte eine Bitstelle nach links. In Bit 0 wird das Carry, in das Carry Bit 7 geschoben.					
	X					X	X						

Befehl: ROR					Funktion: C —> 7 6 5 4 3 2 1 0 —> C								
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex		6A		6E	66			7E		76			
Bytes		1		3	2			3		2			
Takte		2		6	5			7		6			
Flags	N	V	B	I	D	Z	C	Rotiere Byte eine Bitstelle nach rechts. In Bit 7 wird das Carry, in das Carry Bit 0 geschoben.					
	X					X	X						

Bild 1.41 Die Rotationsbefehle ROL und ROR

1.17 Die Stackbefehle

Zwei Befehle, die mit dem Stack zu tun haben, haben wir im Kapitel 1.9.1 schon kennengelernt. Es sind die Prozessorstatusregister-Befehle PHP und PLP. Jetzt wollen wir die vier restlichen Befehle besprechen:

- PHA: Push Accumulator = Bringe den Akkuinhalt auf den Stack
- PLA: Pull Accumulator = Hole Datum vom Stack in den Akku
- TSX: Transfer Stackpointer into X-Register = Übertrage den Stapelzeiger ins X-Register
- TXS: Transfer X-Register into Stackpointer = Übertrage das X-Register in den Stackpointer

Die Akku-Stackbefehle werden in Programmen häufig zum Retten des Akkuinhaltes verwendet. Doch achten Sie darauf: Wenn Sie ein Datum auf den Stack schieben, holen Sie es auch wieder zurück!

Befehl: PHA					Funktion: Stack <— (A), S <— (S)-1								
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex Bytes Takte	48 1 3												
Flags	N	V	B	I	D	Z	C	Bringe den Akkuinhalt auf den Stapel. Setze den Stapelzeiger neu.					

Befehl: PLA					Funktion: A <— (Stack), S <— (S)+1								
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex Bytes Takte	68 1 4												
Flags	N X	V	B	I	D	Z X	C	Lade den Akku mit dem obersten Wert auf dem Stapel. Korrigiere den Stapelzeiger.					

Bild 1.42 Die Akku-Stackbefehle PHA und PLA

Mit dem TSX-Befehl besteht die Möglichkeit, den Stackpointer in einem Programm zu verarbeiten. Die Registeranzeige des Monitors holt sich mit diesem Befehl den Inhalt des Stackpointers und zeigt ihn dann auf dem Bildschirm an. Mit dem Pendant, dem Befehl TXS, hat man die Möglichkeit, den Stapelzeiger auf einen bestimmten Wert zu setzen. Es ist der einzige Weg, diesen Zeiger direkt zu verändern. In den Resetroutinen aller 6500-Computer befindet sich gleich am Anfang dieser Befehle, um das Stackpointerregister zu initialisieren.

So, jetzt die versprochene Auflösung der Frage aus dem letzten Kapitel: In der Schleife, in der die einzelnen Bits getestet werden, wird eine Betriebssystemroutine aufgerufen. Da in dieser Routine das Carryflag verändert wird und der Befehl ROL den Übertrag in das niederwertigste Bit schiebt, kann man eigentlich auch nicht

erwarten, daß der Zustand des Carryflags vor Aufruf der Systemroutine gleich ist dem Zustand nach Verlassen der Routine. Aus diesem Grund wird aller Wahrscheinlichkeit nach das Byte in der Speicherzelle $01330 ebenfalls verändert sein. Um das Ergebnis korrekt zu erhalten, muß das Carryflag gerettet werden. Ich hoffe, es ist Ihnen jetzt ein Licht aufgegangen: Die Befehle PHP und PLP sind genau das, was gebraucht wird. Nach dem ROL $1330 muß ein PHP stehen, um den Status über den Stack zu retten. Vor dem Befehl DEX fügen Sie bitte ein PLP ein, um das Carrybit wiederherzustellen. Wenn Sie Bild 1.20 noch einmal betrachten, werden Sie feststellen, daß DEX das Carryflag nicht beeinflußt. Damit stimmt jetzt unsere Routine. Viel Spaß damit!

Befehl: TSX					Funktion: X <— (S)								
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex Bytes Takte	BA 1 2												
Flags	N X	V	B	I	D	Z X	C	Bringe den Stapelzeiger in das X-Register. Der Stapelzeiger bleibt unverändert.					

Befehl: TXS					Funktion: S <— (X)								
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex Bytes Takte	9A 1 2												
Flags	N	V	B	I	D	Z	C	Bringe den Inhalt des X-Registers in den Stapelzeiger. Das X-Register bleibt unverändert.					

Bild 1.43 Die X-Register-Stapelbefehle TSX und TXS

1.18 Die Unterprogrammbefehle JSR und RTS

Einmal schon haben wir in einem unserer Beispielprogramme einen Unterprogrammaufruf verwendet, ohne genau zu wissen, was hier passiert. Was sind Unterprogramme?

Es gibt in größeren Programmen Teile, die mehrfach benötigt werden. Man kann jeden dieser Teile für sich selbst, wo er benötigt wird, programmieren. Durch diese Vorgehensweise wird aber sehr viel Speicherplatz verbraucht.

Dieser Programmteil kann aber auch nur einmal als Unterprogramm programmiert werden, das dann bei Bedarf aufgerufen wird. Unterprogramme sind normale Sequenzen von Befehlen, die durch den speziellen Befehl JSR aufgerufen und durch den Befehl RTS abgeschlossen werden. Im Englischen nennt man ein Unterprogramm eine Subroutine. Ein Unterprogramm kann von verschiedenen Stellen im Programm aus aufgerufen werden. Es wird dabei immer die gleiche Aufgabe verrichtet. Ein sehr schönes Beispiel für eine Subroutine kennen Sie schon: Die CHRGET-Routine. Sie wird im BASIC-Interpreter ständig in verschiedenen Programmroutinen aufgerufen, um das nächste Zeichen aus dem BASIC-Quelltext zu holen.

Stößt die CPU auf den Befehl

- JSR: Jump to Subroutine = Springe zum Unterprogramm

so legt sie als erstes den Programmzähler + 2 auf den Stapel. Diese erhöhte Adresse zeigt jetzt auf das Byte vor dem nächsten Befehl. Dann wird der Operand (immer 2 Byte) als neue Adresse in den Programmzähler geladen. Die CPU arbeitet anschließend ab dieser Adresse weiter. JSR ist ein unbedingter Sprungbefehl. Benötigt man einen bedingten Subroutinenaufruf, muß man sich mit den Verzweigungsbefehlen und einem JSR, ähnlich wie im Beispiel im Kapitel 1.13.2.2, behelfen. Am Ende des Unterprogrammes muß der Befehl

- RTS: Return from Subroutine = Kehre aus dem Unterprogramm zurück

stehen. Die CPU holt sich die Rücksprungadresse vom Stapel und erhöht diese um eins. Jetzt zeigt der Programmzähler korrekt auf den nächsten Befehl, mit dem die CPU die weitere Abarbeitung des Programmes fortsetzt.

Befehl: JSR				Funktion: PC <— ADR, Stack <— (PC)+2									
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex Bytes Takte				20 3 6									
Flags	N	V	B	I	D	Z	C	Der Programmzähler +2 wird auf den Stapel gebracht. Das Programm wird bei ADR fortgesetzt.					

Befehl: RTS				Funktion: PC <— (Stack), S <— (S)+1, PC <— (PC)+1									
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex Bytes Takte	60 1 6												
Flags	N	V	B	I	D	Z	C	Der Programmzähler wird vom Stapel geholt und inkrementiert. Der Stapelzeiger wird neu gesetzt.					

Bild 1.44 Die Unterprogramm-Befehle JSR und RTS

Wenn Sie in einem Unterprogramm Daten mit den Befehlen PHP oder PHA auf den Stapel schieben, vergessen Sie unter keinen Umständen diese Daten vor einem RTS wieder vom Stapel zu holen! Ich glaube nicht, daß dieser Hinweis noch weiterer Erklärung bedarf.

Wenn Sie das Monitorprogramm im Bereich $FB0D0 bis $FB0D8 disassemblieren, sehen Sie folgenden merkwürdigen Konstrukt:

```
FB0D0 BD FD B0    LDA $B0FD,X
FB0D3 48          PHA
FB0D4 BD FC B0    LDA $B0FC,X
FB0D7 48          PHA
FB0D8 4C A7 B7    JMP $B7A7
```

Es werden anscheinend 2 Byte aus einer Tabelle X-indiziert geladen und auf den Stapel geschoben. Dann erfolgt ein normaler Sprung nach $B7A7. Folgt man dieser

Routine durch alle Verzweigungen, so stellt man fest, daß sie mit einem RTS beendet wird. Frage: Wo wird das Programm wohl fortgesetzt werden?

Richtig – die beiden Bytes, die auf den Stapel geschoben wurden, stellen die Adresse-1 des dann folgenden Programmteiles dar. Im BASIC-Interpreter ist in der Interpretationsroutine ab $F4B80 die gleiche Art des Routinenaufrufes zu finden.

Befehl: NOP					Funktion: keine								
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex Bytes Takte	EA 1 2												
Flags	N	V	B	I	D	Z	C	Der Befehl tut 2 Taktzyklen lang buchstäblich NICHTS.					

Befehl: BIT					Funktion: Z <— (A) und (M), N <— (M7), V <— (M6)								
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex Bytes Takte			*89 2 2	2C 3 4	24 2 3			*3C 3 *4		*34 2 4			
Flags	N 7	V 6	B	I	D	Z X	C	Die Inhalte von Akku und M werden UND-verknüpft. Vom Datenbyte werden Bit 7 ins V-Flag und Bit 6 ins N-Flag kopiert. Das Z-Flag wird vom Ergebnis der Verknüpfung gesetzt. Der Akku bleibt unverändert.					

Bild 1.45 Die Sonderbefehle NOP und BIT

1.19 Die Sonderbefehle

Zwei Befehle gibt es im 6500-Assembler, die ganz besondere Funktionen zur Verfügung stellen:

1.19.1 Der Befehl NOP

Es ist wirklich so: Beim Befehl NOP (Englisch: No Operation) tut die CPU tatsächlich für 2 Takte buchstäblich nichts. Dieser Befehl wird häufig zur Verlängerung von Zeitverzögerungsschleifen eingesetzt, oder er dient als Platzhalter für später einzufügende Befehle. Letztere Möglichkeit ist beim TOP-ASS, wie wir noch sehen werden, bedeutungslos, aber bei der zeilenweisen Assemblierung mit dem Monitor interessant.

1.19.2 Der Befehl BIT

Verglichen mit dem Befehl NOP ist BIT (Englisch: Test Bits in Memory with Accu = Teste Bits im Speicher mit Akku) ein überaus komplexer Befehl. Es wird eine UND-Verknüpfung zwischen dem Inhalt des Akkus und der adressierten Speicherzelle durchgeführt. Das Zeroflag wird abhängig vom Ergebnis der logischen Operation gesetzt. Ist das Resultat der logischen Verknüpfung aller Bitpositionen gleich Null, wird das Zeroflag auf Eins gesetzt, sonst wird es gelöscht. Mit BEQ und BNE kann dann verzweigt werden.

Im weiteren überträgt BIT zusätzlich das Bit 6 der adressierten Speicherzelle in das Überlaufflag und Bit 7 in das Negativflag. Es ist der einzige Befehl, mit dem das Bit 6 eines Bytes direkt abfragt werden kann. Mit den Branchbefehlen BMI, BPL, BVC und BVS kann über die beiden Flags verzweigt werden. Der Akkumulatorinhalt bleibt unverändert erhalten.

Der Befehl BIT kann außer zum Testen von Bits noch zu einem kleinen kodesparenden Trick verwendet werden.

```
01307 2E 30   13   ROL $1330   ;  Rotiere Bit ins Carrybit
0130A B0 03        BCS $130f   ;  Wenn es gesetzt ist, springe
0130C A9 30        LDA #$30    ;  ASCII-Code für das Zeichen 0
0130E 2C A9   31   BIT $31A9   ;  ????
01311 20 D2   FF   JSR $FFD2   ;  Gebe Zeichen auf Bildschirm
                                  aus
```

Es scheint ein Teil aus dem Programm zur Ausgabe von Dualzahlen auf dem Bildschirm zu sein. Doch was soll der BIT-Befehl? Nur Geduld! Es wird in $0130C der ASCII-Code für das Zeichen »0« in den Akku geladen. Wie wir wissen, verändert der darauffolgende BIT-Befehl den Akkuinhalt nicht, er hat also hier keine Auswirkung, da der Zustand der Flags ignoriert werden kann. In $0130A wird auf die Speicherzelle $0130F gesprungen. Hier steht der Kode

```
0130F A9 31    und das ist die Befehlsfolge      LDA #$31
```

(wie das Disassemblieren über »D 0130F« ebenfalls beweist).

Der BIT-Befehl ist damit übersprungen, der Akku wird korrekt mit dem ASCII-Code für das Zeichen »1« geladen. Die Logik des Programmes ist eingehalten und als zusätzliches Geschenk wurde ein Byte eingespart!

1.20 Interrupts mit den Befehlen RTI und BRK

In den sechs höchsten Speicherzellen müssen sich in Computern mit Prozessoren der 6500-Familie drei Vektoren befinden. Diese Vektoren zeigen auf Programme, in denen die Interruptanforderungen abgearbeitet werden.

- NMI-Vektor — in $FFFA und $FFFB: Zeigt auf die Interruptroutine zur Abarbeitung des nicht-maskierbaren Interrupts.
- RST-Vektor — in $FFFC und $FFFD: In dieser Routine müssen alle Register, I/O-Ports, usw. initialisiert werden. Beim Einschalten des Rechners erfolgt ein Sprung in diese Routine.
- IRQ-Vektor — in $FFFE und $FFFF: Hier wird der maskierbare Interrupt, angefordert durch Hard- oder Software, erledigt.

Erfolgt eine Aufforderung zur Unterbrechung, so beendet die CPU auf jeden Fall erst den Befehl, den sie gerade abarbeitet, bevor sie den Interrupt bedient. Dabei werden vom Prozessor nur der Programmzähler und das Prozessorstatusregister automatisch auf den Stapel gerettet. Dann wird der Inhalt des zutreffenden Interrupt-Vektors in den Programmzähler geladen und die Interruptroutine gestartet. Das Interruptflag im Prozessorstatusregister wird gesetzt, um weitere Interruptanforderungen zu verhindern. Möchte man trotzdem weitere IRQ-Anforderungen zulassen, muß das Interrupt-Enable-Flag mit dem Befehl CLI wieder gelöscht werden. NMI-Anforderungen können nicht verhindert werden. Sie besitzen höchste Priorität, deren Anforderungen immer sofort erfüllt werden müssen. Da in den Interruptroutinen normalerweise die drei Registerinhalte verändert werden, sollte der Programmierer beim Eintritt in die Routinen, die die Inhalte von Akku, X- und Y-Register retten.

Befehl: RTI					Funktion: P <— (Stack), PC <— (Stack), S <— (S)+3								
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex Bytes Takte	40 1 6												
Flags	N X	V X	B X	I X	D X	Z X	C X	Der ursprüngliche Zustand des Statusregisters und Programmzählers vorm Interrupt wird hergestellt.					

Befehl: BRK					Funktion: Stack <— (PC)+2, Stack <— (P) PC <— ($FFFE,$FFFF)								
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex Bytes Takte	00 1 7												
Flags	N	V	B *	I 1	D	Z	C	Ist ein Software-Interrupt. PC und Status werden gerettet. In den PC wird die Adresse in $FFFE geholt.					

Bild 1.46 Die Interruptbefehle RTI und BRK

Folgende Befehlsfolge ist deshalb zu Beginn fast jeder Interruptroutine zu finden:

```
PHA   ;   Rette Akku
TXA   ;   X nach A kopieren
PHA   ;   Rette X
TYA   ;   Y nach A kopieren
PHA   ;   Rette Y
```

Vor Verlassen der Interruptroutine müssen die Register selbstverständlich wiederhergestellt werden:

```
PLA   ;   Hole Y zurück
TAY   ;   Nach Y kopieren
PLA   ;   Hole X zurück
TAX   ;   Nach X kopieren
PLA   ;   Hole A zurück
RTI   ;   Rückkehr vom Interrupt
```

Beachten Sie, daß der Akku immer zuerst gerettet und in Analogie dazu zuletzt vom Stapel geholt werden muß.

Das Interruptprogramm muß mit dem Befehl RTI (Englisch: Return from Interrupt = Rückkehr vom Interrupt) verlassen werden. Einzig das Resetprogramm benötigt diesen Befehl nicht, da ja alle Register in dieser Routine erst initialisiert werden. RTI löscht das Interrupt-Enable-Flag wieder, es können die nächsten Anforderungen abgearbeitet werden. Das Statusrregister und der Programmzähler werden vom Stapel geholt. Die CPU arbeitet im alten Programm den nächsten Befehl ab.

Normalerweise erfolgen die Interruptanforderungen durch die Hardware über die Steuerleitungen. Doch stellt die 6500-Assemblersprache einen Befehl zur Verfügung, um softwaremäßig einen Interrupt einzuleiten. Der Befehl heißt BRK (Englisch: Break = Unterbrechung), er simuliert einen IRQ. Wir haben diesen Befehl in unseren Beispielprogrammen immer am Ende des Programmes verwendet. Durch den Befehl BRK werden folgende Vorgänge gestartet: Wie bei den anderen Unterbrechungen wird das Statusregister auf den Stapel gerettet. Im Gegensatz dazu wird aber das Breakflag im Prozessorstatusregister gesetzt und der Programmzähler um zwei erhöht, bevor er ebenfalls auf den Stapel abgelegt wird. Mit folgender Routine ist es möglich, festzustellen, ob der IRQ durch den Befehl BRK oder durch die Hardware ausgelöst wurde:

```
PLA          ;  Hole Statusregister vom Stapel
PHA          ;  Stelle den Stapel wieder her
AND #$10     ;  Ist das B-Flag gesetzt?
BNE BRKPRG   ;  Ja -> springe in BRK-Programm
```

Mit dieser Befehlsfolge verzweigt die CPU immer bei einem BRK in das BRK-Programm, und das ist beim C128 das Monitorprogramm. Jetzt ist auch klar, warum wir beim Beenden unserer Programmbeispiele mit dem BRK-Befehl immer wieder im Monitor gelandet sind.

Auf einen Punkt muß ich noch einmal hinweisen: Nach einem BRK weist der Programmzähler nicht auf den nächsten Befehl! Er weist auf das Byte danach. Der Sinn und Zweck ist einleuchtend: Beim Testen eines Programmes wird man BRK-Anweisungen an besonders markanten Stellen einbauen, um die Registerinhalte auslesen zu können. Da die meisten Befehle des 6500-Assemblers 2-Byte lang sind, kann BRK als Ersatz eingesetzt werden. Unter Umständen ist aber eine Korrektur notwendig.

1.21 CPU-Fehler und Illegale Opcodes

Seit den Zeiten des alten KIM und des PET mit ihren 6502-CPUs, über den C64 mit der CPU 6510 bis zum C128 mit seinem 8502, pflanzen sich einige Fehler bei fast allen Mitgliedern der 6500-Familie über die Jahre fort. Erst mit den CMOS-Versionen 65SC02 und 65C02 sind diese Fehler korrigiert worden.

1.21.1 Die CPU-Maskenfehler

Diese Fehler stammen aus dem Fertigungsprozess. Die Maske, nach der die CPUs hergestellt werden, hat einen Fehler, der sich bei einigen Befehlen und Adressierungsarten unangenehm bemerkbar machen kann.

1.21.1.1 Die indizierten Adressierungsarten

Bei allen Befehlen, die die Adressierungsarten Zeropage-indiziert, X-indiziert-indirekt und Indirekt-Y-indiziert muß ein Überschreiten der Zeropage durch die Indizierung unbedingt vermieden werden, d. h., die Basisadresse + Index darf nicht größer als $FF werden. Siehe dazu auch den Hinweis im Kapitel 1.6.10.

1.21.1.2 Der indirekte JMP-Befehl

Immer wenn das niederwertige Byte des Operanden $FF ist, holt sich der Prozessor das höherwertige Adreßbyte aus der falschen Speicherzelle:

Beispiel:

```
JMP (23FF)
```

Es wird das niederwertige Adreßbyte korrekt aus der Speicherzelle $23FF geholt, doch das höherwertige Byte wird nicht, wie beabsichtigt, aus $2400, sondern aus $2300 geholt. Gute Assembler, wie TOPASS, melden diesen Fehler; der A-Befehl des Monitors bietet diesen Komfort nicht.

1.21.1.3 Die dezimalen Rechenoperationen

Bei dezimalen Rechenoperationen werden die Flags nicht immer richtig gesetzt:

Beispiel:

```
01300 18     CLC       ; Lösche Carry
01301 78     SEI       ; Verbiete Interrupt
01302 F8     SED       ; Schalte auf Dezimalmodus
01303 A9 99  LDA #$99  ; Addiere zu dezimal 99
```

```
01305 69 01   ADC #$01   ;  eine 1.
01307 D8      CLD        ;  Binärmodus wieder einschalten
01308 78      SEI        ;  Erlaube wieder Interrupts
01309 00      BRK        ;  Zeige Statusregister an
```

Das Statusregister sieht jetzt folgendermaßen aus:

```
N V - B D I Z C
1 0   1 0 1 0 1
```

Das Carryflag ist korrekt gesetzt, es trat ein Übertrag auf. Falsch ist das Zeroflag; es müßte ebenfalls gesetzt sein, da das Ergebnis im Akku gleich Null ist. Gleichfalls ist das Negativflag falsch, das Ergebnis ist nie negativ. Richtig müßte das Prozessorstatusregister folgenderweise aussehen:

```
N V - B D I Z C
0 0   1 0 1 1 1
```

1.21.2 Die illegalen Opcodes

Insgesamt könnten für die CPUs der 6500-Familie 256 Operationen kodiert sein und nicht nur die 151, die tatsächlich vorhanden sind. Die nicht gebrauchten Opcodes sind von den Herstellern nicht mit NOPs unterlegt worden. Man könnte sich streiten, ob das ebenfalls zu den Maskenfehlern zu rechnen sei, da ja einige dieser illegalen Operationskodes vernünftige Befehle ausführen - doch Vorsicht beim Einsatz! Diese Befehle sind von den Herstellern der CPU nicht dokumentiert, sind also nicht Standard. Sie können auf einer CPU eines anderen Herstellers zu ganz anderen Reaktionen, z. B. Absturz, führen als beabsichtigt. Sie lassen sich aber bis jetzt beim 6510 und ebenfalls beim 8502 einsetzen. Im folgenden eine Auflistung dieser Befehle mit ihren Mnemonics, die sich seit dem Assembler-Sonderheft (Sonderheft 8/85) des 64er-Magazins eingebürgert haben:

- A11: AND Register with $11
- AAX: AND Accu with X and Store Accu
- ASR: AND Accu and Shift right
- ARR: AND with Accu and Rotate right
- AXS: AND Accu with X and SUBTRACT Data
- DCP: Decrement and Compare with Accu
- DOP: Double NOP
- ISC: Increment and Subtract with Carry
- KIL: Kill the CPU
- LAR: Load Accu, AND with Stackpointer, result to Accu, X and S
- LAX: Load Accu and transfere to X

- NOP: NOP
- RLA: Rotate left, AND with Accu, Store Accu
- RRA: Rotate right, Add with Accu
- SLO: Shift left and OR with Accu
- SRE: Shift left and EXOR with Accu
- TOP: Triple NOP

In den Bildern 1.47a bis 1.47e finden Sie alle weiteren Angaben zu diesen Befehlen.

Befehl: A11					Funktion: M <— (Register) und ($11)								
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex Bytes								9C 3	9E 3				
Flags	N	V	B	I	D	Z	C	AND Register with $11					
	X					X							
Verknüpfe Register mit Speicher durch UND, speichere Ergebnis ab.													

Befehl: AAX					Funktion: M <— (A) und (X)								
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex Bytes			88 2	8F 3	87 2						97 2	83 2	
Flags	N	V	B	I	D	Z	C	AND Accu with X and STORE Accu					
	X					X							
Der Akku und das X-Register werden durch UND verknüpft. Der Akkuinhalt wird abgespeichert.													

Befehl: ASR					Funktion: A <— (A) und Daten, A <— (A) Shift Right								
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex Bytes			6B 2										
Flags	N	V	B	I	D	Z	C	AND with Accu and SHIFT RIGHT					
	0					X	X	Befehlsfolge: AND, LSR					
Der Akku wird mit dem Datum durch UND verknüpft. Das Ergebnis wird im Akku noch um eine Bitposition nach rechts geschoben.													

Bild 1.47a Die illegalen Opcodes

Befehl: ARR					Funktion: A <— (A) und (D), A <— (A) Rotate Right								
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex Bytes			4B 2										
Flags	N	V	B	I	D	Z	C	AND with Accu and ROTATE RIGHT					
	X					X	X	Befehlsfolge: AND, ROR					
Der Akku wird mit dem Datum durch UND verknüpft. Das Ergebnis im Akku noch um eine Bitposition nach rechts rotiert.													

Befehl: AXS					Funktion: A <— (A) und (X), A <— (A) - Daten								
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex Bytes			CB 2										
Flags	N	V	B	I	D	Z	C	AND Accu with X and SUBTRACT Data					
	X	X				X	X						
Der Akku wird mit dem X-Register UND verknüpft. Vom Ergebnis wird der angegebene Wert subtrahiert.													

Befehl: DCP					Funktion: M <— (M)-1, N,Z,C <— (A) - Daten								
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex Bytes				CF 3	C7 2			DF 3	DB 3	D7 2		C3 2	D3 2
Flags	N	V	B	I	D	Z	C	DECREMENT and COMPARE with Accu					
	X					X	X	Befehlsfolge: DEC, CMP					
Die Speicherzelle wird um 1 erniedrigt, dann mit dem Akku verglichen. Der Akkuinhalt bleibt unverändert.													

Bild 1.47b Die illegalen Opcodes

Befehl: DOP	Funktion: Tut nichts Double NOP
Adressierung ist Implied. Folgende Opcodes führen diesen Befehl aus: $04, $14, $34, $44, $54, $64, $74, $80, $89, $93, $D4, $F4	
Es wirkt wie ein NOP, doch wird das folgende Byte übersprungen.	

Befehl: ISC					Funktion: M <— (M) + 1, N,Z,C,(A) <— (A) - (M)								
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex Bytes				EF 3	E7 2			FF 3	FB 3	F7 2		E3 2	F3 2
Flags	N	V	B	I	D	Z	C	INCREMENT and SUBTRACT with Carry					
	X	X				X	X	Befehlsfolge: INC, SBC					
Die Speicherzelle wird um 1 erhöht, das Ergebnis vom Akku subtrahiert. Der Akkuinhalt wird verändert.													

Befehl: KIL	Funktion: Absturz des Prozessors
Adressierung ist Implied. Folgende Opcodes führen diesen Befehl aus: $02, $12, $22, $32, $42, $52, $62, $72, $92, $B2, $D2, $F2	
Die CPU stürzt komplett ab. Nur mehr ein RESET kann helfen.	

Befehl: LAR					Funktion: A,X,S <— (S) und (M)								
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex Bytes									BB 3				
Flags	N	V	B	I	D	Z	C	LOAD Accu, AND with Stackpointer, TRANSFERE Result to Accu, X & S					
	X					X		Befehlsfolge: LDA, AND, TAX, TXS					
Lade Akku, kopiere das Ergebnis der UND-Operation mit dem Stackpointer ins X-Register und Stackpointer. Der Akkuinhalt wird ebenfalls geändert.													

Bild 1.47c Die illegalen Opcodes

Befehl: LAX				Funktion: A,X, <— (M)									
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex Bytes				AF 3	A7 2			BF 3	B7 2			A3 2	B3 2
Flags	N	V	B	I	D	Z	C	LOAD Accu and TRANSFERE to X					
	X					X		Befehlsfolge: LDA, TAX					
Der Akku und das X-Register wird geladen.													

Befehl: NOP	Funktion: Nichts
Adressierung ist Implied. Folgende Opcodes stehen zur Verfügung: $1A, $3A, $5A, $7A, $DA, $FA	
Diese Opcodes haben die gleiche Wirkung wie das normale NOP.	

Befehl: RLA				Funktion: (M) <— ROL, A <— (A) und (M), M <— (A)									
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex Bytes				2F 3	27 2			3F 3	3B 3	37 2		23 2	33 2
Flags	N	V	B	I	D	Z	C	ROTATE LEFT, AND with Accu, STORE Accu					
	X					X	X	Befehlsfolge: ROL, AND, STA					
Rotiere die Speicherzelle nach links, verknüpfe das Ergebnis durch UND mit dem Akku, speichere anschließend den Akku in die Speicherzelle.													

Befehl: RRA				Funktion: (M) <— ROR, A <— (A)+(M)									
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex Bytes				6F 3	67 2			7F 3	7B 3	77 2		63 2	73 2
Flags	N	V	B	I	D	Z	C	ROTATE RIGHT, ADD with Accu					
	X	X				X	X	Befehlsfolge: ROR, ADC					
Rotiere die Speicherzelle nach rechts, addiere Ergebnis zum Akkuinhalt.													

Bild 1.47d Die illegalen Opcodes

Befehl: SLO				Funktion: (M) <— ASL, A <— (A) oder (M)									
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex Bytes				0F 3	07 2			1F 3	1B 3	17 2		03 2	13 2
Flags	N	V	B	I	D	Z	C	SHIFT LEFT and ÖR with Accu					
	X					X	X	Befehlsfolge: ASL, ORA					
Schiebe die Speicherzelle nach links, führe mit Ergebnis und Akkuinhalt eine Oder-Verknüpfung durch. Der Akkuinhalt wird verändert.													

Befehl: SRE				Funktion: (M) <— LSR, A <— (A) exor (M)									
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex Bytes				4F 3	47 2			5F 3	5B 3	57 2		43 2	53 2
Flags	N	V	B	I	D	Z	C	SHIFT LEFT and EXOR with Accu					
	X					X	X	Befehlsfolge: ASL, EXOR					
Schiebe die Speicherzelle nach links, führe mit Ergebnis und Akkuinhalt eine EXOR-Verknüpfung durch. Der Akkuinhalt wird verändert.													

Befehl: TOP	Funktion: Tut nichts Triple NOP
Adressierung ist Implied. Folgende Opcodes führen diesen Befehl aus: $0C, $1C, $3C, $5C, $7C, $DC, $FC	
Es wirkt wie ein NOP, doch werden die folgenden 2 Byte übersprungen.	

Bild 1.47e Die illegalen Opcodes

1.22 Die Befehle der CPU 65C02

In den Jahren 1982 und 1983 erschienen zwei neue Mitglieder der 6500-Familie auf dem Markt. Beide sind CMOS-Versionen des 6502-Prozessors, doch mit einigen neuen Befehlen und Adressierungsarten.

In den Bildern mit den Befehlsbeschreibungen sind die zusätzlichen Adressierungsarten, die die Prozessoren 65C02 und 65SC02 zur Verfügung stellen, im Opcode mit einem Stern gekennzeichnet. Folgende Befehle wurden vollständig neu implementiert:

- BRA: Branch Always = Verzweige immer
- PHX: Push X on Stack = Schiebe X auf den Stapel
- PHY: Push Y on Stack = Schiebe Y auf den Stapel
- PLX: Pull X from Stack = Hole X vom Stapel
- PLY: Pull Y from Stack = Hole Y vom Stapel
- STZ: Store Zero in Memory = Lege Null im Speicher ab
- TRB: Test and Reset Memorybit with Accu = Teste Speicher mit Bitmaske im Akku, Lösche Bits dementsprechend
- TSB: Test and Set Memorybit with Accu = Teste Speicher mit Bitmaske im Akku, setze Bits dementsprechend
- BBR: Branch on Bit Reset = Verzweige, wenn Bit Null ist
- BBS: Branch on Bit Set = Verzweige, wenn Bit gesetzt ist
- RMB: Reset Memory Bit = Lösche Bit im Speicher
- SMB: Set Memory Bit = Setze Bit im Speicher

Die letzten vier Befehle können nur bei der CPU 65C02 angewendet werden. Die restlichen versteht auch die CPU 65SC02. Die Befehle BBR und BBS benötigen 2 Operanden, die durch ein Komma getrennt sind: Das erste Byte ist die Adresse der Speicherzelle in der Zeropage, die getestet wird, das zweite Byte gibt den Sprungoffset an. Die illegalen Opcodes funktionieren nicht mehr. Sie bewirken NOPs, DOPs und TOPs (siehe Kapitel 1.21). Die Befehle sind mit allen wichtigen Informationen in den Bildern 1.48a bis 1.48d dargestellt.

Befehl: BRA					Funktion: Verzweige immer								
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex Bytes Takte						80 2 *3							
Flags	N	V	B	I	D	Z	C	Springe immer zur angegebenen Adresse. Befehlsfolgen wie BEQ dann BNE sind damit unnötig.					
Branch Always = Verzweige immer													

Befehl: PHX					Funktion: Stack <— (X), S <— (S) - 1								
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex Bytes Takte	DA 1 3												
Flags	N	V	B	I	D	Z	C	Der Inhalt von X wird auf den Stapel gebracht, der Stapelzeiger wird korrigiert.					
Push X on Stack = Schiebe X-Register auf den Stapel													

Befehl: PHY					Funktion: Stack <— (Y), S <— (S) - 1								
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex Bytes Takte	5A 1 3												
Flags	N	V	B	I	D	Z	C	Der Inhalt von Y wird auf den Stapel gebracht, der Stapelzeiger wird korrigiert.					
Push Y on Stack = Schiebe Y-Register auf den Stapel													

Bild 1.48a Die Befehle der CPU 65C02

Befehl: PLX					Funktion: X <— (Stack), S <— (S) + 1								
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex Bytes Takte	FA 1 4												
Flags	N X	V	B	I	D	Z X	C	Der oberste Inhalt vom Stapel wird nach X gebracht, der Stapelzeiger wird korrigiert.					
Pull X from Stack = Hole X-Register vom Stapel													

Befehl: PLY					Funktion: Y <— (Stack), S <— (S) + 1								
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex Bytes Takte	7A 1 4												
Flags	N X	V	B	I	D	Z X	C	Der oberste Inhalt vom Stapel wird nach Y gebracht, der Stapelzeiger wird korrigiert.					
Pull Y from Stack = Hole Y-Register vom Stapel													

Befehl: STZ					Funktion: M <— 0								
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex Bytes Takte				9C 3 4	64 2 3			9E 3 5		74 2 4			
Flags	N	V	B	I	D	Z	C	Der Wert 0 wird in die adressierte Speicherzelle geschrieben.					
Store Zero in Memory = Lege Null im Speicher ab													

Bild 1.48b Die Befehle der CPU 65C02

Befehl: TRB				Funktion: M <— (A) und (M)									
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex				1C	14								
Bytes				3	2								
Takte				6	5								
Flags	N	V	B	I	D	Z	C	Teste Speicher mit Bitmaske im Akku und lösche Bits im Speicher.					
						X							
Test and Reset Memorybits with Accu = Vergleiche Speicher mit Akku und lösche Bits entsprechend.													

Befehl: TSB				Funktion: M <— (A) oder (M)									
Adress.	Imp	A	#	Abs	ZP	Rel	( )	,X	,Y	Z,X	Z,Y	(,X)	( ),Y
Hex				0C	04								
Bytes				3	2								
Takte				6	5								
Flags	N	V	B	I	D	Z	C	Teste Speicher mit Bitmaske im Akku und setze Bits im Speicher.					
						X							
Test and Set Memorybits with Accu = Vergleiche Speicher mit Akku und setze Bits entsprechend.													

Befehl: BBR				Funktion: Verzweige, falls Bit Null ist			
Adress.	absolut Zeropage, relativ						
Hex	0F, 1F, 2F, 3F, 4F, 5F, 6F, 7F						Das höherwertige Nibble gibt das zu testende Bit an.
Bytes:	3			Takte: #5			Schreibweise z. B.: BBR 1,$0A,$2F
Flags	N	V	B	I	D	Z	C	Verzweige, falls das getestete Bit gelöscht ist. Befehl nur in der Zeropage möglich.
Branch on Bit Reset = Verzweige, wenn Bit Null ist.							

Bild 1.48c Die Befehle der CPU 65C02

Befehl: BBS	Funktion: Verzweige, falls Bit gesetzt ist	
Adress.	absolut Zeropage, relativ	
Hex	8F, 9F, AF, BF, CF, DF, EF, FF	Das höherwertige Nibble - 8 gibt das zu testende Bit an.
Bytes: 3	Takte: #5	Schreibweise z. B.: BBS 1,$0A,$2F
Flags N V B I D Z C		Verzweige, falls das getestete Bit gesetzt ist. Befehl nur in der Zeropage möglich.
Branch on Bit Set = Verzweige, wenn Bit gesetzt ist.		

Befehl: RMB	Funktion: Mbit <— 0	
Adress.	absolut Zeropage	
Hex	07, 17, 27, 37, 47, 57, 67, 77	Das höherwertige Nibble gibt das zu löschende Bit an.
Bytes: 2	Takte: 5	Schreibweise z. B.: RMB 1,$0A
Flags N V B I D Z C		Lösche das angegebene Bit in der adressierten Speicherzelle.
Reset Memory Bit = Lösche Speicherbit		

Befehl: SMB	Funktion: Mbit <— 1	
Adress.	absolut Zeropage	
Hex	87, 97, A7, B7, C7, D7, E7, F7	Das höherwertige Nibble - 8 gibt das zu löschende Bit an.
Bytes: 2	Takte: 5	Schreibweise z. B.: SMB 1,$0A
Flags N V B I D Z C		Setze das angegebene Bit in der adressierten Speicherzelle.
Set Memory Bit = Setze Speicherbit		

Bild 1.48d Die Befehle der CPU 65C02

1.23 Literaturverzeichnis

Gratuliere, Sie haben sich durchgebissen und sich die notwendigen Grundkenntnisse angeeignet, um eigene Problemlösungen in Assembler formulieren zu können. Damit Sie bei tieferschürfenden Fragen nicht auf sich selbst gestellt bleiben, nenne ich Ihnen eine Reihe von Publikationen, die ich persönlich für gut halte.

Allgemeine 65xx-Literatur:

Rockwell International: R6500 Programming Manual
Martinsried Document No. 29650 N30A
Rockwell International: R6500 Hardware Manual
Martinsried Document No. 29650 N31
Rockwell International: Datenblatt R6500
Martinsried Document No. 29000 D39
Rockwell International: Datenblatt R65C00
Martinsried Document No. 29651N52
GTE: Datenblatt G65SC02
München Document 3002-01-02
Zaks, Rodnay: Programmierung des 6502
SYBEX, Düsseldorf ISBN 3-88745-011-6
Zaks, Rodnay: 6502 Anwendungen
SYBEX, Düsseldorf ISBN 3-88745-014-0
Leventhal, Lance A.: 6502-Programmieren in Assembler
te-wi, Haar ISBN 3-921803-10-1
Scanlon, Leo J.: 6502 Software Design
Howard W. Sams & Co,
Indianapolis ISBN 0-672-21656-6

C-128-Literatur:

Schineis et al: C128 ROM Listing Operating System
Markt & Technik, Haar ISBN 3-89090-221-9
Schineis et al: C128 ROM Listing Basic-7.0-Betriebssystem
Markt & Technik, Haar ISBN 3-89090-220-0
Ruprecht, Dr.: C128 ROM Listing Commodore Sachbuch
Markt & Technik, Haar ISBN 3-89090-212-X
Gerits et al: Commodore 128 Intern
DATA BECKER, Düsseldorf ISBN 3-89011-098-3

Möllmann, Gerd: C128 Programmieren in Maschinensprache
Markt & Technik, Haar ISBN 3-89090-213-8
Müller, Florian: Vom C64 zum C128 Tips & Tricks
Markt & Technik, Haar ISBN 3-89090-402-5
Rosenbeck, Peter: Das Commodore 128-Handbuch
Markt & Technik, Haar ISBN 3-89090-195-6
Ponnath, Heimo: Grafikprogrammierung C128
Markt & Technik, Haar ISBN 3-89090-202-2
Schramm, Karsten: Die Floppy 1570/71
Markt & Technik, Haar ISBN 3-89090-185-9

Als besondere Fundgrube für Commodore-Computer hat sich folgende Zeitschrift erwiesen:

64er – Das Magazin für Computerfans
Markt & Technik, Haar

Das 64er-Magazin erscheint monatlich und vermittelt vor allem zum Thema »Maschinensprache« wichtige Informationen. Ein umfangreicher C128-Teil hält Sie immer über aktuelle Ereignisse rund um Ihr Gerät auf dem laufenden.

Auch eine Vielzahl von Sonderheften hat 64er schon veröffentlicht, wovon folgende speziell die Themen »Maschinensprache« und »C128« aufgreifen:

Sonderheft 8/85 »Assembler«
Sonderheft 1/86 »128er«
Sonderheft 8/86 »128er«
Sonderheft 12/86 »Assembler / Programmiersprachen«

Aber auch zu anderen Themen werden Sonderhefte angeboten, die Ihnen in vielerlei Hinsicht nützen werden. Inhaltlich kann es ein 64er-Sonderheft mit manch gutem Buch aufnehmen, vor allem was die Programmvielfalt anbelangt.

Kapitel 2

Assemblerprogrammierung auf dem C-128

Nachdem wir in Kapitel 1 den Aufbau des Prozessors und seinen Befehlsvorrat unter die Lupe genommen haben, ist es in diesem Kapitel an der Zeit, sich mit der zweiten wichtigen Voraussetzung für ein erfolgreiches Programmieren in Assembler zu beschäftigen: der Kenntnis der Hardware des Rechners, in diesem Fall des C-128. Anders als die Programmierung in einer höheren Programmiersprache wie z. B. Basic erfordert die Programmierung in Assembler immer eingehende Kenntnisse des internen Aufbaus des Computers. Auf Maschinenspracheebene existiert keine Instanz wie der Basic-Interpreter, die den Programmierer von den Hardware-Eigenheiten des Rechners abschirmt. Mehr noch: Logische Programmfehler werden nicht mit einem *syntax error* geahndet, sondern es wird der Normalfall sein, daß bei auftretenden Fehlern nur noch der Reset-Taster den Rechner dazu bewegen kann, seine Arbeit wieder aufzunehmen.

Dieses Kapitel will Ihnen zunächst eine Vorstellung vom Aufbau des C-128 vermitteln. Zunächst wird die Speicherverwaltung des C-128 angesprochen: Wie verwaltet der 128er die RAM-Bänke und die ROMs? Diesem Thema kommt auf dem 128er die beinahe gleiche Bedeutung zu wie der Kenntnis der Prozessorbefehle selbst. Daran anschließend werden einige Routinen näher besehen, die das Betriebssystem der C-128 bereitstellt, um dem Programmierer den zugegebenermaßen am Anfang vielleicht etwas kompliziert scheinenden Umgang mit der Speicherverwaltung zu erleichtern. Drittes Thema dieses Abschnitts bildet dann die Klärung der Frage, wo der Anwender seine Maschinenprogramme im Speicher plazieren kann und welche Adressen in der Zeropage gefahrlos für eigene Zwecke einzusetzen sind. Nachdem Sie dieses Unterkapitel gelesen haben, sollten Sie sich zumindest mit den Grundlagen der Bedienung des ASE-Assemblers vertraut gemacht haben, um die folgenden Beispielprogramme in der Praxis nachvollziehen zu können. Diese Programme werden im Rahmen der Vorstellung der wichtigsten ROM-Routinen des Betriebssystems entwickelt. In der Hauptsache handelt es sich

bei diesen ROM-Routinen um Routinen zur Ein- und Ausgabe. Bekanntlich bestehen die meisten Programme zu mindestens 80 Prozent aus reiner Ein- und Ausgabe; nach der Lektüre des letzten Teils dieses Kapitels sollten Sie unter anderem in der Lage sein, Ein- und Ausgaben am Bildschirm in Assembler zu programmieren. Das gleiche gilt für die Dateibehandlung in Assembler: Wie werden Dateien geöffnet, beschrieben, gelesen, geschlossen.

Daß dabei nicht jede Einzelheit in aller Ausführlichkeit besprochen werden kann, ist in Anbetracht des hier zur Verfügung stehenden Platzes und der auf Assemblerebene doch erheblichen Komplexität des C-128 nicht verwunderlich. Ein letzter Abschnitt dieses Kapitels weist Sie deshalb auf Bücher und Zeitschriften hin, in denen Sie weiterführende Informationen finden können bzw. auch Listings und aktuelle Beiträge zum C-128.

2.1 Speicherverwaltung des C-128

Eine Frage, die sich Ihnen nach Durcharbeiten des ersten Kapitels dieses Buches eigentlich schon stellen müßte, ist folgende: Mit welchen Befehlen erreicht man eigentlich den gesamten Speicher des C-128, der ja allein an RAM-Bänken schon 128 KByte mißt? Im Befehlssatz des Prozessors ist kein Befehl zu finden, der über dessen Adreßraum von 64 KByte hinausreicht. Das ist eine natürliche Frage, die schon zeigt, daß offensichtlich die alleinige Kenntnis des Prozessors nicht für die vollkommene Beherrschung des 128ers ausreicht.

Was der Prozessor nicht leisten kann, muß jemand anderes übernehmen. In allen Rechnern, die einen größeren Speicher verwalten müssen, als der Prozessor direkt ansprechen kann, übernimmt diese Aufgabe der Speicherverwaltung neben dem Prozessor eine sogenannte **Memory Management Unit (MMU).** Diese MMU kann man sich als einen Mittler zwischen dem Prozessor auf der einen Seite und dem gesamten Speicher auf der anderen Seite vorstellen. Der Prozessor hat keine direkte Verbindung zu den Speicherbausteinen, sondern ist mit der MMU verbunden, die die Speicherzugriffe des Prozessors kontrolliert. In verschiedenen Registern der MMU kann der Programmierer Einstellungen vornehmen, die dem Prozessor Zugang zu ausgewählten Speicherbereichen gestatten oder auch verwehren. Auf diese Weise wird aus dem Gesamtspeicher eine 64 KByte lange Auswahl getroffen, die dem Prozessor als Adreßraum zur Verfügung gestellt wird. Der Prozessor selbst nimmt diesen Vorgang nicht wahr. Für ihn sind ständig 64 KByte Speicher sichtbar; woraus sich dieser Speicher letztendlich zusammensetzt – ob aus RAM oder ROM, ob aus dieser oder jener Bank – bemerkt der Prozessor nicht.

Leider hat diese Art und Weise, einen großen Speicher zu verwalten, das sogenannte **Bankswitching,** für den Assemblerprogrammierer zur Folge, daß er nicht allein den Prozessor zu programmieren hat. Auch die geeignete Zusammenstellung des im Programm benötigten Speichers muß er zunächst einmal selbst kontrollieren. Diese

Vorgehensweise kennen Sie vielleicht in einfacher Form von Basic her: Durch den Befehl *bank* beeinflussen Sie – über die Zwischenstation Basic-Interpreter – die MMU. Die hinter *bank* angegebene Nummer entspricht einem **Konfigurationsindex**, der eine Speicherkonfiguration (Speicherzusammenstellung) aus einer Reihe von vordefinierten auswählt. Nachdem die Konfiguration festgelegt ist, kann mit *peek* oder *poke* eine Adresse aus dieser Konfiguration gelesen oder beschrieben werden. In einem nicht gebankten Rechner wie beispielsweise dem C-64, der nur einen so großen Speicher hat, wie der Prozessor direkt adressieren kann, entfällt entsprechend jede Angabe einer Konfiguration. (Für die Kundigen unter Ihnen möchte ich hinzufügen, daß ich hier das nicht direkt von Basic ansprechbare RAM »unter« dem ROM des C-64 einmal unter den Tisch fallen lasse.)

Ein ganz ähnlicher Mechanismus wie in Basic steht uns auch auf Assemblerebene zur Verfügung, allerdings reichen die 16 vorhandenen Konfigurationsindizes nicht aus, um alle benötigten Speicherkonfigurationen herzustellen. Steigen wir deshalb erst einmal eine Stufe tiefer und betrachten, woraus sich der Speicher des C-128 zusammensetzt und in welchen Abschnitten wir die einzelnen Teile des Gesamtspeichers zu einem Adreßraum für den Prozessor zusammensetzen können:

Der im Serien-C-128 verfügbare RAM-Speicher von 128 KByte wird in Form zweier RAM-Bänke verwaltet. Jede RAM-Bank hat eine Länge von 64 KByte und füllt damit für sich genommen schon den Adreßraum des Prozessors voll aus. An der MMU wird eingestellt, welche RAM-Bank die CPU »sieht«.

In die so gewählte RAM-Bank können ROMs oder auch der später noch angesprochene In/Out-Bereich eingeblendet werden. Teile der RAM-Bank werden auf diese Weise überdeckt und werden dadurch für die CPU wieder »unsichtbar«. Liest der Prozessor aus einem solchen überdeckten Adreßbereich, so liest er Daten aus dem ROM aus. Versucht er allerdings, Daten in einen ROM-Bereich zu schreiben – ein solcher Festwertspeicher kann ja nicht beschrieben werden – so lenkt die MMU diesen Schreibzugriff auf das – bildlich gesprochen – darunterliegende RAM um.

Insgesamt gibt es vier Adreßbereiche, in die ROMs eingeblendet werden können:

$4000 - $7fff:

Hierhin kann der untere Teil des Basic-ROMs gelegt werden. Aufgrund seiner Länge – der Basic-Interpreter ist nur wenig kürzer als der ASE-Assembler – mußte der Interpreter in zwei getrennten ROM-Bausteinen untergebracht werden; dies hier ist der untere Teil. Normalerweise wird der untere Teil nicht allein in den Adreßraum des Prozessors eingeblendet, sondern nur zusammen mit dem zweiten Teil des Interpreter-ROMs.

$8000 - $bfff:

Hier liegt der obere Teil des Basic-Interpreters, falls er über die MMU aktiviert wird. Dieser Teil des Interpreters enthält unter anderem den C-128-Bordmonitor (Basicbefehl *monitor*). Unter Umständen kann es sinnvoll sein, diesen Teil des ROMs allein einzublenden, falls ein Maschinenprogramm etwa Routinen des Monitors als Unterprogramme verwenden will. In den gleichen Adreßbereich können auch ROM-Erweiterungen gelegt werden, die allerdings in der Serienausführung nicht vorhanden sind. Zu unterscheiden sind sogenannte *interne* und *externe* ROM-Erweiterungen. Diese Unterscheidung beruht darauf, wo die Erweiterungen angebracht werden: Interne ROMs werden innerhalb des Rechnergehäuses installiert, externe ROMs werden am Expansion-Port angeschlossen.

$c000 - $cfff und $e000 - $ffff:

Diese beiden Adreßbereiche mit einer Lücke zwischen $d000 und $dfff werden von der MMU gemeinsam ein- und ausgeschaltet, da sie gemeinsam das Betriebssystem des C-128 beherbergen. Der untere Teilbereich enthält den sogenannten **Kernal-Editor**, der unter anderem die Routinen zur Tastaturabfrage, die Fensterverwaltung des C-128 und die Routinen zur Ausführung der Escape-Sequenzen beinhaltet. Im oberen Teil des **Kernal-ROMs** finden sich beispielsweise die Routinen zur Ein- und Ausgabe über den IEC-Bus, die Kassetten- und die RS232-Routinen. Auch hier können alternativ zum normalen Kernal-ROM ROM-Erweiterungen über die MMU sichtbar gemacht werden.

$d000 - $dfff:

Dieser Adreßbereich spielt insoweit eine Sonderrolle, als hier keine ROMs eingeblendet werden können, die Programmkode enthalten. Falls hier kein RAM sichtbar ist, überdeckt entweder eines der beiden möglichen **Zeichensatz-ROMs** (DIN oder ASCII) oder aber der schon erwähnte In/Out-Bereich das RAM. Die Zeichensatz-ROMs enthalten die Zeichendefinitionen für die beiden möglichen Zeichensätze, bestimmen also letztlich das genaue Aussehen der Zeichen am Bildschirm. Der In/Out-Bereich enthält die Register der verschiedenen Chips des 128ers, als da sind: die beiden Videocontroller (40 und 80 Zeichen), die beiden CIAs (Complex Interface Adapters), die unter anderem für den Disketten-Betrieb und die Tastaturabfrage benützt werden, der Sound-Generator SID und schlußendlich die MMU. Des weiteren ist im In/Out-Bereich auch das Farb-RAM des 40-Zeichen-Chips zu finden.

Dies sind die vier Teilbereiche, aus denen sich Adreßraum des Prozessors zusammensetzen läßt. Alle oben nicht erwähnten Teile des Adreßraums werden immer durch RAM der gewählten RAM-Bank eingenommen.

Die tatsächliche Auswahl der Speicherbereiche erfolgt in einem Register der MMU: dem sogenannten **Konfigurationsregister**. Von diesem existieren zwei vollkommen identische Ausführungen – eine im In/Out-Bereich, wie oben schon erwähnt, eine

weitere an der Adresse **$ff00**, scheinbar mitten im Kernal-ROM (falls dieses gerade aktiv ist). Dieses Konfigurationsregister, zusammen mit vier weiteren Registern (s.u.), wird durch Manipulationen der Speicherkonfigurationen **nicht** betroffen; die ab $ff00 liegenden Register der MMU sind also immer sichtbar und nicht abschaltbar. Eine sehr sinvolle Einrichtung: Wie oben zu sehen war, liegen die Register der MMU im In/Out-Bereich; der In/Out-Bereich seinerseits kann aber durch entsprechende Einstellung der MMU ausgeblendet werden. Wäre kein Duplikat des Konfigurationsregisters vorhanden, das sich nicht abschalten ließe, bestünde die Gefahr, sich von der MMU selbst auszuschließen, was nicht gerade im Sinne der Erfinder liegen dürfte.

Folgende Duplikate von MMU-Registern existieren:

$ff00	Konfigurationsregister
$ff01	Load-Configuration-Register A
$ff02	Load-Configuration-Register B
$ff03	Load-Configuration-Register C
$ff04	Load-Configuration-Register D

Im Konfigurationsregister können Speicherkonfigurationen direkt eingestellt werden. Insbesondere kann man hier den In/Out-Bereich mit den weiteren MMU-Registern wieder sichtbar machen, falls er abgeschaltet wurde. Die vier darüberliegenden **Load-Configuration-Register** (Englische Schreibweise) bieten eine Möglichkeit, vier ausgewählte Konfigurationen besonders schnell und einfach zu aktivieren; dazu werden die vier Werte des Konfigurationsregisters, die die ausgewählten Speicherzusammenstellungen bewirken, in vier sogenannte **Pre-Configuration-Register** eingetragen. Die vier Pre-Register liegen im In/Out-Bereich (s.u.). Durch einfaches Beschreiben eines der Load-Configuration-Register mit einem beliebigen Wert wird dann der Inhalt des zugehörigen Pre-Configuration-Registers in das eigentliche Konfigurationsregister transportiert. Befindet sich beispielsweise im Pre-Register A der Wert *xy*, so bewirkt ein Befehl wie *stx $ff01* (Schreibzugriff auf das Load-Register A), daß das Konfigurationsregister den Wert *xy* annimmt. Der Basic-Interpreter macht im Gegensatz zum eigentlichen Betriebssystem Gebrauch von den Möglichkeiten der Load- und Pre-Configuration-Register, deshalb sollte man ihre Werte erst dann verändern, wenn man sich über die Wirkung dieser Änderung auf den Basic-Interpreter voll im klaren ist. Die Belegung der Pre-Configuration-Register und die durch sie zu erzielenden Speicherkonfigurationen können Sie der nachfolgenden Beschreibung der MMU-Register entnehmen.

Bevor wir zu dieser Beschreibung kommen, soll ein weiterer Begriff eingeführt werden, der für die Programmierung des C-128 unerläßlich ist: der der **Common Area**. Um die Funktion der Common Area zu verstehen, wollen wir ein Gedankenexperiment anstellen: Nehmen wir dazu an, ein von uns geschriebenes Maschinenprogramm läge in der RAM-Bank 0 und versuche, ein Byte aus der RAM-Bank 1 zu lesen – eine alltägliche Aufgabe. Vollziehen wir nach, was im C-128 vor sich geht:

Wir starten das Maschinenprogramm etwa durch den Befehl *sys* und der Prozessor beginnt, den in Bank 0 liegenden Objektkode unseres Programms abzuarbeiten. Damit wir irgendwann aus der Bank 1 lesen können, ist es erforderlich, die MMU zu manipulieren, genauer gesagt muß unser Maschinenprogramm die RAM-Bank 1 für den Prozessor sichtbar machen. Also beschreibt es das Konfigurationsregister mit dem erforderlichen Wert und ...

... schon können wir mit großer Wahrscheinlichkeit den Griff zum Reset-Taster üben. Was ist passiert?

Im gleichen Augenblick, in dem das Konfigurationsregister mit dem neuen Wert beschrieben wurde, wurde auch unser Programm für den Prozessor unsichtbar! Er beschreibt das Konfigurationsregister und will danach den nächsten Operationskode aus unserem Maschinenprogramm holen und ausführen – nur, das Programm ist nicht mehr da. Statt dessen findet die CPU ein nicht vordefiniertes Stückchen Kode aus der Bank 1 und alle Murphy'schen Gesetze sprechen dafür, daß der Prozessor sich daraufhin zu weiterer Zusammenarbeit mit dem Programmierer nicht mehr in der Lage sieht.

Natürlich gibt es einen Weg, diesen scheinbar unvermeidlichen Absturz zu vermeiden, sonst würde ja der ganze C-128 nicht funktionieren. Dazu muß man lediglich vermeiden, daß der Programmkode für die CPU unsichtbar wird, und dies ist relativ leicht möglich, denn die MMU erlaubt es, RAM-Bereiche zu definieren, die von einer Umschaltung zwischen den RAM-Bänken nicht betroffen werden – die besagten Common Areas. Common Areas können sowohl am unteren als auch am oberen Ende einer RAM-Bank liegen. Ihre Größe kann jeweils von Null bis zu 16 KByte betragen. Im Normalfall – unter Basic – wird nur eine untere Common Area verwendet, die von Adresse $0 bis $400 reicht. Sie umschließt damit die beiden wichtigen Systemspeicher Zeropage und Prozessorstack. Oberhalb des Stackbereiches finden sich eine Reihe von Lade- und Speicherroutinen, die genau die Aufgabe übernehmen, die das Maschinenprogramm unseres Gedankenexperimentes erfüllen sollte. Diese Routinen werden beim Einschalten des Rechners – oder bei einem Reset – aus den ROMs in die Common Area kopiert. Auch zu diesen Routinen finden Sie weiter unten nähere Einzelheiten.

2.1.1 Register der MMU

Alle Register der im C-128 verwendeten Chips finden sich im In/Out-Bereich zwischen $d000 und $dfff, so auch die der MMU. Die außerhalb des In/Out-Bereiches zu findenden Duplikate von MMU-Registern zwischen $ff00 und $ff04 bilden die einzigen Ausnahmen von dieser Regel, sieht man einmal vom Prozessorport und dem Datenrichtungsregister zu diesem Port ab (beide liegen in der Zeropage, an den Speicherstellen 1 und 0). Jedem der Chips ist innerhalb des In/Out-Bereiches eine Basisadresse zugeordnet; ein bestimmtes Register eines Chips findet

man durch die Addition dieser Basisadresse zur betreffenden Registernummer. Die Basisadresse der MMU ist $d500, also liegt Register 0 der MMU an eben dieser Adresse, Register 1 bei $d501 usw.

Insgesamt besitzt die MMU elf Register. Von diesen elf Registern ist das Konfigurationsregister mit der Nummer 0 sicher das mit Abstand am häufigsten benutzte. Ein exaktes Ebenbild dieses Registers befindet sich an der Adresse $ff00.

2.1.1.1 Das Konfigurationsregister der MMU (Register 0)

Durch Setzen oder Löschen einzelner Bits des Konfigurationsregisters werden ROMs oder RAM-Bänke in den Adreßraum des Prozessors eingeblendet. Die Zuordnung der Bits (nach allgemeiner Übereinkunft zählt man in Kreisen der Maschinen-Programmierer immer von 0 aufwärts):

Bits 7 und 6:

Hier findet die Auswahl der RAM-Bank statt, die als »Grundlage« des Adreßraums in einer bestimmten Konfiguration dienen soll. Mit den beiden Bits können insgesamt vier solcher 64 KByte langer Bänke angesprochen werden, wovon in der derzeitigen Serienausführung des C-128 nur zwei wirklich vorhanden sind. Ist in Ihren Rechner keine Speichererweiterung eingebaut, so ist nur das Bit 6 für Sie von Bedeutung. Das gesetzte Bit 6 schaltet die RAM-Bank 1 ein, entsprechend ist bei gelöschtem Bit 6 die RAM-Bank 0 gewählt.

Bit 7	Bit 6	RAM-Bank
1	1	Nr. 3, falls vorhanden – sonst Nr. 1
1	0	Nr. 2, falls vorhanden – sonst Nr. 0
0	1	Nr. 1
0	0	Nr. 0

Bits 5 und 4:

Diese beiden Bits bestimmen über den Inhalt des Adreßraums zwischen $c000 und $ffff (Kernal-Bereich), dabei enthält dieser Bereich eine Lücke zwischen $d000 und $dfff, die nur indirekt von den beiden Bits beeinflußt wird. In den besagten Bereich kann man ROM-Erweiterungen einblenden, die eine ebensolche Lücke aufweisen wie das Kernal-ROM. Bei einem Seriengerät ohne ROM-Erweiterungen bewirken die Bitkombinationen, die externes oder internes Erweiterungs-ROM auswählen würden, nichts.

Bit 5	Bit 4	Bereich $c000-$cfff und $e000-$ffff
1	1	RAM-Bank (s. Bits 7 und 6)
1	0	Externe ROM-Erweiterung
0	1	Interne ROM-Erweiterung
0	0	Kernal-ROM

Wird versucht, ein eingeblendetes ROM zu beschreiben, wird dieser Zugriff durch die MMU auf die zugrundeliegende RAM-Bank umgeleitet. Dies sollten Sie sich noch einmal merken, denn diese automatische Umleitung erspart dem Programmierer eventuell ein explizites Abschalten der ROMs und verkürzt damit sowohl Programmlänge als auch Laufzeit des Programms.

Bits 3 und 2:

Im Prinzip arbeiten diese beiden Bits wie die soeben erwähnten Bits 5 und 4, nur betreffen die beiden vorliegenden den Speicherbereich zwischen $8000 und $bfff, der normalerweise vom oberen Teil des Basic-ROMs eingenommen wird:

Bit 3	Bit 2	Bereich $8000 - $bfff
1	1	RAM-Bank (s. Bit 7 und 6)
1	0	Externe ROM-Erweiterung
0	1	Interne ROM-Erweiterung
0	0	Oberes Basic-ROM

Bit 1:

Das zweitniedrigste Bit des Konfigurationsregisters ist allein zuständig für den Adreßraum von $4000 bis $7fff. In diesem Bereich können keine ROM-Erweiterungen sichtbar gemacht werden. Ist Bit 1 gelöscht, ist die untere Hälfte des Basic-ROMs aktiv, ist es gesetzt, kann der Prozessor die in den Bits 7 und 6 gewählte RAM-Bank auslesen.

Bit 0:

Dieses Bit steuert den Inhalt des Bereiches $d000 bis $dfff. Seine Wirkung ist teilweise abhängig von der Einstellung der Bits 5 und 4: Wird Bit 0 gelöscht, so wird auf jeden Fall der In/Out-Bereich sichtbar. Ist Bit 0 dagegen gesetzt, entscheidet der Zustand der beiden Bits 5 und 4, ob RAM sichtbar ist (Bit 5 und 4 wählen RAM für ihren Bereich aus) oder ob ein Zeichensatz-ROM gelesen werden kann (Bit 5 und 4 wählen irgendein ROM für ihren Bereich aus). Welches Zeichensatz-ROM – DIN oder ASCII –, wird nicht in den MMU-Registern bestimmt, sondern im Pro-

zessorport in der Zeropage; Bit 6 des Ports gibt den Zustand des ASCII/DIN-Schalters der C-128-Tastatur wieder.

Um das Ganze noch ein wenig zu komplizieren, enthält der In/Out-Bereich neben den schon erwähnten Registern der einzelnen Chips des C-128 in sich selbst noch ein Stück RAM, das Farb-RAM des 40-Zeichen-Bildschirms ab Adresse $d800. Dieser Farbspeicher wiederum kann zwischen zwei verschiedenen physikalischen RAM-Stückchen umgeschaltet werden, die die gleiche Adresse einnehmen. Auch dieser Vorgang wird im Prozessorport gesteuert, zuständig ist das Bit 0. Der Umschaltvorgang an sich wird durch die im Betriebssystem unterstützten »Splitscreens« notwendig, geteilten Bildschirmen aus Grafik- und Textbereich. Dies alles sei aber nur der Vollständigkeit halber erwähnt.

Eine andere Sache sollten Sie sich im Zusammenhang mit dem In/Out-Bereich jedoch sorgfältig einprägen:

Bei jeder wie auch immer gearteten Ein- oder Ausgabeoperation Ihrer Programme muß der In/Out-Bereich für den Prozessor sichtbar sein!

Dies gilt ausdrücklich auch dann, wenn Sie zur Ein- oder Ausgabe Routinen des Betriebssystems benützen. Auch diese Routinen setzen voraus, daß die Chips, die letztendlich die Ein- und Ausgabe übernehmen, für den Prozessor ansprechbar sind. Beachten Sie diesen Punkt nicht, müssen Sie damit rechnen, daß die Ein- und Ausgabe nicht regelgerecht funktioniert. Zudem wird der RAM-Bereich im gleichen Adreßraum wie der In/Out-Bereich in Mitleidenschaft gezogen werden, da der Prozessor in ihm versucht, die hier vermuteten Chip-Register anzusprechen. Dies kann unter anderem dazu führen, daß ein dort liegender Basic-Programmtext zerstört wird.

2.1.1.2 Die Pre-Configuration-Register (Register 1 – 4)

Die vier Pre-Configuration-Register wurden weiter oben schon allgemein angesprochen. Im In/Out-Bereich sind sie oberhalb des eigentlichen Konfigurationsregisters an den Adressen $d501 bis $d504 anzutreffen.

Zu jedem der PCRs existiert ein korrespondierendes Load-Configuration-Register (LCR). Die LCRs liegen in derselben Reihenfolge wie die PCRs hinter dem Duplikat des Konfigurationsregisters an den Adressen $ff01 bis $ff04. Die LCRs können durch keine mögliche Manipulation der MMU aus dem Adreßraum des Prozessors ausgeblendet werden. Der Sinn dieser vier PCRs und LCRs wurde weiter oben schon einmal angedeutet: In die vier Pre-Configuration-Register können Werte eingetragen werden, die durch einen einfachen Schreibzugriff auf eines der Load-Configuration-Register in das Konfgurationsregister übertragen werden. Vier Konfigurationen, die innerhalb eines Programms besonders häufig benützt werden, können auf diese Art besonders schnell und bequem erreicht werden.

Da der Basic-Interpreter Gebrauch von dieser Möglichkeit macht, ist es einigermaßen gefährlich – zumindest für den Anfänger – die Werte der PCRs zu verändern. Wollen Sie die PCRs in eigenen Programmen einsetzen und verändern, sollten Sie zumindest dafür sorgen, daß die ursprünglichen Inhalte der vier Register nach Ablauf des Programmes wiederhergestellt werden. Besonders problematisch wird es, wenn Sie die PCRs verändern, aber gleichzeitig Routinen aus dem Basic-ROM als Unterprogramme Ihres Maschinenprogramms verwenden wollen. Die sicherste Methode, um einen Absturz Ihres Programmes durch etwaige Kollisionen innerhalb des Interpreters bei veränderten PCRs zu vermeiden, besteht dann wohl darin, vor jedem Aufruf eines ROM-Unterprogrammes die PCRs auf die Normalwerte zu bringen und nach Rückkehr vom Unterprogramm die von Ihnen bevorzugten Werte wieder einzusetzen. Im allgemeinen sollten Sie aber zuerst einmal besser von einer Veränderung der PCRs absehen, wenn die Konsequenzen nicht absolut klar sind.

Die unter Basic verwendeten Werte für die PCRs und die daraus resultierenden Speicherlandschaften sind aus den folgenden Bildern 2.1 bis 2.4 zu ersehen. Es sei noch bemerkt, daß diese Konfigurationen nur unter Basic gelten; wenn der Top-Ass-Assembler ASE gestartet wird, verändert er beispielsweise die PCRs. Ähnliches gilt für andere professionelle Programme, die auf dem C-128 laufen.

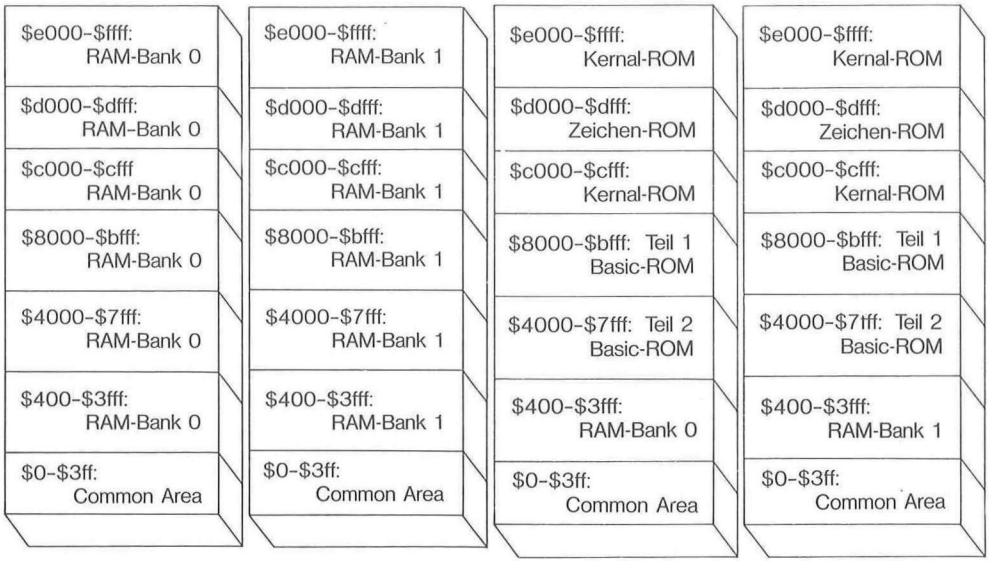


Bild 2.1. Speicherbelegung unter PCRA ($3f).

Bild 2.2. Speicherbelegung unter PCRB ($7f).

Bild 2.3. Speicherbelegung unter PCRC ($01).

Bild 2.4. Speicherbelegung unter PCRD ($41).

Beachten Sie bitte, daß in keiner der vier Konfigurationen, die durch Beschreiben der LCRs aktiviert werden können, der In/Out-Bereich eingeschaltet ist! Der Basic-Interpreter beschreibt bei Ein- und Ausgabeoperationen statt dessen das Konfigurationsregister direkt mit einer Null. Dieser Umstand ist besonders dann zu beachten, wenn man ein Maschinenprogramm etwa als Unterprogramm eines Basic-Programms entwickelt.

2.1.1.3 Das Mode-Configuration-Register (Register 5)

Das Mode-Configuration-Register ist das erste Register der MMU – von denen, die wir bisher kennengelernt haben -, von dem kein Spiegelbild außerhalb des In/Out-Bereiches existiert. Die Schlußfolgerung ist erlaubt, daß dieses Register nicht die zentrale Bedeutung besitzt, die den vorher besprochenen zukommt.

Die Bedeutung der einzelnen Bits dieses Registers:

Bit 7:

Am Zustand dieses Bits läßt sich ablesen, ob die 40/80-Zeichen-Taste beim Start oder Reset des Rechners gedrückt war. Ein gelöschtes Bit an dieser Stelle zeigt an, daß die Einstellung »80 Zeichen« gewählt wurde.

Bit 6:

Wie Sie sicher wissen, besitzt der C-128 verschiedene Betriebsmodi. Zwei dieser Modi benutzen den 8502-Prozessor, während der dritte, der CP/M-Modus, einen Z80-Prozessor benötigt. Im vorliegenden Bit wird bestimmt, welcher der beiden 8502-Modi – der C-64- oder der C-128-Modus – ausgewählt ist, wenn der 8502 aktiv ist. Eine Null bedeutet, daß der C-128-Modus angewählt ist. Nur durch Beschreiben dieses Bits gelangt man noch nicht in den C-64-Modus – vgl. ab $E24B im Kernal (D FE24B).

Bits 5 und 4:

Diese beiden Bits sind im 128-Modus ohne Bedeutung, deshalb soll nur erwähnt werden, daß hier Steckmodule erkannt werden können, die für den C-64-Modus vorgesehen sind. Im C-128-Modus haben diese Bits immer den Wert 0.

Bit 3:

Die Bedeutung dieses Bits ist relativ schwer zu erklären, will man nicht schon einiges an Wissen voraussetzen. Nur soviel: Die neuen Floppies 1571 und 1570 ermöglichen ein gegenüber den älteren Modellen – z. B. der Floppy 1541 des C-64 – erheblich schnelleres Laden von Programmen von Diskette. Das Bit 3 des Mode-Configuration-Registers der MMU besitzt in diesem Zusammenhang eine Bedeutung, da es eine der Übertragungsleitungen zwischen Rechner und Floppy beeinflußt. Diese Leitung kann je nach Stellung von Bit 3 als Eingang (Laden) oder als Ausgang (Speichern) dienen.

Bits 2 und 1:

Werden zur Zeit nicht benutzt.

Bit 0:

Hier wird bestimmt, welcher Prozessor mit dem Speicher verbunden wird – der 8502 (Bit 0 hat den Wert 1) oder der Z80 (Bit 0 = 0).

2.1.1.4 Das RAM-Configuration-Register (Register 6)

Dieses Register ist von größerer Bedeutung als das vorhin besprochene Mode-Register – in ihm wird Größe und Lage der Common Area im Adreßraum des 8502 bestimmt.

Das Prinzip, das zur Einführung einer Common Area führt, wurde in der Einleitung zum Kapitel »Speicherverwaltung« schon vorgestellt. Nur durch eine Common Area wird es möglich, Daten aus anderen RAM-Bänken – genauer gesagt: aus anderen Konfigurationen – zu lesen oder in solche zu schreiben.

Im C-128 können Common Areas sowohl am oberen Ende des Speichers (von $ffff abwärts) als auch am unteren Ende (von $0000 aufwärts) angelegt werden; auch eine Kombination ist möglich. Die normale Common Area unter Basic ist 1 KByte lang und liegt am unteren Speicherende bis $3ff einschließlich. Der ASE-Makroassembler benützt ebenfalls eine Common Area am unteren Ende des Speichers, diese ist allerdings 8 KByte lang und reicht somit bis $1fff einschließlich.

Die Bedeutung der einzelnen Bits des RAM-Konfigurationsregisters:

Bits 7 und 6:

Die ersten beiden Bits des Registers haben gleich schon einmal nichts mit der Common Area zu tun. Viele Chips des C-128 übernehmen auch Aufgabe für andere Bausteine – so ist es auch hier: Die beiden Bits legen die RAM-Bank fest, in der der 40-Zeichen-Bildschirm und die anderen Speicher des 40-Zeichen-Videocontrollers liegen sollen. Im Normalfall liegen diese Speicher des VIC-Chips in der RAM-Bank 0, der Textschirm liegt beispielsweise direkt hinter der Common Area von Adresse $400 bis $7ff, der Grafikschirm von $2000 bis $3fff, falls Grafik verwendet wird. Die Zuordnung der Bitkombinationen zur damit ausgewählten RAM-Bank entspricht derjenigen beispielsweise des Konfigurationsregisters.

Natürlich kann man unter Verwendung der normalen Betriebssystemroutinen nicht einfach die VIC-Speicher in eine andere Bank legen und dann erwarten, daß alles automatisch weiterläuft. Hierzu müßte man auf eigene Ein-/Ausgaberoutinen zurückgreifen.

Bits 5 und 4:

Werden nicht benutzt.

Bits 3 und 2:

Hier wird die Lage der Common Area bestimmt:

Bit 3	Bit 2	Common Area
1	1	Am oberen und unteren Speicherende
1	0	Nur am oberen Ende
0	1	Nur am unteren Ende
0	0	Gar keine Common Area

Die Kombination 01 ist der Einschaltzustand.

Bit 1 und 0:

Diese beiden Bits legen die Größe der Common Area fest. Sind Common Areas sowohl am oberen als auch am unteren Ende des Speichers erwünscht, so gilt die hier getroffene Größenwahl für beide Teilbereiche.

Bit 1	Bit 0	Größe der Common Area
1	1	16 KByte pro Teilbereich
1	0	8 KByte
0	1	4 KByte
0	0	1 KByte (Einschaltzustand)

2.1.1.5 Die Pagepointer (Register 7 bis 10)

Wie weiter vorn schon geschrieben wurde, kann man sich die MMU als Vermittler zwischen den Speicherbausteinen des C-128 und dem 8502-Prozessor vorstellen. Jeder Zugriff des Prozessors auf den Speicher wird von der MMU bewertet und je nach Einstellung der MMU-Register auf bestimmte Speicherbausteine geleitet. So wird es möglich, daß verschiedene RAM-Bänke verwendet werden können und daß ROMs Teile der RAM-Bänke überdecken können.

Die Tatsache, daß die MMU gleichzeitig mit zwei Prozessoren verbunden ist, deren Architekturen z. B. eine verschiedene Lage des In/Out-Bereiches erfordern, hat dazu geführt, daß die MMU noch weitergehende Fähigkeiten als die reine Zugangskontrolle zu verschiedenen Speicherbausteinen erhalten hat: Sie kann sogar Zugriffe der CPU auf bestimmte Adreßbereiche auf solche mit anderen Adressen umrechnen! Auf diese Weise wird es beispielsweise möglich, den wichtigen In/Out-Bereich $d000 - $dfff für den Z80-Prozessor in den unteren Speicherbereich zu spiegeln. Jeder Zugriff des Z80 auf den Adreßbereich, in dem dieser Prozessor die In/Out-Bausteine vermutet, wird durch die MMU physikalisch in den Bereich $d000 - $dfff übersetzt, jeder Zugriff auf den Bereich $d000 - $dfff wiederum wird auf das RAM umgeleitet, dessen Platz der gespiegelte In/Out-Bereich einnimmt. Der Prozessor selbst merkt von diesen Vorgängen – wie gewöhnlich – nichts.

Für den Programmierer des 8502 hat diese Fähigkeit der MMU Folgen, deren Auswirkungen bisher noch nicht ausgelotet worden sind: Er wird in die Lage versetzt, die beiden wichtigsten Systemspeicher des 8502 – die Zeropage und den Prozessorstack – an physikalisch andere Orte zu verlegen! Praktisch bedeutet dies, daß man beispielsweise jedem Programm eine eigene Zeropage oder einen eigenen privaten Stack zuteilen könnte. Es sei hier aber angemerkt, daß mir bisher kein kommerzielles Programm bekannt ist, das von dieser Option Gebrauch macht. Es eröffnet sich also noch ein weites Feld für eigene Experimente auf diesem Gebiet.

Für die Verlegung von Zeropage und Stack sind die sogenannten **Pagepointer** der MMU zuständig. Sie liegen an den Adressen $d507 und $d508 für die Zeropage und bei $d509 und $d50a für den Prozessorstack. Die beiden Pointer sind in der Reihenfolge Low/High zu lesen, d. h. für den Zeropage-Pointer stellt $d508 die niederwertige Angabe dar, $d509 die höherwertige; entsprechendes gilt für den Pointer des Prozessorstacks.

Die Verschiebung der beiden Speicherbereiche ist nicht stufenlos möglich, sondern nur in Schritten von 256 Byte; es ist also nicht möglich, beispielsweise den Stack auf die physikalische Adresse $1234 zu legen; statt dessen wäre es denkbar, die Adressen $1200 oder $1300 zu wählen. Der Low-Anteil des entsprechenden Pagepointers enthielte dann die Angabe der Nummer des ausgewählten 256-Byte-Blocks, im obigen Fall also $12 oder $13. Der High-Anteil des Pagepointers erlaubt die Auswahl der RAM-Bank, der dieser Block entnommen werden soll. In der Serienausführung des C-128 enthält das Highbyte des Pagepointers also eine Null für die RAM-Bank 0 oder eine Eins für die zweite RAM-Bank. Eine Besonderheit ist noch zu beachten, die in den Dokumentationen von Commodore nicht erwähnt wird: Eine Verlegung der Zeropage oder des Stacks in eine andere Bank ist nur dann möglich, wenn der zu verlegende Bereich nicht in einer Common Area liegt. Da die normalerweise unter Basic (und auch ASE) verwendete Common Area die Zeropage und den Stack umfaßt, ist eine Verlegung nur dann möglich, wenn man die Common Area vorher auflöst. Sehen Sie hierzu die Beschreibung des RAM-Configuration-Registers.

2.1.1.6 Das Versionsregister (Register 11)

Dies ist das letzte Register der MMU. Es dient keinen speziellen Einstellungen, sondern enthält lediglich zwei Angaben, die ein Programm abfragen kann:

Bits 7 bis 4:

Wieviele RAM-Bänke sind in den C-128 eingebaut? Die Zahl kann ein Anwendungsprogramm aus diesen Bits ablesen und sich entsprechend darauf einstellen. Angezeigt werden können maximal 16 RAM-Bänke, dies würde einer Speicherkapazität von 16 x 64 KByte = 1 Megabyte entsprechen. Die gegenwärtige MMU-Version ist allerdings nicht in der Lage, einen derart großen Speicher nutzbar zu machen.

In Verbindung mit den Speichererweiterungen auf 256 K oder 512 K kommt diesen Bits durchaus eine sinnvolle Funktion zu.

Bits 3 bis 0:

Hier kann die Versionsnummer der eingebauten MMU abgelesen werden. Sobald mehr als eine MMU-Version für den C-128 erhältlich sein wird, wird diese Ver-

sionsnummer unter Umständen von Bedeutung sein. Bis dahin kann man die angegebene Nummer 0 mit ruhigem Gewissen ignorieren.

2.1.2 Konfigurationsindizes und Speicherlandschaften

Wie Sie in der Zwischenzeit sicher festgestellt haben werden, ist die Beherrschung eines gebankten Systems, wie es der C-128 ist, auf Assemblerebene nicht unbedingt trivial. Kein Wunder also, daß die Software-Entwickler von Commodore sich etwas haben einfallen lassen, um die Speicherverwaltung zu vereinfachen: Zum einen haben sie die sogenannten Konfigurationsindizes ersonnen, zum anderen stellen sowohl Betriebssystem des C-128 als auch das Basic 7.0 Routinen zur Verfügung, die die gröbsten Aufgaben in Zusammenhang mit dem Banking übernehmen (Laden, Speichern, Aufrufen von Programmen in anderen Konfigurationen usw.).

In diesem Kapitel wollen wir die Konfigurationsindizes betrachten, die Routinen des Betriebssystems werden später erläutert.

Konfigurationsindizes sind, wie der Name schon besagt, Indizes. Als solche weisen sie in eine vom Betriebssystem bereitgestellte Tabelle von Konfigurationswerten. Man muß also scharf zwischen der Verwendung von Konfigurationsindizes und den eigentlichen Konfigurationen trennen, die anhand des Index aus einer Tabelle ermittelt werden. Diese Trennung wird leider im Handbuch des C-128 nicht immer genau genommen, wodurch sich immer wieder Begriffsverwirrungen gerade bei Einsteigern in die Assemblerprogrammierung ergeben.

Die unter Basic in verschiedenen Befehlen wie *bank* verwendeten Angaben einer Banknummer sind in Wirklichkeit Konfigurationsindizes. Wenn man im Handbuch liest, mit *bank 0* werde die Bank 0 eingeschaltet, ist dies eigentlich irreführend. Richtig und exakterweise müßte es heißen: *bank 0* schaltet die Konfiguration ein, die unter dem Konfigurationsindex 0 zu finden ist. Dies ist zwar – zufällig oder auch nicht – die Konfiguration, in der nur die RAM-Bank 0 für den Prozessor sichtbar ist, hat aber bei vielen Anwendern erhebliche Konfusion hervorgerufen, wenn sie von Basic auf Assembler umsteigen wollten. Ebenso wird mit *bank 15* nicht die RAM-Bank 15 ausgewählt, sondern die Konfiguration mit Nummer 15.

Sehen wir uns die Tabelle der Konfigurationen, die Indizes und die über die Indizes zu erreichenden Speicherlandschaften im einzelnen an:

Index	Tab. Wert	Konfiguration
0	$3f	Durchgehend RAM-Bank 0
1	$7f	Durchgehend RAM-Bank 1
2	$bf	Durchgehend RAM-Bank 2, falls vorhanden. Falls keine Erweiterung eingebaut, wie Index 0.
3	$ff	Durchgehend RAM-Bank 3, falls vorhanden, sonst wie Index 1.
4	$16	RAM-Bank 0, $c000-$ffff und $8000-$bfff interne ROM- Erweiterung, $d000-$dfff In/Out-Bereich
5	$56	Wie Index 4 für RAM-Bank 1
6	$96	Wie Index 4 für RAM-Bank 2
7	$d6	Wie Index 4 für RAM-Bank 3
8	$2a	Wie Index 4, nur sind externe ROM-Erweiterungen ausgewählt.
9	$6a	Wie Index 8 für RAM-Bank 1
10	$aa	Wie Index 8 für RAM-Bank 2
11	$ea	Wie Index 8 für RAM-Bank 3
12	$06	RAM-Bank 0, $c000-$ffff Kernal-ROM, $8000-$bfff interne ROM-Erweiterung, $d000-$dfff In/Out
13	$0a	RAM-Bank 0, $c000-$ffff Kernal-ROM, $8000-$bfff externe ROM-Erweiterung, $d000-$dfff In/Out
14	$01	RAM-Bank 0, $c000-$ffff Kernal-ROM, $4000-$bfff Basic-ROMs, $d000-$dfff Zeichensatz-ROM
15	$00	RAM-Bank 0, $c000-$ffff Kernal-ROM, $4000-$bfff Basic-ROMs, $d000-$dfff In/Out

Die Tabelle liegt im ROM ab $F7F0 und enthält die Tabellenwerte in Reihenfolge der Indizes.

2.2 Routinen des Betriebssystems in der Common Area

Das Betriebssystem stellt dem Assemblerprogrammierer verschiedene Routinen in der Common Area zur Verfügung, die diesem die eigene Programmierung von Unterprogrammen abnehmen, die beispielsweise das Laden und Abspeichern von Daten in anderen Konfigurationen bzw. anderen RAM-Bänken als der, in der das Maschinenprogramm selbst liegt, übernehmen. Diese Routinen sind ausdrücklich für die Verwendung seitens des Programmierers vorgesehen. Sie werden aus diesem

Grund – mit sehr wenigen Ausnahmen – weder durch das Betriebssystem selbst noch durch den Basic-Interpreter benutzt. Da diese Ausnahmen praktisch vernachlässigbar sind, kann der Programmierer die Routinen ohne jede Einschränkung nutzen, ohne befürchten zu müssen, das System dadurch in irgendeiner Weise negativ zu beeinflussen.

Ähnliche Routinen wie das Betriebssystem besitzt auch der Basic-Interpreter in der Common Area. Da diese jedoch auf die Bedürfnisse des Interpreters zugeschnitten sind, sind sie nicht allgemein genug gehalten, um für die Programmierung beliebiger Programme herhalten zu können. Auf diese Basic-Unterprogramme soll deshalb hier nicht näher eingegangen werden. Wer sich für diese Unterprogramme trotzdem interessiert und keine weitere Literatur zu Hilfe ziehen möchte, kann sie sich auch mit dem Disassembler beispielsweise des DBM, der in Kapitel 4 erläutert wird, ausdrucken lassen. Sie belegen den Adreßraum zwischen $380 und $3d1.

Die fünf Routinen, von denen dieses Kapitel handelt, liegen von $2a2 bis $2fb. Sie haben sämtlich einen bestimmten Namen, unter dem sie allgemein bekannt sind und sollen hier der Reihe nach vorgestellt werden:

FETCH ($2a2):

Diese Routine holt ein Byte aus einer beliebigen Konfiguration, also zum Beispiel aus einer beliebigen RAM-Bank. Sehen wir uns diese Routine zunächst disassembliert an:

```
02a2  ad  00  ff   lda  $ff00
02a5  8e  00  ff   stx  $ff00
02a8  aa           tax
02a9  b1  ff       lda  ($ff),y
02ab  8e  00  ff   stx  $ff00
02ae  60           rts
```

Ich glaube, die Wirkungsweise dieser Routine ist nicht schwer nachzuvollziehen: Als erstes wird der Inhalt des Konfigurationsregisters in den Akkumulator gelesen, um die alte Konfiguration zum Ende der Routine wiederherstellen zu können. Daraufhin wird die im X-Register übergebene Ziel-Konfiguration in das Konfigurationsregister geladen und die alte Konfiguration im X-Register gesichert. Von nun an ist die Konfiguration aktiv, die wir durch den im X-Register übergebenen Wert ausgewählt haben. Da das kleine Unterprogramm in der Common Area liegt, die von keinerlei Änderung der Konfiguration betroffen werden kann, ist kein Absturz des Prozessors zu befürchten, wie dies der Fall wäre, wenn wir ähnliches außerhalb der Common Area versuchten. Der nächste Befehl – *lda ($ff),y* – erscheint zunächst einmal ein wenig seltsam. Sollte es nur möglich sein, Daten zu laden, auf die ein ganz bestimmter Zeiger weist? Das wäre sicher keine sehr glückliche Lösung. Aber – erinnern wir uns daran, daß dieses Unterprogramm im RAM liegt! RAM kann beschrieben und

verändert werden, also kann man auch dieses kleine Stückchen Programmkode verändern, das den Pointer des lda-Befehls festlegt (Adresse $2a2). Schreibt man an diese Adresse, die sinnfälliger Weise **fetvec** genannt wird, beispielsweise eine $24, so lädt man ein Byte, auf das der Pointer $24/$25 zeigt – natürlich wird der Inhalt des Y-Registers noch zur im Pointer enthaltenen Adresse hinzugezählt, aber ich denke, das Prinzip ist klargeworden. Eine solche Vorgehensweise, daß ein Programm eines seiner Unterprogramme während des Programmlaufs verändert, wird **Selbstmodifikation** genannt, was man ungefähr mit Selbstveränderung übersetzen könnte. Programmkode, der sich selbst verändert, wird auch **selbstmodifizierender Kode** genannt. Selbstmodifizierender Kode wird allgemein nicht als feiner Programmierstil angesehen, aber – wie man sieht – gibt es Situationen, in denen dies die einzige Methode darstellt, auf gedrängtem Raum effiziente und kurze Programme oder Unterprogramme zu schreiben. Gehen wir weiter in der Beschreibung der fetch-Routine: Nachdem das Byte, auf das der gewählte Zeropage-Pointer zusammen mit dem Y-Register zeigt, in den Akkumulator geladen ist, wird als letzter Schritt die ursprüngliche Konfiguration wiederhergestellt und das Unterprogramm kehrt zum aufrufenden Programm zurück.

Fassen wir die einzelnen Schritte noch einmal zusammen, die notwendig sind, um die *fetch*-Routine in einem eigenen Programm zu verwenden:

Schritt 1:

Auswahl eines Pointers in der Zeropage, den wir für das Laden verwenden wollen. Das Lowbyte dieses Pointers – nicht das Lowbyte des Inhalts dieses Pointers – wird in den *fetvec* geschrieben.

Schritt 2:

Das Y-Register wird auf den für unsere Zwecke passenden Wert gebracht. In einer Schleife könnte dieser Schritt zum Beispiel aus einer einmaligen Initialisierung des Registers und anschließender Inkrementierung bestehen.

Schritt 3:

Das X-Register wird mit dem Konfigurationswert geladen, der die Konfiguration beschreibt, aus der wir laden wollen. Dieser Schritt ist bei jedem neuen Ladevorgang notwendig, denn *fetch* zerstört den Inhalt des X-Registers.

Schritt 4:

Aufruf von *fetch*. Nach Rückkehr von diesem Unterprogramm befindet sich das spezifizierte Byte im Akkumulator und kann weiter bearbeitet werden.

Das Ganze noch einmal in Form eines kleinen Beispielprogramms; der Kommentar zu jeder Zeile steht hinter einem Semikolon, wie es auch in einem Assembler-Quelltext der Fall ist. Das Beispiel soll ein Byte von Adresse $1234 aus der RAM-Bank 1 lesen:

```
lda   #$12  ;  Highbyte der Zieladresse
sta   $25   ;  In das Highbyte des Pointers schreiben
lda   #$34  ;  Dasselbe mit dem
sta   $24   ;  Lowbyte der Adresse
lda   #$24  ;  Lowbyte des Pointers
sta   $2aa  ;  im fetvec speichern
ldy   #$00  ;  Einen Offset brauchen wir nicht
ldx   #$7f  ;  Konfiguration für RAM-Bank 1
jsr   $2a2  ;  fetch aufrufen
...         ;  Hier geht's dann irgendwie weiter
```

Diese Vorgehensweise sieht nur auf den ersten Blick etwas kompliziert aus; sie wird Ihnen schon bald in Fleisch und Blut übergegangen sein. Zudem ist der Fall, daß Werte aus anderen RAM-Bänken geladen werden müssen, nicht so häufig, wie man dies vielleicht meinen sollte. Das soll aber nicht bedeuten, daß diese Routinen etwa unwichtig seien!

STASH ($2af):

Diese Routine leistet genau das gleiche wie die *fetch*-Routine, sieht man davon ab, daß sie statt ein Byte in den Akkumulator zu laden ein Byte in einer anderen Konfiguration abspeichert. Da alles sonstige ganz entsprechend zu *fetch* verläuft, denke ich, es genügt, *stash* disassembliert mit Kommentar zu zeigen:

```
02af  48          pha           ;  Merke zu speicherndes Byte
02b0  ad 00 ff    lda  $ff00    ;  Rette Original-Konfiguration
02b3  8e 00 ff    stx  $ff00    ;  Ziel-Konfiguration ein
02b6  aa          tax           ;  X = Original-Konfiguration
02b7  68          pla           ;  Zu speicherndes Byte holen
02b8  91 ff       sta  ($ff),y  ;  und abspeichern
02ba  8e 00 ff    stx  $ff00    ;  Originalzustand herstellen
02bd  60          rts           ;  Ende
```

Genau wie *fetch* besitzt auch *stash* eine Speicherstelle, in die der verwendete Pointer für das Abspeichern eingetragen werden muß. Dieser heißt analog zu *fetvec* **stavec** und liegt an der Adresse **$2b9**, wie man auch aus dem Disassembler-Listing schliessen kann.

CMPARE ($2be):

Die dritte Routine der gleichen Art. *cmpare* vergleicht ein im Akkumulator übergebenes Byte mit dem Inhalt einer Adresse in einer durch das X-Register angegebenen Konfiguration. Der Pointer wird in dieser Routine an Adresse $2c8 eingetragen, der Name dieser Speicherstelle ist *cmpvec*. Das kommentierte Disassemblerlisting:

```
02be  48          pha          ;  Vergleichswert merken
02bf  ad 00 ff    lda  $ff00   ;  Rette Original-Konfiguration
02c2  8e 00 ff    stx  $ff00   ;  Setze Ziel-Konfiguration
02c5  aa          tax          ;  X = Original-Konfiguration
02c6  68          pla          ;  Vergleichswert holen
02c7  d1 ff       cmp  ($ff),y ;  Vergleichen
02c9  8e 00 ff    stx  $ff00   ;  Original-Konfiguration ein
02cc  60          rts          ;  Fertig
```

JMPFAR ($2e3) und JSRFAR ($2cd):

In den beiden folgenden Routinen geht es darum, Programme in anderen Konfigurationen anzuspringen *(jmpfar)* bzw. ein Unterprogramm in einer anderen Konfiguration aufzurufen *(jsrfar)*. Dabei ist weniger daran gedacht, daß der Anwender seine Programme über die RAM-Bänke verstreuen könnte als an die Verwendung von ROM-Routinen. Einfachster Fall wäre beispielsweise, daß das aufrufende Programm an einer Adresse liegt, die vom ROM, das die benötigte Routine enthält, überdeckt werden würde. In einem solchen Fall wäre ein direkter Aufruf des Unterprogramms natürlich nicht möglich – sobald das ROM eingeschaltet würde, wäre das aufrufende Programm für den Prozessor wieder einmal unsichtbar geworden. Also geht man wieder den Umweg über die Common Area: Man ruft eine Routine auf, die garantiert nicht von einem Konfigurationswechsel betroffen werden wird, dieses Programm seinerseits stellt die Ziel-Konfiguration her, ruft das benötigte Unterprogramm auf, stellt die Original-Konfiguration wieder her und kehrt danach zum aufrufenden Programm zurück.

Um ein Programm in einer anderen Konfiguration sinnvoll aufrufen zu können, muß zumindest die Möglichkeit bestehen, eine ganze Reihe von Parametern an das aufgerufene Programm zu übergeben – daneben muß natürlich auch die Adresse des aufzurufenden Programms und die Konfiguration, in der es liegt, irgendwie angegeben werden. Diese ganzen Angaben werden in der Zeropage gemacht. Die nachfolgende Tabelle zeigt, wo:

Adresse	Parameter
2	Konfigurationsindex der Konfiguration, in der das Programm oder Unterprogramm zu finden ist.
3	Highbyte der Adresse des Programms.
4	Lowbyte der Adresse. Beachten Sie, daß die Reihenfolge des Low- und des Highbytes hier genau anders herum ist als normalerweise, etwa in einem Vektor!
5	Zu übergebender CPU-Status.
6	Zu übergebender Wert des Akkumulators.
7	X-Register
8	Y-Register
9	Stapelzeiger

Alle hier angegebenen Registerwerte außer dem Stapelzeiger befinden sich beim Eintritt in das aufzurufende Programm an den vorgesehenen Stellen. Falls Sie ein Unterprogramm mit *jsrfar* aufrufen, erhalten Sie bei der Rückkehr auch die entsprechenden Werte wieder zurückgeliefert, darunter auch den Stapelzeiger. Der Wert des Y-Registers befindet sich bei der Rückkehr außerdem im Prozessorregister selbst.

Die beiden Disassemblerlistings:

JSRFAR:

```
02cd  20 e3 02     jsr $2e3     ;  Aufruf jmpfar
02d0  85 06        sta $06      ;  Registerwerte A, X, Y nach
02d2  86 07        stx $07      ;  Rückkehr in der Zeropage
02d4  84 08        sty $08      ;  ablegen
02d6  08           php          ;  Status über den Umweg über
02d7  68           pla          ;  den Stack in die Zeropage
02d8  85 05        sta $05      ;  schreiben
02da  ba           tsx          ;  Stapelzeiger -> X
02db  86 09        stx $09      ;  und abspeichern
02dd  a9 00        lda #$00     ;  Original-Konfiguration
02df  8d 00 ff     sta $ff00    ;  wiederherstellen
02e2  60           rts          ;  und Ende ...
```

Zu *jsrfar* wird eine weitere Angabe innerhalb des Kodesegments selbst erwartet: die Angabe, in welcher Konfiguration eigentlich das Programm zu finden ist, von dem aus *jsrfar* gerufen wird. Dazu wird der Konfigurationswert selbst, nicht ein Index auf einen solchen, an die Adresse **$02d = farvec** geschrieben. Bei *jmpfar*, dem Sprung ohne Rückkehr, ist eine solche Angabe selbstverständlich nicht erforderlich.

JMPFAR:

```
02e3  a2 00        ldx #$00   ;  Adresse und Status auf den
02e5  b5 03        lda $03,x  ;  Stack legen (für rti)
02e7  48           pha
02e8  e8           inx
02e9  e0 03        cpx #$03
02eb  90 f8        bcc $02e5
02ed  a6 02        ldx $02    ;  Zum Konfigurationsindex
02ef  20 6b ff     jsr $ff6b  ;  den Konfigurationswert holen
02f2  8d 00 ff     sta $ff00  ;  und aktivieren
02f5  a5 06        lda $06    ;  Registerwerte laden
02f7  a6 07        ldx $07
02f9  a4 08        ldy $08
02fb  40           rti        ;  Aufruf des Programms ...
```

Sehr wichtig und mit Auswirkungen auf den Einsatz beider Unterprogramme ist, daß *jmpfar* eine Routine aus dem Kernal-ROM benutzt, um den zugehörigen Konfigurationswert zum in der Zeropage übergebenen Konfigurationsindex zu ermitteln (*jsr* $ff6b). Aus dieser Tatsache folgt, daß, während *jsrfar* oder *jmpfar* ablaufen, das Kernal-ROM sichtbar sein muß! Leider befinden sich »unterhalb« des Kernal-ROMs in den beiden RAM-Bänken keine Kopien der Routine *$ff6b*, obwohl z. B. die Interrupt-Einsprungpunkte in die RAMs kopiert werden. Es gibt jetzt zwei Möglichkeiten: Entweder holen wir in unserem Programm das Kopieren nach, oder wir beschränken uns zunächst darauf, *jsrfar* und *jmpfar* nur bei eingeschaltetem Kernal-ROM aufzurufen. Ich denke, in Ihrem Sinn zu handeln, wenn ich zeige, wie man *$ff6b* kopiert, so daß man sich danach an keine Einschränkungen halten muß. Das kleine Kopierprogramm sollten Sie bei Bedarf in Ihre Programme mit aufnehmen und vor dem ersten Aufruf von *jsrfar* oder *jmpfar* einmal starten.

```
copy    lda  #$24     ;  $24/ $25 als Zeiger für
        sta  stavec   ;  stash vorsehen
        lda  #$26     ;  $26/ $27 als Zeiger für
        sta  fetvec   ;  fetch
        lda  #$3f     ;  Konfiguration RAM-Bank 0
        jsr  copy1    ;  Kopieren in Bank 0
        lda  #$7f     ;  Konfiguration RAM-Bank 1

copy1   sta  conf     ;  Konfiguration merken
        lda  #$ff     ;  stash-Zeiger auf Adresse
        sta  $25      ;  $ff6b richten
        lda  #$6b
        sta  $24
```

```
        ldy  #0        ;  Schleifenzähler initialisieren
cp1     lda  jmper,y   ;  Byte aus Tabelle >jmp $ff45<
        ldx  conf      ;  In RAM-Bank
        jsr  stash     ;  abspeichern
        iny            ;  Insgesamt sind es drei Byte
        cpy  #3        ;  Alle kopiert?
        bcc  cp1       ;  Nein: ->

        lda  #$f7      ;  fetch - Zeiger auf Adresse
        sta  $27       ;  $ffec richten, dort befindet sich
        lda  #$ec      ;  die eigentliche Routine im ROM
        sta  $26
        lda  #$ff      ;  stash - Zeiger auf $ff45 richten,
        sta  $25       ;  dahin legen wir unsere Kopie
        lda  #$45
        sta  $24

        ldy  #0        ;  Noch eine Schleife ...
cp2     ldx  #0        ;  Konfiguration mit Kernal-ROM
        jsr  fetch     ;  Byte aus dem ROM holen
        ldx  conf      ;  und in die RAM-Bank kopieren
        jsr  stash
        iny            ;  Haben wir schon alle?
        cpy  #20
        bcc  cp2       ;  Nein: ->
        rts            ;  Ja: Kopie erstellt ...

conf    .byte 0        ;  Speicher für Konfiguration
jmper   jmp  $ff45     ;  Tabelle für den Sprung
```

Ich glaube, dieses Beispiel ist auch gut geeignet, um noch einmal die Anwendung von *fetch* und *stash* zu demonstrieren. Das Verständnis dieser Kopierroutine dürfte nicht schwer fallen. Das einzige, was vielleicht nicht sofort ersichtlich ist, ist, warum unsere Kopie nach $ff45 gelegt wird und warum wir nicht ab der Adresse $ff6b mit dem Kopieren beginnen. Dazu ist zu bemerken, daß der RAM-Bereich oberhalb der Duplikate der MMU-Register prinzipiell nicht von Anwenderprogrammen genutzt wird. In diesem Bereich liegen einige sehr kurze Kopien von wichtigen Routinen aus dem Kernal-ROM, auf die ich hier nicht weiter eingehen werde. Der Bereich ab $ff45 ist frei und ausreichend groß, um unsere Kopie aufzunehmen. An der Adresse $ff6b befindet sich im ROM lediglich ein Sprung zur eigentlichen Routine bei $f7ec. Dieser Sprungbefehl ist einer in einer ganzen Reihe von Sprüngen, die in einer sogenannten **Sprungleiste** am Ende des Kernal-ROMs zusammengefaßt sind. Einige weitere Routinen dieser Sprungleiste werden wir weiter unten noch kennenlernen.

Noch ein Beispiel für die Anwendung von *jsrfar*, damit die Verwendung dieser Routine noch deutlicher sichtbar wird. Dazu will ich die Routine *chrout = $ffd2* anspringen; unser Kopierprogramm soll schon gestartet worden sein, damit die Lage des Hauptprogramms keine Rolle mehr spielt. *chrout* ist ebenfalls eine Routine aus der erwähnten Sprungleiste und gibt ein im Akkumulator übergebenes Zeichen z. B. auf den Bildschirm aus.

```
exmpl   lda  #$ff      ;  Highbyte der Adresse von
        sta  $03       ;  chrout in die Zeropage
        lda  #$d2      ;  und das Lowbyte ...
        sta  $04
        lda  #15       ;  Ein Index, in dessen Konfig.
        sta  $02       ;  das Kernal-ROM sichtbar ist
        lda  zeichen   ;  Das auszugebende Zeichen
        sta  $06       ;  abspeichern
        jsr  jsrfar    ;  und jsrfar rufen
        ...            ;  Hier geht's irgendwie weiter
```

Routinenversionen mit Konfigurationsindizes:

Zu jeder der Unterprogramme *fetch, stash* und *cmpare* gibt es in der besagten Sprungleiste des Kernal-ROMs einen Einsprungpunkt in eine Routine, die ihrerseits die Routinen der Common Area aufruft, aber vorher die entsprechenden Speicherstellen *fetvec*, stavec, cmpvec mit einem übergebenen Pointer initialisiert und einen ebenfalls übergebenen Konfigurationsindex in den zugehörigen Konfigura-tionswert umrechnet. Nachteil dieser Routinen ist zweifellos, daß sie nur verwendet werden können, wenn das Kernal-ROM sichtbar ist. Von Vorteil dagegen ist, daß man sich einige Programmschritte sparen kann, wenn das ROM sowieso einge-blendet ist und natürlich, wenn die Ziel-Konfiguration, die man erreichen will, unter einem Index zu finden ist – für die beiden RAM-Bänke ist dies der Fall.

Alle drei Routinen der Sprungleiste erwarten die Angabe des Pointers und den Konfigurationsindex in der gleichen Anordnung: Lowbyte des Pointers im Akkumulator, Index im X-Register. Das Y-Register wird nicht verändert.

Adresse	Name	Ersatz für ...
$ff74	indfet	fetch
$ff77	indsta	stash
$ff7a	indcmp	cmpare

Für *jsrfar* und *jmpfar* existieren ebenfalls zwei Einsprungpunkte in der Sprungleiste: $ff6b und $ff6e. Die Sprünge, die sich an diesen beiden Adressen befinden, führen jedoch sofort zu den entsprechenden Adressen in der Common Area, dem Programmierer wird also keine Arbeit durch die Benutzung dieser Adressen abgenommen.

Vorteil der Verwendung der Kernal-Adressen: Falls sich Commodore je entschließen sollte, die Lage von *jsrfar* und *jmpfar* innerhalb der Common Area zu verändern, würden die Adressen der Sprungleiste trotzdem konstant bleiben. Wollen Sie erreichen, daß Ihre Programme auf jeden Fall auch auf zukünftigen Versionen des C-128 laufen, empfiehlt sich daher immer der Einsatz der Sprungleisten-Routinen anstelle des direkten Weges in die Common Area. Wie groß die Wahrscheinlichkeit für eine Veränderung der Common Area tatsächlich ist, sollte jeder selbst beurteilen – ich persönlich halte sie für sehr gering.

2.3 Ein Platz für Maschinenprogramme

Langsam aber sicher nähern wir uns dem Ziel, das erste eigene Maschinenprogramm entwickeln zu können. Was uns in diesem Kapitel beschäftigen soll, befaßt sich schon mit der Frage, wo im Speicher noch Platz für so ein Maschinenprogramm sein könnte und welche Adressen in der Zeropage ein eigenes Programm verwenden kann, ohne in die Fallstricke des Systems zu geraten.

Zeropage:

Grundsätzlich muß man davon ausgehen, daß die Zeropage als einer der wichtigsten Speicherbereiche des Prozessors sowohl vom Betriebssystem als auch vom Basic-Interpreter ausgiebig genutzt wird. Da jedoch kein – oder so gut wie kein – längeres Programm ohne zumindest einige wenige Zeropagespeicher auskommt und damit auch ein von uns geschriebenes Programm nicht auf die Benutzung der Zeropage verzichten kann, muß man sich die Zeropage etwas näher betrachten, um freie Stellen für eigene Zwecke zu finden. Zunächst führt die Suche zu fünf Speicherzellen, die von den Commodore-Entwicklern mit Absicht frei gehalten worden sind. Da diese fünf Byte nicht unbedingt das Volle sind, aus dem man schöpfen könnte, verwenden wir im folgenden zusätzlich einige Adressen, die dem Basic-Interpreter als reine Zwischenspeicher dienen, also keine für den Interpreter lebenswichtige Daten enthalten. Nur diese beiden Arten von Zeropagespeichern wollen wir für die Beispielprogramme benutzen. Eine vollständige Analyse der Zeropage – welche Adressen man gefahrlos benutzen kann und welche Speicherzellen welche Bedeutung als Systemspeicher haben, die man besser nicht verändern sollte – soll anderen, weiterführenden Büchern vorbehalten bleiben. Solche Bücher werden am Ende dieses Kapitels genannt.

Folgende Zeropageadressen können benutzt werden (falls Einschränkungen gemacht werden müssen, sind diese angegeben):

Adressen	Einschränkungen
$24 – $2b	Diese Speicherzellen werden von Basic genutzt. Falls Sie Routinen des Basic-ROMs benutzen, können die Inhalte dieser Adressen verändert werden.
$50 – $53	Wie oben
$5e – $61	Wie oben
$63 – $72	Wie oben
$fa – $fe	Keine Einschränkungen

Diese stattliche Anzahl von 37 Byte an Zeropagespeicher dürfte für die meisten Anwendungen schon mehr als ausreichend sein. Beachten Sie aber bitte, daß ausschließlich die Adressen $fa – $fe überhaupt nicht benutzt sind. Nur in diesen Speicherzellen können Sie Werte unterbringen, die garantiert unverändert bleiben, auch wenn Sie in Ihrem Maschinenprogramm Unterprogramme aus den Basic-ROMs aufrufen. Unterprogramme aus dem Betriebssystem nehmen grundsätzlich keinen Einfluß auf die aufgeführten Speicherzellen.

Maschinenprogramme:

Nachdem die Frage nach den für die Assemblerprogrammierung des 8502 so wichtigen Zeropageadressen soweit geklärt ist, bleibt natürlich noch zu überlegen, wohin unser Maschinenprogramm selbst gelegt werden kann, ohne daß es durch das Betriebssystem oder durch den Interpreter überschrieben werden kann und ohne daß es selbst wichtige Speicherbereiche zerstört.

Prinzipiell steht dafür fast der gesamte Adreßraum zur Verfügung; grob gesprochen sind dies die Adressen $1300 bis $feff in der RAM-Bank 0 und $400 bis $feff in der Bank 1. In der Praxis bieten sich aber kleinere Speicherbereiche in der RAM-Bank 0 besonders für die Aufnahme eigener Maschinenprogramme an, wenn man einmal davon ausgeht, daß es sich bei Ihren ersten Programmen nicht gerade um speicherfressende Riesenprogramme handeln wird. In Frage kommen folgende Adressen:

$b00 -$bff:

Diese 256 Byte werden normalerweise als Pufferspeicher bei Kassettenoperationen und beim Boot-Vorgang (Selbststart eines Programmes von Diskette) benutzt. Solange nicht vom Datenrekorder gelesen oder auf eine Datenkassette geschrieben wird, kann man diesen Bereich unbesorgt für kleinere Maschinenprogramme verwenden.

$c00 -$dff:

Ähnliches gilt für diesen 512 Byte großen Speicherbereich, der normalerweise für RS232-Übertragungen benutzt wird. Da für die Nutzung von RS232-Operationen Zusatzgeräte am C-128 notwendig sind, kann man davon ausgehen, daß die beiden

RS232-Puffer in der Regel unbenutzt sind und deshalb einen sicheren Aufbewahrungsort für unsere Programme darstellen. Dieser Bereich wird für die überwiegende Anzahl der noch folgenden Beispielprogramme benutzt werden.

$e00 – $fff:

Hier bringt der Basic-Interpreter Spritedefinitionen unter. Solange keine Sprites verwendet werden, sind auch diese 256 Byte ein sicherer Speicher.

$1300 – $1bff:

Dies ist der größte freie Speicherbereich, der ausdrücklich für eigene Anwendungen freigehalten wurde. Er ist 2304 Byte lang und dürfte damit auch noch Platz für etwas größere Programme bieten. Beachten Sie bitte, daß der ASE-Assembler den Platz unter $1400 als Zwischenspeicher bei manchen Operationen benutzt. Falls Sie ein Maschinenprogramm schreiben wollen, daß bei $1300 beginnen soll, sollten Sie den Objektkode des Maschinenprogramms zunächst an einer anderen Adresse ablegen lassen und dann erst nach $1300 verschieben. Hierzu finden Sie weitere Informationen unter dem Stichwort *Codeverschiebung* in der Beschreibung des ASE-Assemblers im nächsten Kapitel.

2.4 Lesehinweise

Jetzt ist es an der Zeit, daß Sie die Lektüre dieses Kapitels zeitweilig unterbrechen, um sich zumindest mit den Grundlagen der Bedienung des ASE-Assemblers vertraut zu machen. Die im nächsten Abschnitt folgenden Beispielprogamme werden vollständige Assembler-Quelltexte sein, die sie mit Hilfe des Assemblers in der Praxis nachvollziehen können.

Die Beispielprogramme nutzen mit Absicht nicht alle Fähigkeiten des Assemblers, um Sie in möglichst kurzer Zeit zur eigenen Programmierung zu führen. Beispiele für die weiterführenden Features des Assemblers finden Sie innerhalb der Beschreibung des Assemblers selbst.

2.5 Ein- und Ausgabe auf Assemblerebene

Jedes Computerprogramm zerfällt immer in drei Teilaufgaben: Die Eingabe von Daten, die Verarbeitung der eingegebenen Daten und schließlich die Ausgabe des Ergebnisses der Verarbeitung. Teile dieses Schemas können fehlen.

Aufgabe des vorliegenden Abschnittes soll es sein, Ihnen die Programmierung des C-128 zur Dateneingabe und -ausgabe nahezubringen, soweit dies im Rahmen dieses Buches möglich ist. Die Datenverarbeitung selbst ist nicht allgemein genug zu fassen, als daß sie in Form einer kurzen Abhandlung zu beschreiben wäre.

Alle vorgestellten Routinen zur Ein- und Ausgabe sind – mit einer Ausnahme – Kernal-Routinen aus der schon des öfteren erwähnten Sprungleiste des Kernals. Sie ermöglichen die Kommunikation mit folgenden Geräten:

* Bildschirm (40 und 80 Zeichen pro Zeile)
* Tastatur
* Datenrekorder
* Drucker (sofern am IEC-Bus angeschlossen)
* Floppy

Diese Geräte decken 99 Prozent aller Ein- und Ausgaben ab, die der Assemblerprogrammierer in seiner »täglichen Arbeit« benötigt. Alle anderen Eingabe- und Ausgabegeräte erfordern eine Zusatzhardware, die am C-128 angeschlossen werden muß.

2.5.1 Tastatur und Bildschirm

Tastatur und Bildschirm sind die – neben der Floppy – wohl am häufigsten benutzten Geräte zur Dateneingabe und -ausgabe. Das Betriebssystem unterstützt die beiden Geräte im wesentlichen in sechs Routinen:

Adresse	Name	Zweck
$c02d	window	Bildschirmfenster definieren
$ff7d	primm	Ausgabe eines Textes
$ffcf	basin	Eingabe eines Zeichens (wird auch zuweilen *chrin* genannt)
$ffd2	bsout	Ausgabe eines Zeichens (anderer Name *chrout*)
$ffed	scrorg	Breite eines Bildschirmfensters holen
$fff0	plot	Cursor setzen/Cursorposition holen

2.5.1.1 Ausgabe auf den Bildschirm

Zu den Aufgaben, die bei Ausgaben auf den Bildschirm anfallen, gehört außer der eigentlichen Ausgabe von Texten oder Zeichen auch, daß sich ein Programm über den Cursorort informieren kann, den Cursorort bestimmen kann und daß es die zur Verfügung stehende Breite des aktuell eingestellten Bildschirmfensters ermitteln

kann, um seine Ausgaben auf diese Breite einstellen zu können. Diese »Nebenaufgaben« wollen wir als erstes behandeln:

WINDOW ($c02d):

window ermöglicht die Einstellung eines Bildschirmfensters aus einem Assemblerprogramm heraus. Jedes Fenster wird definiert durch die Position der linken oberen und der rechten unteren Ecke des Fensters. Je eine dieser Positionen pro Aufruf wird durch *window* festgelegt. Die Unterscheidung, welche Ecke gesetzt werden soll, wird *window* durch das Carry-Flag mitgeteilt: Gelöschtes C-Flag läßt den Programmierer die linke obere Ecke definieren, entsprechend setzt *window* bei gesetztem Carry die rechte untere Ecke:

Linke obere Ecke setzen:

```
clc          ;  »links oben« wählen
lda zeile    ;  Zeilennummer
ldx spalte   ;  Spalte
jsr window   ;  linke obere Ecke festlegen
```

Rechte untere Ecke setzen:

```
sec          ;  »rechts unten« wählen
lda zeile    ;  Zeilennummer
lda spalte   ;  Spaltennummer
jsr window   ;  rechte untere Ecke festlegen
```

Ein Einschalten des In/Out-Bereichs ist für den Einsatz dieser Routine nicht unbedingt erforderlich, da sie lediglich Verwaltungsaufgaben erfüllt. Sie nimmt keinen Zugriff auf einen der In/Out-Chips. Fenster werden auf ökonomische Weise wieder aufgelöst, indem man über die später beschriebenen Ausgaberoutinen ein doppeltes *HOME* ausgibt.

SCRORG ($ffed):

Diese Routine liefert die Breite des aktuell eingestellten Bildschirmfensters zurück. Falls kein Fenster verwendet wird, ermittelt man so die Breite des eingestellten Bildschirms (40 oder 80 Zeichen). Falls Ihr Programm spezielle Ausgaben vornimmt, die auch dann lesbar bleiben sollen, wenn der Benutzer des Programms verschiedene Fensterformate verwendet, sollten Sie diese Ausgaben in Abhängigkeit von der Fensterbreite programmieren. Nach Aufruf von *scrorg* befindet sich die aktuelle Fensterbreite im X-Register, der Akkumulator enthält die für den gewählten Bildschirm maximale Fensterbreite.

PLOT ($fff0):

Aufgabe dieser Routine ist es, den Cursorort zu ermitteln bzw. einen neuen Cursorort zu bestimmen. Welche Funktion *plot* ausführt, ist abhängig vom Zustand des Carry-Flags beim Einsprung in die Routine: Ist das C-Flag gelöscht, wird der Cursor gesetzt, ist das C-Flag gesetzt, wird die Cursorposition geholt. Die Übergabe der Zeilen- und Spaltennummern erfolgt in den Registern X und Y.

Cursor setzen:

```
lda   #0       ;  In/Out-Bereich einschalten!
sta   $ff00    ;
clc            ;  Lösche C-Flag, um »Setzen« zu wählen
ldx   zeile    ;  Zeilennummer in das X-Register
ldy   spalte   ;  Spaltennummer in das Y-Register
jsr   plot     ;  Cursor setzen
```

Cursor holen:

```
lda   #0       ;  ...
sta   $ff00    ;
sec            ;  Setze C-Flag, um »Holen« zu wählen
jsr   plot     ;  Hole Position
stx   zeile    ;  X enthält Zeilennummer
sty   spalte   ;  Y enthält Spaltennummer
```

Das Einschalten des In/Out-Bereiches wurde in diesen beiden Beispielroutinen noch einmal ausdrücklich gezeigt, um zu demonstrieren, daß dieses **unbedingt** erfolgen muß, bevor die Routine aufgerufen wird. Sollten Sie den In/Out-Bereich schon an anderer Stelle aktiviert haben, entfallen selbstverständlich die beiden ersten Befehle *lda* und *sta*. In Zukunft wird auf die Rolle des In/Out-Bereiches nicht mehr eingegangen werden.

Wichtig, wenn Sie den Cursor an eine bestimmte Position setzen wollen: Versuchen Sie, den Cursor an eine Stelle zu setzen, die die aktuellen Fenstergrenzen über- oder unterschreitet, so liefert *plot* ein gesetztes C-Flag zurück, zum Zeichen, daß ein Fehler während der Ausführung der Routine aufgetreten ist.

Wichtig sowohl beim Holen als auch beim Setzen des Cursors: Die Zeilen- und Spaltennummern werden relativ zu den Fenstergrenzen angegeben. Liegt die obere linke Ecke des Bildschirmfensters beispielsweise bei Zeile 4 und Spalte 10, setzen Sie den Cursor mit

```
ldx    #3
ldy    #6
sec
jsr    plot
```

an die absolute Position (7, 16). Holen Sie die Cursorposition, erhalten Sie ebenfalls relative Angaben zurück. Wollen Sie die absolute Cursorposition in Erfahrung bringen bzw. den Cursor an eine absolute Position setzen, bleibt keine andere Möglichkeit, als auf vier Speicherzellen des Kernal-Editors in der Zeropage zurückzugreifen:

Adresse	Inhalt (absolute Angaben)
$e4	Unterste Zeile des Fensters
$e5	Oberste Zeile des Fensters
$e6	Kleinste Spaltennummer des Fensters (linke Grenze)
$e7	Höchste Spaltennummer (rechte Grenze)

Durch geeignete Addition oder Subtraktion der in diesen Speicherzellen enthaltenen, absolut angegebenen Fenstergrenzen zu oder von der Cursorposition kann man relative und absolute Werte ineinander umwandeln.

Als letztes folgen nun die eigentlichen Ausgaberoutinen:

BSOUT ($ffd2):

Die Abkürzung *bsout* stammt von *basic output*, was nicht einem Zusammenhang mit dem Basic-Interpreter entspringt, sondern zu Deutsch etwa heißen würde »allgmeine Ausgaberoutine«. *bsout* wird nicht nur zur Ausgabe von Zeichen auf den Bildschirm verwandt, sondern dient als allgemeine Routine, mit der beispielsweise auch Ausgaben in eine Datei oder zum Drucker erfolgen können (s. u.). An dieser Stelle interessiert nur die Ausgabe auf den Bildschirm. Die Benutzung der *bsout*-Routine ist sehr einfach: Man übergibt im Akkumulator das auszugebende Zeichen und ruft *bsout* über einen *jsr*-Befehl auf. Beide Indexregister werden durch *bsout* auf dem Stack gesichert, so daß sie nach Rückkehr in das Hauptprogramm wieder ihre ursprünglichen Werte vor dem Aufruf besitzen.

```
lda   zeichen    ;  auszugebendes Zeichen in den Akku
jsr   bsout      ;  bsout aufrufen
```

bsout liefert **keine** Fehlermeldung an das aufrufende Programm, falls während der Ausgabe Fehler auftreten. Dies könnte z. B. der Fall sein, wenn Sie mit *bsout* in eine Datei schreiben und die Diskette überläuft. Zu diesem Thema finden Sie aber weiter unten noch Näheres.

PRIMM ($ff7d):

primm ist eine gelungene Anwendung der *bsout*-Routine. *primm* erlaubt die Ausgabe von Texten aus einem laufenden Programmtext heraus, ohne daß der Text in einem vom Programmkode getrennten Datenfeld abgelegt werden muß. Ein Aufruf von *primm* sieht in Prinzip folgendermaßen aus:

```
...          ; irgendwas vorher
jsr primm    ; Ausgaberoutine rufen
.byte "Text mit Null am Ende",0
...          ; hier geht's ohne Unterbrechung weiter
               im Programm ...
```

Die *primm*-Routine ist so elegant, daß sie unten etwas näher unter die Lupe genommen werden soll:

```
fa17  48        pha          ;  Akku auf Stack retten
fa18  8a        txa          ;  Gleiches
fa19  48        pha          ;  mit dem X-Register
fa1a  98        tya          ;  und
fa1b  48        pha          ;  mit dem Y-Register
fa1c  a0 00     ldy  #$00    ;  Schleifenzähler
fa1e  ba        tsx          ;  Hole Stapelzeiger nach X
fa1f  fe 04 01  inc  $104,x  ;  Addiere 1 zur Rücksprung-
fa22  d0 03     bne  $fa27   ;  adresse, die durch jsr
fa24  fe 05 01  inc  $105,x  ;  auf dem Stapel liegt
fa27  bd 04 01  lda  $104,x  ;  Hole diese Adresse
fa2a  85 ce     sta  $ce     ;  und
fa2c  8d 05 01  lda  $105,x  ;  lege sie in einen
fa2f  b1 cf     sta  $cf     ;  Hilfszeiger in der Zeropage
fa31  b1 ce     lda  ($ce),y ;  Lade ein Zeichen
fa33  f0 05     beq  $fa3a   ;  Falls Null: Ende
fa35  20 d2 ff  jsr  $ffd2   ;  Nicht Null: ausgeben
fa38  90 e4     bcc  $fa1e   ;  Springe immer
fa3a  68        pla          ;  Stelle alle
fa3b  a8        tay          ;  ursprünglichen
fa3c  68        pla          ;  Registerinhalte
fa3d  aa        tax          ;  wieder her
fa3e  68        pla          ;
fa3f  60        rts          ;  und kehre zurück ...
```

Wie Sie im Listing sehen, rettet *primm* sämtliche Registerinhalte auf den Stack, die Register werden zum Abschluß der Routine wiederhergestellt. Die auf dem Stack liegende *rts*-Adresse des Unterprogrammaufrufs unseres Hauptprogramms wird dazu benutzt, einen Zeiger auf die auszugebenden Bytes zu gewinnen. Die Daten werden solange ausgegeben, bis ein abschließendes Nullbyte gefunden wird. Für jedes aus dem Datenbereich entnommene Byte wird ferner die *rts*-Adresse um Eins erhöht, so daß die Rücksprungadresse zum Ende der Routine genau hinter das Nullbyte weist. Kehrt *primm* also durch das abschließende *rts* wieder zum Hauptprogramm zurück, so wird unser Programm direkt nach dem auszugebenden Text fortgesetzt – eine wirklich elegante Lösung!

Folgende Einschränkungen müssen für den Einsatz der *primm*-Routine gemacht werden: Da der Datenbereich lediglich durch eine Indizierung über das Y-Register angesprochen wird, darf der auszugebende Text nicht länger als 255 Byte inklusive des Nullbytes am Ende sein. Zweitens: Durch die Manipulation der *rts*-Adresse innerhalb von *primm* ist es nicht möglich, *primm* über *jsrfar* aufzurufen – die Routine würde durch den Umweg über *jsrfar* den Datenbereich nicht finden können. Daraus folgt, daß *primm* nur von einem Ort aufgerufen werden kann, von dem aus das Kernal-ROM direkt sichtbar ist, *primm* muß außerdem ihrerseits direkt die aufrufende Routine »einsehen« können, um durch das abschließende *rts* wieder korrekt zum Hauptprogramm zurückkehren zu können.

2.5.1.2 Eingaben vom Bildschirm und von der Tastatur

Eingaben vom Bildschirm und von der Tastatur unterscheiden sich in der gleichen Weise, in der sich die Wirkung der Basicbefehle *input* und *get* unterscheidet. Bei Bildschirmeingaben erscheint der Cursor, dann erfolgt eine Eingabe mit Echo auf dem Bildschirm, die durch *Return* abgeschlossen wird. Tastatureingaben erfragen lediglich den Zustand der Tastatur, ohne daß dabei etwas auf dem Bildschirm sichtbar wird.

Diese beiden Eingabearten unterstützt das Kernal-ROM des C-128 durch zwei Routinen:

basin (**$ffcf**) erlaubt eine Eingabe ähnlich zum *input*. Die Routine wird manchmal auch *chrin* genannt.

getin (**$ffe4**) ist die Routine zur Tastaturabfrage wie bei *get*.

Beide Unterprogramme werden, wie *bsout*, auch im Zusammenhang mit Eingaben aus Dateien usw. verwendet.

BASIN ($ffcf):

basin ist die allgemeine Eingaberoutine des Kernals. Sie wird ganz analog zu *bsout* verwendet. Sie rettet wie diese Routine die Prozessorregister auf dem Stack, so daß diese bei der Rückkehr der Routine wieder unverändert zur Verfügung stehen. Sobald *basin* das erste Mal zur Eingabe vom Bildschirm aufgerufen wird, erscheint der Cursor und der Benutzer kann Eingaben vornehmen. Das aufrufende Programm erhält erst dann das erste Zeichen von *basin* geliefert, wenn die Eingabe mit *Return* beendet wurde. Ein zweiter Aufruf von *basin* liefert dann das zweite Zeichen der Eingabe usw. Als Zeichen für das Ende der Eingabezeile liefert *basin ein Carriage Return* (ASCII-Code 13). An einem Beispiel können Sie sehen, wie *basin* verwendet wird. Es wird eine Eingabezeile in einen Eingabepuffer ab $b00 gelesen:

Beispiel:

```
.base $c00              ; Startadresse $c00
.define puffer = $b00   ; Eingabepuffer
.define basin = $ffcf   ; die Routine
.define coreg = $ff00   ; Konfigurationsregister

rdlin   ldx  #0         ; Schleifenindex
        stx  coreg      ; In/Out und Kernal ein
rd1     jsr  basin      ; Zeichen vom Bildschirm lesen
        cmp  #13        ; Ende der Eingabe?
        beq  rd2        ; Ja: ->
        sta  puffer,x   ; Nein: Zeichen abspeichern
        inx             ; Index + 1
        bne  rd1        ; Schleife fortführen
rd2     lda  #0         ; Ende: Puffer mit 0
        sta  puffer,x   ; Abschließen
        rts             ; Fertig ...
```

Nachdem Sie dieses Programm assembliert und gestartet haben, erscheint der Cursor am Bildschirm. Sobald Sie die Eingabe mit *Return* abschließen, wird die Eingabezeile in den Puffer ab $b00 geholt. Sie können dies z. B.dadurch kontrollieren, daß sie sich den Bereich ab $b00 nach Ablauf des Programmes mit dem Monitor anschauen (Monitorbefehl *m b00)*. Diese Eingabezeile kann in der Folge durch ein Hauptprogramm weiter bearbeitet werden.

GETIN ($ffe4):

getin fragt die Tastatur nach etwaig vorhandener Eingabe ab. Sofern eine Taste gedrückt wurde, liefert *getin* den ASCII-Code des gedrückten Zeichens im Akkumulator zurück; liegt keine Eingabe an, erhält der Aufrufer eine Null geliefert. *getin* rettet keine Register! Eine typische Verwendung von*getin*:

```
loop    jsr  getin      ; Tastatur abfragen
        tax             ; Eingabe vorhanden?
        beq  loop       ; Nein: warten
        ...             ; Hier wird das Zeichen verar-
                          beitet
```

2.5.2 Dateiverwaltung in Assembler

Welche Basicbefehle benutzen Sie, um Ausgaben auf den Drucker zu machen, eine Datei auf Diskette zu beschreiben und was der Dinge mehr sind, die unter »Dateiverwaltung« zu verstehen sind? Sie werden finden, daß sie fast genaue Entsprechungen dieser Befehle auch in Assembler wiederfinden, in Form von Kernal-

Routinen. Auch die für die Dateiverwaltung benutzte Basicvariable *ST* finden Sie in Assembler wieder: eine Zeropageadresse. Sehen wir uns die interessierenden Routinen als erstes in einer Zusammenfassung an:

Adresse	Name	Funktion
$ff68	setbnk	Bank für Dateinamen setzen
$ffba	setpar	Dateiparameter festlegen
$ffbd	setnam	Dateinamen festlegen
$ffc0	open	Datei öffnen
$ffc3	close	Datei schließen
$ffc6	chkin	Eingabe auf Datei schalten
$ffc9	ckout	Ausgabe auf Datei legen
$ffcc	clrch	Ein- und Ausgabe auf Normalzustand
$ffcf	basin	Eingabe von Datei
$ffd2	bsout	Ausgabe auf Datei
$ffd5	load	Datei laden
$ffd8	save	Datei abspeichern

Zwei der Routinen haben wir bereits kennengelernt: *basin* und *bsout*. Bei der Beschreibung dieser Routinen wurde außerdem schon darauf hingewiesen, daß sie auch zur Ein- bzw. Ausgabe von bzw. auf Dateien dienen können. Die restlichen Routinen sollen in Gruppen der Reihe nach vorgestellt werden:

SETBNK ($ff68), SETNAM ($ffbd), SETPAR ($ffba), OPEN ($ffc0), CLOSE ($ffc3):

Alle fünf Routinen zusammen ermöglichen das Öffnen und Schließen von Dateien. Wie der *Open*-Befehl unter Basic benötigt auch die *open*-Routine verschiedene Parameter: logische Filenummer, Gerätenummer, Sekundäradresse, Dateiname – sofern vorhanden. Diese Angaben können nicht alle in den Prozessorregistern übergeben werden, dazu bieten die Register einfach nicht genug Raum. Statt dessen werden die drei Routinen *setbnk*, *setnam* und *setpar* zur Hilfe gezogen, die die Parameter in die richtigen Speicherzellen in der Zeropage transportieren. Eine weitere Aufgabe haben die Routinen nicht.

SETBNK:

Die Routine erwartet zwei Angaben in den Registern; der Akkumulator enthält im Fall, daß ein späteres *load* erfolgen soll, den Konfigurationsindex, unter dem die Datei beim Laden abgespeichert werden soll. Diese Angabe kann fehlen, falls ein normales *open* folgen soll. Das X-Register enthält den Konfigurationsindex, unter dem der Dateiname für das *open* zu finden ist, falls ein Dateiname erforderlich ist (Bei einer Ausgabe auf einen Drucker z. B.wäre kein Filename erforderlich, so daß *setbnk* überhaupt nicht aufgerufen werden müßte.)

SETNAM:

setnam legt Länge und Adresse eines Dateinamens in die Zeropage. Die Länge wird im Akkumulator übergeben, das X-Register muß das Lowbyte der Adresse des Filenamens enthalten, das Y-Register das Highbyte. Ist kein Dateiname erforderlich, so wird *setnam* mit einer Namenslänge 0 aufgerufen.

SETPAR:

Mit *setpar* werden die noch verbleibenden Parameter *logische* Filenummer, Gerätenummer und *Sekundäradresse* übergeben. Die Angaben erfolgen in dieser Reihenfolge in den Registern A, X und Y. Verwenden Sie die gleichen Werte für diese Parameter, die Sie auch in einem Basic-*Open* verwenden würden. Beispiel:

```
lda   #1        entspricht   open   1, 8, 15
ldx   #8
ldy   #15
jsr   setpar
(jsr  open)
```

OPEN:

Nachdem alle Parameter gesetzt worden sind, genügt ein einfacher Aufruf der *open*-Routine (s. o.), um den File zu öffnen. *open* liefert im C-Flag eine Fehleranzeige zurück, das gesetzte Carry-Flag deutet auf einen Fehler wie z. B.»file open« hin. Zusammen mit dem C-Flag wird eine Fehlernummer im Akkumulator geliefert, die weiter ausgewertet werden kann. Die folgende Tabelle zeigt die Zuordnung der Fehlernummern:

Nummer	Fehler
1	too many files
2	file open
3	file not open
4	file not found
5	device not present
6	not input file
7	not output file
8	missing filename
9	illegal device number
16	break (durch STOP-Taste)

Ein Teil der hier aufgeführten Fehlernummern kann bei *open* nicht auftreten; da der gleiche Mechanismus zur Anzeige von Fehlern aber auch bei anderen Routinen verwendet wird, sind an dieser Stelle alle möglichen Fehlernummern angegeben. Die Nummern 10 bis 15 existieren nicht.

CLOSE:

close ist das Gegenstück zu *open*, das ein geöffnetes File wieder schließt. Um das richtige File zu identifizieren, wird an *close* die logische Filenummer im Akkumulator übergeben. Auch bei *close* werden Fehler durch C-Flag und Fehlernummer angezeigt.

CHKIN ($ffc6), CKOUT ($ffc9), CLRCH ($ffcc), BASIN ($ffcf), BSOUT ($ffd2):

Nachdem ein File einmal geöffnet wurde, kann es noch nicht unmittelbar beschrieben oder gelesen werden. Vorher muß der Eingabe- oder Ausgabestrom erst auf dieses File gelegt werden. Diese Aufgabe übernehmen *chkin* und *ckout*. Den Routinen wird im X-Register die logische Filenummer des Files übergeben, auf das der Ein- bzw. der Ausgabestrom geschaltet werden soll. Der normale Fall – Eingabe von der Tastatur, Ausgabe auf den Bildschirm – wird durch *clrch* wiederhergestellt. Alle Ein- und Ausgaben selbst erfolgen über die bekannten Routinen *basin* und *bsout*. *chkin* und *ckout* benutzten wieder den Mechanismus C-Flag plus Fehlernummer, um Fehlerbedingungen anzuzeigen. *basin* und *bsout* dagegen liefern immer ein gelöschtes Carry-Flag, deswegen muß eine andere Methode verwendet werden, um beispielsweise das Ende eines Files zu erkennen: Hierzu dient die Zeropageadresse **status = $90.** Diese Zeropageadresse entspricht in jeder Einzelheit der Basicvariablen *ST*. Um Fehler in *status* erkennen zu können, muß *status* zu Beginn einer Übertragung mit Null initialisiert werden. Dies übernimmt, wenn Sie ausschließlich die hier vorgestellten Routinen verwenden, das Betriebssystem für Sie. Ein Beispiel für die Abfrage von *status* auf Erreichen des Endes eines Files finden Sie am Schluß dieses Kapitels.

LOAD ($ffd5) und SAVE ($ffd8):

Die *load*-Routine kennt zwei verschiedene Betriebsarten – das Laden an die Adresse, die im File selbst abgespeichert ist *(bload)* und das Laden an eine beliebige, frei wählbare Adresse *(dload)*. Die beiden folgenden Beispielprogramme zeigen, wie die *load*-Routine angewendet wird:

Absolutes Laden (Laden an die Ursprungsadresse):

```
ldx   #device     ;  Gerätenummer
ldy   #1          ;  Flag für absolutes Laden
jsr   setpar      ;  Parameter setzen
lda   #length     ;  Länge des Filenamens
ldx   #<(name)    ;  Lowbyte der Adresse des Namens
ldy   #>(name)    ;  Highbyte
jsr   setnam      ;  in die Zeropage
lda   #lindex     ;  Konf.-Index für Laden
```

```
ldx   #nindex     ; Dto. für den Filenamen
jsr   setbnk      ; in die Zeropage
lda   #0          ; Flag für »Load«
jsr   load        ; Laden
bcs   fehler      ; Fehler aufgetreten: ->
stx   endadr      ; Endadresse merken, Lowbyte
sty   endadr+1    ; Highbyte
```

Relatives Laden (Laden an beliebige Adresse):

```
ldx   #device     ; Gerätenummer
ldy   #0          ; Flag für relatives Laden
jsr   setpar      ; Parameter setzen
lda   #length     ; Länge des Filenamens
ldx   #<(name)    ; Lowbyte der Adresse des Namens
ldy   #>(name)    ; Highbyte
jsr   setnam      ; in die Zeropage
lda   #lindex     ; Konf.-Index für Laden
ldx   #nindex     ; Dto. für den Filenamen
jsr   setbnk      ; in die Zeropage
lda   #0          ; Flag für »Load«
ldx   #<(ladr)    ; Lowbyte Zieladresse
ldy   #>(ladr)    ; Highbyte der Zieladresse
jsr   load        ; Laden
bcs   fehler      ; Fehler aufgetreten: ->
stx   endadr      ; Endadresse merken, Lowbyte
sty   endadr+1    ; Highbyte
```

Die Wirkungsweise von *load* dürfte durch diese beiden Listings klar sein. Auch ein *Verify* ist mit der *load*-Routine möglich: Dazu wird *load* statt mit einer Null mit einer Eins im Akkumulator aufgerufen, alles andere bleibt gleich.

Bei *save* entfällt die Unterscheidung relativ oder absolut. Sie benötigen allerdings einen Zeropagezeiger, um die Startadresse des abzuspeichernden Adreßbereiches an *save* zu übergeben. Nachdem Sie sich das Listing zur *save*-Routine angesehen haben werden, wird sich eine weitere Erläuterung erübrigen.

Abspeichern (Save):

```
ldx   #device     ; Gerätenummer
jsr   setpar      ; in die Zeropage
lda   #length     ; Namenslänge
ldx   #<(name)    ; und Adresse des Namens
ldy   #>(name)    ; in die Zeropage
lda   #lindex     ; Aus welcher Konfiguration sp.?
```

```
ldx   #nindex     ;  Index des Filenamens
jsr   setbnk      ;  in die Zeropage
lda   #<(start)   ;  Startadresse in einen
sta   zero        ;  Zeropagezeiger bringen
lda   #>(start)   ;
sta   zero+1      ;
lda   #zero       ;  Wo liegt der Zeiger?
ldx   #<(end)     ;  Wie ist die Endadresse?
ldy   #>(end)     ;
jsr   save        ;  und speichern
bcs   fehler      ;  im Fehlerfall: ->
...               ;  und irgendwie weiter ...
```

Beachten Sie noch, daß die Endadresse *end* immer **hinter** das letzte abzuspeichernde Byte weisen muß, kann hier eigentlich nichts mehr schiefgehen.

2.6 Weiterführende Informationen

Das war's in diesem Buch zum Thema »Einführung in die Assemblerprogrammierung«. Mit Sicherheit ist das Thema hier nicht erschöpfend abgehandelt worden und mit ebenso großer Sicherheit werden Sie sich im Laufe der Zeit fragen, wo Sie sich weitere, eingehendere Informationsquellen erschließen können und wo Sie aktuelle Listings und Tips erhalten können.

ROM-Listings: Wenn Sie vorhaben, Befehlserweiterungen für den Basic-Interpreter zu schreiben, oder auch, wenn Sie sich nur für die Funktionsweise des Interpreters und des Betriebssystems interessieren, werden Sie auf Dauer ohne ein kommentiertes ROM-Listing nicht auskommen. Ein solches ROM-Listing enthält den Inhalt der im C-128 vorhandenen ROM-Bausteine in disassemblierter oder auch reassemblierter Form, ähnlich, wie Sie es an den Beispielprogrammen in diesem Buch sehen können (s. Fehlerprotokoll für ausführlichen Einschub). Eventuell finden Sie sogar auf einigen Werbeseiten am Ende dieses Buches einen Hinweis auf ein ROM-Listing zum C-128.

Programmierung des C-128: Dieses Kapitel kann nicht schließen, bevor auch eine kleine Werbung in eigener Sache gemacht wurde. Ich habe Ende 1985 ein Buch speziell zum Thema »Maschinenprogrammierung des C-128« geschrieben. Dieses Buch ist jetzt erhältlich und bietet meiner Ansicht nach den besten Anschluß an das vorliegende Buch und an die Programme, die Sie gleichzeitig erworben haben. Das Buch enthält eine Menge an Tips, darunter unter anderem z. B.auch für den Umgang mit den Fließkommaroutinen des Interpreters und den Bau eigener Befehlserweiterungen – der Titel: »C-128 Programmieren in Maschinensprache".

Kapitel 3

Der ASE-Makroassembler Editor/Linker

Das ASE-Programm – ASE steht als Abkürzung für »Assembler und Editor« – besteht eigentlich aus drei Programmen, die zu einem Programmpaket verschnürt worden sind:

- dem Editor
- dem Makroassembler
- dem Linker.

Eine kurze Erläuterung der drei Begriffe: Mit dem **Editor** werden die Texte geschrieben, die der **Assembler** verarbeitet, sprich in Maschinensprache umsetzt. Damit ist schon umrissen, was der Assembler macht – er verwandelt einen sogenannten **Quelltext**, der mit dem Editor geschrieben wird, in ein Maschinenprogramm. Der Zusatz »Makro« vor dem »Assembler« deutet darauf hin, daß das Assemblerprogramm sogenannte Makros verarbeiten kann, eine wichtige und arbeitssparende Option, auf die weiter unten noch eingegangen wird. Der **Linker** ist eine Besonderheit des ASE-Assemblers. Linker bieten, allgemein gesprochen, ebenfalls eine Arbeitserleichterung für den Programmentwickler. Auf anderen Systemen und unter anderen Betriebssystemen (CP/M, TOS, MS-DOS ...) sind Assembler mit Linkern gang und gäbe, für 6502-Assembler stellt der Linker des ASE eine ziemlich ungewöhnliche Sache dar. Auf die Gründe hierfür und die Aufgabe des Linkers wird ebenfalls etwas später in der Übersicht über die Funktionen des Assemblers noch eingegangen.

Die drei Programmteile wurden aus einem einfachen Grund zu einem Gesamtprogramm zusammengefaßt: um die Bedienung zu vereinfachen. Wären beispielsweise Editor und Assembler getrennte Programme, sähe die Erstellung eines Maschinenprogramms ziemlich trostlos für den Programmierer aus – ein ständiger Wechsel mit Neuladen und Starten der beiden Programmteile ließe sich nicht vermeiden. Nachdem der Quelltext für den Assembler das erste Mal geschrieben wurde, würde

er durch den Assembler geschickt, dieser fände eventuell einige Fehler in der Syntax; um diese zu verbessern, müßte der Editor wieder geladen werden, der Quelltext würde umgeschrieben, der Assembler wieder geladen ... Ich denke, Sie können sich dieses Spiel weiterdenken. Von anderen Computern ist mir der Umgang mit solcherart getrennten Systemen vertraut und lästig. Eine der Entwurfsentscheidungen bei der Entwicklung des Assemblers war deshalb, zusammengehörige Programme auch immer zusammen im Speicher halten zu können.

Aus dieser Entwurfsentscheidung resultierte in gewisser Hinsicht – mit anderen Überlegungen – die Art des verwendeten Editors. Denkbar waren zwei prinzipiell unterschiedliche Editortypen: Zeileneditoren, wie der Basic-Programmeditor einer ist, und Editoren, die in der Art einer Textverarbeitung Eingaben auf dem gesamten Bildschirm akzeptieren (Full-Screen-Editoren). Beide Editortypen unterscheiden sich als erstes in der Programmlänge: Der bei ASE verwendete Zeileneditor ist in seiner jetzigen Fassung ungefähr so lang, wie es ein sehr einfacher Full-Screen-Editor wäre. Die Editortypen unterscheiden sich zum zweiten in der Art der Bedienung: Full-Screen-Editoren haben zumeist einen Befehlssatz, der durch die Betätigung bestimmter Tasten oder Tastenkombinationen ausgelöst wird. Welche Tasten zu welchem Ergebnis führen oder führen sollten, darüber scheint kein allgemeiner Konsens zu existieren. Mit anderen Worten, die Bedienung jedes Full-Screen-Editors muß von Grund auf erlernt werden, während ich bei einem Zeileneditor darauf vertrauen kann, daß die Bedienung desselben von der Basic-Programmierung her vertraut ist. Diese beiden Gründe haben letztlich zum jetzt in den ASE integrierten Zeileneditor geführt, den Sie weiter unten noch kennenlernen werden.

Die Integration des Linkers in den Assembler war folgerichtig. Da der Editor kurz gehalten werden konnte, konnte auch der Linker noch mit in das System integriert werden, so daß tatsächlich kein einziges Nachladen eines Programmes notwendig wird, um eine wie auch immer geartete Assemblierung vollständig durchführen zu können.

Um den ASE zu laden und zu starten, gibt es zwei Möglichkeiten. Sie können die Masterdiskette in das Diskettenlaufwerk einlegen und die Diskette booten lassen. Dies geschieht automatisch, wenn Sie den Rechner erst einschalten, nachdem Sie die Masterdiskette in das eingeschaltete Laufwerk eingelegt haben. Alternativ dazu können Sie auch den Basicbefehl boot verwenden, der den gleichen Vorgang bei schon eingeschaltetem Rechner auslöst. Die zweite Möglichkeit besteht darin, den Assembler durch die Befehlsfolge

```
run »ase*«
```

in den Speicher zu holen und zu starten. Diese Vorgehensweise möchte ich Ihnen in Verbindung mit einer anderen empfehlen: Der ASE-Assembler kann durch ein einfaches *dload* und anschließendes *dsave* auf eine andere Diskette kopiert werden. Da Sie sowieso nicht die Masterdiskette für die tägliche Arbeit verwenden sollten, sollten Sie den ASE auf diese Weise auf Ihre jeweilige Arbeitsdiskette aufbringen.

Dies ist zeitsparend und erleichtert die Entwicklung von Programmen, da nicht nach jedem Austesten eines gerade entwickelten Programmes (Absturz des Rechners?) ein Diskettenwechsel erfolgen muß, um den Assembler neu starten zu können.

In diesem Zusammenhang des Kopierens möchte ich einige Bemerkungen machen und eine Bitte an Sie richten: Wie Sie sehen, sind die auf der Diskette liegenden Programme sämtlich ungeschützt. Sie können auf die für den Anwender bequemste Art und Weise kopiert werden. Honorieren Sie bitte auch dieses Entgegenkommen meinerseits und von seiten des Verlages: Reichen Sie die Programme nicht einfach weiter, nach dem Motto »ist ja nicht geschützt, also darf ich ruhig kopieren«. Ich glaube, es hat sich mittlerweile ausreichend herumgesprochen und es ist ausreichend oft publiziert worden, daß als bekannt vorausgesetzt werden darf: **Jegliches Kopieren von Programmen, außer für den persönlichen Bedarf als Sicherheitskopie oder Arbeitskopie, ist strafbar!** Es gibt immer wieder einige Leute, die sich trotzdem auf den Standpunkt stellen »wo kein Kläger, da kein Richter«. Für diese möchte ich noch ein Argument anfügen, sofern sie überhaupt noch Argumenten zugänglich sind: Das kriminelle Raubkopieren macht es in kürzerer Sicht unmöglich, Programme wie beispielsweise den Top-Ass weiter so günstig anzubieten, wie dies zur Zeit der Fall ist. Welcher Programmautor wird sich die Zeit nehmen wollen, die zur Entwicklung kommerzieller Software notwendig ist, wenn sein Einsatz für ihn zum Verlustgeschäft wird? Er wird also entweder auf einem so hohen Preis bestehen müssen, daß die Verluste durch die Raubkopierer abgefangen werden, oder die Programme werden auf immer raffiniertere Weise geschützt werden, wodurch ihre Gebrauchstüchtigkeit mit Sicherheit nicht zunimmt. Raubkopieren schadet also letztlich auch dem normalen Anwender! Ich hoffe, daß sich diese Einsicht irgendwann auch bei denjenigen einstellen wird, die heute noch stolz auf ihre Sammlung Hunderter Disketten zeigen, von denen sie die wenigsten ehrlich erworben haben.

3.1 Terminologie

Dieses Kapitel soll für diejenigen vor die eigentliche Beschreibung des Assemblers vorgeschaltet werden, die nicht so vertraut mit der in Kreisen der Maschinenprogrammierer verwendeten Terminologie sind und die auch noch keine Vorstellung darüber haben, welche genaue Arbeit ein Assembler im Zuge der Erstellung eines Maschinenprogramms leistet. Wenn Sie wollen, können Sie es als eine Art Glossar auffassen, dessen Einträge in einen verständnisfördernden Sinnzusammenhang gebracht sind. Mögliche Wiederholungen von Dingen, die schon in Kapitel 1 von Michael Bauer erläutert wurden, bitte ich zu verzeihen. Zum Zeitpunkt, an dem ich dieses niederschreibe, ist mir der Inhalt des ersten Kapitels noch nicht im einzelnen bekannt. Außerdem – es ist nie verkehrt, wichtige Dinge zweimal erläutert zu bekommen, gerade von verschiedenen Personen.

Lassen Sie mich als erstes raten, in welchem Zusammenhang Sie erstmals auf Maschinensprache als solche getroffen sind: Haben Sie etwa ein Basicprogramm abgetippt, das in der Hauptsache aus Datazeilen bestand, deren Funktion zunächst nicht einmal zu erraten war? Ich glaube, in der Mehrzahl der Fälle liege ich mit dieser Schätzung ganz richtig. Mit Sicherheit wissen Sie in der Zwischenzeit, daß die Datazeilen das Maschinenprogramm selbst waren, was gleich eines deutlich werden läßt: Maschinenprogramme sind Daten. Das Basicprogramm wird diese Daten im allgemeinen an irgendeine Adresse »poken«, wir können unsere Definition also präzisieren: Ein Maschinenprogramm ist eine Folge von Daten im Speicher des Rechners. Als Datenfolge unterscheidet sich ein Maschinenprogramm prinzipiell nicht von anderen Dingen, die im Speicher des Rechners untergebracht werden – Texte, Fließkommazahlen, Basicprogramme usw. Alles wird in der gleichen Grundeinheit, dem **Byte**, abgespeichert. Ein Byte ist eine Speicherzelle aus acht **Bit** und kann damit Zahlen, nichts anderes, zwischen 0 und 255 aufnehmen. Alles weitere ist eine Sache der Interpretation dieser Zahlen; man kann unter einer bestimmten Zahl ein bestimmtes Zeichen verstehen und stößt so auf die bekannten **ASCII-** und **Bildkode**-Tabellen. Man kann auch mehrere Bytes zusammenhängend lesen und interpretieren – Fließkommazahlen bestehen beispielsweise aus vier Byte. Wie auch immer – alles führt auf die Byte-Ebene zurück, einen anderen Typ Daten besitzt der Speicher nicht und kennt der Prozessor nicht. Jedes Byte liegt innerhalb des Speichers an einer festen Position, der Adresse dieses Bytes. Ein gewöhnlicher 8-Bit-Mikroprozessor versteht Adressen einer Größenordnung von Null bis 65535. Eine solche Zahl kann man gerade in zwei Byte unterbringen, wenn man die beiden Bytes als untere (**Lowbyte**) und obere Hälfte (**Highbyte**) eines 16-Bit-breiten **Wortes** auffaßt. Laut Übereinkunft und nach dem, was der Prozessor uns an Möglichkeiten vorgibt, liegt die untere Hälfte eines Wortes immer an der niedrigeren Adresse. Dies muß nicht so sein; betrachten wir etwa den im C-128 auch eingebauten Z80-Prozessor, so propagiert dieser genau die andere Reihenfolge von Low- und Highbyte.

Mit diesen Termini können wir nun den Begriff »Maschinenprogramm« schon etwas »professioneller« definieren: Ein Maschinenprogramm ist eine Folge von Bytes mit einer bestimmten Startadresse.

Was macht nun aus einer Folge von Bytes ein lauffähiges Maschinenprogramm? Nun – der Prozessor kann veranlaßt werden, eine solche Bytefolge als Maschinenprogramm zu interpretieren. Dazu wird der sogenannte **Programmzähler (PC für Program Counter)** des Prozessors auf eine bestimmte Adresse gesetzt und der Prozessor auf die dort vorhandenen Bytes »losgelassen«. Daraufhin liest er das erste Byte und untersucht, ob er den dort vorhandenen Zahlenwert als Befehl kennt. Falls ja, wird eine Aktion des Prozessors ausgelöst, er lädt, speichert, rechnet und was er sonst noch alles kann. Ein solches als Befehl des Prozessors zu interpretierendes Byte heißt entsprechend **Opcode (Operation Code)**. Bestimmte Operationen bedingen Parameter; beispielsweise wäre ein Befehl »addiere zum Akkumulator« sinnlos, wenn nicht angegeben wäre, welche Speicherzelle z. B. zum Akku zu addieren

ist. Solche Angaben werden hinter dem Opcode in maximal 2 Byte (Adresse) angegeben, man nennt sie folgerichtig auch die **Adressierung** des Befehls.

Nicht alle möglichen Zahlenwerte zwischen Null und 255 entsprechen einem Prozessorbefehl, wie Sie sicher in Kapitel 1 von Michael Bauer erfahren haben. Was der Prozessor macht, wenn er auf einen Nicht-Befehl trifft, ist nicht mit letzter Sicherheit zu sagen, zumindest garantiert kein Hersteller dafür, daß der Prozessor daraufhin in einer festgelegten Art und Weise reagiert (andernfalls hätte man diese Zahl ja als Opcode in den Befehlssatz des Prozessors aufnehmen können). Die Erfahrung zeigt zwar, daß bestimmte Nicht-Befehle, die **illegalen Opcodes**, in den meisten Fällen doch eine bestimmte Wirkung haben, es gilt aber nicht als »feine englische Art« diese Opcodes zu nutzen – kein Top-Ass-Programm unterstützt aus diesem Grund Illegale. REASS hingegen erkennt diese Opcodes und zeigt sie an.

Was ein Maschinenprogramm ist, haben wir jetzt in den Griff bekommen. Auf welche Art und Weise programmiert man aber diese Folge von Opcodes und Adressierungen?

Möglichkeit 1: Man legt sich eine Tabelle der Opcodes neben den Rechner und gibt die entsprechenden Werte beispielsweise über ein Monitorprogramm in den Speicher ein. Alle Monitorprogramme bieten diese Möglichkeit – meist erfolgt eine solche Eingabe über einen **Hex-Dump** in sedezimaler Schreibweise. Gibt es nicht, dauert ja viel zu lange? Hat's aber mal gegeben!

Der Einwand ist aber berechtigt und auch schon recht früh in der Computer-Geschichte aufgetaucht. Was macht man also, um sich die Arbeit zu erleichtern? Die Tabelle kommt vom Schreibtisch in den Rechner, dieser soll sich gefälligst mit der Tabelle herumschlagen! Was braucht man dazu? Ein Programm und eine Art, die Opcodes und Adressierungsarten zu benennen, und zwar so, daß sie einigermaßen leicht zu behalten sind, sonst könnte man ja gleich wieder zur Tabelle greifen. Das Programm muß die neue Benennungsart in Opcodes und Adressierungen umrechnen. Damit ist der **Assembler** geboren. Opcodes ähnlicher Funktion wurden zu einem **Mnemonic** zusammengefaßt, das zusammen mit einer **Adressierungsart** und der Adresse selbst – sofern notwendig – einen bestimmten Opcode bezeichnet. Auf diese Weise erhält man Listings wie

```
lda  ($ff),y
jmp  $4567
adc  ($d7,x)
beq  $1234    usw.
```

Den umgekehrten Weg – die Übersetzung der Opcodes und Adressierungen zurück in Mnemonics und Adressierungsarten – geht ein zweites Übersetzungsprogramm: der **Disassembler**. Listings der obigen Art können recht leicht in Opcodes verwandelt werden, man verwendet dazu eine bestimmte Art Assembler, den sogenannten **Direkt-** oder **Zeilen-Assembler** oder auch **Line Assembler**. Die Bezeichnung rührt aus

der Bedienung dieser Programme her: Man gibt jeweils eine Zeile Quelltext am Bildschirm ein, diese wird direkt nach der Eingabe – eben zeilenweise – durch den Zeilen-Assembler in Opcodes umgewandelt. Im Top-Ass-System ist ein Line-Assembler verfügbar, er findet sich im Monitorprogramm namens DBM.

So ganz das Richtige ist diese Art der Notation aber auch noch nicht. Wie schreibe ich denn beispielsweise einen Text in den Speicher, den das Maschinenprogramm ausgeben soll? Dafür gibt es kein Mnemonic. Und überhaupt könnte das alles noch viel bequemer sein, wo man einmal dabei ist ...

Die erste Maßnahme, um eine weitere Erleichterung für den Programmierer zu erreichen, bestand im weiteren darin, den zu assemblierenden Text nicht jedes Mal neu am Bildschirm eingeben zu müssen, sondern ihn in Form eines **Quelltextes** aufzubewahren, der vom Assembler gelesen und verarbeitet wird. Einen Quelltext kann man leicht verändern, drucken, weiterreichen und was der Dinge mehr sind – eine praktische Einrichtung. Gleichzeitig wurde die Möglichkeit geschaffen, die vielen verwendeten Adressen und weiteren Zahlenangaben für den Programmierer leichter handhabbar zu machen: sie wurden durch sogenannte **Label** oder Marken ersetzt. Label sind Namen aus Buchstaben und Sonderzeichen, die symbolisch für eine bestimmte Adresse stehen. Namen kann man sich halt wesentlich besser merken als nackte Zahlen. Ich vermute, das Aussehen eines Quelltextes mit Labeln ist Ihnen bekannt; sollte dies jedoch nicht der Fall sein – hier noch ein einfaches Beispiel, das die wesentlichen Elemente eines Quelltextes zeigt:

```
.base $c00
.define label_1 = $1234

label_2    lda #0
           jsr label_1
           bcc label_2
```

Sie sehen die bekannten Mnemonics der Prozessorbefehle. Was Sie außerdem erkennen, sind einige offensichtliche Befehlsworte, die durch einen Punkt vor dem eigentlichen Wort eingeleitet werden. Dies sind Anweisungen an den Assembler selbst; in der Regel werden diese Anweisungen nicht in Maschinenkode übersetzt, sondern lösen assemblerinterne Aktionen aus. ».base« (Basis) beispielsweise teilt dem Assembler mit, welche Startadresse das Maschinenprogramm haben soll, das der Assembler aus dem Quelltext erzeugen soll. ».define« weist einem Label einen bestimmten Wert zu, bestimmt also, für welche Adresse der Label als Symbol herhalten soll. Es gibt sehr viele dieser Assembleranweisungen, die treffend auch **Pseudoops** genannt werden. Viele Assembler verwenden zudem auch noch unterschiedliche Pseudoops, deswegen läßt sich keine Angabe über die vorhandenen und nicht vorhandenen Befehle von Assemblern allgemein machen. Beim ASE habe ich so gut wie alle Pseudoops implementiert, die auch bei anderen Assemblern in der

einen oder anderen Form zu finden sind, außerdem sind noch einige Vorstellungen verwirklicht, die andere Assembler nicht aufweisen können – insgesamt besitzt der ASE einen Befehlssatz, der auch für Profis mehr als ausreichend ist.

Aber fahren wir in der Betrachtung des Beispiel-Quelltextes fort: Wie Sie sehen, steht einer der Label – *label_2* – vor einem Mnemonic. Dies ist die zweite Möglichkeit, einen Label mit einer Adresse zu identifizieren. Ein solcher vor einem Befehl stehender Label bezeichnet die Adresse des Befehls, vor dem der Label steht. Man kann begriffsmäßig zwei Labelarten unterscheiden – **interne Label** und **externe Label.** Beides sind erst einmal ganz normale Label, die Bezeichnungen *intern* und *extern* dienen dazu, hervorzuheben, welche Adressen sie repräsentieren. Liegt ein Label innerhalb des assemblierten Maschinenprogramms, wird er *intern* genannt, liegt er dagegen außerhalb des Assemblierungsbereichs – Beispiel wäre eine Adresse in der Zeropage – heißt er *extern.* Die beiden Labelarten unterscheiden sich auch in der Form, in der sie definiert werden: interne Label wird man in der Regel so festlegen, daß man sie direkt vor den betreffenden Befehl im Quellprogramm schreibt. Da dies bei externen Labeln nicht möglich ist, gibt es bei allen Assemblern einen Pseudoop, mit dem man diesen Labeln einen Wert in Form einer Wertzuweisung zuordnen kann, bei ASE heißt dieser Befehl ».define«.

Ein Quelltext mit Labeln läßt sich nicht mehr unmittelbar in Opcodes umsetzen. Alle Label, die vor einem Mnemonic stehen, haben zunächst ja keinen festgelegten Wert. Die Adresse dieser Label stellt der Assembler selbst fest, indem er den gesamten Quelltext einmal durchläuft und sich die dabei auftretenden Labelwerte in einer **Symboltabelle** notiert. Erst nach diesem Durchlauf (engl. **Pass**) kann er dann die eigentliche Übersetzung des Quelltextes vornehmen, wobei die festgestellten Labelwerte in die Adressierungen eingesetzt werden können. Jede Assemblierung besteht aus diesem Grund aus mindestens zwei Passes durch den gesamten Quelltext.

Welche Rolle spielen die erwähnten Makros bei der Assemblierung? Sie werden im Laufe der Zeit feststellen, daß auch einfache Problemlösungen in Assembler nur in relativ vielen Quelltextzeilen zu formulieren sind. Es fällt jedoch ebenso auf, daß manche Teile von Quelltexten eine gewisse Ähnlichkeit miteinander besitzen, ohne daß man gleich ein Unterprogramm schreiben könnte, das diese ähnlichen Aufgaben zusammenfassend löst. Solche Teile kann man zu einem **Makro** zusammenfassen. Ein Makro ist eine Anweisung an den Assembler; beim ASE werden Makros dadurch definiert, daß Quelltextzeilen zwischen die beiden Befehle ».macro« und ».macend« geschrieben werden. Durch diese Definition identifiziert der Assembler die angegebenen Quelltextzeilen der **Makrodefinition** mit einem Namen, dem **Makronamen.** Wird der Makroname im weiteren Verlauf des Quelltextes über einen speziellen Befehl (»..«) aufgerufen, fügt der Assembler die Zeilen der Makrodefinition am Ort des Aufrufes ein. Damit spart man häufig eine ganz erhebliche Menge an Tipparbeit. Es gibt noch eine ganze Reihe weiterer Möglichkeiten, die sich durch die Makroverarbeitung ergeben. Diese möchte ich hier aber nicht aufzählen, schließlich sollen Sie an dieser Stelle nur eine Übersicht über Eigenschaften von Assemblern und

Maschinenprogrammen und über die in diesem Zusammenhang wichtigsten Begriffe erhalten.

Nur noch zwei Begriffe in Kürze, dann kommen wir zur eigentlichen Beschreibung des Assemblers ASE, auf die sie sicher schon warten. Der erste lautet **bedingte Assemblierung**. Es hat sich als sehr vorteilhaft erwiesen, bestimmte Teile eines Quelltextes nur dann assemblieren zu lassen, wenn bestimmte Bedingungen erfüllt sind. Auf diese Weise lassen sich beispielsweise Quelltexte schreiben, die durch minimale Veränderungen – eben der Veränderung einer Bedingung, indem z. B. eine Variable auf einen bestimmten Wert gebracht wird – Maschinenprogramme generieren, die unterschiedliche Ausbaustufen eines Gesamtprogramms bilden, die auf verschiedenen verwandten Rechnertypen laufen usw. Gerade in Verbindung mit der Makrofähigkeit eines Assemblers spielt die gleichzeitige Fähigkeit zur bedingten Assemblierung eine überragende Rolle, wie Sie später noch an Beispielen sehen werden.

Last aber bestimmt nicht least der **relokatible Kode.** Die spezielle Art der Assemblierung, bei der nicht direkt das Maschinenprogramm erzeugt wird, sondern erst eine Art Zwischenkode in der Rangfolge zwischen Quelltext und Opcode, ist unter anderen Betriebssystemen eigentlich die häufiger zu findende – ich denke beispielsweise an CP/M, das Sie auf dem 128er ebenfalls zur Verfügung haben. Der Zwischenkode bietet gegenüber dem Opcode den Vorteil, daß er nicht an eine bestimmte Startadresse gebunden ist, er nennt sich aus diesem Grund auch *relokatibel* (verschieblich). Ein spezielles Programm, meist **Loader** genannt, kann ein im Zwischenkode vorliegendes Maschinenprogramm an beliebige Speicheradressen laden. Damit erspart man sich zum Beispiel eine nochmalige Assemblierung, wenn sich nur die Startadresse eines Programms verändert hat, oder man spart sich auch vielleicht Dutzende Programmversionen auf Diskette, die sich nur in der verwendeten Startadresse unterscheiden (s. den beiliegenden Monitor DBM). Was aber noch viel wichtiger ist – mehrere vollkommen unabhängig voneinander assemblierte Programmteile können, sofern sie im Zwischenkode vorliegen, durch einen **Linker** (Verbinder) zu einem Gesamtprogramm verbunden werden! Dieser Vorgang ist um mindestens eine Größenordnung schneller, als würde man einen entsprechend großen Quelltext schreiben und das Programm durch eine Gesamt-Assemblierung erzeugen. Der Geschwindigkeitsvorteil nimmt sogar mit wachsender Programmlänge noch zu!

Mit diesen letzten Betrachtungen soll die Übersicht nun endlich beschlossen werden. Für die folgenden Beschreibungen sollten Sie den ASE laden und starten, so daß Sie die einzelnen Beschreibungen nachvollziehen können.

3.2 Der ASE-Editor

Wie schon ganz zu Anfang des dritten Kapitels erwähnt wurde, gehört der ASE-Editor zur Familie der Zeileneditoren. Einen Vertreter dieser Gattung kennen Sie mit Sicherheit bereits, nämlich den Editor, mit dem Sie Ihre Basicprogramme bisher geschrieben haben. Die prinzipielle Bedienung des ASE-Editors – Zeilen werden mit einer Zeilennummer eingegeben, die Eingabe wird mit einem *Return* abgeschlossen – dürfte Ihnen damit bekannt sein. Die Ähnlichkeit der beiden Editoren geht so weit, daß Sie fast alle Befehle, die Sie zur Gestaltung des Textes in Basic benutzen, auch im ASE-Editor wiederfinden. Teils tragen die Befehle beim ASE einen anderen Namen, teils, z. B. bei *delete* zum Löschen von Zeilen oder bei *new* zum Löschen des gesamten Textes, sind sie sogar namensgleich.

Auf der anderen Seite gehört der Basic-Editor nicht gerade zu den komfortabelsten Vertretern seiner Art. Einige sehr wichtige Befehle wie das Suchen und das automatische Ersetzen innerhalb eines Programmtextes – oder hier besser: Quelltextes – kennt der Basic-Editor z. B. nicht. Um diese beiden Befehlsarten und andere wurde der ASE-Editor gegenüber dem Basic-Editor erweitert, um dem Programmierer eine möglichst bequeme Umgebung zu bieten.

Der ASE-Editor besitzt auch einige gegenüber dem Basic-Editor grundsätzlich neue Eigenheiten, an die sich der Benutzer mit Sicherheit erst gewöhnen muß: Schauen Sie sich eines der Beispielprogramme einmal an, so werden Sie feststellen, daß jede Zeile dieser Programme einem bestimmten Format folgt – links kann ein Label stehen oder ein Leerraum; dahinter folgt das Mnemonic, dahinter wieder ein Zwischenraum und die Adressierung; als letztes folgt dann ein durch ein Semikolon eingeleiteter Kommentar; die Mnemonics und Kommentare stehen säuberlich untereinander. Ich hielt es beim Entwurf des ASE (und auch schon beim Hypra-Ass für den C-64) für eine gute Idee, ein solches formatiertes Aussehen des Quelltextes schon bei der Eingabe sichtbar zu machen, ohne daß sich der Programmierer dazu besonders anzustrengen braucht. Der ASE besitzt deshalb eine Funktion, die die eingegebenen Quelltextzeilen sofort nach Abschluß der Eingabe formatiert und die Eingabezeile durch die formatierte Version der Zeile überschreibt. So entsteht der subjektive Eindruck, die Zeile würde am Bildschirm auf ein bestimmtes Format »auseinandergezogen«. Auf diese Weise erhält man schon bei der Eingabe eines Quelltextes einen Überblick darüber, wie der Text bei der späteren Erzeugung eines Assemblerlistings auf dem Drucker aussehen wird. Außerdem wird die Eingabe selbst durch die grössere Übersichtlichkeit des Textes nach der Formatierung wesentlich erleichtert.

Eine zweite Sache ist die automatische Überprüfung des Quelltextes auf vorhandene Syntaxfehler. Da jede Quelltextzeile bereits bei der Formatierung bearbeitet werden muß, versucht bereits der Editor des ASE, Fehler innerhalb des Quelltextes zu erkennen. In der gegenwärtigen Version umfaßt die Fehlerprüfung eine Kontrolle der

Mnemonics und Pseudoops des Assemblers. Es ist absolut unmöglich, ein falsches Mnemonic wie beispielsweise »ldc« oder »rst« in den Quelltext einzugeben; der Editor weist solche Falscheingaben mit einer Fehlermeldung »not a mnemonic« bzw. »not implemented« im Falle eines falschen Pseudoops zurück. Eine praktische Einrichtung, die ca. 90 Prozent der anfallenden Tippfehler vermeiden hilft.

Genug der einleitenden Vorrede. Starten Sie nun den ASE, falls Sie es noch nicht getan haben. Sofort nach dem Start wird eine Copyright-Meldung ausgegeben, Sie befinden sich nun bereits im Editor.

3.2.1 Quelltext-Eingabeformat

Um die oben erwähnte formatierte Ausgabe des Quelltextes vernünftig bewerkstelligen zu können, muß sich der Programmierer bei der Eingabe von Zeilen an bestimmte Regeln halten, die nun der Reihe nach besprochen werden sollen:

(1) Geben Sie vor jeder Quelltextzeile eine Zeilennummer ein. Diese Zeilennummer **muß** von einem beliebigen Zeichen außer dem Leerzeichen gefolgt werden, das das Ende der Zeilennummer und damit den Beginn der eigentlichen Quelltextzeile bezeichnet. Dieses Zeichen dient ausschließlich Formatierungs-zwecken, es wird nicht in den Speicher übernommen! Vergessen Sie es, wird eventuell das erste Zeichen Ihrer Eingabezeile hinter der Zeilennummer und ungleich dem Leerzeichen bei der Formatierung »verschluckt«, so daß Sie in den meisten Fällen eine der beiden Fehlermeldungen »not a mnemonic« oder »not implemented« erhalten werden. Das normale Aussehen einer Zeilennummer für den ASE-Editor ist damit wie folgt:

```
          100-
oder      120.
oder       90+
```

usw. Falls Ihnen diese Vorgehensweise ziemlich lästig erscheinen sollte – in den meisten Fällen übernimmt die automatische Zeilennumerierung des ASE-Editors neben der Ausgabe einer neuen Zeilennummer auch gleich die Ausgabe des Begrenzungszeichens in Form eines »-«. Sie brauchen sich also nur dann an diese Regel zu erinnern, wenn Sie Zeilen ohne die automatische Numerierung eingeben wollen.

(2) Jede Eingabezeile darf höchstens einen Befehl an den Assembler, also einen Pseudoop oder einen Prozessorbefehl, ein Mnemonic, enthalten. Wenn Sie sich die Listings der Beispielprogramme ansehen, wird dies offensichtlich. Diese Beschränkung ist bei Assemblern allgemein üblich und soll eine bessere Übersichtlichkeit des Quelltextes gewährleisten.

(3) Wollen Sie Kommentar in eine Quelltextzeile einfügen, so wird dieser Kommentar **immer** durch ein einleitendes Semikolon begonnen. Auch dies ist allgemein üblich und anhand der Beispielprogramme nachzuvollziehen.

```
          100-      lda #0   ;  Kommentar
oder      110-      .byte 1  ;  Kommentar
```

(4) Soll eine Quelltextzeile nur aus Kommentar bestehen – beispielsweise eine Überschrift über einem Unterprogramm – so erwartet der Formatierer des ASE das einleitende Semikolon direkt hinter dem Begrenzungszeichen der Zeilennummer. Steht zwischen dem Begrenzer und dem Semikolon ein Leerzeichen, wird eine Fehlermeldung »not a mnemonic« ausgegeben, da der Formatierer in diesem Fall einen Befehl erwartet. Reine Kommentarzeilen sehen also folgendermaßen aus:

```
          100-;
oder      110-; Ueberschrift
oder      120-;----------------------------
```

(5) Soll ein interner Label vor einem Befehl in einer Quelltextzeile stehen und damit die Adresse dieses Befehls symbolisieren, so muß auch der Label direkt hinter dem Begrenzer der Zeilennummer beginnen. Sie erhalten, wie gehabt, die Fehlermeldung »not a mnemonic«, wenn Sie Leerzeichen zwischen Begrenzer und Label lassen. Beispiele:

```
          100-label    lda #0
oder      110-marke    .byte 0,1,2,3
oder      120-_debug   jsr uprog1
```

Der Pseudoop ».byte« sollte Sie hier nicht weiter stören. Er soll lediglich andeuten, daß Label ja nicht nur vor Prozessorbefehlen stehen können, sondern genausogut auch vor Pseudoops.

(6) Zwischen Begrenzungszeichen der Zeilennummer und einem Mnemonic **muß** mindestens ein Leerzeichen stehen. Das Leerzeichen hinter dem Begrenzer wertet der ASE-Formatierer als Zeichen für ein folgendes Mnemonic, deshalb gab es in den beiden letzten Regeln auch eine entsprechende Fehlermeldung für ein fehlendes oder falsches Mnemonic, wenn ein Leerzeichen hinter dem Begrenzer folgte. Die gleiche Anordnung gilt auch für einen Label, dem ein Mnemonic folgt: Label und Mnemonic werden durch mindestens ein Leerzeichen voneinander getrennt. Wieder einige Beispielzeilen:

```
          100- lda#0
oder      110-label ldxmarke,y
```

So sehen zwei Zeilen aus, bevor sie durch den Formatierer gelaufen sind. Die gleichen Zeilen, wie sie nach der Eingabe die Eingabezeilen überschreiben:

```
          100  -          lda  #0
und       110  -label     ldx  marke,y
```

(7) In einer Quelltextzeile kann immer auch nur ein Label stehen. Es ist nur ein Zeilenbegrenzungszeichen hinter der Zeilennummer vorhanden, so daß die Regel 7 sich logisch schon aus weiter oben stehenden Regeln ableiten ließe. Sie sei trotzdem noch einmal ausdrücklich erwähnt.

(8) Label können auch ohne folgenden Befehl in einer Quelltextzeile stehen. Dies kann verwendet werden, falls man eine Adresse mit einem Label markieren will, die schon von einem anderen Label besetzt ist, oder wenn ein bestimmter Pseudoop kein Label zuläßt (auch solche gibt es). Beispiele:

```
          100-label
          110-       .macend

oder      100-label1
          110-label2      lda #0
```

Im ersten Fall ist vor ».macend« kein Label erlaubt. Der Label wird in einem solchen Fall in eine eigene Zeile geschrieben. Der zweite Fall weist zwei Labeln die gleiche Adresse zu – die des Befehls »lda #0«. Wie aus den beiden Beispielen abzulesen ist, ist hinter Einzellabeln in einer Zeile **kein Kommentar** erlaubt. Sie erhalten wieder die Fehlermeldung »not a mnemonic«, falls Sie es trotzdem einmal versuchen wollen.

(9) Pseudoops – die Anweisungen an den Assembler, die mit einem Punkt beginnen – können an zwei Stellen innerhalb der Quelltextzeile positioniert werden. Wird der Pseudoop mit einem führenden Leerzeichen vor dem Punkt eingegeben, erscheint er nach der Formatierung in einer Reihe mit den Mnemonics. Zweite Möglichkeit: der Punkt wird wieder direkt hinter dem Zeilenbegrenzer eingegeben. In diesem Fall erscheint der Pseudoop in einer Reihe mit den Labeln. Beide Möglichkeiten sind für bestimmte Pseudoops sinnvoll. Beispiele:

```
100-.define label = $1234
110-            .byte 0,0,0
```

(10) Da der ASE eine strukturierte Programmierung in Form von speziellen Strukturpseudos unterstützt – Pseudoops wie z. B. ».repeat« oder ».until« – wäre eine Positionierung von Pseudoops an nur zwei Stellen innerhalb einer Quelltextzeile nicht genug. Strukturierte Programme zeichnen sich ja unter anderem auch dadurch aus, daß die Programmstruktur durch das Aussehen des Programm-

textes – Einrückungen von Befehlen – deutlich wird. Aus diesem Grund erlaubt der Formatierer des ASE-Editors Leerzeichen zwischen dem einleitenden Punkt eines Pseudoops und dem Schlüsselwort selbst. Beispiele:

```
        100-.   repeat
        110-.   until #e
oder    120-            .   do
        130-            .   while #e
```

An dieser Stelle wäre es zweifellos das beste, wenn Sie nun einige Quelltextzeilen versuchsweise eingäben, um die Wirkung des Editors und seine Reaktionen beobachten zu können. Nehmen Sie dazu beispielsweise einige Zeilen aus einem Beispielprogramm. Versuchen Sie auch einmal, was passiert, wenn Sie einzelne der obigen Regeln nicht einhalten!

Eine Anmerkung noch, wie die einzelnen Quelltextzeilen in den Speicher des Rechners gebracht werden: Sie wissen vielleicht, daß Basic-Befehlswörter bei der Übernahme des Programmtextes in den Speicher »tokenisiert« werden, d. h. die Befehlswörter werden zu einem Byte, dem Token, verdichtet. Eine ähnliche Tokenisierung unternimmt auch der ASE-Editor – Pseudoops und Mnemonics werden in Form von Token im Quelltext aufbewahrt. Gleichzeitig werden alle überflüssigen Leerzeichen aus der Quelltextzeile entfernt, wodurch sich insgesamt ein erheblich geringerer Speicherplatzbedarf für Quelltexte ergibt, Sie können also längere zusammenhängende Quelltexte schreiben. Die Werte der verwendeten Token können Sie dem Anhang entnehmen, falls Sie einmal eine Anwendung entwickeln wollen, die Top-Ass-Quelltexte verarbeiten soll.

Alle Pseudobefehle des Assemblers können bei der Eingabe auch abgekürzt angegeben werden; eine ähnliche Einrichtung bietet auch Basic. Die Abkürzungen können Sie ebenfalls dem Anhang entnehmen oder sich auch selbst überlegen. Das Prinzip der im ASE-Editor verwendeten Abkürzungen ist recht einfach – der erste für ein Befehlswort einzigartige oder ein späterer Buchstabe wird großgeschrieben. Beispiele:

```
.byte     =     .bY
.base     =     .bA
```

Das »b« ist beiden Befehlswörtern gemeinsam, muß also kleingeschrieben werden. Es gibt allerdings keinen Befehl mit einem führenden »b« und einem nachfolgenden »y«, so daß ».bY« eine gültige Abkürzung für ».byte« ist. Das gleiche gilt dann natürlich auch für die Abkürzung ».byT«. Sollten Sie im Zweifelsfall eine Abkürzung nicht mehr in der Erinnerung haben, können Sie sie nach diesen Überlegungen leicht am Bildschirm rekonstruieren, auch ohne dafür unbedingt den Anhang bemühen zu müssen.

3.2.2 Editorbefehle im einzelnen

Sämtliche Editorbefehle – bis auf die berühmte Ausnahme – werden am Bildschirm durch einen führenden Punkt eingeleitet, wie es bei den Pseudoops auch schon der Fall war. Diesem Punkt folgt ein Buchstabe, der den Befehl bezeichnet, sowie eventuelle Parameterangaben, die der Befehl benötigt. Parallel zu den Editorbefehlen können Sie die meisten Basicbefehle im Direktmodus weiter benutzen! Geben Sie beispielsweise »directory« ein, wird das Inhaltsverzeichnis einer Diskette angezeigt, mit »scratch« löschen Sie Dateien einer Diskette, mit »print« können Sie wie gewohnt rechnen. Wenn Sie so wollen, können Sie den gesamten ASE als eine Befehlserweiterung des Basic-Interpreters betrachten – allerdings eine Erweiterung, die größer ist als der gesamte Interpreter selbst.

Die einzelnen Befehle des Editors sollen im folgenden in Funktionsgruppen vorgestellt werden, so, wie sie in der täglichen Arbeit benötigt werden.

3.2.2.1 Edierbefehle

Unter *Edierung* eines Textes versteht man alles, was zum Schreiben eines Textes gehört: Löschen und Einfügen von Text, Ansehen des geschriebenen Textes am Bildschirm und alle Hilfsfunktionen, die damit in engerem oder weiterem Zusammenhang stehen.

Automatische Zeilennumerierung:

Der Befehl **».a nn«** schaltet die automatische Zeilennumerierung des ASE-Editors ein. »nn« ist dabei eine Schrittweite; diese Schrittweite wird jeweils zur Nummer der letzten eingegebenen Zeile addiert, um die neue Zeilennummer zu erhalten. Der ASE-Editor gibt außer der Zeilennummer auch das Begrenzungszeichen hinter der Nummer aus, deswegen sollten Sie diesen Befehl statt des Basicbefehls »auto« benutzen. Der Cursor wird durch die automatische Zeilennumerierung, mit einem Leerzeichen Abstand, hinter das Begrenzungszeichen »-« gesetzt. Da die Mehrzahl der einzugebenden Quelltextzeilen ein Leerzeichen am Zeilenstart erfordert, ist diese Vorgehensweise vorteilhaft.

Falls Sie einmal mit dem Hypra-Ass auf dem C-64 gearbeitet haben oder falls Sie den Top-Ass V1.0 kennengelernt haben, seien hier die wichtigen Änderungen der Wirkungsweise der Auto-Funktion noch einmal erwähnt: Der Cursor kann auch während die Auto-Funktion aktiviert ist, beliebig über den Bildschirm geführt werden. Die nächste ausgegebene Zeilennummer ergibt sich aus der Nummer der Zeile, in der Sie *Return* geben und der eingestellten Schrittweite.

Die automatische Zeilennumerierung kann auf zweierlei Art und Weise durch den Anwender desaktiviert werden. Die einfachste und wohl gebräuchlichste Methode ist die, in einer neuen Zeile, direkt, nachdem die neue Zeilennummer ausgegeben wur-

de, einfach *Return* zu geben. Die zweite besteht in der Eingabe von ».a« ohne Schrittweite. Um eine neue Schrittweite zu wählen, braucht die Zeilennumerierung nicht vorher abgeschaltet zu werden, es genügt die Eingabe des Befehls mit der neuen Schrittweite. Auch eine automatische Abschaltung der Zeilennumerierung durch den Editor findet statt. Diese folgt exakt der des Basic-Editors (nach »load« etc.).

Im Zusammenhang mit der Eingabe von Quelltexten und da die Eingabe zu fast 100 Prozent unter Zuhilfenahme der automatischen Zeilennumerierung geschieht, sei hier auf ein spezielles Problem hingewiesen: Sicher werden Sie im Laufe der Zeit beabsichtigen, Ihre Quelltexte inklusive erzeugten Maschinenkodes usw. in Form eines Assemblerlistings ausdrucken zu lassen. Normalerweise passen die einzelnen Quelltextzeilen samt Zeilennummer, Opcodes und Adressen in eine Druckerzeile, wenn Sie die Schriftart »Elite« verwenden (96 Zeichen/Zeile). Die Schriftauswahl Ihres Druckers werden Sie in Ihrem Druckerhandbuch finden. Ich mußte aber leider feststellen, daß einige gebräuchliche Drucker-Interfaces mit der gewählten maximalen Zeilenlänge von 96 Zeichen nicht zurechtkommen. Diese Interfaces erzwingen eine Zeilenlänge von 80 Zeichen durch interne Erzeugung eines *Carriage Returns* auch dann, wenn druckerseitig eine größere Zeilenlänge eingestellt ist. In einem solchen Fall bleibt Ihnen fast nichts anderes übrig, als die Quelltextzeilen bereits bei der Eingabe so kurz zu halten, daß sie zusammen mit den anderen Ausgaben eines Assemblerlistings in eine Druckzeile passen. Diese Anpassung sollten Sie **unbedingt** vornehmen, wenn Sie saubere Listings auf dem Drucker erhalten wollen! Folgende Vorgehensweise hat sich auf dem 80-Zeichen-Schirm bewährt, um sicherzugehen, daß die Zeilenlänge paßt: Löschen Sie den Bildschirm mit »CLR/HOME«, geben Sie zweimal »TAB« ein und als letztes die Tastenfolge »ESC« und »B«. Sie definieren damit ein schmaleres Bildschirmfenster, dessen Breite fast genau der maximalen Breite einer Quelltextzeile in einem Assemblerlisting entspricht.

Löschen von Zeilen und Zeilenbereichen:

Quelltextzeilen werden **immer** durch den Befehl »delete«, abgekürzt »deL«, gelöscht. Die Eingabe einer alleinstehenden Zeilennummer bewirkt nichts – anders als im Basic-Editor, der dann die betreffende Zeile löscht. Die Parameter hinter »delete« und ihre Eingabe dürften Sie kennen, sie sollen aber der Vollständigkeit halber noch einmal gezeigt werden:

```
deL   100        - Löschen einer Zeile
deL   -100       - Löschen bis Zeile (einschließlich)
deL   100-       - Löschen von Zeile (einschl.) bis
                   zum Ende des Textes
deL   100-200    - Löschen eines Zeilenbereiches
                   (beide Zeilennummern einschließ-
                   lich)
```

Zum Löschen des gesamten Textes sollte nicht der Befehl »deL« verwendet werden, sondern das ebenfalls von Basic bekannte »**new**«.

Formatiertes Listen:

Das besondere Format der Quelltextzeilen, das durch den ASE-Editor hergestellt wird, erfordert natürlich auch eine besondere Art von List-Routine, die den Quelltext formatiert auf dem Bildschirm zeigen kann. Der Basicbefehl »list« liefert zwar noch eine Ausgabe, diese ist aber nicht zu verwerten. Probieren Sie einmal aus, welches Ergebnis das Listen eines Quelltextes mit »list« ergibt!

Der Ersatzbefehl zu »list« heißt ».e«. Hinter ».e« folgt die Angabe einer Zeilennummer oder eines Zeilenbereiches, wie sie auch hinter »list« erwartet wird. ».e« ohne Parameter listet den gesamten Quelltext; dieser Befehl findet sich auch auf der Funktionstaste »F5« (nicht auf F 7!!!).

Die Ausgabe am Bildschirm kann durch einfaches Drücken der Leertaste zeitweilig angehalten werden; ein zweiter Druck auf die Leertaste läßt das Listen wieder anlaufen. Zum endgültigen Abbruch des Listens dient nicht die Taste »Stop«, sondern »Return«!

Tabulatoren:

Das Format der Quelltextzeilen kann durch drei Tabulatoren beeinflußt werden:

(1) ».**t0,nn**« bestimmt Spalte »nn« als Tabulator für die Position der Mnemonics in einer formatierten Zeile.

(2) ».**t1,nn**« bestimmt auf die gleiche Weise die Position des Kommentars hinter den Mnemonics und Pseudoops.

(3) ».**t2,nn**« setzt die Position des Gleichheitszeichens hinter dem Pseudoop ».define« auf Spalte »nn«.

Die beim Start des Assemblers verwendeten Tabulatoren sind abhängig vom Bildschirmformat, das der Assembler beim Start vorfindet.

Arbeitsbereiche (Pages):

Ein Arbeitsbereich oder eine »Page« ist nichts anderes als ein fester Zeilenbereich, dem Sie einen Index zuweisen können. Die fünf Arbeitsbereiche (0 bis 4), die Sie im ASE-Editor definieren können, dienen hauptsächlich zwei Zwecken: Zum einen können Sie einen bestimmten Bereich markieren, so daß Sie ihn schnell wieder aufsuchen (listen) können. Geben Sie mit

```
.pl,nn,mm
```

die Page 1 mit einem Zeilenbereich von »nn« bis »mm« ein, so listet der Befehl

```
.1
```

die Page – natürlich nur, wenn die angegebenen Zeilen existieren. Diese Möglichkeit, schnell zwischen verschiedenen Zeilenbereichen hin- und herschalten zu können, ist in längeren Quelltexten sehr nützlich, da man doch dazu neigt, im Laufe der Zeit die Übersicht zu verlieren.

Die zweite Verwendungsart der Pages ist die als Parameter in anderen Befehlen. Da sehr häufig eine Reihe von Befehlen auf gleiche Zeilenbereiche angewendet wird, kann man besser den kurzen Index eines Arbeitsbereiches eingeben als die beiden Zeilennummern, die den Bereich festlegen.

Neu-Durchnumerieren von Pages:

Pages können durch Eingabe des Befehls **».n p,nn,mm«** mit neuen Zeilennummern versehen werden; »p« ist die Nummer der zu numerierenden Page, »nn« ist die Nummer, die die erste Zeile dieses Bereiches erhalten soll, und »mm« ist die Schrittweite, mit der das »Renumber« erfolgen soll. Wird »p« weggelassen und nur das Komma eingegeben, so betrachtet der Editor dies als Eingabe der Pagenummer 0. Beachten Sie folgenden Umstand: Sie können einen Zeilenbereich inmitten des Quelltextes mit beispielsweise größeren Zeilennummern versehen als der Bereich dahinter – zum Ende des Quelltextes hin – aufweist. Dies bedeutet nicht, daß dieser Zeilenbereich in irgendeiner Weise verschoben worden wäre! Die physikalische Reihenfolge der Zeilen im Speicher ist vielmehr vollkommen unverändert geblieben, wie Sie sich durch ein Listen des Gesamttextes mit ».e« überzeugen können!

Verwenden Sie nicht den Basicbefehl »renumber«! Dieser versucht, die Zeilennummern hinter »goto«, »gosub« usw. an die bei der Numerierung veränderten Zeilennummern anzupassen. Da die Token eines ASE-Quelltextes eine andere Bedeutung haben als die eines Basicprogrammes, wird dieser Versuch nicht »nur nicht von Erfolg gekrönt« sein, sondern sogar Ihren Quelltext beschädigen!

Einfügen von Zeilen:

Ein normales Einfügen von Zeilen kann wie in Basic erfolgen, indem man einfach eine Reihe von Zeilen mit Zeilennummern im Bereich zwischen denen eingibt, die die Zeilen unterhalb und die oberhalb des Einfügepunktes besitzen.

Sollen mehr Zeilen eingefügt werden als der zu Verfügung stehende Bereich am Einfügepunkt erlaubt – Beispiel: Zwischen die Zeilen 10 und 15 sollen 10 Zeilen eingefügt werden – so gibt es zwei praktikable Methoden, um dies zu erreichen:

(1) Sie numerieren den Bereich hinter dem Einfügepunkt um, so daß die Zeilennummer hinter dem Einfügepunkt entsprechend groß ist, daß alle einzufügenden

Zeilen vor diese Zeilennummer passen. Anschließend fügen Sie Zeilen unter Verwendung des Befehls ».a« ein.

(2) Da die Methode (1) bei größeren Bereichen nicht sehr effektiv ist, sollten Sie in diesem Fall anders vorgehen. Schreiben Sie zunächst den einzufügenden Text in einen Quelltext, den Sie auf Diskette sichern. Als nächstes laden Sie den Text, in den Sie einfügen möchten, und definieren einen Arbeitsbereich, der von der Zeile hinter dem Einfügepunkt bis zum Textende reicht. Diesen Bereich können Sie auf Diskette speichern und anschließend mit »deL« löschen. Mit dem später noch beschriebenen Befehl ».m« (merge) laden Sie nun nacheinander den einzufügenden Textteil und den soeben gesicherten Textrest hinter den im Speicher befindlichen Text. Führen Sie anschließend noch ein Renumber mit einem Zeilenbereich von bespielsweise 0 bis 65000 durch, haben Sie die Einfügung perfekt vorgenommen.

Ausdrucken des Quelltextes:

In einigen Fällen kann es erwünscht sein, einen Quelltext nicht als Assemblerlisting, sondern so, wie er ist, auszudrucken. An dieser Stelle wird beschrieben, wie Sie hierzu vorzugehen haben; die Methode entspricht ganz dem, was Sie auch unter Basic vornehmen würden, um einen Programmtext zu drucken.

Schritt 1: Geben Sie die Befehlsfolge ein »**open 1,4,7: cmd 1**«. Hiermit öffnen Sie einen Ausgabekanal zum Drucker. Die Sekundäradresse im Befehl »open« kann von Bedeutung sein, um den Drucker oder das von Ihnen verwendete Interface auf eine bestimmte Betriebsart einzustellen. Sehen Sie hierzu die Beschreibungen dieser Geräte.

Schritt 2: Geben Sie nun den Befehl zum formatierten Listen des Bereiches, den Sie drucken wollen, und warten Sie das Ende des Ausdrucks ab.

Schritt 3: Als letztes geben Sie den Befehl »**print #1: close 1**«, um die Ausgabe zu beenden und den Ausgabekanal zu schließen. Wenn Sie diesen letzten Schritt vergessen, wird u. U. ein Teil der Quelltexte (z. B. die letzte Zeile) nicht gedruckt, sondern »vom Drucker verschluckt«.

3.2.2.2 Laden, Speichern, DOS-Support

Einmal geschriebene Quelltexte müssen natürlich auch auf Diskette gesichert und wieder geladen werden können, man muß die Inhaltsverzeichnisse von Disketten einsehen können, und was in diesem Zusammenhang noch erforderlich ist.

Für diese Arbeiten kommt uns wieder einmal zugute, daß der ASE-Editor als »Basic-Befehlserweiterung« arbeitet. Sämtliche Befehle, die unter Basic 7.0 verwendet werden können – »directory«, »scratch«, »rename« und »print ds$«, um die wichtigsten zu nennen – stehen uns damit auch im Editor des Assemblers zur Ver-

fügung. Quelltexte werden wie Basicprogramme geladen (»dload«) und gespeichert (»dsave«). Man braucht also keine neuen Befehle zu lernen, um auch diesen Teil des Editors vollkommen zu beherrschen.

Nur zwei Befehle vermißt man bei der Arbeit mit dem Basic-Editor in diesem Zusammenhang: Ein Zusammenfügen zweier Dateien von Diskette (»merge«) und die Abspeicherung von Teilen eines Programmes. Diese beiden sind im ASE-Editor hinzugekommen.

»Mergen« von Quelltexten:

Befindet sich bereits ein Quelltext im Speicher des Rechners, an dessen Ende ein weiterer von Diskette angehängt werden soll, so kann man hierfür den Befehl

```
.m "dateiname", u9
```

verwenden. Die Parameter hinter ».m« entsprechen denen des Basicbefehls »dload«; das Anhängsel », u9« soll andeuten, daß auch Quelltexte von einem zweiten oder weiteren Laufwerk geladen werden können. Erfolgt das Laden vom Laufwerk mit der Nummer 8, kann das Kürzel entfallen, in diesem Fall genügt also die einfache Angabe eines Filenamens.

Im allgemeinen wird nach erfolgtem Nachladen eine Neunumerierung des gesamten Quelltextes notwendig werden. Sehen Sie hierzu die Befehlsbeschreibung im letzten Abschnitt.

Abspeichern einer Page:

Der Befehl zum Abspeichern einer Page hat folgendes Format:

```
.s p, "dateiname", u9
```

»p« bezeichnet wieder eine Pagenummer, das Kürzel », u9« kann entfallen, falls Sie auf dem Laufwerk 8 abspeichern wollen. Der abgespeicherte Quelltextteil bleibt unverändert im Speicher stehen.

3.2.2.3 Suchen und Ersetzen im Quelltext

Dies sind ebenfalls zwei Befehlsarten, die ein guter Editor unbedingt beherrschen sollte, die der Basic-Editor aber leider nicht zur Verfügung stellt.

Prinzipiell muß man beim Suchen und Ersetzen innerhalb von Quelltexten zwei Befehls-Unterarten unterscheiden – das Suchen und Ersetzen von normalen Strings und das Suchen und Ersetzen von Befehlen.

3.2.2.3.1 Suchen und Ersetzen von Befehlen

Wie Sie wissen, wird jede Quelltextzeile bei der Übernahme in den Speicher tokenisiert, d. h. die in der Zeile enthaltenen Mnemonics oder Pseudoops werden in To-ken umgewandelt; gleichzeitig werden alle überflüssigen Leerzeichen aus der Quelltextzeile entfernt, um Speicherplatz zu sparen. Diese Vorgehensweise bedingt, daß ein Suchstring – ein String, der gefunden werden soll –, der Mnemonics, Pseudoops und Leerzeichen enthält, vor dem eigentlichen Suchen in der gleichen Art und Weise bearbeitet werden muß. Die Schlüsselwörter im String müssen in Token verwandelt werden, die überflüssigen Leerzeichen, beispielsweise zu Beginn des Suchstrings, müssen entfernt werden. Erst das Ergebnis dieser Umwandlung kann dann mit dem Inhalt des Quelltextes verglichen werden.

Ich gebe zunächst einmal die Form eines Suchbefehles an, dessen Suchstring umgewandelt wird:

```
.fm p "suchstring"
```

Beachten Sie, daß zwischen der Pagenummer »p« und dem folgenden String kein Komma steht! Die Pagenummer kann wieder weggelassen werden, sie wird in diesem Fall gleich Null gesetzt; »fm 0 "string"« ist also gleichbedeutend zu »fm "string"«.

Zur Tokenisierung des Suchstrings verwendet der ASE-Editor die gleiche Routine, die auch schon für die Umwandlung von Quelltextzeilen herangezogen wurde. Dies bedingt, daß innerhalb des Suchstrings die gleichen Formatregeln wie in einer Quelltextzeile gelten! Der Zeilenbegrenzer ist hier nicht das »-« hinter der Zeilennummer, sondern das Anführungszeichen. Gleichzeitig werden bei der Umwandlung die gleichen Fehlermeldungen wie bei der Eingabe einer Quelltextzeile ausgegeben; Mnemonics und Pseudoops werden auf ihre Zulässigkeit überprüft. Das beste wird sein, einige Beispiele für diese Art von Suchbefehl durchzuspielen, damit die ganze Sache noch deutlicher wird. Um Leerzeichen innerhalb der Suchstrings besser sichtbar zu machen, verwende ich hier den Unterstrich an ihrer Stelle.

(1) ».fm 1 "_lda#0"«: Sucht in Page 1 alle Vorkommnisse des Befehls »lda #0«. Vor den gefundenen Befehlen können Label stehen, dahinter kann Kommentar folgen.

(2) ».fm 1 "lda#0"«: Sucht einen ASCII-String »lda#0«, der sich mit großer Wahrscheinlichkeit nirgends im Quelltext finden wird. Das fehlende Leerzeichen zu Beginn des Suchstrings läßt den Editor vermuten, daß ein Label folgt.

(3) ».fm 1 "dies_und_jenes«: Ergibt eine Fehlermeldung »not a mnemonic«. »dies« wird als Label interpretiert und das Wort »und« ist zweifellos kein Mnemonic.

Innerhalb der Suchstrings sind Fragezeichen als Joker erlaubt. Jedes Fragezeichen steht für eine Zeichenposition, die in Suchstring und gefundenem String nicht übereinzustimmen braucht. Joker sind innerhalb von Schlüsselwörtern, also innerhalb von Mnemonics und Pseudoops, nicht erlaubt! Drei weitere Beispiele:

(4) ».fm 1 "_jmp_lab??"«: Sucht alle »jmp«-Befehle, denen Label aus mindestens fünf Buchstaben folgen, deren erste drei Zeichen »lab« sind.

(5) ».fm 1 "???_.bY": Sucht alle Vorkommnisse des Pseudoops ».byte« (Abkürzung), vor denen ein mindestens drei Zeichen langer Label steht.

(6) .fm 1 ".mA_a??(": Sucht alle Makrodefinitionen von Makros, die mit dem Buchstaben »a« beginnen und deren Name genau drei Zeichen lang ist.

Wird ein String innerhalb des Quelltextes gefunden, so wird die gesamte Zeile am Bildschirm gelistet. Tritt der String ein zweites Mal in der gleichen Zeile auf, erhalten Sie sie auch zweimal angezeigt. Die Suche kann vorzeitig abgebrochen werden, indem Sie die Taste »Return« drücken, »Stop« hat keine Wirkung.

Das Ersetzen von Befehlen verläuft in der gleichen Art. Die beiden Ersetzbefehle lauten:

```
        .rm p "ersetzstring","suchstring"
und     .Rm p,"ersetzstring","suchstring"
```

Beiden Befehlsarten ist gemeinsam, daß der Suchstring wieder Fragezeichen als Joker enthalten darf. Unterschiedlich ist die Art, in der die Ersetzung vorgenommen wird: Der zweite Befehl fragt den Benutzer vor jeder Ersetzung, ob sie wirklich stattfinden soll; der erste Befehl nimmt die Ersetzung automatisch und ohne Rückfrage vor.

».rm« listet alle Zeilen nach erfolgter Änderung am Bildschirm. Sollte eine Ersetzung mehrfach in einer Zeile möglich gewesen sein, erhält man die Zeile auch entsprechend oft gelistet.

».Rm« geht anders vor: Zunächst wird die Zeile angezeigt, in der der Suchstring gefunden wurde. Darunter erscheint die Nachfrage:

```
»replace, skip or cancel (r, s, c) ?«
```

Die Routine wartet nun auf eine Eingabe von der Tastatur. Geben Sie ein »r« ein, wird die Ersetzung durchgeführt. Die geänderte Zeile wird mit dem Vermerk »new line« gelistet. Geben Sie dagegen ein »s« ein, wird dieses Vorkommen des Suchstrings übersprungen, die Zeile bleibt soweit unverändert. Mit »c« für »cancel« brechen Sie die gesamte Ersetzfunktion vorzeitig – bevor das Ende des Quelltextes erreicht wurde – ab, und der Cursor erscheint wieder auf dem Bildschirm.

In allen besprochenen Fällen ist die Länge der Such- und Ersetzstrings begrenzt. Sie darf 49 Zeichen nicht überschreiten, sonst wird die Fehlermeldung »string too long« ausgegeben.

3.2.2.3.2 Suchen und Ersetzen von ASCII-Strings

Bei der Suche und auch beim Ersetzen von reinen Strings ohne Tokenisierung treffen wir auf Probleme, die sich ebenfalls wieder aus dem Quelltextformat des ASE-Editors ergeben: Die Mnemonics und Pseudoops kann man am Bildschirm nicht, oder nur sehr schlecht unter Zuhilfenahme von Grafikzeichen, eingeben und alle überflüssigen Leerstellen (Blanks) wurden aus den Quelltextzeilen vor der Übernahme in den Speicher entfernt. Der erste Punkt wurde durch die Aufnahme besonderer Such- und Ersetzbefehle in den ASE-Editor gelöst. Der zweite Punkt mahnt uns bei den hier besprochenen Befehlen zur Vorsicht: Entfernte Leerzeichen kann man natürlich erstens nicht finden; zweitens sollte man bei Ersetzungen darauf achten, nicht überflüssige Leerstellen in die Quelltextzeilen einzubringen. Der ASE zeigt sich zwar unbeeindruckt von solchen Leerstellen, die nicht seinem Quelltextformat entsprechen, bei weiteren Ersetzungen kann es aber zu Kollisionen kommen. Haben Sie beispielsweise durch eine der hier besprochenen Ersetzungen hinter einige Mnemonics – z. B. hinter einige »lda«-Befehle – ein Leerzeichen eingefügt und versuchen Sie anschließend, mit den Befehlen ».rm« oder ».Rm« weitere Ersetzungen – z. B. »lda #« durch »lda« – vorzunehmen, werden diese Befehle die unzulässig durch das Leerzeichen erweiterten Quelltextzeilen nicht mehr zu ihrem Suchbereich zählen! In ihrem Suchstring ist das Leerzeichen durch die Tokenisierung nicht mehr enthalten, deshalb fällt ein Vergleich zwischen Suchstring und Quelltextzeile negativ aus und eventuell bleiben einige eigentlich zu ändernde Quelltextzeilen unberührt! Ich denke, das Problem ist klar geworden.

Trotz einiger Schwierigkeiten, wie oben zu sehen, sind die Funktionen zum Suchen und Ersetzen von ASCII-Strings nützlich, wenn Such- und Ersetzstrings keine Mnemonics oder Pseudoops enthalten. In diesen Fällen sind sie absolut gleichwertig zu den weiter oben beschriebenen Befehlen mit Tokenisierung – solange man Leerstellen weder einfügt noch nicht vorhandene Leerstellen zu suchen versucht.

Die Befehle:

```
.f p "suchstring"
.r p "ersetzstring","suchstring"
.R p "ersetzstring","suchstring"
```

In allen weiteren Belangen gelten die Beschreibungen der oben besprochenen Befehle mit Tokenisierung.

3.2.2.4 Rechnen im Editor/Die Integerarithmetik

Wie Sie wissen, kann man die Arbeit des ASE-Editors als Funktion einer sehr umfangreichen Basicerweiterung auffassen. Basicbefehle können im Direktmodus weiter wie gewohnt verwendet werden. Als Basicerweiterung greift der Editor aber auch in die Funktion des Basic-Interpreters selbst ein – er fügt in den Interpreter die gewöhnliche Schreibweise für sedezimale Zahlen ein, z. B. »$c000« für »49152« oder »$100« für »256«. Dieses Zahlenformat kann nach Start des ASE-Editors in allen Basicbefehlen verwendet werden. Beispiele:

```
print $c000/ 2
sys $c00
poke $1234, $1a
```

Insbesondere die Verwendung innerhalb des Befehles »print« ist für Rechnungen mit sedezimalen Zahlen, die man immer wieder einmal braucht, interessant. Nebenher erleichtert sich natürlich auch der Umgang mit »sys« usw., weil man sich die Verwendung der Umrechnungsfunktion »dec« des Interpreters sparen kann.

Eine zweite Möglichkeit, um im ASE-Editor zu rechnen, bietet der Einsatz des Editorbefehls »_« **(in der ASCII-Belegung entspricht dies dem Pfeil nach links)**. Der ASE-Assembler besitzt wie der Basic-Interpreter eigene Routinen zur Bewertung arithmetischer Ausdrücke. Mit »_« rechnen Sie, ähnlich wie bei »?«, mit dieser Arithmetik statt die Interpreter-Routinen zu benützen. »_« gibt Ergebnisse **immer** sedezimal aus, während »?« immer ein dezimales Ergebnis liefert. Damit ergibt sich ein einfacher Weg, Dezimalzahlen und Sedezimalzahlen ineinander umzurechnen, was letztlich auch der Zweck der Einführung dieses Editorbefehls war.

Im Gegensatz zum Interpreter benötigt der Assembler weder Fließkommazahlen noch negative Zahlen, die Arithmetik des Assemblers rechnet aus diesem Grunde nur mit Integerzahlen im Bereich zwischen Null und 65535 (sedezimal 0 – $ffff). Daß diese Integerarithmetik für einen Assembler trotzdem extrem leistungsfähig ist, werden Sie den folgenden Beschreibungen entnehmen können. Das beste wird sein, Sie testen die einzelnen Funktionen der Integerarithmetik zunächst einmal mit »_« aus, denn einige Funktionen und Rechnungsarten arbeiten anders, als Sie dies von Basic her gewohnt sind.

Beschreibung der Integerarithmetik:

Die Integerarithmetik beherrscht drei verschiedene Zahlenformate: Binärzahlen werden durch ein vorgestelltes Prozentzeichen, Sedezimalzahlen durch ein Dollarzeichen eingeleitet, Dezimalzahlen werden wie gewohnt verwendet:

```
$c000
%1000'0001
1234
```

Wie Sie sehen, können Binärzahlen durch eingestreute Hochkommata lesbarer gestaltet werden. Die Zahl dieser Hochkommata ist nicht begrenzt. Führen Sie einmal einige Beispielrechnungen durch:

```
_256
 $0100          erscheint als Ergebnis
```

Oder:

```
_%1000'1101, %0111'1000
 $008d $0078
```

Mehrere Rechnungen können gleichzeitig mit »_« ausgeführt werden. Hierzu werden die einzelnen arithmetischen Ausdrücke hinter »_« einfach durch Kommata voneinander getrennt.

Zwei vielleicht etwas merkwürdig aussehende Funktionen ermitteln den ASCII-Wert bzw. den Bildkodewert eines Zeichens:

```
_"a"
 $0041          liefert den ASCII-Wert

_'a'
 $0001          liefert den Bildkode
```

Zwei weitere Funktionen liefern das Highbyte bzw. das Lowbyte eines Ausdrucks:

```
_<(12345)
 $0039          Lowbyte eines Ausdrucks

_>(12345)
 $0030          Highbyte eines Ausdrucks
```

Alle vier Funktionen und natürlich die Zahlenkonstanten können untereinander durch eine ganze Reihe von Operatoren verknüpft werden. Eine beliebige Klammerung nach den gewohnten Regeln ist erlaubt. Die Liste der möglichen Operationen:

Grundrechenarten:

»+«	Addition
»-«	Subtraktion
»*«	Multiplikation
»/«	Division

Logische Operationen:

».and.«	logisches Und
».or.«	logisches Oder
».xor.«	exklusives Oder
».not.«	Negation (Einerkomplement)

Vergleiche:

».eq.«	gleich (equal)
».ne.«	ungleich (not equal)
».lt.«	kleiner als (less than)
».le.«	kleiner gleich (less or equal)
».gt.«	größer (greater than)
».ge.«	größer gleich (greater or equal)

Beachten Sie, daß einige Operatoren denen von Fortran nachgebildet sind: Sie werden in Punkte eingeschlossen, die zur korrekten Schreibweise der Operatoren gehören. Einige Beispielrechnungen:

```
_(1+"a")*2
 $0084
_%1101'1000.and.%1001'0000
 $0090

_.not.$12
 $ffed
```

Beachten Sie weiter, daß die Vergleiche der Integerarithmetik andere Ergebnisse liefern, als Sie es von Basic gewohnt sind: Ein Vergleich liefert den Wert 1, falls er wahr ist, sonst 0:

```
_1.eq.2
 $0000

_1.ne.2
 $0001
```

Der Stern »*« wird nicht nur als Operator für die Multiplikation benutzt, sondern auch als Zeichen für den aktuellen Inhalt des Programmzählers während der Assemblierung. Vor erfolgter Assemblierung weist der PC den Wert 0 auf, nach der Assemblierung enthält der PC die Adresse des letzten Bytes des erzeugten Maschinenprogramms plus Eins. Innerhalb eines Quelltextes – hier ist das eigentliche Einsatzgebiet der beschriebenen Arithmetik verwendet man den Stern, um Label zu sparen. Beispiel:

```
beq *+5 = marke beq marke+5
```

Der Stern enthält hier die laufende Adresse des Befehls »beq« innerhalb des Maschinenprogramms. Ich empfehle die Rechnung mit dem Stern für den Anfänger ausdrücklich nicht! Sie setzt auf jeden Fall voraus, daß Sie beispielsweise die Befehlslängen der im obigen Beispiel übersprungenen Befehle kennen. Ganz nebenbei wird der Quelltext im allgemeinen auch unübersichtlich, wenn Sie viele Sprünge relativ zum Programmzähler angeben.

Eine kleine, aber vermeidbare, Kollision bei Verwendung der Integerarithmeitik innerhalb des Quelltextes sei noch ausdrücklich erwähnt: Leider werden die runden Klammern innerhalb der Syntax der Prozessorbefehle benutzt, um die Adressierungs-arten »(),y« und »(,x)« anzuzeigen. Da die Klammern gleichzeitig auch innerhalb arithmetischer Ausdrücke erlaubt sind, muß der Assembler eine Prioritätsentschei-dung treffen: Wird hinter einem Mnemonic als erstes eine öffnende runde Klammer angetroffen, gilt diese Klammer als Zeichen der Adressierungsart. Dies ist der Grund, warum ein Befehl wie

```
lda (1+marke)*3
```

zu einer Fehlermeldung (»illegal address«) führt. Der Assembler erwartet nach der runden Klammer eine der Adressierungsarten »(),y« oder »(,x)«.

In diesem Fall kann man sich durch Umstellen behelfen:

```
lda 3*(1+marke)
```

wird korrekt verarbeitet.

Fehlermeldungen der Arithmetik:

Natürlich kann es vorkommen, daß von Ihnen eingegebene arithmetische Ausdrücke nicht korrekt sind. In diesen Fällen gibt die Arithmetik verschiedene Fehlermeldungen aus:

»illegal expression«: Ein Zeichen oder ein Operator eines Ausdrucks ist falsch oder ein Operator wird falsch verwendet. Beispiele:

```
_-1 oder _?
```

Im ersten Beispiel schlägt zu Buche, daß die Integerarithmetik ausschließlich Zahlen größer gleich Null kennt. Auch negative Zwischenergebnisse führen deshalb zur Fehlermeldung. Gegebenenfalls kann man Ausdrücke so umordnen, daß keine negativen Zwischenergebnisse erscheinen (»-1+2« zu »2-1« z. B.).

»number out of range«: Diese Fehlermeldung zeigt ebenfalls an, daß der Zahlenbereich der Integerartihmetik über- oder unterschritten wurde. Beispiel:

```
_$e000+$e000
```

»not implemented«: Dieser Fehler tritt ein, wenn Sie eine logische Operation oder einen Vergleich falsch abkürzen. Beispiel:

```
_1.gl.2
```

Nicht alle offensichtlich falschen Ausdrücke werden bei Rechnungen innerhalb des Editors richtig mit einer Fehlermeldung quittiert. Dieser Umstand erklärt sich daraus, daß eine wichtige Testfunktion für Ausdrücke nur innerhalb des Quelltextes angewendet wird. Eine entsprechende Testfunktion für Rechnungen im Editor habe ich mir geschenkt, denn so wichtig sind Rechnungen im Editor nicht und außerdem stehen hier die Ausdrücke auf dem Bildschirm, so daß grobe Fehler sofort sichtbar sind.

Variablen:

Die ASE-Arithmetik kennt Variablen, diese können jedoch vom Editor aus nur gelesen, nicht mit einem Wert versehen werden. Die Variablen werden während der Assemblierung als Speicher für die anfallenden Label verwendet, dienen aber auch als Variablen selbst – hier sei die später noch beschriebene »bedingte Assemblierung« angeführt.

Variablennamen können maximal bis zu 72 Zeichen lang sein. Alle Zeichen des Namens sind signifikant, d. h. die beiden Variablen »abspeicherungsbereich1« und »abspeicherungsbereich2« werden als verschiedene Variablen behandelt. Für den Einsatz der Variablen als Labelspeicher heißt diese sehr große Namenslänge, daß sich der Programmierer nicht mit Kunstbezeichnungen herumschlagen muß, wie sie häufig bei Assemblern und Compilern zu finden sind, die die Länge von Bezeichnern einschränken. Sie sollten sich allerdings nicht exzessiv nur sehr lange Namen einfallen lassen – darunter leidet die Geschwindigkeit der Assemblierung und auch das Druckbild der Symboltabelle (wir werden etwas später noch sehen, daß die während der Assemblierung aufgenommene Symboltabelle auf dem Bildschirm angezeigt oder auch gedruckt werden kann). Ich empfehle eine maximal zu benutzende Label- und Variablenlänge von 15, wobei die Länge des Labels oder der Variablen gleichzeitig etwas über die Wichtigkeit aussagen sollte: Verwenden Sie längere Namen für die Bezeichnung von Einsprüngen in Unterprogramme beispielsweise, kürzere Namen für Label innerhalb dieser Unterprogramme. Beispiel:

```
unterprogramm  lda  #0
               ldx  speicher
               bmi  upr1
                 ...
upr1             ...
               rts
```

Dies alles hilft, Ihre Programme übersichtlich zu gestalten, eine Hauptvoraussetzung für die Wartbarkeit der Programme.

Als Zeichen innerhalb eines Variablennamens sind erlaubt:

- große und kleine Buchstaben
- Ziffern
- Sonderzeichen: Unterstrich »_«, Hochkomma »'«, Fragezeichen »?«, eckige Klammern »[« und »]«

Das erste Zeichen des Namens **muß** entweder aus einem Buchstaben oder aus dem Unterstrich bestehen. Ich selbst verwende den Unterstrich als erstes Zeichen meist, um Routinen aus den ROMs und Speicherzellen des Systems – beispielsweise aus der Zeropage – zu bezeichnen. Auch dies dient der Übersichtlichkeit des Quelltextes, da man so sofort erkennen kann, ob der Label eine Adresse innerhalb des eigenen Programms oder eine Adresse außerhalb bezeichnet, auch wenn man die genaue Bedeutung des Labelnamens nicht mehr im Gedächtnis hat. Natürlich stellt diese Vorgehensweise keine verbindliche Regel dar, Sie werden sicher im Laufe der Zeit Ihren eigenen Stil finden.

Eine weitere Eigenschaft der ASE-Variablen soll hier kurz angerissen werden: Die Arithmetik kennt lokale und globale Variablen, eine Eigenschaft, die Basic unbekannt ist. Lokale Variablen sind Variablen, die nur innerhalb bestimmter Bereiche des Quelltextes gelesen werden können, man sagt, sie sind nur in einem lokalen Bereich des Quelltextes definiert. Lokale Variablen dienen dem Assembler während der Makroverarbeitung und innerhalb von Blockstrukturen (s. dort Näheres).

3.2.2.5 Weitere Befehle des Editors

Mit den bisher beschriebenen Befehlen ist der Vorrat des ASE-Editors noch lange nicht erschöpft, nur passen diese in kein einheitliches Schema mehr, unter dem sie zusammengefaßt werden könnten. Sie sollen deshalb einfach der Reihe nach angesprochen werden, ohne Platz durch Überschriften für Unterkapitel zu verschwenden.

Dumps der Symboltabelle:

Nach erfolgter Assemblierung – und nur dann mit sinnvollem Ergebnis – kann die während der Assemblierung aufgebaute Symboltabelle auch auf dem Bildschirm angezeigt werden. Eine Sortierung nach verschiedenen Kriterien ist möglich. Die Editorbefehle hierfür sind:

.d	–	unsortierten Dump ausgeben
.dw	–	nach Wert sortierten Dump ausgeben
.da	–	alphabetisch sortierten Dump zeigen

Die beiden sortierenden Dumpbefehle bewirken eine physikalische Sortierung der ansonsten in einem speziellen Variablenraum nach einem Hashverfahren verteilten Variablen. Die Variablen werden dazu verdichtet und danach mit dem Quicksort-Algorithmus sortiert. Die Verdichtung bedingt, daß nach der Sortierung keinerlei Variablenwerte mehr mit »_« abgefragt werden können – die Variablen befinden sich nicht mehr an der Stelle innerhalb des Variablenraums, an der sie nach dem Hashverfahren abgelegt wurden, deshalb findet die Arithmetik sie nicht mehr wieder. Ebenso ist nicht mehr gewährleistet, daß eine zweite Sortierung des verdichteten Variablenraums noch korrekte Ergebnisse liefert. Ein einmal nach einem Kriterium sortiertes Variablenfeld kann in der gleichen Reihenfolge ein zweites Mal durch einfache Eingabe von ».d« ausgegeben werden.

Die Ausgabe des Dumps erfolgt dreispaltig am Bildschirm. Sie kann wie das formatierte Listen durch »Leertaste« angehalten und wieder gestartet werden, »Return« bricht die Ausgabe ab.

Die drei Befehle sind eigentlich Relikte aus der Testphase des Assemblers. Der Assembler besitzt einen Pseudoop ».symbols«, mit dem die Symboltabelle nach der Assemblierung beispielsweise zum Drucker, zur Floppy oder auch zum Bildschirm gesendet werden kann. Die drei Editorbefehle benutzen die gleichen Routinen wie dieser Pseudoop des Assemblers. Da die Dump-Ausgabe aber ganz nett aussieht, insbesondere auf dem 80-Zeichen-Bildschirm, habe ich sie auch nach Austesten des ASE stehengelassen. Wer weiß, welcher Benutzer diese Ausgabe nicht doch irgendwie nützlich findet?

Beachten Sie, daß auch die Label ausgegeben werden, die Makronamen darstellen. Der für diese Label ausgegebene Wert ist keine Adresse innerhalb des assemblierten Maschinenprogramms, sondern eine Adresse innerhalb des Quelltextes, an der die Makrodefinition zu finden ist! Dies sei schon einmal vorbemerkt, obwohl die Makros noch nicht näher vorgestellt wurden.

»Old« – Aufheben des Befehles »New«:

Ein Befehl, der eigentlich in keinem Basicdialekt fehlen sollte. Leider enthält auch das Basic 7.0 keinen derartigen Befehl, deswegen wird er durch den Editor eingeführt.

»new« bewirkt keine physikalische Vernichtung eines Basicprogrammes oder hier eines Quelltextes, sondern verändert im wesentlichen nur einen Zeiger auf das Ende des Textes. Der Befehl

```
.o
```

stellt diesen Zeiger wieder her, so daß ein mit »new« gelöschter Quelltext wieder gelistet und bearbeitet werden kann. ».o« funktioniert nicht mehr, wenn beispielsweise in den Quelltext »gepoked« wurde oder irgendeine andere physikalische Veränderung größeren Ausmaßes vorgenommen wurde.

Ein- und Ausschalten des automatischen Formatierens am Schirm:

Wenn Sie keinen 80-Zeichen-Monitor besitzen, schaltet der ASE-Assembler die höhere Taktfrequenz (2 MHz) des C-128 beim Start nicht ein, da der 40-Zeichen-Videocontroller nicht mit dieser Taktfrequenz arbeiten kann. Folglich laufen alle Vorgänge nur mit der Hälfte der möglichen Geschwindigkeit ab. Dieses Manko macht sich insbesondere bei der Bearbeitung langer Quelltexte bemerkbar, etwa, wenn Sie Zeilen zu Beginn des Quelltextes einfügen, was größere Blockverschiebungen innerhalb des Quelltext-Speichers notwendig macht. Einen kleinen Geschwindigkeitsvorteil kann man sich allerdings beschaffen, indem man den ASE-Editor anweist, das automatische formatierte Listen nach erfolgter Eingabe zu unterlassen. Das Bild des Quelltextes wird dadurch zwar unübersichtlicher, was aber immer noch besser ist als eine zu langsame Eingabemöglichkeit. Da der Assembler ASE ausdrücklich auch für diejenigen nutzbar sein soll, die über keinen 80-Zeichen-Monitor verfügen, sind die beiden Editorbefehle

».-« – Ausschalten der Formatierung
».+« – Einschalten der Formatierung

mit in den Editor aufgenommen worden. Falls dies noch nicht vollkommen klar sein sollte: Die Abschaltung der Formatierung betrifft lediglich die Eingabe von neuen Quelltextzeilen. Durch sie wird weder das Format der Zeilen im Speicher noch das formatierte Listen durch den Befehl ».e« betroffen.

Anzeige der aktuellen Speicherbelegung:

Mit dem Befehl ».b« können Sie sich jederzeit über den Zustand des Quelltextspeichers informieren. Sie erhalten eine Ausgabe der folgenden Art:

```
$61d1 is start of text
$61d3 is end of text
40237 bytes free
```

Die erste Angabe bezeichnet die Adresse, an der das erste Byte des Quelltextes zu finden ist. Diese Adresse fällt zusammen mit dem Ende des Assemblerprogramms. Die zweite Angabe enthält analog dazu die Adresse des letzten Bytes des Quelltextes. Im obigen Fall wäre der Quelltextspeicher leer, die unterschiedlichen Angaben für den Start und das Ende des Quelltextes ergeben sich dadurch, daß das Ende des Quelltextes durch zwei zusätzliche Nullbytes angezeigt wird. Die letzte ausgegebene Zahl ist wohl selbsterklärend – hier können Sie ablesen, wieviel Platz Ihnen für weitere Erweiterungen des gerade im Speicher befindlichen Quelltextes noch verbleibt. Empfehlung meinerseits in diesem Zusammenhang: Machen Sie Ihre Quelltexte nicht zu groß; es sollte Ihnen immer noch genügend Platz bleiben, um den Quelltext nach abgeschlossener Entwicklungsarbeit ausreichend kommentieren zu können. Im allgemeinen bedeutet diese Devise, daß ca. 30 Prozent – dies entspricht

ca. 10 KByte – als Reserve gehalten werden sollten. Natürlich können Sie auch anders vorgehen, diese ganzen Hinweise sollen ja nur dazu dienen, Ihnen zu zeigen, welche Arbeitsmethoden andere Programmierer verwenden.

Verlassen des Assemblers:

Haben Sie Ihr Pensum geschafft oder einfach keine Lust mehr, geben Sie ».x« für »exit« ein. Daraufhin wird ein Software-Reset durchgeführt, der allerdings wieder einen Boot-Vorgang einleitet. Sollten Sie also die Masterdisk selbst oder eine exakte Kopie derselben im Laufwerk haben, wird der Assembler erneut gestartet. Weiter oben wurde aber schon gesagt, daß keine Notwendigkeit besteht, mit der Masterdisk selbst oder einer Kopie zu arbeiten. Kopieren Sie statt dessen besser den ASE-Assembler auf die Arbeitsdiskette, die auch Ihre Quelltexte aufnehmen soll – hierzu reicht ein »dload« und ein »dsave«. Da es nicht oft genug wiederholt werden kann, an dieser Stelle noch einmal: Honorieren Sie bitte das Entgegenkommen, daß die Programme so leicht kopierbar gehalten wurden, und geben Sie sie nicht weiter – auch nicht an Freunde und Bekannte.

Starten der Assemblierung:

Eine ganz wichtige Sache ist bisher noch überhaupt nicht angeklungen: Wie wird die Assemblierung des im Speicher stehenden Quelltextes eingeleitet?

Hierzu genügt der Befehl »run«, der auch auf einer der Funktionstasten zu finden ist. Falls sich kein Quelltext im Speicher befindet, hat der Befehl keine Wirkung. Es ist möglich, gleichzeitig einen Quelltext zu laden und gleich anschließend die Assemblierung zu starten. Hierzu geben Sie entsprechend zu Basic einen Befehl »run "filename"« ein.

Nach Abschluß der Assemblierung wird eine Meldung auf den Bildschirm ausgegeben, der Sie verschiedene Informationen entnehmen können:

```
end of assembly at 0:00.2

origin is $1000
code from $1000 to $1000 in bank 0
0 bytes total, 0 labels used
```

Die erste Zeile enthält die für die Assemblierung benötigte Zeit in Minuten, Sekunden und Zehntelsekunden – ein kleiner Gag, der Sie vielleicht animieren sollte, die benötigte Zeit mit der anderer Assembler für den 128er zu vergleichen, sofern Sie die Möglichkeit haben. Sie werden feststellen, daß der ASE sehr schnell ist. Zum Vergleich: Die ASE-Version für den C-64 (zur Zeit nicht erhältlich) ist so schnell wie der vielgepriesene Assembler mit dem geschwindigkeitsverheißenden Vorsatz, Sie wissen vielleicht, welchen ich meine. Nur – der ASE kann mehr!

Die zweite Zeile der abschließenden Ausgabe enthalt die Startadresse, die Sie für das assemblierte Programm gewählt haben. In der dritten Zeile erhalten Sie noch einmal die Information, wo das erzeugte Maschinenprogramm innerhalb des Speichers abgelegt wurde – es ist möglich, für die Ablage des Codes eine andere als die Startadresse zu wählen. Die letzte Zeile schließlich zeigt an, wieviele Bytes das erzeugte Programm umfaßt und wieviele Label im Quelltext insgesamt verwendet wurden. Die letzte Angabe ist dann wichtig, wenn Sie besonders lange Quelltexte entwickeln, da die Anzahl der Label, die der ASE verwalten kann, wenn auch sehr groß, natürlich nicht unbegrenzt ist.

Belegung der Funktionstasten:

Die normale Belegung der Funktionstasten wie unter Basic ist für die Arbeit mit dem ASE-Assembler nicht sehr geeignet; sie wurde deshalb von mir durch eine Belegung ersetzt, die sich aus meiner eigenen Arbeit mit dem Assembler als Erfahrungswert ergeben hat:

```
F1              –   dload"
F2              –   dsave"
F3              –   directory + chr$(13)
F4              –   print ds$ + chr$(13)
F5              –   .e + chr$(13)
F6              –   run + chr$(13)
F7              –   .n,100,10 + chr$(13)
F8              –   monitor + chr$(13)

HELP            –   .b + chr$(13)
SHIFT + RUN/ STOP   – boot "dbm 2000"on b1 + chr $(13)
```

Wieder eine kleine Feinheit, die Sie gegebenenfalls beachten müssen: Der Befehl »boot« funktioniert nur, wenn Sie eine Floppy 1570 oder 1571 oder eine andere mit der neuen schnellen Übertragungsweise des »Burst-Modus« verwenden. Verantwortlich hierfür ist ein Fehler innerhalb des Betriebssystem-ROMs des 128ers – nobody is perfect. Besitzen Sie eine der angegebenen Floppies, lädt der Befehl das Monitorprogramm »DBM« des Top-Ass-Pakets und startet ihn.

3.3 Der Makroassembler

Jetzt müßten Sie eigentlich soweit sein, daß Sie die Bedienung des Editors zumindest in den Grundzügen beherrschen. Im folgenden wird nicht mehr auf die Funktionen des Editors eingegangen. Das Hauptthema dieses Kapitels sind vielmehr die vom ASE unterstützten Pseudoops, also die Anweisungen an den Assembler, die innerhalb des Quelltextes erteilt werden können (Festlegung der Startadresse der Assem-

blierung, Einfügen von Tabellen usw.). In Art und Anzahl der unterstützten Pseudoops unterscheiden sich die verschiedenen Assembler untereinander am deutlichsten – neben der Geschwindigkeit, in der die Assemblierung erfolgt. Ich denke, die reine Zahl von insgesamt 96 verfügbaren Pseudoops des ASE – wenn ich mich nicht verzählt habe – vermittelt schon einen Eindruck, wie leistungsfähig dieses Programm ist. Ich lade Sie gern zu Vergleichen mit anderen Programmen ein.

Andererseits bedeutet diese Anzahl an Befehlen auch, daß Sie noch eine ganze Menge an Arbeit vor sich haben, bis Sie alle Möglichkeiten des ASE voll nutzen können. Besser, als wenn Sie die Fähigkeiten des Assemblers schon nach einem halben Jahr bis ins letzte ausgelotet hätten und sich dann erst nach einem professionellen System umsehen müßten, oder nicht?

Damit Sie zu Beginn nicht von der Komplexität des ASE erschlagen werden, finden Sie als erstes Unterkapitel dieses Abschnitts einen Wegweiser, dem Sie entnehmen können, welche Teile der Gesamtbeschreibung des Assemblers Sie für die Entwicklung Ihrer ersten eigenen Programme bearbeiten sollten. Sie können dann nach und nach die weiteren Pseudoops und Features ausprobieren, wenn Sie sich in der Maschinensprache überhaupt und in der Bedienung des Assemblers in stärkerem Maße sicher fühlen.

Nach diesem kurzen Abschnitt folgt die eigentliche Beschreibung aller Pseudoops. Ich habe diese zu Unterkapiteln organisiert, die jeweils alle Pseudoops zu einem bestimmten Problemkreis enthalten. Jeder Pseudoop selbst erhielt nach Möglichkeit ein eigenes Unterkapitel, damit die Beschreibungen möglichst leicht wieder aufzufinden sind.

3.3.1 Wegweiser für Erstbenutzer

Was müssen Sie wissen, um die ersten elementaren Maschinenprogramme mit dem ASE schreiben und assemblieren zu können? Zunächst einmal müssen Sie natürlich die Quelltexte schreiben können, ich hoffe, dieses Thema ist mit der Beschreibung des ASE-Editors erledigt. Als zweites müssen Sie die Prozessorbefehle an sich und ihre Notation kennen, diese haben Sie in Kapitel 1 erlernt. Kennen Sie auch noch die Verwendung von Labeln innerhalb des Quelltextes, sind 90 Prozent des Textes schon erschlagen, die fehlenden 10 Prozent setzen sich aus den Pseudoops zusammen.

Von diesen Pseudoops reichen für die ersten Anwendungen einige wenige aus:

.base – Festlegen der Startadresse der Assemblierung (an welcher Adresse soll Ihr Programm einmal laufen?)

.define – Wertzuweisung an einen Label oder eine Variable (Definition von Zeropageadressen oder ROM-Routinen)

.byte – Einfügen von Byte-Tabellen oder Text in den Quelltext

.word – Einfügen von Adreßtabellen

Mit diesen vier Pseudos kommen Sie schon ganz erheblich weiter, was aber auf keinen Fall heißen soll, daß Sie die anderen Befehle einfach vergessen sollten.

Wenn Sie weiter vordringen möchten, rate ich Ihnen, das dritte Kapitel zumindest einmal zu überfliegen, um sich einen Überblick über die vorhandenen Möglichkeiten zu verschaffen. Insbesondere die Minimacs und die strukturierte Programmierung sind sicher Kapitel, die die Programmierung in Assembler bequemer und leichter einsehbar machen. Makros sollten Sie sich dann vielleicht etwas später vornehmen, wenn Sie schon etwas Erfahrung besitzen, welche Probleme in der laufenden Programmierung auftauchen – sonst könnte es sein, daß Sie die Notwendigkeit der Makros gar nicht einzusehen vermögen. Für die Fortgeschrittenen unter Ihnen möchte ich dann die Kapitel über relokatiblen Kode, den Relativlader VLO und den ASE-Linker empfehlen. Ich persönlich zähle diese Teile des ASE zu den praktischen und mächtigsten – sie haben mir schon manche Arbeitsstunde erspart.

```
ready.

monitor
    pc  sr ac xr yr sp
; fb000 00 00 00 00 f8

>12000 00 00 00 0c 0b 0c 00 00 00 53 54 41 52 54 00 03:
>12010 0c 00 00 00 00 00 53 50 45 49 43 48 45 52 00 00:
>12020 0c 00 00 5f 0f 1f 00 6f 00 00 3e 28 53 54 41 52:
>12030 54 29 00 1f 00 2f 02 0a 0c a9 00 0f 0f 0f 00 00:
>12040 3e 28 53 50 45 49 43 48 45 52 29 00 a9 0c 20 0a:
>12050 0c 60 0b 42 59 20 20 c7 45 52 44 20 cd 4f 45 4c:
>12060 4c 4d 41 4e 4e 0d 1b 51 23 05 28 43 29 31 39 38:
>12070 36 20 cd 41 52 4b 54 20 26 20 d4 45 43 48 4e 49:
>12080 4b 20 d6 45 52 4c 41 47 0d 1b 51 23 0b c1 4b 54:
>12090 49 45 4e 47 45 53 45 4c 4c 53 43 48 41 46 54 1b:
>120a0 51 0d 1b 51 0d 00 a9 20 20 cd 35 ca d0 fa 60 a0:
>120b0 00 b9 2e 20 f0 1d c9 23 d0 13 24 d7 10 05 a2 14:
>120c0 20 a6 20 c8 be 2e 20 20 a6 20 c8 d0 e4 20 cd 35:
>120d0 c8 d0 de 60 20 1f 21 a9 3e 20 cd 35 20 18 36 a0:
>120e0 00 f0 05 a9 20 20 cd 35 20 55 22 20 2c 36 c8 cc:
>120f0 4d 21 90 ef c0 08 90 22 a9 3a 20 cd 35 a9 12 20:
>12100 aa 22 a0 00 20 55 22 48 29 7f c9 20 68 b0 02 a9:
>12110 2e 20 cd 35 c8 cc 4d 21 90 ea a9 0d 4c cd 35 a5:
>12120 e7 38 e5 e6 c5 ee d0 03 a9 ff 2c a9 00 8d 4f 21:
>12130 aa e8 24 d7 10 02 e8 e8 bd 45 21 8d 4d 21 bd 49:
>12140 21 8d 4e 21 60 08 04 10 08 20 10 40 20 00 00 00:
>12150 20 1f 21 a9 3c 20 cd 35 20 18 36 a9 12 20 aa 22:
>12160 a0 00 20 55 22 48 29 7f c9 20 68 b0 02 a9 2e 20:
>12170 cd 35 c8 cc 4e 21 90 ea 4c 1a 21 20 8b 23 1b 51:
>12180 00 a9 2e 20 cd 35 20 18 36 20 1f 21 20 80 26 24:
>12190 d7 30 05 2c 4f 21 10 22 a0 00 ae b7 25 a9 20 20:
>121a0 cd 35 b9 ba 25 20 2c 36 ca f0 03 c8 d0 ef c0 03:
>121b0 f0 08 a2 03 20 a6 20 c8 d0 f4 a9 20 20 cd 35 4c:
>121c0 d7 27 85 06 86 07 84 08 ad 00 ff 8d de 02 a9 0f:
>121d0 85 02 60 a4 08 a6 07 a5 05 48 a5 06 28 60 20 c2:
>121e0 21 a9 e9 a2 b8 85 04 86 03 20 cd 02 4c d3 21 20:
>121f0 c2 21 a9 e7 a2 b8 85 04 86 03 20 cd 02 4c d3 21:
>12200 20 c2 21 a9 2a a2 b1 85 04 86 03 20 cd 02 4c d3:
>12210 21 20 c2 21 a9 58 a2 cb 85 04 86 03 20 cd 02 4c:
>12220 d3 21 20 c2 21 a9 ce a2 b7 85 04 86 03 20 cd 02:
>12230 4c d3 21 20 c2 21 a9 5c a2 c1 85 04 86 03 20 cd:
>12240 02 4c d3 21 20 c2 21 a9 01 a2 b9 85 04 86 03 20:
```

3.3.2 Startadresse der Assemblierung / Ablegen des erzeugten Codes

3.3.2.1 Festlegen der Startadresse mit ».base«

Wie Sie wissen, ist jedes Maschinenprogramm nur an einer bestimmten Adresse lauffähig, dies unterscheidet Basicprogramme und Maschinenprogramme prinzipiell. Welche Adresse Sie für ein zu assemblierendes Programm vorgesehen haben, müssen Sie dem Assembler natürlich auch mitteilen, damit dieser die absoluten Adressen innerhalb der Prozessorbefehle richtig berechnet. Zuständig hierfür ist der Befehl

».base«. ».base« wird gefolgt von einem arithmetischen Ausdruck, der die Startadresse des Programms angibt. Der Ausdruck wird durch die Integerarithmetik des Assemblers berechnet wie sämtliche anderen Ausdrücke, die innerhalb des Quelltextes verwendet werden (s. Beschreibung der Integerarithmetik in Kapitel 3.2). Die Startadresse bestimmt die Adresse, an der das erste Byte des Maschinenprogramms stehen soll, wenn es gestartet wird. Beispiele:

```
.base $1400
.base 4567
.base origin
```

».base« setzt den assemblerinternen Programmzähler auf die angegebene Startadresse. Der assemblerinterne Programmzähler wird mit jedem durch den Assembler erzeugten Byte des Maschinenprogramms um Eins weitergezählt. Bei Rechnungen, die sich auf den Programmzähler des Assemblers beziehen (Rechnungen mit »*«) bezeichnet der Zähler gerade die Adresse des Befehls, in dem die Rechnung erfolgt. Erst nach erfolgter Assemblierung wird die Länge des assemblierten Befehls zum Programmzähler addiert – die Berechnung der notwendigen Distanz für einen Branch ist also verschieden von der, die der Prozessor durchführt! Beispiel:

```
c7da a2 0f          ldx #15
c7dc dd 4c ce  ma1  cmp $ce4c,x
c7df f0 04          beq ma2
c7e1 ca             dex
c7e2 10 f8          bpl ma1
c7e4 60             rts
c7e5           ma2  ...
```

Die linke Seite soll den erzeugten Maschinenkode mit Inhalt des Programmzählers bei der Erzeugung darstellen, die rechte Seite sei ein Auszug aus einem Quelltext. Wie werden die beiden Branches ausgedrückt, wenn man den Stern für den Programmzähler verwendet? Der erste Branch geht vorwärts, die Länge des Branches von zwei Byte ist während der Rechnung mit dem Stern noch nicht zum Programmzähler addiert, also müssen die zwei Byte des Branches mit zur Distanz addiert werden, und man erhält ersatzweise – statt Verwendung des Labels »ma2« – einen Befehl »beq *+7«. (Zählen Sie einfach die Bytes der Opcodes herunter, bis Sie »ma2« erreicht haben.) Der zweite Branch führt rückwärts nach »ma1«. Auch hier hilft das Abzählen der Bytes weiter; in diesem Fall dürfen Sie aber nicht vergessen, die Bytes des Zielbefehls – des Befehls, auf dem der Branch landen soll – mit zur Distanz zu zählen. Die Zählung beginnt eine Zeile höher, als der Branchbefehle steht. Auf diese Weise erhält man eine Ersatzdarstellung für die Angabe des Labels »ma1« von »bpl *-6«. Wie Sie an den beiden Beispielen sehen, müssen Sie die Längen der einzelnen Prozessorbefehle im Kopf haben, um Angaben relativ zum Programmzähler machen zu können; im Quelltext stehen die einzelnen Opcodes ja nicht neben

dem Text. Im Laufe der Zeit wird sich diese Kenntnis sicher von allein einstellen, dem Einsteiger in die Assemblerprogrammierung möchte ich aber noch einmal von der Verwendung des Sterns abraten, denn eine falsche Distanzangabe führt mit fast tödlicher Sicherkeit zum Absturz Ihres Programmes und Fehler innerhalb von Angaben relativ zum PC sind erfahrungsgemäß schwer zu finden.

Zurück zu ».base«: Außer daß ».base« die Startadresse des zu assemblierenden Programms festlegt, bestimmt dieser Pseudoop auch gleichzeitig, wohin der während der Assemblierung anfallende Kode geschrieben wird: Im Normalfall – wenn keine anderen Vorkehrungen getroffen sind (s. u.) – erscheint der Maschinenkode an der Startadresse in der RAM-Bank 0.

Bleibt als letztes zu klären, wo innerhalb des Quelltextes ».base« zu stehen hat. Folgende Regeln sind einzuhalten:

- ».base« muß vor der ersten kodeerzeugenden Quelltextzeile stehen, also vor der ersten Zeile, die ein Mnemonic, einen Makroaufruf, einen Pseudoop zur strukturierten Programmierung oder einen Pseudoop zur Tabellenerzeugung enthält.
- Der Pseudoop muß natürlich vor der ersten Zeile stehen, in der Bezug auf den Programmzähler genommen wird (Rechnung mit »*«).
- ».base« muß vor dem ersten internen Label stehen (Label, das rechts an das Minuszeichen hinter der Zeilennummer anschließt).
- Der Befehl muß vor dem ersten Auftreten von ».extern«, von ».object« und von ».code« stehen (s. u.).

Eine ganze Reihe von Regeln, die sich aber im Laufe der Zeit sicher von selbst einschleifen werden. Wenn ».base« als erste Anweisung innerhalb des Quelltextes verwendet wird, können Sie zunächst einmal nicht viel verkehrt machen, obwohl es Fälle gibt, in denen ».base« anders verwendet wird. Auf diese Fälle wird aber noch speziell eingegangen (bedingte Assemblierung, Verkettungen mit ».append«). Vergessen Sie bitte die Angabe der Startadresse nicht, Sie arbeiten sonst mit einer undefinierten Startadresse und überschreiben vielleicht sogar den Assembler selbst. Was dies für Folgen hat, brauche ich wohl nicht weiter auszuschmücken.

3.3.2.2 Verschiebung der Kodespeicherung

Die mit ».base« festgelegte Adresse zur Ablage des assemblierten Kodes innerhalb der RAM-Bank 0 ist nicht immer sinnvoll zu verwenden. Man denke sich etwa das Beispiel, daß Sie ein Programm assemblieren wollen, das genau dort starten soll, wo während der Assemblierung der ASE steht oder der Quelltext des Programms. Diesen Fall fängt ein weiterer Pseudoop des Assemblers auf, der es erlaubt, den erzeugten Maschinenkode an eine beliebige Adresse in einer beliebigen RAM-Bank zu schreiben. Der Befehl lautet:

```
».code b,a«
```

»b« ist die reine Nummer der RAM-Bank, in der der Kode abgelegt werden soll, also weder Konfiguration noch Konfigurationsindex. »a« ist die Adresse innerhalb der Bank. Wollen Sie z. B. eine Kodeverschiebung in die RAM-Bank 1 nach $2345 vornehmen, wobei das zu assemblierende Programm eine richtige Startadresse von $1234 besitzen soll, müßte am Anfang Ihres Quelltextes die Befehlsfolge stehen:

```
.base $1234
.code 1,$2345
```

Das Ganze an einem Beispiel, das Sie in der Praxis nachprüfen können: Wir nehmen ein einfaches Maschinenprogramm, das nur aus einem »lda #0« sowie der Angabe der Startadresse und der Kodeverschiebung besteht:

```
100   -.base $4c00
110   -.code 1,$2000
120   -          lda #0
```

Starten Sie die Assemblierung mit »run«, nachdem Sie den kleinen Quelltext eingegeben haben. Der Endmeldung der Assemblierung entnehmen Sie die Start- und Endadresse der Kodespeicherung – $2000 und $2002. Aktivieren Sie jetzt einmal den Bordmonitor des C-128 und sehen sich das Ergebnis mit »d 12000« an. Sie sehen den erzeugten Kode an der Kodeablageadresse. Würden Sie jetzt den Assembler mit ».x« verlassen, so könnten Sie Ihr Maschinenprogramm mit dem Monitorbefehl

```
t kodestart kodeende startadresse
```

an die vorgesehene Startadresse verschieben und anschließend starten. Die drei Angaben hinter dem Monitorbefehl »t« entsprechen genau denen, die der ASE-Assembler zum Ende der Assemblierung ausgibt – mit anderen Worten: der ASE gibt diese Adressen aus, weil die Kodeverschiebung hierdurch unterstützt wird.

Wie immer sind auch einige Fallstricke zu beachten, wenn Sie eine Kodeverschiebung vornehmen wollen:

(1) Die Common Area unter ASE reicht bis $2000 hinauf. Eine Kodeverschiebung an eine Adresse unterhalb von $2000 ist deshalb nur innerhalb der RAM-Bank 0 möglich – die Common Area setzt sich aus RAM der Bank 0 zusammen. Da der ASE- Assembler selbst seinen Start bei $1c00 hat, verbieten sich Verschiebungen einer Adresse zwischen $1c00 und $2000 damit von selbst.

(2) In der RAM-Bank 1 liegen die verschiedenen Datenbereiche des Assemblers, beginnend an der Adresse $6a00 bis zum Ende der Bank. Alle Adressen oberhalb $6a00 in der Bank 1 sind damit erst einmal tabu.

Damit bleibt Ihnen in der Bank 1 der Adreßbereich von $2000 bis $69ff für eigene Verwendungszwecke, diesen Bereich möchte ich persönlich Ihnen auch für die Kodeverschiebung empfehlen, er ist genau für solche Zwecke freigehalten worden. Außerdem bleibt in der Bank 0 immer noch der Bereich hinter dem Quelltext ab der Adresse plus Eins, die der Befehl ».b« als Endadresse des Quelltextes amgibt. Haben Sie Speichererweiterungen in Form weiterer RAM-Bänke eingebaut, so können Sie diese natürlich beliebig verwenden – bis auf den Bereich der Common Area. Zum Zeitpunkt der Entwicklung des ASE waren noch keine Speichererweiterungen erhältlich, aber ich denke, die Kodeverschiebung so programmiert zu haben, daß sie auch mit einer dritten und vierten Bank arbeitet.

3.3.2.3 Abspeichern auf Diskette (des Maschinenkodes)

Bei längeren Programmen und solchen, die sich mit dem ASE nicht vertragen, also allein gestartet werden müssen, ist es sinnvoll, statt einer Kodeverschiebung den Maschinenkode direkt zur Floppy oder zu einem anderen Massenspeicher zu senden. Im Fall der Floppy wird auf diese Weise eine Programmdatei mit der richtigen Startadresse erzeugt, die Sie nach der Assemblierung und dem Verlassen des Assemblers mit »bload« und »sys« oder auch »boot« laden und starten können. Der Befehl hierzu lautet:

```
».object "filename",u«
```

Der Assembler hängt an den Filenamen intern einen Zusatz »,p,w« an, es wird also ein Filetyp »PRG« erzeugt. Verwenden Sie innerhalb des Filenamens nie ein »@:«, um ein File zu überschreiben! Diese Einschränkung wird nicht vom Assembler verlangt, sondern folgt aus einem Fehler innerhalb des Floppy-ROMs. Das Betriebssystem der Floppies ist nicht in der Lage, ein Überschreiben in jedem Fall korrekt vorzunehmen. Es kann tausend Mal gutgehen, das tausendunderste Mal wird Ihre Diskette so durcheinandergewürfelt werden, daß einige Files verloren sein werden! Deshalb grundsätzlich **Hände weg vom Klammeraffen!** Sollen Dateien überschrieben werden, »scratchen« Sie sie erst, bevor Sie die Datei ganz normal neu erzeugen.

Zurück zur Parameterliste von ».object«: Folgt dem Filenamen ein Komma, erwartet der Assembler die nachfolgende Angabe einer Gerätenummer (»u«). Fehlt dieser Teil der Parameterliste ganz – inklusive Komma -, nimmt der Assembler automatisch an, daß der Kode zur Floppy mit der Gerätenummer 8 gesendet werden soll. Beispiele:

```
.object "filename"      - sendet zur Floppy #8
.object "filename",9    - sendet zur Floppy #9
```

Das Senden des Objektkodes endet, wenn der Assembler das Ende des Quelltextes erreicht. Die erzeugte Datei wird geschlossen.

In Zusammenhang mit der später beschriebenen bedingten Assemblierung und der Erzeugung relokatibler Moduln kann es von Vorteil sein, ein Objektfile schon vor Erreichen des Endes der Assemblierung zu schließen. Zu diesem Zweck existiert der Befehl

```
».objend«
```

Dieser Pseudoop bricht die Assemblierung **nicht** ab, sondern schließt lediglich ein mit ».object« – oder ».modul« – erzeugtes File.

».object« steht im Quelltext an der gleichen Position wie auch ».code«. Beide Befehle schließen sich gegenseitig aus, d. h. wird ».object« im Quelltext verwendet, sollte kein ».code« im Quelltext vorhanden sein und umgekehrt. Sollte man ».code« trotzdem verwenden, erhält man eine Programmdatei, die zwar den richtigen Inhalt, aber eine falsche Startadresse besitzt: die der Kodespeicherung.

3.3.2.4 Testläufe ohne Kodespeicherung

Ein weiterer Pseudoop verhindert die Abspeicherung des während der Assemblierung erzeugten Maschinenkodes total. Der entsprechende Befehl lautet einprägsam:

```
».syntax«
```

Diese Bezeichnung deutet schon darauf hin, das der Pseudoop noch eine andere Wirkung als die reine Verhinderung der Kodespeicherung besitzt, dazu aber gleich mehr.

».syntax« kann zusammen mit ».code« verwendet werden, kann allerdings nicht verhindern, daß ein leeres File erzeugt wird, falls Sie gleichzeitig ein ».object« verwenden. Ich habe mir eine bestimmte Arbeitsweise mit diesem Befehl angewöhnt, wenn ich zur eigentlichen Erzeugung eines Maschinenprogramms ein ».object« verwende. Diese Arbeitsmethode möchte ich kurz schildern. Ich mache mir zunutze, daß die restliche Quelltextzeile hinter ».syntax« vom Assembler ignoriert wird, also auf kein bestimmtes Format festgelegt ist. Füge ich neue Befehle in einen Quelltext ein, so verwende ich immer als erstes einen schnellen Testlauf mit dem Pseudoop ».syntax«, bevor ich daran gehe, ein File auf Diskette zu erzeugen. Dies hat auf lange Sicht Vorteile, denn es kommt doch das eine oder andere Mal vor, daß fehlerhafte Quelltextzeilen die Assemblierung abbrechen lassen, wobei ein unvollständiges File auf Diskette zurückbliebe, das vor einem neuen Anlauf erst einmal wieder »gescratched« werden müßte. Will ich einen Testlauf durchführen, schreibe ich einfach das ».syntax« in dieselbe Zeile wie das ».object«:

```
100   -.syntax    .object "filename"
```

Will ich wieder ein Objektfile erzeugen, überschreibe ich das ».syntax« wieder mit Leerzeichen – fertig. Eine andere Möglichkeit wäre der Einsatz der bedingten Assemblierung, um zwischen Testlauf und Fileerzeugung zu wählen, dies wird später noch beschrieben.

Wie angedeutet, besitzt ».syntax« noch andere Eigenschaften als die der reinen Verhinderung der Kodespeicherung. Wie das Schlüsselwort schon andeuten soll, ist mit diesem Pseudoop außerdem die Überprüfung des Quelltextes auf etwaige Fehler möglich. Normalerweise führt jeder auftretende Fehler innerhalb des Quelltextes zum sofortigen Abbruch der Assemblierung – Ausnahme: Auffinden eines unbekannten Labels. Hat man einen neuen Quelltext erstellt, ist jedoch die Wahrscheinlichkeit ziemlich groß, daß sich gleich mehrere Fehler eingeschlichen haben werden, es wäre also eine gewisse Arbeitserleichterung, wenn man auch mehrere Fehler gleichzeitig entdecken und verbessern könnte. Diese Möglichkeit bietet der Testlauf mit ».syntax«.

Ein gefundener Fehler führt nicht mehr automatisch zum Abbruch der Assemblierung; statt dessen gibt der Assembler die Fehlermeldung aus und listet die fehlerhafte Zeile – das macht er sonst auch – und wartet daraufhin in einer Tastaturabfrage. Sie können sich jetzt gegebenenfalls den Fehler und die Zeilennummer der fehlerhaften Zeile notieren; besser wäre allerdings, Sie merken sich diese Angaben einfach soweit Sie können und brechen dann den Testlauf ab – der Assembler ist sehr schnell, so daß ein zweiter Start des Testlaufs fast keine Zeit kostet. Dieser Abbruch des Testlaufes erfolgt durch Drücken von »Return« nach Ausgabe der Fehlermeldung. Betätigen sie statt »Return« die Leertaste, setzt der Assembler den Testlauf **hinter** der fehlerhaften Zeile fort, sucht also weitere Fehler. Die fehlerhafte Zeile selbst wird nicht weiter bearbeitet, deshalb kann es sein, daß das Fehlen dieser Zeile bei der Assemblierung weitere Fehler nach sich zieht. Dies wäre z. B. der Fall bei einer Zeile wie

```
100   -&arke      .repeat
```

Der Label »marke« wurde hier falsch geschrieben, was zu einem Fehler »illegal label« führt. Der folgende Pseudoop ».repeat«, der immer von einem ».until« gefolgt werden muß, geht durch die Fehlermeldung unter; trifft der Assembler nun auf das folgende ».until«, kennt er das zugehörige ».repeat« nicht, was zu einer neuen Fehlermeldung führt.

Daß die Assemblierung in zwei Durchläufen geschieht, bewirkt, daß einige Fehlermeldungen doppelt ausgegeben werden – einmal in Pass 1, einmal im zweiten Pass. Dies sollte Sie nicht weiter irritieren, Sie kennen ja jetzt den Grund.

Eine Sonderbehandlung erfährt, nicht nur bei der Assemblierung im Testlauf, der Fehlerfall, daß Sie einen unbekannten Label oder eine unbekannte Variable verwenden. Dieser Fall tritt relativ häufig ein, wenn man sich in der Schreibweise eines Labels irrt. Der Assembler gibt die Fehlermeldung »unknown label« aus; die Feh-

lermeldung selbst enthält noch einmal den Labelnamen in eckigen Klammern – falls in einer Quelltextzeile mehrere Label auftreten sollten, erleichtert dies die Orientierung innerhalb der Zeile. Anschließend erscheint die Frage:

```
»break or definition (b/d)?«
```

Drücken Sie die Taste »b«, wird die Assemblierung abgebrochen, ein eventuell offener File (».object« oder ».modul«) wird geschlossen. Über die Taste »d« erhalten Sie die Gelegenheit, den Wert des Labels für diese eine Zeile nachzutragen. Es erscheint

```
»? =«
```

mit dem Cursor, und Sie können einen arithmetischen Ausdruck eintragen, der dem Labelwert entspricht – z. B. die richtige Schreibweise des Labels. Das einzige, was nicht nachgetragen werden kann, sind Zeropagelabel. Trifft der Assembler im ersten Pass auf einen noch nicht bekannten Label, nimmt er zunächst an, daß die Definition später im Quelltext noch erfolgt, er antizipiert also einen internen Label, der nicht in der Zeropage liegt. Folgt die Definition nicht mehr, können Sie nun natürlich in Pass 2 nicht einen Labelwert kleiner als 256 eingeben, die Längenberechnung dieses einen Befehls und damit die gesamte Symboltabelle würde hierdurch hinfällig.

Probieren Sie die Definition eines unbekannten Labels einmal an folgendem Beispiel aus:

```
100   -.base $c00
110   -          lda marke
```

Starten Sie die Assemblierung mit »run« oder »F6« und führen Sie die oben beschriebenen Schritte durch, um den Labelwert nachzuliefern. Betrachten Sie nun mit Hilfe des Monitors den Befehl an der Adresse $c00. Sie werden sehen, daß der Befehl genau die nachgelieferte Labeladresse enthält. Versuchen Sie, den Labelwert über die Editorfunktion »_« abzufragen, werden Sie bemerken, daß dem Label kein Wert zugewiesen wurde, sondern lediglich der Labelwert für eine Zeile nachgeliefert wurde.

3.3.3 Label und Variablen

Label und Variablen bilden einen der Eckpfeiler für das vernünftige Funktionieren eines Assemblers. Im ASE-Assembler halten die Variablen der assemblereigenen Integerarithmetik für die Speicherung von Labeladressen her, der Variablenname und der Labelname werden synonym benutzt; ob sich hinter einem Namen eine Variable verbirgt oder ein Label, ist eine reine Frage der Interpretation. Es gibt andere Möglichkeiten, eine Symboltabelle zu speichern – z. B. in einer besonderen Tabelle, wie

es viele Assembler tun, die über keine ausgeprägte Arithmetik verfügen. Die Abspeicherung in Variablen bietet jedoch die größte Flexibilität, zumal die später besprochene bedingte Assemblierung die Verfügbarkeit von echten Variablen erfordert.

Vielleicht haben Sie sich bei der Lektüre der letzten Sätze ein wenig gewundert, wieso ich Label und Variablen wie Begriffe für völlig verschiedene Dinge benutzt habe? Dann sollten Sie das folgende noch einmal genau durchlesen, um sich den Unterschied zwischen Labeln und Variablen zu verdeutlichen:

Was sind Variablen? Variablen – in der Computer-Terminologie – sind Speicher für Werte. Variablenwerte können jederzeit verändert werden und müssen jederzeit veränderlich sein.

Dagegen die Label: Label sind Symbole für die Bezeichnung von Adressen – von festen Adressen! Label stehen für genau eine bestimmte Adresse, die sich für die Dauer einer Assemblierung im allgemeinen nicht ändern sollte. Der ASE verwendet als Symbole die Namen von ASE-Variablen, die Adresse, die der Label bezeichnet, wird im Variablenwert gespeichert.

Wie man sieht, sind Label und Variable durchaus nicht ein und dasselbe. Diese Tatsache sollten Sie immer im Hinterkopf behalten, obwohl im folgenden nicht jedesmal auf die unterschiedliche Verwendungsart der ASE-Variablen einmal als Variablen selbst, einmal als Speicher und Symbol für einen Label hingewiesen wird – ist in den Beschreibungen von Variablen und Labeln die Rede, können je nach Sinnzusammenhang auch die gerade nicht erwähnten Verwendungsarten mit gemeint sein.

3.3.3.1 Lokale und globale Variablen/Blöcke

In der Beschreibung der ASE-Arithmetik in Kapitel 3.2.2.4 ist schon angeklungen, daß der ASE grundsätzlich lokale Variablen kennt. Es stellt sich natürlich die Frage, was unter lokalen Variablen zu verstehen ist.

Das beste wird sein, die Erzeugung von lokalen Variablen an einem Beispiel zu zeigen und dieses Beispiel anschließend näher zu erläutern:

```
100   -.base $1234
101   -.define bsout := $ffd2
110   -;
120   -lab1      .byte "text",0
130   -lab2:     .byte "noch ein text",0
140   -.begin
150   -          ldx#0
160   -lab1      lda lab2,x
170   -          beq  lab4
180   -          jsr  bsout
```

```
180   -          inx
190   -          bne  lab1
200   -.end
210   -lab4:     ...
```

Dieses Beispiel entspricht nur sehr bedingt einem Quelltext, der in der Programmierpraxis verwendet werden würde, enthält aber die wesentlichen Elemente, die für eine Erläuterung globaler und lokaler Label notwendig sind:

Als erstes fallen die beiden Pseudoops mit den verheißenden Namen ».begin« und ».end« auf – sehen wir zunächst von ».define« und ».byte« ab, deren genaue Bedeutung wir zwar noch nicht abgehandelt haben, deren Funktion aber schon allgemein angedeutet wurde. Die Namen der beiden neuen Pseudoops besagen schon, daß sie eine gewisse »Klammerfunktion« besitzen. Wenn Sie sich an die fast gleichlautenden Befehle des Basic 7.0 erinnern – die beiden Befehle fassen in Basic mehrere Anweisungen zu einer syntaktischen Einheit, dem Anweisungsblock, zusammen. Ganz ähnlich, aber vollkommen anders, die Wirkung der beiden Pseudoops ».begin« und ».end«: Auch hier werden die zwischen den beiden Befehlen liegenden Anweisungen – besser und genauer gesagt, die Quelltextzeilen zwischen den Befehlen – zu einem Block zusammengefaßt. Dieser Block nennt sich bei ASE ein **lokaler Block,** was nichts anderes aussagen soll, als daß alle Label und Variablen innerhalb dieses Blocks – sofern nicht ausdrücklich anders angegeben – lokal zum Block sind. Lokale Label sind nur innerhalb des Blocks bekannt, in dem sie angelegt wurden. »Anlegen« bezeichnet den Vorgang der ersten Wertzuweisung an einen Label oder an eine Variable, die Wertzuweisung geschieht beispielsweise dadurch, daß ein Label links von einem Befehl steht oder durch eine Zuweisung mit ».define« (es gibt auch noch andere Möglichkeiten, siehe unten). Neben lokalen Labeln und Variablen existieren entsprechend **globale Variablen.** Globale Variablen sind unabhängig von den Blöcken überall erreichbar. Die Klammerung mit ».begin« und ».end« ist nur eine der Möglichkeiten, lokale Blöcke zu erzeugen, der gleiche Vorgang wird auch bei der Makroverarbeitung genutzt, um eine vollkommen freie Benutzung von Labeln innerhalb von Makros erlauben zu können. Dies nur am Rande, zu den Makros später noch mehr.

Zurück zum Beispiel: In der Zeile 1 sehen Sie eine Wertzuweisung an einen externen Label. Diese Wertzuweisung erfolgt mit dem Pseudoop ».define«, gefolgt vom Labelnamen, von einem Doppelpunkt, einem Gleichheitszeichen und schließlich dem Labelwert; ich glaube, die Vorgehensweise bei der Wertzuweisung spricht im Prinzip für sich selbst, so daß ich hierüber nicht viele Worte zu verlieren brauche. Der einzige interessante Aspekt in dieser Zuweisung ist der Doppelpunkt: Er bewirkt, daß der Label gleichzeitig damit, daß er einen Wert erhält, auch als global deklariert wird. Wie Sie weiter unten im lokalen Block sehen, können globale Label auch in den einzelnen Blöcken verwendet werden. Dies sollte gleichzeitig die praktische Verwendung der globalen Label klar machen: Verwendet man innerhalb des Quell-

textes Blockstrukturen, so dienen die globalen Label für Definitionen, die allen Blöcken gleichzeitig zu Verfügung stehen sollen – beispielsweise Routinen des Betriebssystems, Zeropageadresse, wichtige Konstanten usw. Soll ein mit ».define« deklarierter Label lokal sein, also nicht für andere Blöcke erreichbar, genügt es, in der ».define«-Zeile den Doppelpunkt fehlen zu lassen. Ich habe mir diese Variante des Befehls erspart, da mir der Sachverhalt doch relativ einfach scheint.

Unter der eben besprochenen Zeile finden wir die zweite Möglichkeit, wie man Labeln einen Wert zuweisen kann: indem man sie vor den Befehl schreibt, dessen Adresse sie bezeichnen sollen. In unserem Beispiel bezeichnen »lab1« und »lab2« zwei Texte, die beispielsweise im weiteren Verlauf des Quelltextes ausgegeben werden könnten. Zeile 120 ist der Normalfall dieser Wertzuweisung: der Label wird lokal angelegt – der gesamte Quelltext zwischen Anfang und Ende wird automatisch als der erste lokale Block des Programms angesehen. In Zeile 130 geschieht praktisch der gleiche Vorgang, nur folgt hier dem Label ein Doppelpunkt; die Folgen überlasse ich diesmal Ihrer Phantasie.

Darunter beginnt schon der lokale Block. Alle Label innerhalb dieses Blocks sind von »außen« nicht »einsehbar«, ebensowenig sind die lokalen Label anderer Blöcke von »innen« zu »sehen«. Die nochmalige Verwendung des Labels »lab1« innerhalb des Blocks, diesmal als eine Sprungmarke, soll dies noch einmal deutlich machen. Globale Label sind dagegen überall sichtbar, die Verwendung der Label »bsout« und »lab4« zeigt auch diesen Umstand.

Lokale Blöcke können – fast – beliebig ineinander geschachtelt werden, die maximale Schachtelungstiefe liegt bei etwas über 80, was mehr als ausreichen sollte für alle denkbaren Anlässe. Überschreiten Sie die Schachtelungstiefe, gibt der Assembler die Fehlermeldung »structure too complex« aus. Die Anzahl der insgesamt verwendeten Blöcke unterliegt einer ebenso restriktiven Beschränkung: Verwenden Sie bitte nicht mehr als 65534 Blöcke in einem Quelltext, sonst erhalten Sie eine Fehlermeldung »too many structures«. Diese immense Anzahl von Blöcken dürfte aber in der Praxis kaum vorkommen, zumal der Quelltext-Speicher auch nur begrenzt Platz bietet.

Wozu dienen lokale Blöcke? Diese Frage werden Sie sich in der Zwischenzeit sicher gestellt haben. Zunächst einmal: Die Klammerung lokaler Blöcke mit ».begin« und ».end« ist eigentlich ein »Abfallprodukt« der Makrofähigkeit des ASE-Assemblers. Wie Sie bei der Beschreibung der Makros noch sehen werden, legt der Assembler um jedes Makro automatisch einen Block, so daß man innerhalb der Makros beliebig viele Label ohne irgendwelche weiteren Einschränkungen verwenden kann – dies ist nicht selbstverständlich. Wenn Sie Vergleichsmöglichkeiten mit anderen Assemblern haben, werden Sie sehen, daß viele dieser Programme recht große Einschränkungen für die Labelverwendung innerhalb von Makros machen. Obwohl aber die lokalen Blöcke mit ».begin« und ».end« eigentlich nur ein Seiteneffekt der Makroverarbeitung sind, können sie dennoch einige nützliche Arbeit leisten. Nehmen wir als Beispiel folgenden Fall, der gar nicht so selten auftritt, wenn man längere Quelltexte

schreibt, in denen eine Vielzahl – Hunderte – von Labeln verwendet werden: Sie schreiben einen neuen Teil des Quelltextes, etwa ein neues Unterprogramm, und verwenden als untergeordnete Label für die Branches usw. eine Bezeichnung, die an einem anderen Ort des Quelltextes schon einmal vorkommt. Bei der Assemblierung erhalten Sie natürlich eine Fehlermeldung »label twice«, denn da ein Label immer nur ein Symbol für genau eine Adresse sein kann, muß der Assembler die Assemblierung notgedrungen abbrechen. Die erste Möglichkeit, den Quelltext zu verbessern, wäre nun offensichtlich, die doppelt verwendeten Labelbezeichnungen über die Replacefunktionen des Editors auszutauschen. Eine elegantere Methode, die außerdem nicht so arbeitsaufwendig ist, haben wir aber durch die lokalen Blöcke zur Verfügung. Man lege einen Block aus ».begin« und ».end« um das neue Unterprogramm und alle Kollisionen sind vermieden!

3.3.3.2 Wertzuweisungen im Dialog

Zwei mögliche Arten der Wertzuweisung haben wir schon kennengelernt: die durch den Pseudoop ».define« und die implizite durch die Verwendung eines Labelnamens am Beginn einer Quelltextzeile.

Eine dritte, sehr nützliche Art der Definition von Variablen bietet der Pseudoop

```
                ».request "dialogstring",v«
bzw.            ».request "dialogstring":v«
```

Die beiden Unterarten des Befehls unterscheiden sich darin, daß der erste Befehlstyp eine lokale Wertzuweisung an die Variable »v« vornimmt, der zweite Typ macht das gleiche für den globalen Fall. ».request« ähnelt in seiner Funktionsweise dem Basicbefehl »input«. Der Dialogstring wird auf den Bildschirm ausgegeben, anschließend erscheint direkt hinter der Ausgabe der Cursor auf dem Bildschirm. Sie können nun einen arithmetischen Ausdruck eingeben, die Eingabe wird mit »Return« abgeschlossen. Der Wert des Ausdrucks wird an die Variable »v« zugewiesen.

Der Dialogstring darf fehlen, in jedem Fall müssen aber das Komma bzw. der Doppelpunkt vorhanden sein, um anzuzeigen, welche Art der Wertzuweisung erwünscht ist. Ist der angegebene arithmetische Ausdruck fehlerhaft, wird die Assemblierung abgebrochen, sofern sie nicht in einem Testlauf geschieht. Enthält der arithmetische Ausdruck einen unbekannten Label oder eine unbekannte Variable, tritt nicht der normale Prozeß der Nachtragung des Variablenwertes in Gang, denn die Eingabe von Label- und Variablenwerten erfolgt aus naheliegenden Gründen nur im ersten Pass der Assemblierung.

».request« wird in der Hauptsache in Verbindung mit der bedingten Assemblierung benutzt. Obwohl die Befehle zur bedingten Assemblierung noch nicht behandelt wurden, soll das hierfür geeignete Beispiel an dieser Stelle gegeben werden. Die

nachfolgende Beschreibung des Beispiels wird Ihnen außerdem schon einen ersten Eindruck von der bedingten Assemblierung verschaffen.

Beispiel:

```
100   -.request "Startadresse: ",origin
110   -.base origin
120   -.define d     = 0
130   -.define t     = 1
140   -.request "(d)isk oder (t)estlauf: ",option
150   -.cond option.eq.d
160   -         .object "name"
170   -.alter
180   -         .syntax
190   -.econd
200   -         ...
```

Die ersten Zeilen dürften klar sein: Die gewünschte Startadresse für die Assemblierung wird am Bildschirm erfragt, der eingegebene Wert wird in den Pseudoop ».base« eingesetzt. Der zweite Teil des Beispiels soll bewirken, daß der Quelltext in Abhängigkeit von einer Eingabe entweder nur gegen Fehler getestet wird oder aber der erzeugte Maschinenkode direkt zur Floppy gesendet wird. Die Steuerung hierzu soll sinnfällig durch die Eingabe von Buchstaben statt von Zahlen erfolgen, deshalb die Definition zweier Konstanten »d« und »t« in den Zeilen 120 und 130. Die Konstruktion ».cond«/ ».alter«/ ».econd« wirkt entsprechend zu einem »if«/ »else«/ »endif«, dessen Wirkung ich als bekannt voraussetzen möchte. Die Wirkung der bedingten Assemblierung könnte man in Worte fassen: Falls (».cond«) der Benutzer ein »d« eingegeben hat, sende den Kode, ansonsten (».alter«) mache einen Testlauf; ».econd« zeigt das Ende dieser Konstruktion an.

Das kleine Beispiel kann natürlich nicht alle Möglichkeiten aufdecken, die in der bedingten Assemblierung verborgen sind, ich hoffe aber, Ihr Interesse für die späteren Beschreibungen schon einmal geweckt zu haben.

3.3.3.3 Labelredefinitionen

Im vorigen Kapitel habe ich schon einmal kurz durchblicken lassen, daß der Assembler bei doppelt auftretenden Labeln innerhalb eines Quelltextes eine Fehlermeldung »label twice« (Label doppelt) ausgibt. Warum erfolgt hier eigentlich eine Fehlermeldung?

Vergegenwärtigen wir uns noch einmal in Kürze, wie der Assembler einen Quelltext verarbeitet und in ein Maschinenprogramm verwandelt: Er geht dazu in zwei Durchläufen durch den gesamten Quelltext. Im ersten Pass interessieren ihn praktisch nur die im Quelltext verwendeten Label; er baut eine Symboltabelle auf, die alle Label des

Quelltextes mit ihren Adreßwerten enthält. Dazu benötigt er die Startadresse der Assemblierung und den internen Programm- oder Adreßzähler. Für jeden Befehl, den der Assembler im ersten Pass verarbeitet hat, setzt er den Programmzähler um die Befehlslänge weiter; folgt nun irgendwann die Definition eines Labels, der vor einem Mnemonic oder Pseudoop steht, so erhält der Label als Adreßwert den aktuellen Zählerstand des Programmzählers.

Am Ende des ersten Passes liegen – hoffentlich – alle benötigten Label in der Symboltabelle vor, der zweite Pass kann beginnen, in dem die Labelwerte in die Adressierungen beispielsweise der mnemonischen Befehle eingesetzt werden können und damit der endgültige Kode erzeugt werden kann. Im ersten Pass wurde zur Längenbestimmung der Befehle noch eine einfache Regel angewendet: Jeder unbekannte Label gilt als nicht in der Zeropage liegend. Diese Regel wurde aus dem einfachen Grund gewählt, daß Zeropagelabel bei weitem nicht so häufig auftreten wie andere. Trifft der Assembler im zweiten Pass auf einen unbekannten Bezeichner, gibt er die Fehlermeldung »unknown label« aus. Sie könnten nun versuchen, auch Labelwerte für Zeropagelabel nachzutragen; die oben erwähnte Vorgehensweise in Pass 1 macht diese Prozedur jedoch unmöglich, da sich dadurch in Pass 1 und 2 unterschiedliche Befehlslängen ergeben würden, die Symboltabelle würde folglich nicht mehr stimmen.

Die doppelte Verwendung eines Labels für zwei zu bezeichnende Adressen ist ein ähnlich fragwürdiger Fall. ASE erlaubt diese Vorgehensweise zwar durch einen besonderen Pseudoop, Sie sollten sich aber absolut klar über die Problematik sein, bevor Sie diese Möglichkeit zu nutzen versuchen. Nehmen wir ein Beispiel:

```
100   -.base $1234
110   -;
120   -label        lda #0
130   -             ...
200   -             jmp label
210   -label        ldx #1
220   -             ...
```

Dieses Beispiel wäre so nicht lauffähig, der ASE würde die Fehlermeldung »label twice« ausgeben. Nehmen wir aber weiter an, der Assembler würde diese Konstruktion durchgehen lassen, wie würde der Quelltext übersetzt werden? Pass 1 baut die Symboltabelle auf, darin enthält »label« zunächst die Adresse des »lda #0« in Zeile 120. Dieser Wert wird aber im Laufe der weiteren Assemblierung durch die Adresse des »ldx #1« überschrieben, so daß »label« bei Beginn des zweiten Passes diese Adresse enthält. Bei der Kodeerzeugung nun, bei der die Labelwerte in die Adressierungen eingesetzt werden, würde deshalb der Befehl »jmp label« immer mit einem Sprung zum »ldx #1« übersetzt werden, was sicher nicht beabsichtigt ist.

Anders sähe die Sache aus, wenn der Assembler auch während des zweiten Passes die Symboltabelle aktualisieren würde. In diesem Fall würde »label« noch vor der Assemblierung des »jmp« wieder den gewünschten Wert enthalten und die Assemblierung würde das richtige Ergebnis erzielen – vorausgesetzt, der Programmierer ist sich bewußt, was er macht.

Der ASE-Assembler beeinflußt die Symboltabelle während des zweiten Passes nicht mehr – ausgenommen sind hier die Zuweisungen mit ».define«, die ja auch Variablen betreffen können, die für die bedingte Assemblierung unbedingt in beiden Passes verändert werden müssen -, folglich wird die obige Methode der doppelten Labelverwendung nur durch einen besonderen Pseudoop möglich:

```
.set label
```

».set« weist dem angegebenen Label im zweiten Pass der Assemblierung den Wert des aktuellen Programmzählers zu. Das obige Beispielprogramm wäre korrekt, wenn Sie die Zeilen 120 und 210 folgendermaßen ändern:

```
120  ->   120    -.set label
          121    -               lda #0
210  ->   210    -.set label
          211    -               ldx #1
```

Wozu der ganze Aufwand? Ich hätte es ja vorher schreiben können, aber dann hätten Sie die Erklärungen sicher nicht gelesen: Diese Redefinition wird in der Hauptsache von Compilern genutzt, die beispielsweise bestimmte Teile des kompilierten Kodes immer mit den gleichen Labelnamen versehen wollen – dies verringert den Aufwand der Labelbehandlung des Compilers selbst. Es ist aber bestimmt nicht von Schaden, daß Sie nun noch ein wenig mehr über die Labelbehandlung wissen. Vielleicht finden Sie ja noch eine weitere Verwendungsmöglichkeit für die Redefinition?

3.3.4 Tabellen, Texte, Speicherbereiche

In den bisher verwendeten Beispielquelltexten waren schon mehrfach Pseudoops zu sehen, die der Eingabe von Tabellen und Texten dienen, ohne jedoch eine genauere Erläuterung der Funktion dieser Pseudoops zu liefern. Dies soll hier nachgeholt werden.

3.3.4.1 Bytetabellen und Texte mit ».byte«

».byte« ist der wohl flexibelste Pseudoop zur Tabellenerzeugung – er dient zur Eingabe von ASCII-Texten, von Bildkode-Texten, von Texten in reversem Bildkode und nicht zuletzt von Bytewerten in Form arithmetischer Ausdrücke. Alle möglichen Formate können außerdem auch noch gemischt miteinander verwendet werden.

Leser, die bereits den Top-Ass der Version 1.0 kennen, werden bemerken, daß der ».byte«-Pseudoop um die Bildkode-Funktionen erweitert worden ist. Diese Lösung ist flexibler als die durch spezielle Pseudoops für die Bildkode-Texte. Trotzdem werden die in der Version 1.0 verwendeten Pseudoops für diesen Zweck weiter unterstützt, Sie werden Ihre Quelltexte also nicht umschreiben müssen, wenn Sie in Zukunft die Versionen 2.0 der Programme verwenden. Die verschiedenen Verwendungsarten in Beispielen:

```
ASCII-Texte:          .byte "dies ist ein text"
normaler Bildkode:    .byte '"dies ist normaler bildkode"
reverser Bildkode:    .byte &"dies ist reverser bildkode"
Bytewerte:            .byte 12,$ca,<(marke),2*konst/3
```

Mischformen werden durch einfache Aneinanderreihung der einzelnen Elemente – durch Kommata getrennt – eingegeben:

```
Mischformen:          .byte "text mit Nullbyte",0
                      .byte &"bild",0,"ascii",0
```

Die Verwendungsweise dieses Pseudoops wird wohl am deutlichsten durch drei typische Beispielprogramme, die so, wie sie sind, häufig in Quelltexten anzutreffen sind:

Beispiel 1:

Ausgabe eines ASCII-Textes über die Betriebssystemfunktion »bsout«

```
100     -.base $c00
110     -.define _cr          =  $ff00
120     -.define bsout        =  $ffd2
130     -;
140     -start     ldx#0       ;  Index auf 0
150     -          stx _cr     ;  I/O-Bereich ein!
160     -loop      lda text,x  ;  Textzeichen
170     -          beq ende    ;  Null:   -> Ende
180     -          jsr bsout   ;  sonst ausgeben
190     -          inx         ;  Index + 1
200     -          bne loop    ;  springt immer
210     -ende rts
220     -;
230     -text      .byte "hello world!",0
```

Beispiel 2:

Verwendung interner Adressen als Zwischenspeicher für Daten

```
100     -uprog  sta ac      ;  Register in internen
110     -       stx xr      ;  Zwischenspeichern
120     -       sty yr      ;  sichern
                ...
200     -       ldy yr      ;  Zum Ende eines Unter-
210     -       ldx xr      ;  programms wieder
220     -       lda ac      ;  laden
230     -       rts
240     -;
250     -ac     .byte 0     ;  Je ein Byte Speicher
260     -xr     .byte 0     ;  für jedes
270     -yr     .byte 0     ;  Register
```

Beispiel 3:

Sprungverteiler

```
100   -loop     jsr basin
110   -lp1      ldx #0
120   -         cmp abk,x
130   -         beq lp2
140   -         inx
150   -         cpx #max
160   -         bcc lp1
170   -         bcs loop
180   -;
190   -lp2      lda highadr,x
200   -         pha
210   -         lda lowadr,x
220   -         pha
230   -         rts
240   -;
250   -abk      .byte  "abc"
260   -lowadr   .byte  <(up_a-1),<(up_b-1),<(up_c-1)
270   -highadr  .byte  >(up_a-1),>(up_b-1),>(up_c-1)
```

Das letzte Beispiel ist vielleicht für Ihre eigenen Programmierungen das interessanteste. Es ist ein typischer Befehlsverteiler, wie er in Programmen verwendet wird, die z. B. aufgrund verschiedener Tastatureingaben zu unterschiedlichen Unterprogrammen verzweigen. Im Fall des Beispiels wird ein Zeichen vom Bildschirm geholt; dieses Zeichen wird mit einer Liste von möglichen Befehlen verglichen (»abk«); bei

Übereinstimmung mit einem möglichen Befehl wird zum betreffenden Unterprogramm verzweigt. Dazu wird die Adresse des Unterprogramms minus Eins auf den Stack gelegt und der PC des Prozessors durch »rts« auf das Unterprogramm gerichtet. Die um Eins verminderten Adressen der Unterprogramme befinden sich in zwei Tabellen, die jeweils das Lowbyte und das Highbyte der Adresse enthalten. Die Adressenbestandteile liegen in den Tabellen in der gleichen Abfolge vor wie die Befehlskürzel, so daß ein einfacher indizierter Zugriff auf sie über das X-Register möglich ist.

3.3.4.2 Adreßtabellen mit ».word«

Der obige Fall einer Sprungtabelle ist gar nicht so selten, wie man vielleicht annehmen sollte. Leider ist die Notation einer solchen Tabelle mit ».byte« aber gleichzeitig recht umständlich – man muß das »<«- oder das »>«-Zeichen eingeben, außerdem muß immer das »-1« getippt werden. Das konnte auf die Dauer so nicht bleiben, da Programmierer in der Regel recht schreibfaule Leute sind – außerdem: jedes Zeichen, das getippt werden muß, ist eine mögliche Quelle von Syntaxfehlern. Dies hat schließlich zur »Erfindung« eines weiteren Pseudoops geführt:

```
».word a1,a2,a3...«
```

».word« legt die Adressen »a1,a2,...« als Folge von Low- und Highbytes hintereinander im Speicher ab bzw. schreibt sie auf Diskette. Das Format der Adressen entspricht dem beim 6502 üblichen Standard: das Lowbyte steht an der niedrigeren Adresse.

Der Pseudoop ».word« ist ausschließlich für die Verwendung in Adreßtabellen vorgesehen, er erlaubt aus diesem Grunde lediglich die Angabe arithmetischer Ausdrücke in der Parameterliste. Ein Sprungverteiler wie im Beispiel oben sieht mit ».word« ein wenig anders aus:

```
100   -loop   jsr basin
110   -lp1    ldx #0
120   -       cmp abk,x
130   -       beq lp2
140   -       inx
150   -       cpx #max
160   -       bcc lp1
170   -       bcs loop
180   -;
190   -lp2    txa         ;  X-Register mal 2
200   -       asl
210   -       tax
220   -       lda adr+1,x
230   -       pha
```

```
240   -       lda adr,x
250   -       pha
260   -       rts
270   -;
280   -abk    .byte  "abc"
290   -adr    .word  up_a-1,up_b-1,up_c-1
```

Diese Schreibweise bedeutet mit Sicherheit weniger Tipparbeit als die mit ».byte«. Das ständige »-1« kann man sich auch noch sparen, wenn man statt des »rts« einen Sprung über einen Vektor benutzt. Das folgende Beispiel zeigt einen Sprungverteiler dieser Art; der Teil bis zur Marke »lp2« ist der gleiche wie oben:

```
190   -lp2    txa
200   -       asl
210   -       tax
220   -       lda adr,x
230   -       sta vec
240   -       lda adr+1,x
250   -       sta vec+1
260   -       jmp (vec)
270   -;
280   -abk    .byte "abc"
290   -adr    .word up_a,up_b,up_c
```

Das dritte Beispiel zum Thema Sprungverteiler zeigt einen kleinen Kniff, der selbstmodifizierenden Kode erzeugt. Dieser Trick erlaubt es, Unterprogramme aus einer Sprungleiste mit »jsr« aufzurufen – die bisherigen Verteiler führten ja einen direkten Sprung zu den Unterprogrammen aus, wodurch die Unterprogramme selbst mit einem Sprung zur Eingabeschleife enden mußten. Es wird wieder nur der Teil ab »lp2« gezeigt:

```
190   -lp2    txa
200   -       asl
210   -       tax
220   -       lda adr,x
230   -       sta jmper+1
240   -       lda adr+1,x
250   -       sta jmper+2
260   -jmper  jsr $ffff
270   -       jmp loop
280   -;
290   -abk    .byte "abc"
300   -adr    .word up_a,up_b,up_c
```

Eine solche Programmierung würde natürlich jedem Guru der strukturierten Programmierung die Haare zu Berge stehen lassen – das soll uns aber nicht weiter stören: Hauptsache, es funktioniert!

3.3.4.3 Speicher reservieren/».space of«

In einem der Beispiele zu ».byte« wurde schon gezeigt, daß Maschinenprogramme in der Regel eine Anzahl an Speicherzellen für die Speicherung anfallender Daten benötigen. Nicht alle dieser Daten können logischerweise in der Zeropage untergebracht werden, dazu ist diese einfach zu überlaufen und es gibt auch Fälle, in denen die 256 Byte der Zeropage einfach nicht genug wären, alle Daten aufzunehmen. In einem solchen Fall geht man gewöhnlich so vor, daß ein gewisser Platz innerhalb des Maschinenprogramms selbst für die Datenspeicherung vorgesehen wird. Diese Vorgehensweise ist günstiger als die Wahl fester Speicherbereiche außerhalb des Programms, da man nie wissen kann, ob diese Speicherbereiche nicht wichtige Daten enthalten, die von anderen Programmen benötigt werden könnten. Vielleicht kommt ja eines Tages ein Programm zur schnellen Abspeicherung auf Diskette heraus, das gerade den Kassettenpuffer benötigt, den Sie als Datenbereich für ein Maschinenprogramm ausgewählt haben?

Es gibt nun verschiedene Möglichkeiten, innerhalb eines Maschinenprogramms freie Datenbereiche zu erhalten. Zwei dieser Möglichkeiten haben Sie bereits kennengelernt, die Pseudoops ».byte« und ».word«, die jeweils ein bzw. zwei Bytes belegen, in denen man Daten abspeichern kann.

Was aber tun, wenn beispielsweise 200 Byte Speicher benötigt werden, um einen Eingabestring festzuhalten? Hier hilft der Pseudoop ».space of« weiter: Mit diesem Pseudoop kann man bis zu 255 Byte am Stück reservieren. Wahlweise kann dieser Speicherbereich mit einem bestimmten Wert vorbesetzt werden oder undefiniert bleiben. Der letzte Fall trifft natürlich nur zu, wenn Sie den Maschinenkode direkt im Speicher ablegen lassen. In diesem Fall wird einfach die Adresse für die weitere Speicherung von Kode nach ».space of« um die Länge des reservierten Bereiches weitergesetzt. Im Fall, daß Sie den Kode zur Floppy schreiben (».object« oder ».modul«), werden statt dessen Nullbytes ausgegeben.

Die Schreibweise der beiden Befehlsarten:

```
.space of n,w
.space of n
```

Die erste Variante besetzt den Speicherplatz mit dem Wert »w« vor, die zweite hinterläßt undefinierten Speicherplatz bzw. gibt Nullbytes zur Floppy aus.

Ein Beispiel für eine Verwendung dieses Pseudoops; es wird eine Eingabezeile mit »basin« in einen Puffer geholt:

```
100   -.base $b00
105   -.define _cr   = $ff00
110   -.define basin = $ffcf
120   -;
130   -puffer    .space of 255," "
140   -;
150   -start     ldx #0
160   -          stx _cr
170   -loop      jsr basin
180   -          cmp #13
190   -          beq lp1
200   -          sta puffer,x
210   -          inx
220   -          bne loop
230   -;
240   -lp1       lda #0
250   -          sta puffer,x
260   -          rts
```

Der reservierte Speicherplatz wird hier mit einem Leerzeichen gefüllt, damit Sie die Wirkung mit dem Monitor besser verfolgen können. Das Ende der Eingabezeile wird mit einem Nullbyte gekennzeichnet, damit weitere Unterprogramme, die diese Eingabezeile auswerten würden, das Ende problemlos feststellen können. Beachten Sie auch noch einmal, daß der In/Out-Bereich eingeschaltet werden muß; diese Tatsache kann man nicht oft genug wiederholen.

Bei der Beschreibung der bedingten Assemblierung finden Sie noch ein Beispielprogramm, das zeigt, wie man bequem auch größere Bereiche als 255 Byte reservieren kann.

3.3.4.4 Bildkodetexte mit ».screen« und ».revers«

Diese beiden Pseudoops sind nur aus Gründen der Aufwärtskompatibilität mit den Versionen 1.0 des ASE-Assemblers noch Bestandteil des Befehlsvorrats. Sie dienen dem gleichen Zweck, den auch ».byte« in bezug auf Bildkodewerte erfüllt, sind jedoch nicht so flexibel wie dieser, da sie keinerlei Mischformen mit arithmetischen Ausdrücken erlauben. Aufgrund der geringeren Bedeutung dieser Befehle sollte es genügen, die Syntax anzugeben:

```
».screen "bildkode"«
```

und

```
».revers "bildcode"«
```

3.3.5 Einbindungen und Verkettungen von Quelltexten

Bisher sind wir stets von der Voraussetzung ausgegangen, daß der zu bearbeitende Quelltext im Speicher des Rechners liegt. Dies ist jedoch nicht immer vollständig möglich – wie Sie schon an den Beispielprogrammen sehen, sind Quelltexte umfangreich, erst recht, wenn sie in der endgültigen Fassung vernünftig auskommentiert wurden. Ich darf vielleicht als Beispiel die Länge der ASE-Quelltexte angeben, die ca. 400 KByte umfaßt.

Der ASE-Assembler bietet nun insgesamt vier Möglichkeiten, um mit langen Quelltexten umzugehen; drei davon fallen unter den hier anstehenden Themenkreis, die vierte schlägt etwas aus der Art und wird deshalb unter dem Titel »relokatibler Kode/Linker« vorgestellt.

Allen Verfahren ist gemeinsam, daß ein langer Quelltext in mehrere Teile gespalten werden muß, bevor der Assembler ihn verarbeiten kann. Unterschiedlich sind die Vorgehensweisen, in denen der Assembler die einzelnen Teile des Gesamtquelltextes bei der Verarbeitung behandelt. Der ASE ist in dieser Beziehung sehr flexibel gehalten; es sollte hier jeder das finden, was er für seine Arbeit gerade braucht.

3.3.5.1 Einbindungen mit ».source«

Möglichkeit 1: Nehmen wir den Fall, daß wir einen Quelltext erweitern wollen, die Erweiterung aber nicht mehr in den Speicher paßt. Wir können nun so vorgehen: Wir suchen uns einen Teil des Quelltextes, der mit einiger Sicherheit schon seine endgültige Form hat, und speichern diesen mit Hilfe des Editorkommandos ».s« (Abspeichern von Arbeitsbereichen) auf Diskette ab. Anschließend löschen wir den Bereich mit »delete« und setzen genau an die Stelle des Quelltextes, an der der abgespeicherte Block vorher stand, den Pseudoop

```
».source "filename",u«
```

Dies ist die Anweisung an den Assembler, an genau der Stelle des Pseudoops vorübergehend Quelltextzeilen direkt von Diskette zu lesen. »filename« enhält den Namen der Datei, die gelesen werden soll, »,u« bietet die Möglichkeit, eine Gerätenummer anzugeben, falls Sie mit mehr als einer Floppy arbeiten. Wie bei ».object« kann dieser Teil der Parameterliste fehlen, wenn vom Gerät mit der Num-mer 8 gelesen werden soll.

Diese Art der Behandlung von Quelltexten nennt man eine **Einbindung.** Einbindung deshalb, weil das Lesen der Quelltextzeilen an einem beliebigen Ort innerhalb des Quelltextes erfolgen kann. Ist das Ende der gelesenen Datei erreicht, setzt sich die Assemblierung im Quelltext fort, der im Speicher steht.

Von Diskette mit ».source« verarbeitete Quelltexte werden während der Assemblierung zunächst einmal genauso behandelt wie Quelltexte im Speicher. Deshalb muß

ein Quelltext in der Regel nicht umgeschrieben werden, wenn er, statt im Speicher zu stehen, von Diskette gelesen werden soll. Eine Einschränkung ist jedoch zu machen, die die Makrobehandlung betrifft; diese wird bei der Beschreibung der Makros erläutert, wo dieses Thema meiner Meinung nach hingehört.

Nicht erlaubt sind Einbindungen innerhalb von Einbindungen. In einem mit ».source« gelesenen Quelltext darf also kein weiteres ».source« vorkommen. Diese Regel wird vom Assembler nicht ausdrücklich überwacht; probieren wir also einmal aus, was passiert, wenn Sie sie nicht beachten. Schreiben Sie die drei folgenden kleinen Quelltexte:

```
100   -       lda #1
110   -       .source "fl2"
```

Speichern Sie diesen Text als »fl1« auf Diskette ab.

```
100   -        lda #2
```

Diese Zeile speichern Sie als »fl2«. Die Einbindung von »fl1« erfolgt aus einem Quelltext heraus, der im Speicher steht:

```
100   -.base $c00
110   -       .source "fl1"
```

Wenn Sie die Assemblierung dieses Quelltextes mit »run« starten, werden Sie bemerken, wie der Assembler auf die Diskette zugreift, um »fl1« zu lesen. Sobald er auf die ».source«-Anweisung innerhalb von »fl1« trifft, macht der In/Out-Handler des Assemblers nicht mehr mit, da er nicht auf eine Schachtelung von Einbindungen vorbereitet ist. Der Cursor erscheint aufgrund dessen auf dem Bildschirm, aber das System nimmt keine Eingaben mehr an. Verlust des Quelltextes im Speicher, wenn dies der Ernstfall wäre? Mitnichten! Diese Situation gibt die willkommene Gelegenheit, zu zeigen, daß der ASE recht resistent gegen auftretende Fehler ist: Drücken Sie gleichzeitig auf die Tasten »RUN/STOP« und »RESTORE«, um einen Neustart des Systems zu veranlassen. Danach befinden Sie sich nach wie vor im Assembler! Der Quelltext steht wie zuvor im Speicher und kann entweder neu assembliert werden oder was Sie sonst vorhaben. Diese Eigenschaft des ASE ist besonders wichtig, wenn Sie ein Maschinenprogramm austesten wollen, den Assembler aber gleichzeitig mit dem Programm im Speicher halten.

3.3.5.2 Verkettungen

Verkettungen gehen grundsätzlich anders vor als Einbindungen von Quelltexten. Der Gesamttext wird zwar wie bei den Einbindungen in mehrere Teile gespalten, diese werden aber nicht aus einem einheitlichen Basistext im Speicher aufgerufen und direkt von Diskette gelesen, sondern nach und nach selbst in den Speicher geladen.

Trotz dieser Gemeinsamkeit, die alle Arten von Verkettungen aufweisen, gibt es doch eine Reihe Variationsmöglichkeiten, wie die Verarbeitung der Quelltexte im einzelnen vorgenommen werden kann. Im ASE-Assembler sind zwei davon realisiert.

3.3.5.2.1 Chain-Verkettung

Die Chain-Verkettung wird durch drei Pseudoops gesteuert:

```
.chain
.continued "filename",u
.chainend "filename",u
```

».chain« leitet die Verkettung am Beginn des ersten Quelltextteiles ein. ».chain« **muß** vor der ersten Wertzuweisung an einem Label stehen, damit die nachfolgenden Quelltexte Zugriff auf alle Label und Variablen des ersten Textes haben. Die Position des Pseudoops wird durch den Assembler nicht überwacht, da eventuell Anwendungen denkbar sind, die ».chain« an einer anderen Position verwenden wollen. Für Benutzer, die dieses vorhaben, sei die interne Funktion von ».chain« kurz angerissen, obwohl man diese Kenntnis nicht unbedingt benötigt, um die Verkettung anwenden zu können: Alle nach ».chain« folgenden Label- und Variablennamen werden in einem besonderen Stack aufbewahrt, so daß auch nachgeladene Texte Zugriff auf Variablen und Label der folgenden Texte nehmen können.

Am Ende des ersten werden bis zum vorletzten Quelltextteil die nachfolgenden Teile durch die Anweisung ».continued "name",u« aufgerufen. Die Anweisung wirkt so, daß das angegebene File »name« vom Gerät mit der Nummer »u« in den Speicher geladen wird. Auch hier greift wieder der Mechanismus Platz, daß die Gerätenummer in der Parameterliste fortfallen kann, wenn das Gerät 8 benutzt werden soll. Es wäre einigermaßen günstig, wenn ».continued« die jeweils letzte Anweisung der Quelltexte wäre – alle Quelltextzeilen hinter dieser Anweisung werden durch den Assembler ignoriert.

Der letzte Quelltext in der Kette wird nicht durch ».continued«, sondern durch den Pseudoop ».chainend "name",u« abgeschlossen. »name« bezeichnet den ersten Quelltext in der Kette – jenen, der im Speicher stand, als die Assemblierung gestartet wurde. Daraus ersieht man schon, daß sämtliche Quelltextteile für die Chain-Verkettung auf Diskette liegen müssen. Wird eine Änderung an einem beliebigen Teil des Quelltextes durchgeführt, muß das Ergebnis der Änderung zunächst abgespeichert werden, bevor eine neue Assemblierung erfolgen kann. Dies ist bei Einbindungen mit ».source« nicht zwingend erforderlich – der Quelltext, in den eingebunden wird, kann im Speicher verändert werden, da keine Quelltextteile in den Speicher nachgeladen werden. Als Ausgleich dafür bietet die Chain-Verkettung mehr Möglichkeiten für Makroaufrufe, dies wird bei der Besprechung der Makros noch genauer erläutert. Es sei hier nur erwähnt, um zu zeigen, daß die Chain-Verkettung durchaus ihre Daseinsberechtigung hat.

Bild 3.1 zeigt noch einmal, wie der Gesamt-Quelltext bei der Chain-Verkettung aufgeteilt wird.

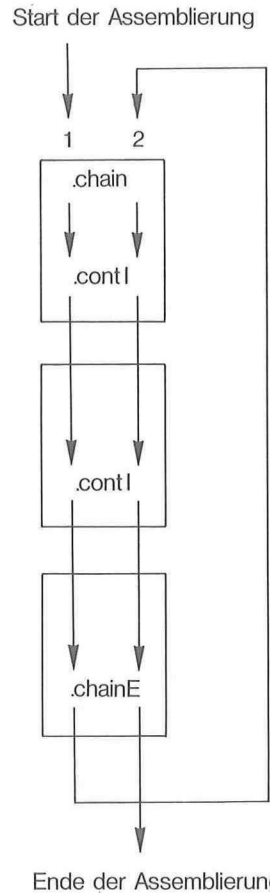


Bild 3.1. Verlauf der Assemblierungsdurchläufe in einer Chain-Verkettung.

».continued« und ».chainend« wirken in jedem Pass der Assemblierung, d. h. jeder Quelltextteil wird für jeden Durchlauf der Assemblierung einmal in den Speicher geholt. Bei der Erzeugung eines normalen Maschinenprogramms bedeutet dies, daß – bis auf den ersten Teil – alle weiteren Teile zweimal geladen werden. Wird ein relokatibles Modul erzeugt (s. u.), werden die Teile sogar dreimal geladen. Diese Vorgehensweise ist aber zeitmäßig fast gleichwertig zur Einbindung mit ».source«.

Beachten Sie besonders, daß zum Ende der Assemblierung nicht das Ausgangsfile im Speicher steht, sondern das letzte Glied der Verkettung. Dies kann bei nur gelegentlicher Verwendung dieser Verkettungsart ein wenig verwirrend sein.

Zum Schluß noch ein ganz frugales Beispiel für die Verwendung der drei Pseudoops. Geben Sie dazu die drei folgenden Quelltexte ein:

```
90    -          .chain
100   -.base $c00
105   -.object "file.o"
110   -          lda #0
120   -          .continued "file2"
```

Speichern Sie diesen Text als »file1« auf Diskette ab.

```
100   -          lda #1
11ß   -          .continued "file3"
```

Dieser Text sollte unter dem Namen »file2« gespeichert werden.

```
100   -          lda #2
110   -          .chainend "file1"
```

Mit diesem Quelltext, der als »file3« gespeichert werden sollte, endet die Verkettung. Laden Sie nun wieder »file1« und starten Sie die Assemblierung mit »run«. Nach Abschluß der Assemblierung können Sie das Ergebnis mit »bload "file.o"« laden und mit dem Monitor betrachten (d c00). Der Befehl ».object« wurde hier nur gewählt, um zu zeigen, daß Einbindungen und Verkettungen keine Auswirkungen auf die gewählte Art der Speicherung des erzeugten Maschinenkodes haben.

3.3.5.2.2 Append-Verkettung

Wenn Sie schon Erfahrung mit meinem Public-Domain-Assembler Hypra-Ass auf dem C-64 gesammelt haben, werden Sie diese Art Verkettung bereits kennen. Sie arbeitet beim ASE auf exakt die gleiche Art wie beim Hypra-Ass.

Die Append-Verkettung wird durch zwei Pseudoops gesteuert:

```
          .append "filename",u
          .common var1,var2...
bzw.
          .common var1:var2:...
```

Der Gesamt-Quelltext wird wie bei der Chain-Verkettung in Teile gesplitted; jeder Quelltextteil außer dem letzten wird mit ».append« beendet, diese Anweisung lädt das angegebene File in den Speicher. Es existieren keine Entsprechungen zu ».chain« und ».chainend«; woran dies liegt, wird klar, wenn Sie wissen, daß das Nachladen des nächsten Quelltextteils erst dann stattfindet, wenn die Assemblierung des vorigen Teils **vollständig** abgeschlossen ist. »Vollständig« bedeutet, daß jeder Quelltextteil alle Passes der Assemblierung – zwei bei der Erzeugung eines normalen

Maschinenprogramms, drei bei einer Modulerzeugung – durchläuft, bevor der nächste Teil zur Bearbeitung in den Speicher geholt wird. Das Nachladen geht einher mit einer vollständigen Löschung des Variablenfeldes des Assemblers. Um es noch einmal anders auszudrücken: Wenn keine anderen Maßnahmen getroffen worden sind, erfolgt die Assemblierung der einzelnen Teile inklusive Kodeerzeugung vollkommen unabhängig voneinander, als wenn Sie die einzelnen Teile nacheinander mit »run "name"« oder »dload "name"; run« geladen und assembliert hätten. Bild 3.2 zeigt den Aufbau der Append-Verkettung noch einmal:

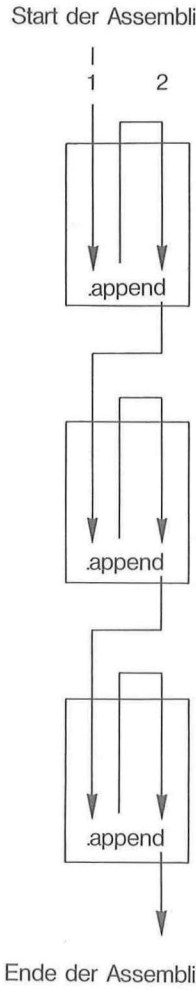


Bild 3.2. Verlauf der Assemblierungsdurchläufe in einer Append-Verkettung

Schon mit dieser einfachen Art der Verkettung sind interessante Effekte möglich – z. B. in Zusammenhang mit der bedingten Assemblierung. Nachfolgend wieder ein solches Beispiel, obwohl die Pseudoops für die bedingte Assemblierung noch nicht besprochen worden sind. Sie können es ja noch einmal nachschlagen, wenn Sie das Kapitel über die bedingte Assemblieung bearbeitet haben. Sinn des Beispiels: Es soll durch eine Tastatureingabe mit ».request« entschieden werden, welcher Quelltext bearbeitet werden soll; dieser Text wird in den Speicher geladen und assembliert. Nach der Assemblierung soll wieder der Abfrage-Quelltext im Speicher stehen:

Geben Sie zunächst den folgenden Quelltext ein, und speichern Sie ihn unter dem Namen »choice«:

```
90    -.cond p1
100   -.print "Quelltext-Auswahl"
110   -.print "=================="
120   -.print
130   -.print "1: file1"
140   -.print "2: file2"
150   -.print
160   -.request "auswahl: ", auswahl
160   -;
165   -.econd
170   -.control auswahl
180   -.= 1
190   -          .append "file1"
200   -.= 2
210   -          .append "file2"
220   -.-
```

Die beiden Quelltexte »file1« und »file2« können für unser Beispiel identisch sein:

```
100   -.base $c00
110   -          lda #0
120   -.append "choice"
```

Natürlich könnten die Quelltexte »file1« und »file2« wieder bedingte Assemblierung enthalten, die beispielsweise eine feinere Auswahl treffen könnte, während »choice« nur eine thematische Auswahl trifft – die Möglichkeiten, die die bedingte Assemblierung hier bietet, sind praktisch unbeschränkt.

Was bringt die Append-Verkettung für eine »normale« Assemblierung? Sicher nicht sehr viel, wenn der Pseudoop ».append« keine weitere Unterstützung erfahren würde. Stellt man sich die Teile der Append-Kette als Teile eines großen Gesamttextes vor, der als Gesamtheit ein einziges Maschinenprogramm beschreibt, muß man davon ausgehen, daß die Quelltextteile untereinander abhängig sind: In Teil 1

wird beispielsweise ein Unterprogramm aufgerufen, dessen Quelltext in Teil 2 steht; in Teil 2 wird eine Speicherzelle verwendet, die in Teil 1 reserviert wurde usw. Mit anderen Worten gesagt: die einzelnen Kettenglieder werden Label aus anderen Gliedern der Kette benötigen, während ».append« alle Label aus vorangegangenen Assemblierungen löscht.

An dieser Stelle tritt ».common« auf den Plan. Dieser Pseudoop hat zwei Auswirkungen:

- Alle Quelltextzeilen bis zur letzten Zeile, in der ein ».common« steht (Common-Zeile), bleiben beim nächsten Nachladen im Speicher stehen (Common-Bereich). Der mit ».append« nachgeladene Quelltextteil wird hinter den Common-Bereich »gemerged«. Enthält auch der nachgeladene Teil einen Common-Bereich, so akkumulieren sich die einzelnen Bereiche während der Assemblierung, was sich logisch auch schon aus der obigen Erläuterung ergibt.
- Alle Label, die in einer Parameterliste hinter ».common« stehen, werden von der sonstigen Löschung des Variablenfeldes ausgenommen und stehen damit während der Assemblierung des nächsten Teils zur Verfügung. Lokale und globale Label/Variablen werden in unterschiedlichen Parameterlisten angeführt:

```
.common var1, var2 ...    Lokale Label/Variablen
.common var1: var2: ...   Globale Label/Variablen
```

  Im Fall der Übergabe globaler Variablen ist zu beachten, daß der Doppelpunkt auch hinter dem letzten Variablennamen in der Parameterliste stehen muß.

Folgende Einschränkung ergibt sich hinsichtlich der Weitergabe von Labelwerten, wenn man keine besonderen Tricks anwenden will – was beim ASE wegen der vorhandenen anderen Möglichkeiten zur Verkettung und Einbindung nicht notwendig ist: Labelwerte werden immer von einem vorausgegangenen zu einem nachfolgenden Quelltextteil weitergereicht. Daraus folgt, daß der vorangehende Quelltextteil keine Label aus späteren Teilen verwenden kann. Beim Hypra-Ass, der als einzige Möglichkeit die Append-Verkettung bietet, wurde diese Einschränkung auf eine wirklich trickreiche Weise umgangen; auf diese möchte ich aber hier nicht eingehen, da der ASE, wie gesagt, weitergehende Möglichkeiten bietet.

Was bietet der Common-Bereich sonst noch für Möglichkeiten außer der Übergabe von Labeln: Man könnte z. B. Makrodefinitionen im Common-Bereich unterbringen – diese Makros wären auch für die nachfolgenden Quelltexte verfügbar. Es könnten auch die Definitionen für Betriebssystemroutinen und Zeropageadressen für alle Quelltextteile gemeinsam in einem Common-Bereich stehen und was der Dinge mehr sind, die für alle Quelltexte gleichermaßen verfügbar sein sollen. Zu beachten ist allerdings, daß nicht alle Anweisungen an den Assembler in den Common-Bereich gehören: Würde beispielsweise ».base« innerhalb eines Common-Bereiches stehen, würde diese Anweisung für jeden nachgeladenen Teil wieder assembliert; alle

Quelltextteile würden also für die gleiche Startadresse assembliert, was sicher nicht immer erwünscht ist.

Ein anderer Punkt ist, daß ».common« auch zusammen mit der Chain-Verkettung eingesetzt werden kann. Auf diese Weise könnte man z. B. erreichen, daß einige Makros, die von allen Chain-Kettengliedern gebraucht werden, in einem Common-Bereich übergeben werden. Die Labelübergabe des Pseudoops spielt bei der Chain-Verkettung natürlich keine Rolle, da hier keine Löschung des Variablenfeldes stattfindet. Ein Beispiel hierzu:

Geben Sie die beiden folgenden Quelltexte ein:

```
»file1«: 100     -; hier ist der Common-Bereich
         110     -;
         120     -.common
         130     -;
         140     -.chain
         150     -          .base $c00
         160     -;
         170     -.continued "file2"

»file2«: 100     -; dies ist file 2
         110     -;
         120     -.chainend "file1"
```

Wenn Sie die Assemblierung mit dem Quelltext »file1« starten, erhalten Sie als zweiten Quelltext im Speicher (nach dem Nachladen) folgendes:

```
         100     -; hier ist der Common-Bereich
         110     -;
         120     -.common
         100     -; dies ist file 2
         110     -;
         120     -.chainend "file1"
```

Die teilweise doppelten Zeilennummern stören den Assembler nicht weiter. Was Sie aber sehen können, ist die Übergabe des Common-Bereiches auch an die nachgeladenen Glieder der Chain-Kette. Diese Vorgehensweise wird zwar nicht direkt vom Assembler unterstützt, so daß ich keine nützliche Einsatzmöglichkeit für solche Mischformen angeben kann, andererseits überwacht der Assembler ausdrücklich nicht, ob z. B. auf ein ».chain« ein ».continued« folgt oder ob einmal ein ».append«, einmal ein ».chainend« verwendet wird. Diese Vorgehensweise soll eine möglichst große Flexibilität gewährleisten und stellt sicher für Sie als Anwender noch ein gewisses Feld für Experimente dar – besonders in Verbindung mit der bedingten Assemblierung.

3.3.5.3 Variablenfeld und Namensstack

In Zusammenhang mit Einbindungen und Verkettungen von Quelltexten ist ein Pseudoop zu erwähnen, den Sie nicht anwenden können, ohne zumindest im Groben zu wissen, wie der Assembler seine Variablen verwaltet. Deshalb soll dieser Punkt zuerst erläutert werden:

Der ASE benützt für die Beschreibung jeder Variablen neun Byte Speicherplatz innerhalb eines insgesamt 20 KByte langen sogenannten Variablenraums; dieser Variablenraum liegt in der RAM-Bank 1 zwischen den Adressen $8000 und $cfff. Aus der Länge eines Variablendeskriptors und der Länge des Variablenraums ergibt sich, daß der Assembler maximal 2275 Variablen aufnehmen kann. Verwenden Sie mehr Label, erhalten Sie die Fehlermeldung »too many labels«. Die Verteilung der Variablendeskriptoren im Variablenraum erfolgt nach einem Hashkode-Verfahren, was sich so auswirkt, daß die einzelnen Deskriptoren nicht sequentiell im Variablenraum liegen, sondern scheinbar willkürlich verteilt. Die Betonung liegt hier auf dem »scheinbar«, denn das Hashverfahren erlaubt eine maximale Suchgeschwindigkeit bei der Suche nach bestimmten Variablen, was sich in einer deutlichen Steigerung der Assembliergeschwindigkeit gegenüber Assemblern mit sequentieller Variablenverwaltung äußert. Um Ihnen einen Begriff von der Maximalzahl von 2275 Labeln zu geben, sei wieder der ASE selbst herangezogen: Für die Assemblierung dieses 17 KByte langen Maschinenprogramms wurden ca. 1500 Label benötigt.

Die Variablendeskriptoren sind wie folgt aufgebaut:

Byte 0: Hat nur für den Linker Bedeutung und soll nicht näher betrachtet werden.

Byte 1: Das Variablen-Statusbyte. Hier wird z. B. festgehalten, ob ein Label global ist, ob er »common« ist usw. Insbesondere findet sich hier auch die Information, in welcher der beiden RAM-Bänke der Variablenname gespeichert ist.

Byte 2 und 3: Variablenwert.

Byte 4 und 5: Adresse des Variablennamens.

Byte 6: Länge des Variablennamens.

Byte 7 und 8: Ordnungszahl der Variablen, sofern sie lokal ist. Mit anderen Worten: In welchem Block wurde die Variable angelegt?

Betrachten wir den Aufbau des Variablendescriptors genauer, fällt auf, daß der Variablenname nicht im Deskriptor erscheint. Statt dessen wird ein Zeiger auf den Namen und eine Längenangabe verwendet. Diese Methode hat eine einfache Ursache: Ich bin davon ausgegangen, daß in der Mehrzahl der Fälle bei Assemblierungen der Quelltext im Speicher des Rechners steht. Innerhalb des Quelltextes sind aber gleichzeitig die Variablennamen zu finden, folglich brauche ich keinen weiteren Speicherplatz für die Speicherung der Namen zu verschenken. Nur im Fall, daß der Quelltext von Diskette gelesen wird, muß ich die Variablennamen irgendwo aufbewahren; dies geschieht in

der RAM-Bank 1, oberhalb des Variablenraums, in einem sogenannten Namensstack, der zwischen den Adressen $d000 und $feff liegt. Dies erklärt auch das Vorhandensein eines Bits im Variablenstatus, das anzeigt, in welcher Bank der Variablenname gesucht werden muß.

Der Namensstack bietet Platz für ungefähr 1600 Label, wenn man davon ausgeht, daß die durchschnittliche Variablenlänge bei 5 Zeichen liegt. Auch der Overflow des Namensstacks wird mit einer Fehlermeldung (»too many names«) quittiert – nebenbei bemerkt, ist mir die Erzeugung dieses Fehlers bislang noch nicht gelungen.

Sollte es bei Ihnen irgendwann einmal notwendig werden, mehr Raum für die Variablennamen zur Verfügung stellen zu müssen, weil Sie obige Fehlermeldung erhalten, kann Ihnen der Pseudoop

```
».names adr«
```

weiterhelfen. Mit diesem Pseudoop verlegen Sie die Startadresse des Namensstacks an eine in »adr« angegebene Adresse innerhalb der RAM-Bank 1. Der Namensstack wächst von unten nach oben, Sie müssen also die untere Grenze des neuen Stackbereiches angeben. Der neue Stack unterliegt keiner Überwachung, deshalb sollten Sie mit dem Befehl gegebenenfalls etwas vorsichtig sein, um nicht andere wichtige Datenbereiche des Assemblers zu überschreiben. Sollte dieser Befehl überhaupt irgendwann einmal eingesetzt werden müssen, so empfehle ich als neue Startadresse die Adresse $2000, da dieser Bereich bis $6a00 speziell für solche und ähnliche Fälle freigehalten wurde. Beachten Sie, daß die Verlegung des Stacks an jedem beliebigen Ort des Quelltextes vorgenommen werden kann. Es wird günstig sein, soviel Labelnamen wie möglich in den ursprünglichen Namensstack schreiben zu lassen und dann erst umzuschalten. Den Standort von ».names« innerhalb des Quelltextes bestimmt man dabei am besten experimentell. Ein Beispiel für die Verwendung von ».names« möchte ich mir sparen.

3.3.6 Die Minimacs

Der Begriff »Minimacs« wurde von mir selbst erfunden, um damit eine für Assembler ebenfalls neuartige Einrichtung zu bezeichnen. Die Minimacs haben eine gewisse Ähnlichkeit mit den später besprochenen Makros, werden jedoch vom Assembler anders behandelt, weshalb sie auch mit Absicht getrennt von den Makros besprochen werden sollen – beim Top-Ass v1.0 gab es doch einige Mißverständnisse hinsichtlich der Minimacs.

Was leisten die Minimacs? Sie ersparen in erster Linie Tipparbeit. Während der Arbeit mit 6502-Maschinensprache fiel mir im Laufe der Zeit auf, daß einige Operationen aus mehreren Prozessorbefehlen geradezu extrem häufig gebraucht werden. Als Beispiele seien etwa folgende Fälle aufgeführt:

```
                lda adr
                ldx adr+1

oder            lda adr
                sta adr1
                lda adr+1
                sta adr1+1
```

Es gibt eine ganze Reihe solcher Befehlsfolgen, deren Programmierung praktisch nur stumpfsinniges Tippen einer immer gleichen Befehlsanordnung ist. Um diesen »Overhead« zu vermeiden, nimmt man normalerweise die später beschriebenen Makros zu Hilfe; es geht aber noch um einiges bequemer: Warum nicht besondere Pseudoops schaffen, die genau den benötigten Maschinenkode solcher Befehlsfolgen direkt erzeugen? Genau diese Pseudoops sind die Minimacs.

Insgesamt gibt es 32 Minimacs; der von ihnen erzeugte Maschinenkode und die Art, wie sie aufgerufen werden, sind im folgenden, thematisch geordnet, aufgeführt.

Minimacs zum Laden der Prozessorregister: Laden zweier Prozessorregister mit dem Inhalt zweier aufeinanderfolgender Speicherzellen bzw. mit Low- und Highbyte einer Konstanten.

```
.lax nn         =       lda nn
                        ldx nn+1

.lay nn         =       lda nn
                        ldy nn+1

.lxy nn         =       ldx nn
                        ldy nn+1

.liax nn        =       lda #<(nn)
                        ldx #>(nn)

.liay nn        =       lda #<(nn)
                        ldy #>(nn)

.lixy nn        =       ldx #<(nn)
                        ldy #>(nn)
```

Abspeichern der Prozessorregister: Abspeichern zweier Register in zwei aufeinanderfolgenden Speicherzellen.

```
.sax nn         =       sta nn
                        stx nn+1

.say nn         =       sta nn
                        sty nn+1
```

```
.sxy nn      =      stx nn
                    sty nn+1
```

Speicherbewegungen: Laden zweier aufeinanderfolgender Speicherzellen mit dem Inhalt zweier weiterer Speicherzellen oder mit einer Konstanten.

```
.mov nn,mm   =      lda nn
                    sta mm
                    lda nn+1
                    sta mm+1

.mvi nn,mm   =      lda #<(nn)
                    sta mm
                    lda #>(nn)
                    sta mm+1
```

Bitschiebeoperationen: Links- und Rechtsverschiebung zweier aufeinanderfolgender Speicherzellen mit Carry und ohne.

```
.shl nn      =      asl nn
                    rol nn+1

.slc nn      =      rol nn
                    rol nn+1

.shr nn      =      lsr nn+1
                    ror nn

.src nn      =      ror nn+1
                    ror nn
```

Inkrementieren und Dekrementieren: Inhalt zweier Speicherzellen um Eins erhöhen bzw. erniedrigen.

```
.icr nn      =      inc nn
                    bne lab
                    inc nn+1
                lab

.dcr nn      =      lda nn
                    bne lab
                    dec nn+1
                lab dec nn
```

Adreßarithmetik: Addieren und Subtrahieren zweier Adressen voneinander; Addieren und Subtrahieren von Konstanten zu oder von Adressen; Addieren des Akkumulators zu einer Adresse mit Übertrag.

```
.add nn,mm    =     clc
                    lda nn
                    adc mm
                    sta mm
                    lda nn+1
                    adc mm+1
                    sta mm+1

.adi nn,mm    =     clc
                    lda #<(nn)
                    adc mm
                    sta mm
                    lda #>(nn)
                    adc mm+1
                    sta mm+1

.adl nn       =     clc
                    adc nn
                    sta nn
                    bcc lab
                    inc nn+1
                lab

.sub nn,mm    =     sec
                    lda mm
                    sbc nn
                    sta mm
                    lda mm+1
                    sbc nn+1
                    sta mm+1

.sbi nn,mm    =     sec
                    lda mm
                    sbc #<(nn)
                    sta mm
                    lda mm+1
                    sbc #>(nn)
                    sta mm+1
```

Vergleiche: Vergleich einer Adresse mit zwei Byte in einer weiteren Adresse oder mit einer Konstanten.

```
.cpr nn,mm    =     lda mm+1
                    cmp nn+1
                    bne lab
                    lda mm
```

```
                        cmp nn
                    lab

.cpi nn,mm      =       lda mm+1
                        cmp #>(nn)
                        bne lab
                        lda mm
                        cmp #<(nn)
                    lab
```

Das Ergebnis der Vergleiche ist wie bei normalen Vergleichen mit »cmp« abzufragen. Beispiel:

```
.cpi grenze,pointer   ; Hat pointer Grenzwert erreicht?
bcc label             ; Nein, ist noch kleiner
```

Alle Adressen bzw. Konstanten »nn« und »mm« in den Beschreibungen sind beliebig, insbesondere können Sie auch in der Zeropage liegen. Die Prozeduren, die den Maschinenkode der Minimacs erzeugen, berücksichtigen die unterschiedlichen Befehlslängen, die sich aus einer Zeropageadressierung ergeben, automatisch, folglich geht dem Programmierer kein Byte Speicherplatz durch die Verwendung der Minimacs verloren.

Bedingte Sprünge:

In der Programmierpraxis kommt es recht häufig vor, daß Schleifen mit Branches zu lang werden, so daß der Assembler die Fehlermeldung »branch too far« ausgibt. Sie erinnern sich, daß die Sprungweite der Branchbefehle auf ca. 128 Byte vor- und rückwärts begrenzt ist. Normalerweise würde man nun so vorgehen, daß man den einzelnen Branch durch einen dazu komplementären und ein zusätzliches »jmp« zum Schleifenanfang ersetzt, d. h., man würde eine Quelltextzeile einfügen. Beispiel:

```
loop        ...
            ...   ;   mehr als 128 Byte
            ...
            beq loop
```

wird so zu:

```
loop        ...
            ...   ;   mehr als 128 Byte
            ...
            bne lab
            jmp loop
lab         ...
```

Auch diese Arbeit können uns aber einige Minimacs erleichtern, die genau die passenden Kombinationen aus Branch und Jump erzeugen:

```
.jeq nn    =    bne *+5
                jmp nn

.jne nn    =    beq *+5
                jmp nn

.jcc nn    =    bcs *+5
                jmp nn

.jcs nn    =    bcc *+5
                jmp nn

.jmi nn    =    bpl *+5
                jmp nn

.jpl nn    =    bmi *+5
                jmp nn

.jvc nn    =    bvs *+5
                jmp nn

.jvs nn    =    bvc *+5
                jmp nn
```

Wie Sie sehen, entspricht die Schreibweise dieser Minimacs ziemlich der der Branches, die einen Sprung über die kürzere Distanz durchführen. Im Fall, daß Sie die Fehlermeldung »branch too far« erhalten, genügt es, das »b« des Branches in der bei der Fehlermeldung angezeigten Zeile mit einem ».j« zu überschreiben und die Assemblierung kann sofort wieder gestartet werden. Ganz nebenbei ergibt sich als weiterer Effekt, daß die Quelltexte durch Verwendung gerade der Sprung-Minimacs erheblich an Transparenz gewinnen können. Schauen Sie sich das obige Beispiel noch einmal mit einem Minimac an:

```
loop        ...
            ...   ;   mehr als 128 Byte
            ...
            .jeq loop
```

Diese Fassung ist sicher besser lesbar und übersichtlicher als die obige, in der Branch und Jump explizit angegeben sind.

Da Minimacs für alle Top-Ass-Anwender gleich sind, ergibt sich ein hoher Grad an Portabilität, wenn ich einmal davon ausgehe, daß der Top-Ass recht verbreitet ist. Was aber für die eigene Programmentwicklung noch wichtiger ist, ist die Einsparung an Programmierzeit bei der Entwicklung von Maschinenprogrammen. Es ist doch

recht häufig der Fall, daß man ein neues Unterprogramm erst einmal in den Grundzügen ausprobieren will, um zu sehen, ob die Lösung, die man sich ausgedacht hat, auch wirklich praktikabel ist. Solche Unterprogramme sollte man unbedingt unter Zuhilfenahme der Minimacs und der später noch erläuterten »Strukturpseudos« schreiben. Entspricht die Lösung Ihren Vorstellungen, können Sie das Programm dann immer noch von Hand optimieren. Als Beispiel hierfür möchte ich ein kleines Blockverschiebe-Programm anführen:

```
100     -.base $c00
110     -.define von   =  $1300
120     -.define bis   =  $1415
130     -.define nach  =  $1456
140     -.define pn1   =  $24
150     -.define pn2   =  $26
160     -;
170     -shift     .mvi von,pn1
180     -          .mvi nach,pn2
190     -          ldy #0
200     -loop      lda (pn1),y
210     -          sta (pn2),y
220     -          .icr pn1
230     -          .icr pn2
240     -          .cpi bis+1,pn1
250     -          bcc loop
260     -          rts
```

Dieses kleine Programm führt eine Verschiebung des Adreßbereiches $1300 bis $1415 nach $1456 durch. Vollziehen Sie den von den Minimacs erzeugten Kode einmal selbst zur Übung nach. Ich glaube kaum, daß man selbst ein so einfaches Programmstückchen ohne die Minimacs in vergleichbarer Zeit schreiben könnte. Beachten Sie auch, daß die Minimac-Parameter, wie immer, arithmetische Ausdrücke sein dürfen!

3.3.7 Strukturen

Sie werden sicher bemerkt haben, daß wir in der Zwischenzeit schon längst Gefilde betreten haben, die sozusagen zu den »höheren Weihen« der Assemblerprogrammierung gehören. Angefangen mit den Minimacs sind alle folgenden hier beschriebenen Features das, was letztlich den Unterschied zwischen einem Hobbyprogramm und einem professionellen Entwicklungspaket ausmacht.

Dazu gehört auch die direkte Unterstützung der strukturierten Programmierung durch den ASE-Assembler: Wie Sie sicher wissen, bietet die strukturierte Programmierung

einige Vorteile hinsichtlich der Wartbarkeit und Überschaubarkeit von Programmen. Bisher hat sie sich jedoch nur im Bereich der höheren Programmiersprachen durchsetzen können – die bekanntesten Beispiele sind die Programmiersprachen C, Pascal und Modula. Da strukturiertes Programmieren aber eine probate Methode gerade auch für eine schnelle Programmentwicklung darstellt, habe ich mich schon in eigenem Interesse entschlossen, einen notwendigen und praktischen Subset von Pseudoops einzuführen, die der Strukturierung dienen.

Die Struktur-Pseudoops gehen im Prinzip ganz ähnlich vor wie die Pseudoops der Minimacs: Sie erzeugen direkt Maschinenkode. Wofür sie Kode erzeugen, wird deutlich, wenn wir betrachten, welche Programmstrukturen wir für einen minimalen Set zur strukturierten Programmierung benötigen (in Klammern sind einige Pascal-Entsprechnungen angegeben):

- Unterprogramme (procedure)
- Schleifenkonstruktionen (repeat/until)
- Fallunterscheidungen (if/else)

Die Verwendung von Unterprogrammen unterstützt der Prozessor selbst mit dem Befehl »jsr«, für diesen Teil brauchen wir also keine eigene Konstruktion.

Was der Prozessor nicht kennt, sind Schleifen und Fallunterscheidungen. Diese muß sich der Programmierer selbst unter Zuhilfenahme von Verzweigungsbefehlen zusammenbasteln, wenn er nicht eine andere Lösung zur Verfügung hat – wie die Struktur-Pseudoops. Sowohl Schleifen als auch Fallunterscheidungen sind abhängig von Bedingungen. Im Fall der höheren Programmiersprachen sind diese Bedingungen durch arithmetische oder logische Ausdrücke angegeben – eine Schleife wird solange durchlaufen, bis ein Ausdruck logisch wahr ist, eine Fallunterscheidung verzweigt in Abhängigkeit von einem logischen Ausdruck. Diese Vorgehensweise kennen Sie auch von Basic: »if ausdruck then«.

Will man ähnliche Konstruktionen wie in den Hochsprachen auch auf Assemblerebene einführen, stellt sich zunächst die Frage, was auf der Ebene des Prozessors eine Entsprechung zu den Bedingungen darstellt. Diese Frage ist schnell geklärt: Der Prozessor kennt nur eine begrenzte Zahl von Bedingungen, die er direkt versteht – dies sind die bekannten Flags des Prozessor-Statusregisters. Auf diese Bedingungen muß man beim Entwurf von Schleifen und Fallunterscheidungen zunächst einmal zurückgreifen. Auf die genaue Anwendung der Bedingungen in den Pseudoops zur strukturierten Programmierung wird etwas später noch eingegangen; hier soll eine Auflistung der Schreibkonventionen für die Bedingungen innerhalb der Pseudoops, sofern ein Pseudo eine Bedingung in seiner Parameterliste erwartet, reichen:

e	=	equal (Z-Flag gesetzt)
n	=	not equal (Z-Flag gelöscht)
c	=	carry clear (C-Flag gelöscht)
s	=	carry set (C-Flag gesetzt)

m	=	minus (N-Flag gesetzt)
p	=	plus (N-Flag gelöscht)
v	=	overflow clear (V-Flag gelöscht)
o	=	overflow set (V-Flag gesetzt)

Jedem der Buchstaben, die für eine Bedingung stehen, darf ein beliebiger weiterer Text folgen; d. h. es ist zulässig, beispielsweise für die Bedingung »e« auch ausgeschrieben »equal« zu schreiben. Vor jeder Bedingung wird **immer** ein Doppelkreuz erwartet, die Normalform für eine Bedingung sieht also etwa wie »#e« aus. Das Doppelkreuz soll lediglich die Lesbarkeit des Textes erhöhen, es darf aber nicht weggelassen werden, da sonst eine Fehlermeldung ausgegeben wird.

Wie die Minimacs sind auch die Pseudoops zur Herstellung von Programmstrukturen in der Hauptsache zu dem Zweck gedacht, eine schnelle Programmentwicklung zu ermöglichen. Steht das Programm einmal in seiner endgültigen Form, kann man anschließend eine Optimierung durchführen, sofern eine solche notwendig sein sollte, was bei einigen Strukturen nicht einmal zutrifft.

3.3.7.1 Schleifenkonstruktionen

Der ASE kennt zwei strukturierte Schleifenkonstruktionen:

```
.repeat/ .until
.do/ .loop
```

Die beiden Konstrukte sollen nacheinander etwas ausführlicher betrachtet werden, da auch dieses Feature des ASE-Assemblers von einigen Anwendern der Top-Ass-Programme der Version 1.0 manchmal falsch interpretiert wurde.

3.3.7.1.1 Die Repeatschleife

Die Repeatschleife ist die am häufigsten verwendete strukturierte Schleifenkonstruktion des ASE. Sie wird durch vier Pseudoops gesteuert:

.repeat	(Schleifenanfang)
.during #b	(Ausgang aus der Repeatschleife)
.unless #b	(Ausgang aus der Repeatschleife)
.until #b	(Schleifenende)

Als Bedingungen »#b« sind die weiter oben beschriebenen einzusetzen. Die Wirkung der Repeatschleife sei an einem Beispiel erläutert für diejenigen, die diese Schleifenart nicht kennen:

```
100   -.base $c00
101   -;
```

```
102   -.define getin = $ffe4
110   -;
105   -.repeat
120   -.  repeat
130   -           jsr getin
140   -           tax
150   -.  until #n
160   -           cmp #"a"
170   -.  unless#e
180   -           cmp #"b"
190   -.until #e
200   -           rts
```

Dieses Beispiel könnten Sie so assemblieren lassen, wie es hier aufgeführt ist. Frage: Welcher Maschinenkode wird bei der Assemblierung erzeugt? Diese Frage möchte ich durch ein Disassemblerlisting beantworten, bevor die Rolle der Pseudoops erläutert wird:

```
0c00  20   e4  ff      jsr   $ffe4
0c03  aa               tax
0c04  f0   fa          beq   $c00
0c06  c9   41          cmp   #"a"
0c08  d0   03          bne   $c09
0c0a  4c   0f  0c      jmp   $c0f
0c0d  c9   42          cmp   #"b"
0c0f  d0   ef          bne   $c00
0c0f  60               rts
```

Das erste, was wir noch einmal registrieren, ist, daß die Struktur-Pseudoops keine irgendwie gestalteten Anweisungen sind, die auf Quelltextebene arbeiten. Sie bewirken also nicht, daß die zwischen ».repeat« und ».until« stehenden Zeilen in einer Schleife assembliert werden, sondern daß die entsprechende Schleifenstruktur im Maschinenprogramm erscheint!

Vergleichen wir jetzt die äußere Schleife ».repeat/.until #e« mit dem Maschinenkode: Sie wird durch eine Befehlsfolge »marke/bne marke«, also praktisch im Disassemblerlisting nur durch den Befehl »bne $c00« an der Adresse $c0f, übersetzt. Das ».repeat« setzt intern eine entsprechende Marke, das ».until« erzeugt einen Sprungbefehl zu der Marke. Die Schleife ».repeat/.until #e« erzeugt damit insgesamt einen Maschinenkode, dessen Zweck man in Worten so beschreiben könnte: »Wiederhole die zwischen ».repeat/marke« und ».until #/Sprung zur Marke« liegenden Prozessorbefehle solange, bis die hinter dem Doppelkreuz stehende Bedingung erfüllt ist.« Wie Sie an der Schachtelung der beiden Strukturen ».repeat/.until« im obigen Quelltext erkennen können, berücksichtigt die Kode-

erzeugung des ».until« die Schachtelungstiefe. Jeder Sprung, der von einem ».until« erzeugt wird, führt genau zu der Adresse, die durch das zum ».until« gehörende ».repeat« gesetzt wurde. Die theoretische maximale Schachtelungstiefe beträgt über 80; theoretisch wird sie wohl auch bleiben, denn bei einer derartigen Verschachtelung ist eine Übersichtlichkeit des Quelltextes, wie sie durch die Pseudoops zur strukturierten Programmierung erreicht werden soll, wohl nicht mehr gegeben.

Die Anweisung ».unless #« ist ein Ausgang aus der Repeatschleife, wie schon oben erwähnt wurde. Im Disassemblerlisting können Sie die Übersetzung dieses Pseudoops nachvollziehen: ».unless #e« wird zu »bne *+5/jmp exit«, wobei »exit« die Adresse des ».until« der innersten Repeatschleife ist, innerhalb derer ».unless« verwendet wird; die Ausgänge aus der Repeatschleife führen also immer nur eine Schleifenebene nach außen. Die Kombination ».repeat/.unless/.until« könnte man übersetzen: »Wiederhole (repeat), bis (until), wenn nicht vorher (unless).« Der Ausgang ».during« wird ganz analog zu ».unless« behandelt, die Kombination ».repeat/.during/.until« ist zu übesetzen mit: »Wiederhole (repeat), bis (until), solange.« Jede Anweisung ».unless« kann in einen Befehl ».during« umgewandelt werden und umgekehrt, indem man einfach die komplementäre Bedingung einsetzt; »during #e« ist somit absolut gleichbedeutend zu »unless #n«, dasselbe gilt für »unless #c« und »during #s«. Innerhalb jeder Repeatschleife können beliebig viele Ausgänge verwendet werden, die beiden Ausgangstypen können gemischt werden.

Interessant für die Programmentwicklung: Solange Sie keinen der beiden Ausgänge der Repeatschleife anwenden, ist die Schleife genauso lang, als wenn Sie auf herkömmliche Art programmieren, indem Sie Label und einen Sprung zum Label verwenden. Die Kodeerzeugung des ASE ist außerdem »intelligent« genug, um zu erkennen, wenn der Schleifenkörper – die Anweisungen zwischen ».repeat« und ».until« – länger als die maximale Sprungweite für einen Branch ist. In diesem Fall wird statt des Branches eine Kombination aus Branch und Jump als Kode erzeugt, wie sie bei den Minimacs für die bedingten Sprünge schon einmal gezeigt wurde. Die beiden Ausgänge aus der Schleife werden immer durch eine Branch/Jump-Kombination übersetzt. Diese Tatsache ist technisch nicht anders zu lösen, wenn die Repeatschleife allgemeingültig bleiben soll. Es ist vorgesehen, dieses Manko in späteren Versionen des Assemblers noch auszuräumen, indem weitere Versionen der Pseudoops zur strukturierten Programmierung eingeführt werden.

Zum Abschluß noch einmal die beiden möglichen Übersetzungen der Repeatschleife – je nach Größe des Schleifenkörpers:

```
.repeat          =    marke
    ...                          ...
.unless #e                       bne *+5
    ...                          jmp exit
    ...                          ...
.during #e                       beq *+5
```

```
        ...                         jmp exit
        ...                         ...
    .until #e                       bne marke
                            exit    ...
```

für Schleifenkörper kleiner als 128 Byte, und

```
    .repeat         =   marke
        ...                         ...
    .unless #e                      bne *+5
        ...                         jmp exit
        ...                         ...
    .during #e                      beq *+5
        ...                         jmp exit
        ...                         ...
    .until #e                       beq *+5
                                    jmp marke
                            exit    ...
```

für den Fall, daß der Schleifenkörper für einen Branch zu groß ist.

3.3.7.1.2 Die Do-Schleife

Die Do-Schleife stellt die Realisierung einer unbedingten Schleife dar. Wieder steuern vier Pseudoops die Anweisung:

```
.do         (Schleifenstart)
.while #b   (Ausgang 1)
.exloop #b  (Ausgang 2)
.loop       (Schleifenende)
```

Nach den vorangegangenen eingehenden Erläuterungen der Repeatschleife sollte hier eine kürzere Abhandlung genügen: ».do« bezeichnet den Schleifenstart, indem der Assembler eine interne Marke setzt – die von den Pseudoops zur strukturierten Programmierung gesetzten Marken werden übrigens nicht im Variablenfeld, sondern in einem getrennten Stack verwaltet und können deswegen nicht durch die Dump- und Rechenbefehle des Editors nachgefragt werden. ».loop« erzeugt die Befehlsfolge zum unbedingten Sprung zu der Marke bzw. dem Label des zum ».loop« gehörenden ».do«. Jeweils ein Paar aus ».do« und ».loop« bildet eine vollständige Do-Schleife; das Vorhandensein eines vollständigen Paares aus diesen beiden Anweisungen wird vom Assembler wie bei ».repeat/.until« überwacht, wobei gegebenenfalls die Fehlermeldung »illegal structure« ausgegeben wird, falls ein Teil fehlt. Die beiden Ausgänge aus der unbedingten Schleife werden wie die Ausgänge der Repeatschleife durch eine Kombination aus Branch und Jump übersetzt. Ihre

Bedeutung dürfte schon aus der Namensgebung folgen: »while #b« erzeugt einen bedingten Sprung aus der Schleife heraus, der dann ausgeführt wird, wenn die Bedingung »b« nicht mehr erfüllt ist; ».exloop #b« erzeugt einen Sprung, der ausgeführt wird, wenn »b« erfüllt ist.

Die Übersetzung der Do-Schleife:

```
.do             =    marke
      ...                        ...
.while #s                        bcs *+5
      ...                        jmp exit
      ...                        ...
.exloop #s                       bcc *+5
      ...                        jmp exit
      ...                        ...
.loop                            jmp marke
                          exit   ...
```

Die Ausgänge aus der Do-Schleife können wieder in beliebiger Zahl verwendet werden; die Schleifenstruktur kann wie bei ».repeat« bis zu einer maximalen Tiefe von über 80 geschachtelt werden. Die reine Do-Schleife ohne ».while« und ».exloop« ist nicht länger, als wenn man die unbedingte Schleife mit Labeln programmieren würde – davon abgesehen ist sie aber wesentlich übersichtlicher.

Verwechseln Sie die Ausgänge der Repeat- und der Do-Schleife, erhalten Sie eine Fehlermeldung »illegal structure«.

3.3.7.2 Fallunterscheidungen

Der ASE-Assembler unterstützt zwei Arten Fallunterscheidungen durch Pseudoops zur strukturierten Programmierung:

3.3.7.2.1 ».if/.else/.enif«

Besonders die Konstruktion ».if/.else/.enif« hat den Anwendern des Top-Ass V1.0 offenbar Schwierigkeiten gemacht. Deshalb zum letzten Mal: Auch diese Anweisungsfolge bewirkt keine Aktion des Assemblers innerhalb des Quelltextes in Form einer bedingten Assemblierung oder wie auch immer. Statt dessen erzeugen die Pseudoops Maschinenkode. Dieser direkt erzeugte Maschinenkode spiegelt die angegebene Programmstruktur – hier die Fallunterscheidung ».if/.else/.enif« – im erzeugten Maschinenprogramm wieder.

Die Bedeutung der If-Konstruktion darf ich wohl als bekannt voraussetzen. Das einzige, was das Basic 7.0 nicht als Entsprechung enthält, ist die Anweisung ».enif«. Diese ist beim ASE unbedingt erforderlich, um das Ende der If-Konstruktion zu

kennzeichnen. Wenn Sie sich die Kodeerzeugung der Pseudoops ansehen, wird deutlich, wie dies gemeint ist:

```
.if #v          =               bvc *+5
      ...                       jmp alter
      ...                       ...
      ...                       jmp ende
.else                 alter     ...
      ...                       ...
      ...                       ...
      ...                       ...
.enif                 ende
```

Sie sehen, der Kodeteil zwischen ».if« und ».else« wird nur dann angesprungen, wenn die Bedingung, die hinter ».if« angegeben ist, erfüllt ist. Das Kodesegment wird abgeschlossen durch ein »jmp« aus der If-Konstruktion heraus, damit der alternative Kodeteil nicht ausgeführt wird. Dieser Teil wird angesprungen, wenn die Bedingung hinter ».if« nicht erfüllt ist.

Die If-Konstruktion kann auch unvollständig – ohne die Angabe einer Alternative mit ».else« – verwendet werden:

```
.if #n          =               bne *+5
      ...                       jmp ende
      ...                       ...
      ...                       ...
.enif                 ende      ...
```

Die If-Struktur kann wie die Schleifenstrukturen auch wieder verschachtelt werden. Die maximale Schachtelungstiefe liegt bei über 80. Das für Compiler übliche Problem mit dem »If« stellt sich beim ASE wegen der Verwendung des ».enif« als Zeichen des Endes der Konstruktion nicht; nehmen wir folgende Verschachtelung an:

```
.if #e                          if
.if #n                          if
      ...                       else
.else
      ...
.enif
.enif
```

Links sehen Sie eine Verschachtelung, wie sie beim ASE geschrieben würde, rechts die Entsprechung in Pascal, das kein besonderes Schlüsselwort für das Ende der If-Struktur kennt. Dadurch wird aus dem Programmtext nicht eindeutig ersichtlich, zu welchem der beiden »ifs« das »else« gehört. Diese Tatsache habe ich noch dadurch

zu unterstreichen versucht, daß ich keine Einrückungen vorgenommen habe. Der Pascal-Compiler – und andere Compiler mit dem gleichen Problem – geht daher immer so vor, daß das »else« zum letzten »if« geschlagen wird, falls keine andere Klammerung der Zugehörigkeit explizit vorgenommen wurde (durch die Pascal-Schlüsselwörter »begin« und »end«). Beim ASE tritt dieses Problem gar nicht erst auf: Das ».else« gehört logisch immer zum letzten ».if«, das noch nicht durch ».enif« abgeschlossen wurde.

In einer Hinsicht ist die If-Konstruktion des ASE von gewöhnlichen If-Strukturen anderer Compiler verschieden: Der Assembler überprüft ausdrücklich nicht, ob innerhalb eines ».if« mehrere Alternativen mit ».else« verwendet werden. Diese Eigenschaft sollte man jedoch nur im Notfall anwenden, da sie einer strukturierten Programmierweise doch ziemlich widerspricht. Auf jeden Fall muß aber einmal gezeigt werden, welcher Kode bei einer solchen Konstruktion entsteht:

```
100     -.base $c00
110     -;
120     -.if #e
130     -           lda #0
14ß     -.else
150     -           lda #1
160     -.else
170     -           lda #2
180     -.enif
```

wird zu

```
0c00 f0 03          beq  $c05
0c02 4c 0a 0c       jmp  $c0a
0c05 a9 00          lda  #0
0c07 4c 11 0c       jmp  $c11
0c0a a9 01          lda  #1
0c0c 4c 11 0c       jmp  $c11
0c0f a9 02          lda  #2
0c11 ...            ...
```

Wie Sie sehen, erzeugt das zweite ».else« einen »versteckten« Kodebereich, der im Rahmen der If-Konstruktion nicht angesprungen wird. Das könnte uns aber natürlich nicht daran hindern, dieses Kodestückchen mit Hilfe eines Labels zu adressieren und anzuspringen.

3.3.7.2.2 Die Case-Konstruktion

Die Case-Konstruktion ist die einzige Programmstruktur des ASE, die nicht auf der Beschreibung von Bedingungen in Form der Angabe von Prozessorflags beruht. Statt dessen wird der aktuell im Akkumulator befindliche Wert mit einer Reihe von Werten verglichen, die in einzelnen Alternativen der Struktur angegeben sind. Für jede Alternative wird dann ein bestimmtes Stück Programmkode durchlaufen.

Die Case-Struktur wird von vier Pseudoops gebildet:

```
.case of exp
.of exp
.otherwise
.caseend
```

».case of« leitet die Konstruktion ein; der Akkumulator wird mit dem Wert verglichen, der im arithmetischen Ausdruck »exp« angegeben ist. Verläuft der Vergleich positiv – ist also der Akkumulator gleich dem Ausdruck – wird der Maschinenkode hinter ».case of« bis zum nächsten ».of«, ».otherwise« oder ».caseend«« ausgeführt. Die Alternative wird mit einem Sprung aus der Case-Konstruktion heraus beendet.

».of« arbeitet in exakt der gleichen Weise wie ».case of«, mit dem offensichtlichen Unterschied, daß ».of« intern keine neue Case-Konstruktion einleitet. Innerhalb einer Case-Konstruktion können beliebig viele ».of«'s verwendet werden.

».otherwise« bildet die Alternative, falls der Akkumulator keinen der in den ».of«'s oder im ».case of« angegebenen Werte enthält. ».otherwise« darf fehlen, falls innerhalb der Case-Struktur nur für bestimmte Fälle eine Aktion erfolgen soll.

».caseend« beendet die Case-Struktur und setzt intern einen Label, der während der Kodeerzeugung benützt wird.

Der bei ».case of« erzeugte Kode noch einmal in der Übersicht:

```
.case of "a"    =               cmp #"a"
      ...                       beq *+5
      ...                       jmp of1
      ...                       ...
      ...                       jmp ende

.of "b"               of1       cmp #"b"
      ...                       beq *+5
      ...                       jmp of2
      ...                       ...
      ...                       jmp ende
```

```
.of   "c"          of2      cmp #"c"
      ...                   beq *+5
      ...                   jmp other
      ...                   ...
      ...                   jmp ende
.otherwise         other
      ...                   ...
.caseend           ende
```

Eine praktische Anwendung bei der schnellen Programmentwicklung wäre beispielsweise der Aufbau eines Sprungverteilers wie des folgenden:

```
100   -.base $c00
101   -.define getin = $ffe4
110   -;
115   -.do
120   -.   repeat
130   -          jsr getin
140   -          tax
150   -.   until #n
160   -.   case of "a"
170   -;
180   -.     of "b"
190   -;
191   -.     of "x"
192   -.      exloop #e
200   -.   caseend
210   -.loop
220   -          rts
```

Dieses kleine Programm ist eine Endlosschleife, in der aufgrund einer Tastatureingabe zu verschiedenen Unterprogrammen verzweigt werden könnte – die entsprechenden Unterprogrammbefehle müßten in die Alternativen der Case-Struktur eingetragen werden. Gibt der Benutzer ein »x« ein, wird die äußerste Schleife mit ».exloop« verlassen und die Eingabeschleife ist beendet.

3.3.7.3 Beispielprogramm zur strukturierten Programmierung

Um die Leistungsfähigkeit der Pseudoops zur strukturierten Programmierung noch einmal zu demonstrieren, hier ein etwas längeres Beispielprogramm. Es handelt sich dabei um die Realisierung einer Quicksort-Routine; eine Routine dieser Art wird auch vom Assembler benutzt, um das Variablenfeld bei einem Dump zu sortieren. Das Beispiel kann natürlich kein vollständiges Programm sein, da hierfür irgend etwas

vorhanden sein müßte, was zu sortieren wäre. Immerhin können Sie das Programm als Muster für den verwendeten Algorithmus verwenden sowie die Realisierung komplexerer Algorithmen gerade in Assembler verfolgen.

```
1000 -;=====================================================
1001 -; QUICKSORT - ROUTINE
1002 -;=====================================================
1003 -;
1004 -unten     .word 0       ; untere Grenze Sortierfeld
1005 -oben      .word 0       ; obere Grenze Sortierfeld
1006 -zsp       .byte 0       ; Zwischenspeicher
1007 -;
1008 -.define lablen = xxx   ; Länge eines Elements im
1009 -;                          Sortierfeld
1010 -.define usp = $xx       ; Zeropage-Stackpointer
1011 -;                          2 Byte
1012 -.define un = $xx        ; laufende Hilfpointer für
1013 -.define ob = $xx        ;    Quicksort
1014 -;
1015 -.define conf = xx       ; Konfiguration des Sortier-
1016 -;                          feldes
1017 -xber    .space of lablen
1018 -;                        Bereich zum Tauschen von
1019 -;                          Elementen
1020 -.define base = $xxxx    ; Bereich für Userstack
1021 -;
1022 -;
1900 -quick   jsr stackinit   ; Userstack Init
1910 -;
2000 -.do
2010 -        .mov unten,un   ; untere + obere Grenze des
2020 -        .mov oben,ob    ; Sortierbereiches in Pointer
2030 -        jmp mq          ; Einsprung
2040 -. do
2050 -.  repeat
2060 -        .adi lablen,un ; Suche von unten erstes
2070 -mq      jsr uneqob?     ; kleineres Element
2080 -        beq mq2         ;    bis man sich in der Mitte
2090 -        jsr cmpelem ;           trifft
2100 -.  until #c
2110 -        jsr xchge       ; und tausche aus
2120 -.  repeat
2130 -        .sbi lablen,ob ; suche von oben
```

```
2140 -         jsr uneqob?     ; erstes kleineres Element
2150 -         beq mq2         ;    bis man sich in der Mitte
2160 -         jsr cmpelem     ;         trifft
2170 -.  until #c
2180 -         jsr xchge       ; und tausche
2190 -.  loop
2200 -;
2210 -mq2      .adi lablen,ob  ; spalte Bereich in 2 Teilb.
2220 -         .cpr oben,ob    ; wenn's geht
2230 -. if #c
2240 -         jsr qupush      ; und merke einen auf dem Us.
2250 -. enif
2260 -         .sbi lablen,un  ; Ist noch ein Bereich übrig
2270 -         .cpr un,unten   ; zum Sortieren?
2280 -. if #s                  ; Nein:
2290 -         jsr stckempty?  ; die auf dem Stack gemerkten
2300 -.  if #e                 ; alle sortiert, dann bin ich
2310 -         rts             ; fertig
2320 -.  else
2330 -         jsr qupull      ; sonst hole Bereich vom St.
2340 -.  enif                  ; und sortiere ihn
2350 -. else
2360 -         .mov un, oben   ; war noch so einer: sortiere
2370 -. enif
2380 -.loop
2390 -;
2391 -uneqob? .cpr un,ob       ;
2392 -         rts
2393 -;
2400 -;=================================================
2410 -; STACK-ROUTINEN SPEZ. FÜR QUICKSORT
2420 -;=================================================
2430 -;
2440 -stckinit .mvi base,ust   ; Initialisierung des User-
2450 -          rts            ; stacks
2460 -;
2470 -stckemp? .cpi base,ust   ; Ist der Stack leer?
2480 -          rts
2490 -;
2500 -qupush    ldy #0          ; <ob,ob+1> und <oben,oben+1>
2510 -          lda ob+1        ; auf den Userstack legen
2520 -          jsr stausty     ;
2530 -          iny             ;
```

```
2540 -          lda ob          ;
2550 -          jsr stausty     ;
2560 -          iny             ;
2570 -          lda oben+1      ;
2580 -          jsr stausty     ;
2590 -          iny
2600 -          lda oben        ;
2610 -          jsr stausty     ;
2640 -          .adi 4,ust      ;
2650 -          rts             ;
2660 -;
2670 -qupull     .sbi 4,ust      ; <unten,unten+1> und
2680 -          ldy #0          ; <oben,oben+1> vom Stack
2690 -          jsr ldausty     ; holen
2700 -          sta unten+1     ;
2710 -          iny
2720 -          jsr ldausty     ;
2730 -          sta unten       ;
2740 -          iny
2750 -          jsr ldausty     ;
2760 -          sta oben+1      ;
2770 -          iny
2780 -          jsr ldausty     ;
2790 -          sta oben        ;
2800 -          rts
2900 -;
2910 -ldausty  lda #ust         ; Laden eines Bytes aus der
2920 -          sta fetvec      ; Konfiguration conf über den
2930 -          ldx #conf       ; Zeropagepointer ust
2940 -          jmp fetch       ;
2950 -;
2960 -stausty  ldx #ust         ; Speichern eines Bytes in die
2970 -          stx stavec      ; Konfiguration conf über den
2980 -          ldx #conf       ; Zeropagepointer ust
2990 -          jmp stash       ;
3000 -;
3010 -;=================================================
3020 -; ELEMENTABHÄNGIGE ROUTINEN:
3030 -;=================================================
3040 -;
3041 -; Zwei Elemente vertauschen
3042 -;
3050 -xchge     ldy #lablen-1
```

```
3060  -;
3070  -. repeat
3080  -          jsr ldauny    ;  un in Zwischenspeicher
3090  -          sta xber,y    ;  transportieren
3100  -          dey           ;
3110  -. until #m
3120  -. repeat
3130  -          iny
3140  -          jsr ldaoby    ;  ob nach un bringen
3150  -          jsr stauny    ;
3160  -          cpy #lablen-1;
3170  -. until #e
3180  -. repeat
3190  -          lda xber,y    ;  Zwischenspeicher nach ob
3200  -          jsr staoby    ;
3210  -          dey
3220  -.until #m
3230  -          rts
3240  -;
3250  -; Vergleich zweier Elemente nach dem Alphabet
3260  -;
3270  -cmpelem ldy #$ff        ;  Startwert wg. iny
3271  -;
3272  -. repeat
3273  -          iny           ;  alle Zeichen verglichen?
3274  -          cpy #lablen-1;
3275  -. unless #s            ;  ja: -> Ende
3280  -          jsr ldaoby    ;  Zeichen von ob
3290  -          sta zsp       ;  zwischenspeichern
3300  -          jsr ldauny    ;  Zeichen von un holen
3310  -          cmp zsp       ;  und vergleichen
3320  -. until #n             ;  bis zum 1. Unterschied
3330  -          rts           ;  liefere Flags zurück
3340  -;
3350  -ldauny    lda #un       ;  Laden eines Bytes aus der
3360  -          .byte $2c     ;  Konfiguration conf über die
3370  -ldaoby    lda #ob       ;  Zeropagepointer un und ob
3380  -          sta fetvec
3390  -          ldx #conf
3400  -          jmp fetch
3410  -;
3420  -stauny    ldx #un       ;  Speichern eines Bytes in die
3430  -          .byte $2c     ;  Konfiguration conf über die
```

```
3440  -staoby   ldx #ob       ;  Zeropagepointer un und ob
3450  -         stx stavec
3460  -         jmp stash
```

Erläuterungen:

Um die obige Quicksortroutine unverändert anwenden zu können, müssen die folgenden Gegebenheiten vorliegen:

(1) Das zu sortierende Feld besteht aus gleich langen Elementen der Länge »lablen«. Das Feld liegt in der Speicherkonfiguration, die durch »conf« definiert ist; dies könnte z. B. die RAM-Bank 0 (conf = $3f) oder die RAM-Bank 1 (conf = $7f) sein. Die beiden Speicheradressen »unten« und »oben« zeigen auf den Beginn und das Ende des Sortierfeldes.

(2) In der gleichen Konfiguration wie das Sortierfeld wird durch die obige Routine ein sogenannter Userstack gehalten; für diesen Stack muß natürlich Raum vorhanden sein. Der Stack arbeitet nach dem gleichen Prinzip, das auch der Prozessorstack benutzt: Der zuletzt gemachte Eintrag wird zuerst wieder gelesen (last in, first out). Userstacks werden recht häufig benutzt, um rekursive Lösungen zu realisieren, da der Prozessorstack der 6502-CPU für umfangreiche Speicherungen wegen seiner geringen Abmessungen nicht geeignet ist.

Die Sortierung selbst erfolgt nach dem bekannten Quicksort-Algorithmus, der in Fachbüchern zur Informatik oder auch in einigen Fachzeitschriften bei Bedarf nachzulesen ist. Im vorliegenden Fall wird alphabetisch sortiert. Das Sortierkriterium kann jedoch durch einfache Verwendung einer anderen Routine »cmpelem« beliebig gewechselt werden. Liefert die Vergleichsroutine für zwei Einträge des Sortierfeldes ein gelöschtes Carry, so werden die Elemente des Sortierfeldes getauscht, d. h. das Element »un« tauscht mit »ob« seinen Platz. Indem man den Vergleich umkehrt, so daß die Prozessorflags beim Vergleich komplementär gesetzt werden, erhält man eine umgekehrte Sortierreihenfolge: aus einer Sortierung mit aufsteigender Elementenfolge wird eine solche mit absteigender Folge.

Auch der Fall, daß ein Feld von Elementen sortiert werden muß, die nicht alle die gleiche Länge haben, kann mit der Routine erfaßt werden: Wieder ist hierfür lediglich die Änderung der Routine »cmpelem« notwendig. Verwenden Sie als Elemente des Sortierfeldes nicht die zu sortierenden Elemente selbst, sondern lediglich Zeiger auf die Elemente und führen Sie den Elementenvergleich indirekt durch! Die Elemente werden dann nicht selbst sortiert, sondern lediglich die Zeiger auf die Elemente, was nebenbei noch den Vorteil hat, daß weniger Speicherverschiebungen notwendig werden.

Die Quicksortroutine stellt meines Erachtens ein sehr gutes Beispiel für den Einsatz der Pseudoops zur strukturierten Programmierung zugunsten einer schnellen Entwicklung von Programm-Prototypen dar – ein solches Verfahren wird auch

häufig als »rapid prototyping« beschrieben. Sie ermöglichen die schnelle Entwicklung lauffähiger Programme, an denen der Entwickler die Durchführbarkeit und Praktikabilität seiner Vorstellungen erproben kann. Die Pseudoops sind nicht in erster Linie dazu gedacht, einen vollkommenen Ersatz für die herkömmlichen Methoden der Assemblerprogrammierung zu bieten; eine abschließende Optimierung der Programme von Hand wird sich in professionellen Programmen nicht vermeiden lassen, um ein optimales Laufzeitverhalten und eine ebensolche Speicherausnutzung zu erreichen. Allerdings ist auf der anderen Seite zu berücksichtigen, daß der Overhead bei Beibehaltung der ASE-Strukturen immer noch wesentlich geringer ist als der von Hochsprachen-Compilern erzeugte.

3.3.8 Bedingte Assemblierung

Hier ist sie nun – die schon des öfteren angesprochene bedingte Assemblierung. Was leistet sie?

Im Prinzip genau das, was die Pseudos zur strukturierten Programmierung gerade nicht vollzogen: Sie arbeiten auf der Ebene des Quelltextes und erzeugen damit keinen Maschinenkode. Der Assembler faßt die Anweisungen zur bedingten Assemblierung als Anweisungen auf, Teile des Quelltextes nur unter gewissen Bedingungen zu assemblieren, die in den Parametern der Pseudoops angegeben sind. Ist die Bedingung nicht erfüllt, ignoriert der Assembler einen Bereich von Quelltextzeilen. Damit ist die Herkunft der Bezeichnung »bedingte Assemblierung« auch schon geklärt.

Der ASE besitzt zur Unterstützung der bedingten Assemblierung zwei Kontrollstrukturen für Fallunterscheidungen, Schleifenkonstruktionen werden nicht unterstützt, können jedoch unter Zuhilfenahme der Makrofähigkeit des Assemblers leicht realisiert werden, wie bei der Besprechung der Makros noch gezeigt werden wird.

3.3.8.1 Fallunterscheidung mit ».cond/.alter/.econd«

Die Benennung dieser Konstruktion liest sich vielleicht im ersten Augenblick etwas ungewohnt, ergibt sich jedoch aus der Tatsache, daß ».cond/.alter/.econd« prinzipiell wie die bekannte Konstruktion »if/else/endif« arbeitet – hier sind nicht die Struktur-Pseudoops des ASE gemeint, sondern Entsprechungen in den höheren Programmiersprachen. Da die bekannten Benennungen aber für die Struktur-Pseudos vergeben wurden, ergaben sich halt die Ersatzbezeichnungen

```
.cond - »condition«
.alter - »alternative«
.econd - »end condition«
```

Die Wirkung der hier betrachteten Pseudos wird am besten sichtbar, wenn Sie einmal das folgende Beispielprogramm eingeben:

```
100   -.base $c00
110   -;
120   -.request "Bedingung: ",bed
130   -;
140   -.cond bed.eq.1
150   -          lda #0
160   -.alter
170   -          lda #1
180   -.econd
190   -          rts
```

Starten Sie die Assemblierung mit »run«, und geben Sie auf die Nachfrage mit ».request« einen arithmetischen Ausdruck ein, um die Variable »bed« mit einem Wert zu versehen. Je nachdem, ob der eingegebene Wert gleich Eins war oder nicht, erhalten Sie an der Startadresse $c00 ein verschiedenes Ergebnis der Assemblierung. War die Bedingung »bed.eq.1« erfüllt, steht bei $c00 ein »lda #0«, andernfalls (».alter«) finden Sie dort den Befehl »lda #1«. Das »rts« ist in beiden Fällen gleich vorhanden; ».econd« zeigt das Ende der bedingten Assemblierung an.

Dies ist auch der Ort, an dem sichtbar wird, warum die ASE-Arithmetik Operatoren für Vergleiche enthält. In den bisher betrachteten Pseudoops und den Mnemonics waren die Vergleiche kaum sinnvoll einzusetzen, hier spielen sie jedoch eine zentrale Rolle. Viele Assembler ohne eigene Arithmetik behelfen sich stattdessen mit speziellen Pseudoops, die bestimmte Fälle auffangen, wie z. B. ».ifneq a,b« für den Vergleich »wenn a ungleich b«, der ASE ist dort jedoch durch die eigene Arithmetik ungleich flexibler.

Zurück zum Thema: Die Alternative mit ».alter« darf fehlen. Die Wirkung einer solchen Konstruktion dürfte wohl offensichtlich sein. Die Cond-Struktur darf nicht verschachtelt werden! Für Fallunterscheidungen mit mehreren Fällen steht eine weitere Konstruktion zur Verfügung, die weiter unten beschrieben wird.

Mit speziellen Bedingungen kann der aktuell durchgeführte Pass der Assemblierung abgefragt werden:

```
.cond p1  - falls Pass 1
.cond p2  - falls Pass 2
.cond p3  - falls Pass 3 (nur bei der Erzeugung von
                          Modulen)
```

Diese Abfragen sind in Verbindung mit einigen weiteren Pseudos sinnvoll, wobei man sich daran erinnern muß, daß die Assemblierung immer in mehreren Durchläufen vorgenommen wird – dadurch bedingt werden auch die von den Pseudos

ausgelösten Aktionen mehrfach durchlaufen. Ein Beispiel finden Sie bei der Besprechung des Pseudoops ».break«.

Ein wichtiger Hinweis zum Schluß:

Vor keinem der drei oben beschriebenen Pseudoops darf ein Label stehen!

Diese Einschränkung wird notwendig, da der Assembler bei der bedingten Assemblierung Zeilen überspringt und dabei die Anweisungen ».alter« bzw. ».econd« suchen muß. Diese Suche ist wesentlich effizienter und schneller, wenn die Pseudoops nicht durch vorstehende Label »verdeckt« werden.

3.3.8.2 Der Kontrollausdruck mit ».control«

Der sogenannte Kontrollausdruck bietet die soeben angekündigte Möglichkeit zur Unterscheidung von mehr als zwei Fällen. Er wird durch drei Anweisungen kontrolliert:

```
.control exp
.= exp
.-
```

».control« setzt den Kontrollausdruck für die folgenden weiteren Pseudoops der hier beschriebenen Fallunterscheidung. Der arithmetische Ausdruck »exp« wird bewertet und assemblerintern gespeichert, so daß er durch die folgenden Pseudoops abgefragt werden kann. Es kann nur ein Kontrollausdruck verwendet werden, eine Verschachtelung ist also auch hier nicht möglich. Allerdings kann der aktuelle Kontrollausdruck jederzeit dadurch gewechselt werden, daß ein neuer mit ».control« bestimmt wird.

Die Abfrage des Kontrollausdrucks erfolgt durch ».=«. Der Wert des hinter ».=« folgenden Ausdrucks wird mit dem gespeicherten Wert des Kontrollausdrucks verglichen. Verläuft der Vergleich positiv, werden die Quelltextzeilen hinter ».=« bis zum nächsten ».=« oder bis zum ».-« assembliert, ansonsten werden sie übersprungen.

».-« beendet die hier beschriebene Art bedingter Assemblierung. Die hinter ».-« folgenden Quelltextzeilen werden wieder »bedingungslos« assembliert.

Ein Beispiel:

```
100   -.base $c00
110   -;
120   -.request "Wahl: ",wahl
130   -;
140   -.control wahl
150   -.= 1
```

```
160   -           lda #0
170   -.= 2
180   -           lda #1
190   -.-
```

Starten Sie die Assemblierung dieses Quelltextes, so erhalten Sie je nach Eingabe, die Sie auf ».request« tätigen, verschiedenen Maschinenkode bei $c00. Die Zahl der einzelnen Alternativen mit ».=« ist im übrigen nicht auf zwei begrenzt.

Auch vor den hier beschriebenen Pseudoops dürfen aus ähnlichen Gründen wie zuvor **keine Label** verwendet werden!

3.3.8.3 Ausgabe von Meldungen auf den Bildschirm

Im Zuge der bedingten Assemblierung müssen zwei Pseudoops genannt werden, die zwar nicht direkt eine bedingte Assemblierung einleiten, die aber nur in Zusammenarbeit mit dieser Assemblierungsart einen sinnvollen Einsatz finden können. Der erste dieser Pseudoops nennt sich

```
.print "string"
```

».print« gibt einen String auf den Bildschirm aus. Der String darf Steuerzeichen enthalten, die in diesem Fall auf die bekannte Art wie in Basic als reverse Zeichen nach dem Anführungszeichen eingegeben werden. Fehlt die Angabe der auszugebenden Meldung hinter ».print«, so gibt dieser Pseudoop lediglich einen Zeilenvorschub aus. Beachten Sie, daß ».print« in beiden Passes der Assemblierung arbeitet! Sollen Ausgaben nur in einem bestimmten Pass erfolgen, müssen Sie die oben beschriebene Anweisung ».cond« benützen.

Ein Beispiel-Quelltext:

```
100   -.cond p1
110   -         .print "Beispielprogramm"
120   -         .print "================"
130   -         .print
140   -.econd
141   -;
150   -.request "Startadresse = ",base
160   -         .base base
170   -;
180   -.cond p1
190   -         .print "Kodeerzeugung:"
200   -         .print
210   -         .print "1 - Testlauf"
220   -         .print "2 - RAM"
```

```
230   -         .print "3  - Disk"
240   -         .print
250   -.econd
260   -;
270   -.request "Auswahl = ",choice
280   -.control choice
290   -. = 1
300   -         .syntax
310   -. = 2
320   -         .request "Kodeablage = ",code
330   -         .code 1,code
340   -. = 3
350   -         .object "file"
360   -.-
```

So könnte ein fast schon allgemeingültiger Kopf für einen Quelltext aussehen, der dann z. B. auf Diskette gespeichert werden könnte und zu Beginn jedes zu assemblierenden Quelltextes mittels ».source« eingebunden werden könnte. Der Assembler erfragt sich daraufhin alle für die Assemblierung notwendigen Angaben am Bildschirm. Natürlich sind die Ausgaben mit ».print« nicht sonderlich ausgebaut – in jeder Anweisung kann nur eine Bildschirmzeile ausgegeben werden und die Ausgabe beschränkt sich auf reine Strings – andererseits sollen ja auch nicht seitenweise Texte am Bildschirm gezeigt werden. Für die hier vorgesehenen Zwecke scheint mir die Funktion ausreichend.

3.3.8.4 Warten auf »Return« mit ».wait«

Auch eine rudimentäre Tastaturabfrage ist in den ASE implementiert:

```
.wait
```

wartet solange, bis der Benutzer die Return-Taste drückt, alle anderen Tastatureingaben werden ignoriert. Die Anweisung wirkt wieder in allen Durchläufen der Assemblierung, muß also gegebenfalls zusammen mit der bedingten Assemblierung eingesetzt werden.

Beispiel-Quelltext:

```
100   -.print "Bitte Disk A einlegen"
110   -.print "und <Return> drücken !"
120   -.wait
130   -.source "file"
```

In diesem Fall muß natürlich in jedem Pass der Assemblierung auf den Diskettenwechsel gewartet werden. Nebenbei bemerkt ist diese Art der Verwendung von

Quelltexten von mehreren Disks nur dann möglich, wenn der Maschinenkode im RAM abgelegt wird bzw. wenn einzelne Module erzeugt werden. Sobald Sie eine neue Diskette in die Floppy einlegen, würde auf der ersten Diskette ein offenes File zurückbleiben, falls Sie versuchen sollten, den Maschinenkode für die gesamte Assemblierung auf einmal zur Floppy zu senden.

3.3.8.5 Abbruch der Assemblierung mit ».break«

».break« ist die einfache Anweisung an den Assembler, die Assemblierung unverzüglich zu beenden. Es wird daraufhin die übliche Endmeldung des Assemblers ausgegeben. Auch ».break« wirkt in allen Durchläufen der Assemblierung.

Beispiele:

```
100   -.cond p2
110   -     .clist
120   -     .break
130   -.econd
```

Dieser Quelltextteil bricht die Ausgabe des Assemblerlistings an der Stelle ab, wo er im Quelltext verwendet wird. Die Ausgabe von Assemblerlistings wird später noch beschrieben.

```
100   -.cond *.gt.$4000
105   -     .print "PC unter ROM"
110   -     .break
120   -.econd
```

Diese vier Zeilen überwachen, ob der Programmzähler eine bestimmte obere Grenze überschreitet. Dies könnte beispielsweise von Nutzen sein, damit ein Programmteil zur Laufzeit nicht unter eines der ROMs zu liegen kommt.

In der folgenden Beschreibung der Makros werden Sie unter dem Stichwort »Makrobibliotheken« sehen, wie man sich eine Reihe solcher nützlicher Anwendungen der bedingten Assemblierung einmal schreibt und dann dauerhaft in Form einer Bibliothek in seinen Quelltexten verwenden kann.

3.3.9 Die Makros

Schon des öfteren wurde in den vorangegangenen Kapiteln auf die Makrofähigkeit des ASE-Assemblers verwiesen, ohne allerdings genau zu definieren, was unter einem Makro zu verstehen ist und wie man Makros im ASE einsetzt. Das soll nun nachgeholt werden.

Die Idee, die hinter den Makros steckt, ist eigentlich inspiriert von einer gewissen Tippfaulheit. Übersetzt man den Begriff »Makro« ins Deutsche, erhält man in etwa »Großbefehl«. Damit ist schon fast gesagt, was ein Makro darstellt: Eine Zusammenfassung mehrerer Befehle zu einem Ganzem, dem »Großbefehl«. Findet der Assembler einen Makroaufruf im Quelltext, bedeutet dies für ihn, die Befehlsfolge des Makros an der Stelle des Aufrufs in den Quelltext einzuassemblieren. Der ungewöhnliche Ausdruck »einassemblieren« soll hier deutlich machen, daß der Assembler keine wirkliche Einfügung von Quelltext an der Stelle des Makroaufrufs vornimmt, sondern lediglich die Makrodefinition aufsucht, diese assembliert und dann zur Quelltextzeile hinter dem Makroaufruf zurückkehrt.

Was fängt man mit dieser prinzipiellen Fähigkeit des Assemblers an? Eine ganze Menge, wie Sie in den folgenden Beschreibungen sehen werden. Aber sehen wir uns zunächst den formalen Aufbau der Makrodefinitionen und Aufrufe an:

3.3.9.1 Definition und Aufruf von RAM-Makros

Zunächst zum Begriff »RAM-Makros«: Unter dieser Bezeichnung möchte ich hier Makros verstanden wissen, deren Definitionen in einem Quelltext stehen, der bei der Assemblierung im Speicher des Rechners liegt, deshalb »RAM-Makros«. Die Makroaufrufe erfolgen entweder aus dem gleichen Quelltext heraus, der auch die Definitionen enthält, oder aber aus eingebundenen Quelltexten (».source«). Daß auch andere Fälle denkbar sind, werden wir später noch sehen.

Um ein Makro zu definieren, benötigt man lediglich zwei Pseudoops:

```
.macro name <par>
.macend
```

».macro« leitet die Makrodefinition ein. Der Pseudoop wird gefolgt von einem Makronamen. Der Makroname gehorcht denselben Regeln wie die Variablennamen; man kann ihn als Variablennamen auffassen, denn wieder dient eine Variable der ASE-Arithmetik als Speicher – diesmal wird der Ort der Makrodefinition in ihr aufbewahrt. Der Makroname dient beim Aufruf des Makro zur Identifizierung des Makros; der Assembler schlägt anhand der im Makronamen gespeicherten Adresse die Makrodefinition nach, assembliert die Definition und setzt danach die Assemblierung wieder hinter dem Aufruf fort.

Hinter dem Makronamen wiederum folgt eine optionale Parameterliste, hier mit »<par>« bezeichnet. »Optional« bedeutet dasselbe wie die häufig gebrauchte Wendung »fakultativ« – nämlich, daß die Parameterliste fehlen darf, hinter dem Pseudo ».macro« steht in diesem Fall also lediglich der Makroname. Die Parameterliste besteht – sofern vorhanden – aus einer Folge von Variablennamen, eingeschlossen in runde Klammern; die Namen werden untereinander durch

Kommata voneinander getrennt. Wenn es einmal niedergeschrieben ist, sieht man es vielleicht besser:

```
.macro name(var,var1,var2)
```

Auf die Bedeutung der Makroparameter wird gleich noch eingegangen; sehen wir uns zuerst den weiteren Aufbau der Makrodefinition hinter ».macro« an:

Die einleitende Zeile der Makrodefinition mit Makronamen und Parametern wird gefolgt von den Quelltextzeilen, die den eigentlichen Inhalt des Makros bilden sollen. Diese Zeilen des »Makrorumpfes« werden bei jedem Aufruf assembliert, so daß der von ihnen erzeugte Maschinenkode – sofern der Rumpf Kode erzeugt – am Ort des Aufrufes im erzeugten Maschinenprogramm erscheint. Bei jedem Aufruf eines Makros legt der ASE-Assembler automatisch einen lokalen Block um die Makrodefinition, hierdurch sind sowohl die Parameterlabel als auch die innerhalb des Makrorumpfes verwendeten Label lokal. Der Makroname ist ebenso automatisch immer global; ein Makro kann demzufolge in jedem beliebigen lokalen Block erreicht werden. Die Eigenschaft der Lokalität der in einer Makrodefinition verwendeten Label ist von zentraler Bedeutung für die Verwendbarkeit der Makros: Sie ermöglicht es, innerhalb eines Makros Label in beliebiger Zahl und ohne Rücksicht auf irgendwelche Einschränkungen zu benutzen – bei jedem Aufruf des Makros erhält es ja einen neuen Block, damit sind alle in der Definition verwendeten Label bei einem zweiten Makroaufruf tatsächlich physikalisch vollkommen verschieden von denen des ersten Aufrufs. Diese Eigenschaft ist durchaus nicht selbstverständlich, ich möchte mir aber an dieser Stelle ersparen, zu erläutern, mit welchen »Klimmzügen« andere Assembler Ähnliches zu erreichen versuchen.

Die Makrodefinition wird durch ».macend« beendet. Zu jedem ».macro« erwartet der Assembler genau ein ».macend«. Vor keinem der beiden Pseudoops der Makrodefinition darf ein Label stehen; dies ist wie bei der bedingten Assemblierung eine Folge davon, daß der Assembler Quelltextzeilen überlesen muß: Da die Makrodefinition für sich keinen Kode erzeugen soll, sondern nur die Makroaufrufe – anderes wäre ja auch unsinnig – werden die Makrodefinitionen selbst während der Assemblierung überlesen. Lediglich die Adressen der Definitionen werden in den Makronamen gespeichert.

Der Aufruf eines Makro erfolgt mit dem Pseudoop

```
.. name <par>
```

Der Pseudoop besteht wirklich nur aus zwei Punkten – beziehungsweise aus einem Punkt, wenn man den ersten Punkt abzieht, der zur Einleitung von Pseudoops allgemein benutzt wird. Hinter dem Pseudo folgt der Makroname, der dem Assembler den Ort der Makrodefinition mitteilt; der Makroname wiederum wird wieder gefolgt von einer Parameterliste, auf die etwas später noch genauer eingegangen wird. Für jetzt nur soviel: Die Parameterliste besteht im Aufruf – sofern

das aufgerufene Makro mit Parameteren definiert wurde – aus einer Folge von arithmetischen Ausdrücken, die durch Kommata voneinander getrennt werden; die gesamte Liste wird in runde Klammern eingefaßt.

Erst einmal genug der Theorie! Jetzt wird es Zeit, eine erste Makrodefinition zu zeigen:

```
100     -.base $c00
110     -;
120     -.macro mak
130     -           lda #0
140     -           ldx #2
150     -.macend
160     -;
170     -           ..mak
```

Das ist, für sich betrachtet, sicher keine sehr sinnvolle Anwendung eines Makro, für ein erstes Beispiel soll es aber erst einmal ausreichen. Sie sehen die Makrodefinition zu Beginn des Quelltextes. Das Makro enthält in diesem Fall keine Parameter. Im Makronamen »mak« wird die Adresse der Definition gespeichert; dies können Sie einmal nachprüfen, indem Sie den Quelltext assemblieren und sich anschließend einen Variablendump ausgeben lassen. Bei meiner Assemblerversion erhalten Sie dann eine Ausgabe von »mak = $61eb«. Sehen Sie sich diese Adresse innerhalb der RAM-Bank 0 mit dem Monitor an, können Sie erkennen, daß sie genau in die Makrodefinition weist. Fügen Sie weitere Zeilen vor der Makrodefinition ein oder löschen Sie beispielsweise die Kommentarzeile vor ».macro«, können Sie nachvollziehen, daß sich der Wert von »mak« bei einer neuen Assemblierung ändern wird. Assemblieren Sie nicht neu und lassen sich nach einer Änderung des Quelltextes noch einmal den alten Dump anzeigen, erhalten Sie eine unsinnige Anzeige. Preisfrage, weil es gerade hierher paßt, warum? Wenn Sie nicht darauf kommen, lesen Sie doch bitte noch einmal die Beschreibung unter dem Stichwort ».names«. Sie werden sehen, daß im Variablenfeld lediglich Zeiger auf die Variablennamen verwaltet werden – diese stimmen nach einer Änderung des Quelltextes natürlich nicht mehr. Das, was jetzt auf dem Bildschirm erscheint, ist das, was nach der Änderung des Quelltextes an dem Ort steht, wo vorher der Variablenname stand.

Zurück zum Thema: Die Makrodefinition wird durch ».macend« abgeschlossen. Dahinter folgt der eigentliche Start des Quelltextes, der Maschinenkode erzeugen soll. Die Makrodefinition wird ja überlesen. Der Aufruf des Makro erfolgt nun in der Zeile 170. Der Assembler findet den Namen »mak« und schlägt die in »mak« enthaltene Adresse der Definition nach. Anschließend merkt er sich die Zeile des Makroaufrufs und setzt die Assemblierung in der Makrodefinition fort. Er legt einen Block um die Definition, was in unserem Beispiel keine Auswirkungen hat, da in der Definition keine Label verwendet werden. Daraufhin assembliert er die beiden Zeilen des Makrorumpfes, bis er das ».macend« findet. Das ».macend« wirkt ähnlich wie das

»return« eines Basic-Unterprogramms: Der Assembler schließt den lokalen Block, als hätte er ein ».end« gefunden und kehrt zum Ort des Aufrufes zurück. Hier findet er das Ende des Quelltextes, das gleichzeitig das Ende der Assemblierung bedeutet. Als Ergebnis der Assemblierung erhalten wir die beiden Befehle »lda« und »ldx« an der angegebenen Startadresse $c00.

Ich hoffe, das Prinzip der Makroverarbeitung durch den Assembler ist hiermit klar geworden, so daß wir die Sache jetzt ein wenig komplizieren können.

Die Makroparameter: Wie Sie am obigen Beispiel sehen, ist die Verwendung von Makros in der oben gezeigten Art doch ziemlich unflexibel.Wieviele Vorkommnisse der Befehlsfolge »lda #0, ldx #2« befinden sich denn in einem durchschnittlichen Quelltext? Sicher nicht genügend, daß sich der Aufwand einer eigenen Makrodefinition lohnen würde. Ein einfacher Mechanismus der Übergabe von Parametern an ein Makro macht die ganze Sache doch schon wesentlich brauchbarer. Sehen Sie hierzu den folgenden Quelltext:

```
100   -.base $c00
110   -;
120   -.macro mak(a,b)
130   -          lda #<(a)
140   -          ldx #>(b)
150   -.macend
160   -;
170   -          ..mak(label+7,2*(label+2))
180   -          ..mak(2000,3000)
```

Dieser Quelltext ist dem vorher verwendeten sehr ähnlich, bis auf die Tatsache, daß diesmal Parameter eingesetzt werden, um das mögliche Einsatzgebiet des Makros zu vergrößern.

Die Vorgehensweise des Assemblers bei der Verarbeitung des Quelltextes entspricht der weiter oben beschriebenen, bis es zum Makroaufruf kommt: Der Assembler liest wieder den Makronamen und erhält so die Adresse der Makrodefinition. Diesmal allerdings folgt dem Makronamen eine runde Klammer – für den Assembler das Zeichen, daß eine Parameterliste folgt. Daraufhin sieht der Assembler in der Makrodefinition nach, ob auch die Definition mit Parametern versehen ist. Ist dies nicht der Fall, erfolgt eine Fehlermeldung »macro parameter error«. Besitzen sowohl Definition als auch Aufruf eine Parameterliste, so legt der Assembler wieder einen lokalen Block um die Definition. Alle in der Folge angelegten Variablen und Label sind somit lokal. Danach liest der Assembler den ersten Parameter der Definition und legt eine Variable des angegebenen Namens lokal an. Hier wird deutlich, warum die Parameterliste der Makrodefinition nur aus einer Aufzählung von Variablennamen besteht. Anschließend kehrt der Assembler an den Ort des Aufrufes zurück; damit verläßt der den lokalen Block der Definition und kehrt in den beim Aufruf herrschen-

den Block zurück. In der Parameterliste des Makroaufrufs liest er nun den ersten arithmetischen Ausdruck – die Parameterliste des Aufrufs darf aus allgemeinen Ausdrücken bestehen. Den Wert des Ausdrucks transportiert er anschließend in die soeben angelegte lokale Variable der Makrodefinition. Dieses Spiel setzt sich fort, bis alle Variablen der Parameterliste der Definition und alle Ausdrücke des Aufrufes verarbeitet sind. Sollte die Anzahl der Parameter in Definition und Aufruf nicht übereinstimmen, erhält man wieder die oben erwähnte Fehlermeldung. Sind die Parameter alle übertragen, setzt sich die Assemblierung wie sonst auch in der Makrodefinition fort, bis das ».macend« gefunden wird. Die in die Parametervariablen übertragenen Werte stehen jetzt im lokalen Block der Definition zur Verfügung, so daß das Makro verschiedenen Kode für Aufrufe mit verschiedenen Parametern erzeugen kann. Diese Eigenschaft sehen Sie ausgenutzt in den beiden Makroaufrufen des obigen Beispiels.

Makros können ineinander verschachtelt werden, d. h., innerhalb einer Makrodefinition können weitere Makros aufgerufen werden. Dies geht soweit, daß sogar ein rekursiver Makroaufruf möglich ist, wobei sich das Makro in der Makrodefinition selbst aufruft. Die letzte Möglichkeit ist natürlich nur sinnvoll in Verbindung mit der bedingten Assemblierung; ein Beispiel für einen rekursiven Makroaufruf finden Sie am Schluß der Makrobeschreibung. Die Schachtelungstiefe kann maximal wieder über 80 betragen.

Gewisse Probleme treten bei Makroaufrufen innerhalb mit ».source« eingebundener Quelltexte auf, die sich aus den obigen Beschreibungen, wie der Assembler bei Aufrufen und Makrodefinitionen vorgeht, eigentlich schon ergeben: In einem eingebundenen Quelltext können prinzipiell keine Makros aufgerufen werden, deren Definitionen im eingebundenen Text selbst stehen. Da eingebundene Quelltexte direkt von der Diskette verarbeitet werden, gibt es einfach keine RAM-Adresse, unter der sich der Assembler die Definition merken könnte. Folglich führen solche Makroaufrufe ins »Abseits«. Nichtsdestotrotz werden Makrodefinitionen innerhalb eingebundener Quelltexte korrekt überlesen. Dies erleichtert in gewisser Hinsicht die Einbindung schon vorhandener Quelltexte in einen anderen. Aufrufe von RAM-Makros aus eingebundenen Quelltexten bereiten keine Schwierigkeiten. Falls Sie also Makros in eingebundenen Quelltexten verwenden wollen, achten Sie darauf, daß die Makrodefinitionen in dem Text stehen, in den eingebunden wird!

3.3.9.2 Makrobibliotheken

Zwei Ideen führen von der einfachen Verarbeitung von RAM-Makros zur Einführung von Makrobibliotheken: Zum einen kann man in der Regel davon ausgehen, daß Makros nicht nur innerhalb eines Quelltextes, sondern in vielen Quelltexten verwendet werden können. Es gibt ausreichend viele Befehlsfolgen, die in allen Quelltexten gleich verwendet werden, daß sich die Einführung einer Makrobibliothek lohnt, aus der die Makros einfach ausgelesen werden können, also nicht jedesmal

neu geschrieben werden müssen. Zum zweiten ist natürlich die Einschränkung der Makroverwendung innerhalb eingebundener Quelltexte ein Dorn im Auge, so daß dieses Manko gleich durch die Makrobibliotheken aufgehoben werden sollte.

Was sind Makrobibliotheken und wie werden sie eingesetzt? Der Aufbau der Bibliothek ist schnell erklärt: Es handelt sich dabei lediglich um eine Sammlung von Makrodefinitionen, die hintereinander geschrieben werden. Der so entstehende Quelltext wird auf Diskette abgespeichert. Durch zwei spezielle Pseudoops des ASE können Makros aus einer angegebenen Bibliothek aufgerufen werden, als wenn sie im RAM stünden. Dieser Aufruf funktioniert auch in eingebundenen Quelltexten ohne Einschränkung.

Die Vorgehensweise des Assemblers bei der Verarbeitung von Bibliotheksmakros ist grundsätzlich gleich der normalen Arbeitsweise. Unterschiede ergeben sich aus der Tatsache, daß die Makrodefinition diesmal nicht im RAM zu finden ist. Bei einem Aufruf eines Bibliotheksmakros öffnet der Assembler statt dessen die aktuelle Bibliothek und sucht die Definition des Makros mit dem im Aufruf angegebenen Namen. Wird die Definition gefunden, geht alles weitere den schon beschriebenen Gang, ansonsten erhalten Sie natürlich eine Fehlermeldung.

Die Verwendung von Bibliotheken wird durch die folgenden beiden Pseudos gesteuert:

```
.maclib "bibliotheksname",u
... makroname <par>
```

Die erste Anweisung teilt dem Assembler den Namen der aktuellen Makrobibliothek mit und setzt gleichzeitig die Gerätenummer, unter der der Assembler die Bibliothek finden kann. Hier greift der gleiche Mechanismus Platz, wie er schon beispielsweise bei der Beschreibung von ».source« dargestellt wurde: Ist keine Gerätenummer angegeben, wird das Gerät mit der Nummer 8 angesprochen. Alle nach ».maclib« aufgerufenen Bibliotheksmakros werden in der durch ».maclib« gesetzten Bibliothek vermutet. Sollen mehrere Bibliotheken verwendet werden, muß zwischen den Bibliotheken mit ».maclib« hin- und hergeschaltet werden.

Ein kleines Beispiel zum Ausprobieren:

Geben Sie zunächst den folgenden Quelltext ein, der unsere kleine Beispiel-Bibliothek darstellen soll:

```
100   -.macro mac1
110   -.cond p1
120   -.print "dies ist mac1"
130   -.econd
140   -.macend
150   -.macro mac2
160   -.cond p1
```

```
170   -.print "dies ist mac2"
180   -.econd
190   -.macend
```

Wie Sie bemerken werden, ist dieser Quelltext ohne jede mögliche Einrückung oder Kommentierung geschrieben. Dies hat seinen guten Grund, auf den ich Sie noch hinweisen möchte: Da die Makrodefinitionen der Bibliothek direkt von Diskette gelesen werden und da eventuell eine ganze Menge Text überlesen werden muß, bis die gesuchte Definition gefunden ist, empfiehlt es sich, zwei Versionen seiner Bibliotheken zu verwenden: Eine Version, aus der alle überflüssigen Zeichen entfernt sind, die den engen Flaschenhals des IEC-Bus zwischen Rechner und Floppy unnötig verstopfen könnten, und eine weitere Version, die vollständig dokumentiert ist. Nebenbei sollten Sie auch eine schriftliche Dokumentation Ihrer Makrobibliotheken anfertigen, z. B. durch einen entsprechenden Ausdruck.

Speichern Sie nun den oben beschriebenen Quelltext unter dem Namen »l.lib1« ab, und geben Sie den aufrufenden Quelltext ein:

```
100 -.base $c00
110 -;
115 -.maclib "l.lib1"
120 -...mac1
130 -...mac2
```

Wenn Sie die Assemblierung nun mit »run« starten, erhalten Sie die erwarteten beiden Bildschirmausgaben, wobei Sie hören werden, wie der Assembler auf die Diskette zugreift, um die Makros aus der Bibliothek auszulesen.

3.3.9.3 Praktische Beispiele für die Makroverwendung

Die Bedeutung der Makros und, in Verbindung mit ihnen, der bedingten Assemblierung wird wohl am besten deutlich, wenn Sie einige praktische Anwendungen kennenlernen:

Beispiel 1:

Wie Sie wissen, ist der Befehl ».space of« auf eine maximale Zahl zu reservierender Bytes von 255 beschränkt. Durch ein einfaches Makro unter Verwendung der bedingten Assemblierung läßt sich diese Einschränkung aber leicht umgehen:

```
10    -;=========================================
20    -;                TAB(SIZE)
30    -;=========================================
40    -; reserviert SIZE mit Null vorbesetzter Bytes
50    -; und gibt bei möglicher Überschreitung der
60    -; maximalen Schachtelungstiefe eine spezielle
70    -; Fehlermeldung aus
80    -;-----------------------------------------
90    -;
100   -.macro tab(size)
110   -.  cond size.gt.80*255
120   -         .print "Fehler in Makro 'tab'"
130   -         .print "Parameter zu groß"
140   -         .break
150   -.  econd
160   -.  cond size.gt.255
170   -         ..tab(size-255)
180   -         .space of 255,0
190   -   alter
200   -         .space of size,0
210   -   econd
220   -.macend
```

Dieses Makro ist ein sehr gutes Beispiel für die praktische Verwendung des rekursiven Makroaufrufs und zeigt gleichzeitig, warum innerhalb der bedingten Assemblierung keine Schleifenstrukturen benötigt werden: Durch Rekursion kann man diese Schleifen leicht nachbilden. Die Überprüfung auf eine mögliche Überschreitung der maximalen Schachtelungstiefe zu Beginn der Makrodefinition könnte man sich auch sparen; der Assembler gibt von sich aus auch eine Meldung »structure too complex« aus, wenn die maximale Tiefe überschritten wird.

Beispiel 2:

Der ASE-Assembler unterstützt keinerlei illegale Opkodes, da ich die Verwendung dieser Kodes sowohl für überflüssig halte als auch für einen bedenklichen Programmierstil. Dennoch will ich hier zeigen, wie Sie sich Makros basteln können, die Ihnen die Arbeit mit den illegalen Kodes erleichtern, wenn Sie denn gar nicht davon absehen wollen:

```
100   -.macro illdef
101   -;
110   -.define ipl  := 0 ; implizite Adressierung
120   -.define imm  := 1 ; unmittelbare Adr.
130   -.define zep  := 2 ; Zeropage
```

```
140   -.define zpx  := 3 ; Zeropage X-indiziert
150   -.define zpy  := 4 ; Zeropage Y-Indiziert
160   -.define abs  := 5 ; absolut
170   -.define abx  := 6 ; absolut X-indiziert
180   -.define aby  := 7 ; absolut Y-indiziert
190   -.define ix   := 8 ; (,x)
200   -.define iy   := 9 ; (),y
201   -;
210   -.macend
```

Diese für die bedingte Assemblierung innerhalb des folgenden Makros gebrauchten globalen Definitionen könnten Sie beispielsweise aus einer Bibliothek heraus in Ihre Quelltexte einbringen.

Der Aufruf eines bestimmten illegalen Opkodes erfolgt durch den Einsatz spezieller Makros für jeden Befehl – eventuell lassen sich auch andere, allgemeinere Lösungen finden, hier ist Ihre Phantasie gefragt. Ich möchte als Beispiel lediglich die Behandlung des Befehls »rla« (rotate left, and with A, store A) zeigen:

```
100   -.macro rla(adr,art)
110   -. cond adr.lt.256
120   -.  control art
130   -.  = zep
140   -           .byte $27
150   -.  = zpx
160   -           .byte $37
170   -.  = ix
180   -           .byte $23
190   -.  = iy
200   -           .byte $33
210   -.  -
220   -           .byte adr
230   -. alter
240   -.  control art
250   -.  = abx
260   -           .byte $3f
270   -.  = aby
280   -           .byte $3b
290   -.  = abs
300   -           .byte $2f
310   -.  -
320   -           .word adr
330   -. econd
340   -.macend
```

Innerhalb dieses Makros wird nicht geprüft, ob eine nicht zulässige Adressierungsart gewählt wurde, diese Prüfung könnten Sie ja eventuell noch selbst einbauen. Der Aufruf des illegalen Befehls sähe dann beispielsweise so aus:

```
100 -       ..rla(label,zx)
```

oder

```
110 -       ..rla($1234,abx)
```

3.3.10 Dokumentationsunterstützung

Das vorliegende Kapitel ist das letzte, das die »normalen« oder halbwegs normalen Fähigkeiten des ASE-Assemblers beschreibt, in dem Sinne, daß die bis dato beschriebenen Features entweder in ähnlicher Form auch in anderen 6502-Assemblern zu finden sind oder aber es handelte sich um Einrichtungen, die zumindest in das von 6502-Assemblern gewohnte Konzept paßten. Im nächsten Kapitel werden Sie dann eine Arbeitsweise kennenlernen, die auf Computern mit Standardbetriebssystemen üblich ist, sich aber für den 6502-Prozessor bislang nicht durchsetzen konnte – auf die Gründe hierfür wird an geeigneter Stelle noch eingegangen.

Im hier anliegenden Abschnitt geht es um das Problem der Dokumentation von Assemblerprogrammen. Der ASE unterstützt Sie im Rahmen seiner Möglichkeiten bei der Ausgabe formatierter Assemblerlistings sowie der Ausgabe formatierter Symboltabellen, die als Teil der abschließenden Programmdokumentation eine wichtige Rolle spielen.

3.3.10.1 Ausgabe formatierter Assemblerlistings

Während des zweiten Durchlaufs der Assemblierung kann ein Assemblerlisting anstelle des Maschinenkodes erzeugt werden. Solche Assemblerlistings haben Sie sicher schon in Zeitschriften oder anderswo gedruckt gesehen. Sie enthalten den Inhalt der Quelltextzeilen, daneben den erzeugten Maschinenkode und die Adresse, an der der Maschinenkode abgelegt wird. Einige Beispielzeilen zeigen das allgemeine Aussehen eines Assemblerlistings:

```
a876   4cbbc4  :200   -marke        jmp label ; ...
a879   a900    :210   -             lda #0    ; ...
a87b   20d2ff  :220   -             jsr bsout
```

Die linke Spalte enthält die Adresse, für die der Maschinenkode bestimmt ist. Die Spalte rechts daneben zeigt die einzelnen Bytes des erzeugten Maschinenkodes; sie werden ohne trennendes Leerzeichen ausgegeben, um möglichst Platz auf der Druckseite zu sparen. Hinter einem trennenden Doppelpunkt folgt dann die Quelltextzeile. Diese wird in der gleichen Art formatiert, wie dies auch beim Listen auf dem Bild-

schirm mit dem Editorbefehl ».e« der Fall wäre. Die Positionen der Mnemonics, Kommentare usw. können wie beim Listen durch die Befehle ».t« – eingegeben im »Direktmodus« – beeinflußt werden.

Zeilen, die Pseudoops zur Tabellenerzeugung enthalten, werden ohne Adresse und Opkodes ausgegeben:

```
:400  -          .byte 2,3,4
:410  -          .word adr
:420  -          .space of 255
:430  -          .screen "abc"
:440  -          .revers "abc"
```

Jede Druckseite wird durch eine Kopfzeile eingeleitet, die eine Seitennumerierung enthält. Die Kopfzeile wird vom eigentlichen Listing durch eine Leerzeile getrennt.

Das Listen wird durch zwei Pseudoops eingeleitet bzw. abgeschaltet:

```
.list <openpar>
.clist
```

».list« startet das Listen; die hinter ».list« folgende Parameterliste »<opnpar>« entspricht exakt derjenigen hinter dem Basicbefehl »open«. Einige Beispiele zeigen verschiedene Möglichkeiten:

».list 1,3«: Öffnet das Gerät 3 (Bildschirm) und zeigt damit das Listing auf dem Schirm an.

».list 1,4,7«: Sendet das Listing zum Drucker mit der Geräteadresse 4.

».list 2,8,2,"listing,s,w"«: Sendet das gleiche Listing in ein sequentielles File namens »listing« auf Disk. Von dort könnte das Listing beispielsweise mit der Textverarbeitung PROTEXT weiterverarbeitet werden, die sequentielle Files aufnehmen kann.

Bei der gleichzeitigen Erzeugung von Maschinenkode und eines Listings ist ein wenig Vorsicht geboten: Der ASE-Assembler ist nicht dazu ausgelegt, gleichzeitig ein Listing und den Maschinenkode über den IEC-Bus zu leiten. Mit anderen Worten: Ist Ihr Drucker über ein Interface an die gleichen Leitungen angeschlossen wie die Floppy, so sollten Sie die Pseudoops ».object« oder ».modul« nicht gleichzeitig mit ».list« verwenden, sofern ».list« das Assemblerlisting an ein Gerät am IEC-Bus sendet.

Im Zusammenhang mit der Ausgabe der Assemblerlistings ist noch ein anderer Punkt anzusprechen: Der Assembler ersetzt kein Drucker-Interface! Falls Sie einen Drucker verwenden, der nicht direkt an einen Commodore-Rechner anschließbar ist, benötigen Sie unbedingt ein Interface, das die spezielle Zeichenkodierung, die von Commodore verwendet wird, in eine für den Drucker verständliche Form umwan-

delt. Verwenden Sie aber ein Interface, kann es gerade dadurch trotzdem zu Komplikationen kommen, die Sie nicht dem Programm anlasten können. Als Beispiel sei hier das vielfach verwendete Wiesemann-Interface genannt: Schalten Sie beispielsweise einen Epson-FX-Drucker auf die Schriftart »Elite« um (96 Zeichen/Druckzeile), so zieht das Interface nicht mit – die einzelnen Zeilen werden trotz der neuen Einstellung nach 80 Zeichen gebrochen. Solche und ähnliche Probleme können Sie praktisch nur zuverlässig vermeiden, indem Sie eines der Software-Interfaces benutzen, die schon vielfach in Zeitschriften und Büchern veröffentlicht worden sind. Zum C128 wurden solche Software-Interfaces speziell für den C128 im Heft 2/86 von 64er und im Sonderheft 10/86 dieser Zeitschrift abgedruckt. Diese unterstützen DIN- und ASCII-Zeichensatz.

Die Ausgabe des Assemblerlistings wird abgeschaltet, indem Sie den Befehl ».clist« hinter die letzte Zeile setzen, die noch ausgegeben werden soll. ».clist« wirkt nur in Pass 2 und bricht die Assemblierung **nicht** ab. Ein automatisches Abschalten des Assemblerlistings findet zudem zum Ende der Assemblierung statt.

Die für die Druckausgabe verwendete Seitenlänge wird durch den Befehl

```
.skip m,n
```

eingestellt. Die Parameter »m« und »n« werden wie folgt berechnet: Es werden »m« Quelltextzeilen pro Druckseite ausgegeben. Es wird dabei stillschweigend davon ausgegangen, daß eine Quelltextzeile nicht länger als eine Druckzeile ist, ansonsten wird die Berechnung der Druckseitenlänge unsauber. Nach Ausgabe der »m« Quelltextzeilen werden »n« Leerzeilen ausgegeben, um die nächste Druckseite zu erreichen. Die beiden für die Kopfzeile und die nachfolgende Leerzeile benötigten Druckzeilen müssen gesondert hinzugezählt werden, um die Länge einer Gesamtseite zu erhalten:

$$\text{Seitenlänge} = m + n + 2$$

Die meisten Endlospapiere haben eine Seitenlänge von 72 Druckzeilen, so daß man mit einer Einstellung von ».skip 60,10« ein ansprechendes Druckbild erhält. ».skip« muß **nach** ».list« im Quelltext erscheinen.

Eine Reihe weiterer Pseudoops unterstützen die Ausgabe von Assemblerlistings. Sie werden in der Folge der Reihe nach aufgeführt:

».**page**«: Erzwingt eine neue Druckseite. Der Pseudoop erwartet keine Parameter.

».**lprint <liste>**«: Ermöglicht das Senden von Steuerkodes vor der Ausgabe der ersten Zeile des Listings an den Drucker. Die Parameterliste »<liste>« kann aus ASCII-Strings und Bytewerten bestehen – ähnlich der Parameterliste hinter ».byte«, mit der Ausnahme der Bildkodewerte. Um die Steuerkodes senden zu können, ist es notwendig, daß ».lprint« hinter ».list« in den Quelltext gesetzt wird. Ein Beispiel:

```
.lprint 27,"r",0
```

Diese Anweisung würde einen Epson-Drucker (oder einen Kompatiblen) auf den amerikanischen Zeichensatz umschalten. Beachten Sie die Unterschiede zwischen dem Commodore-ASCII und dem Standard-ASCII innerhalb der Strings! Kleine und große Buchstaben tauschen ihre Plätze, so daß das »"r"« des Commodore-ASCII einem »"R"« des Standard-ASCII entspricht.

»**.nops**«: Schaltet die Ausgabe der beiden ersten Spalten des Assemblerlistings – Adressen und Opkodes – ab.

»**.nonum**«: Schaltet auf die gleiche Weise die Ausgabe der Zeilennummern ab.

»**.malins**«: Der Assembler ist so voreingestellt, daß die während eines Makroaufrufes durchlaufenen Quelltextzeilen der Makrodefinition nicht gelistet werden. Dies kann mit ».malins« geändert werden.

»**.cdlins**«: Bewirkt das gleiche für die Ausgabe bedingt assemblierter Zeilen. Nach Voreinstellung listet der Assembler nur diejenigen Quelltextzeilen, die tatsächlich auch assembliert werden. Verwenden Sie ».cdlins«, so werden auch die Zeilen gelistet, die die nicht durchlaufenen Teile der bedingten Assemblierung enthalten.

3.3.10.2 Ausgabe der Symboltabelle

Soll eine Symboltabelle ausgegeben werden, so können Sie den Befehl

```
.symbols sort,<openpar>
```

verwenden. »<openpar>« ist wie bei ».list« eine Parameterliste, die bestimmt, wohin die Symboltabelle gesendet werden soll. Der Schalter »sort« legt fest, ob – und wenn, wie – die Symboltabelle sortiert werden soll. Es stehen drei Möglichkeiten zur Verfügung, die durch jeweils einen Buchstaben abgekürzt werden:

u	=	unsortiert
w	=	nach Wert sortiert
a	=	alphabetisch sortiert

Während einer Assemblierung kann immer nur eine Symboltabelle gedruckt werden. Die Gründe, warum eine Umsortierung nicht möglich ist, wurden weiter oben schon erläutert. Der Pseudoop ».symbols« kann an einer beliebigen Stelle des Quelltextes gesetzt werden, die Ausgabe der Symboltabelle erfolgt aber grundsätzlich erst zum Ende der Assemblierung.

Einige Beispiele für die Verwendung von ».symbols«:

.symbols u,1,3	–	unsortiert auf den Bildschirm
.symbols w,1,4,1	–	nach Wert sortiert auf den Drucker
.symbols a,2,8,2,"syms,s,w"	–	alphabetisch sortiert in einen File auf Diskette

Die Form der ausgegebenen Symboltabelle entspricht der, die Sie über die Editorbefehle ».d«, ».dw« und ».da« am Bildschirm erhalten.

3.3.11 Relokatibler Kode/Linker und Lader

Alle bisher besprochenen Features des ASE-Assemblers bewegten sich in dem Rahmen: Man nehme einen Quelltext, schicke ihn durch den Assembler und man erhält ein fertiges Maschinenprogramm. Das dieses Prozedere nicht unbedingt so ablaufen muß – ja, daß es sogar erhebliche Vorteile bringt, einen anderen Weg zu beschreiten – soll in diesem Kapitel aufgezeigt werden. Ich hoffe, Sie werden von den hier gezeigten Möglichkeiten ebenso überzeugt werden, wie ich selbst es wurde, nachdem der ASE einmal fertiggestellt war.

3.3.11.1 Das Prinzip relokatibler Module

Vergleicht man einen Quelltext und das aus dem Quelltext erzeugte Maschinenprogramm miteinander, stellt man fest, daß beide Formen Vor- und Nachteile bei einer weiteren Verwendung aufweisen:

- Vorteile des Quelltextes: Man kann jederzeit ein neues Maschinenprogramm erzeugen, das eine andere Startadresse besitzt. Dies ist zugleich ein eklatanter Nachteil des fertigen Maschinenprogramms, das an eine einmal bei der Assemblierung gewählte Startadresse gebunden ist. So kommt es beispielsweise zustande, daß man zehn verschiedene Versionen eines Programms aufbewahren muß, die sich lediglich in der Startadresse unterscheiden, für die sie bestimmt sind. Änderungen in einem Programm sind auf Quelltextebene leicht durchzuführen – dazu genügt es ja, einige Zeilen neu oder umzuschreiben. Ein fertiges Maschinenprogramm dagegen läßt sich praktisch überhaupt nicht mehr verändern.
- Nachteile des Quelltextes: Zunächst einmal ist ein Quelltext in der Regel sehr lang gegenüber dem fertigen Maschinenprogramm. Beispiel ASE – aus mehreren hundert KByte Quelltext entsteht schließlich ein 17 KByte langes Maschinenprogramm. Bei großen Programmvorhaben kann dies sogar zu echten Problemen führen, die schließlich nur noch zu lösen sind, wenn man ein zweites Laufwerk zu Verfügung hat. Ein weiterer schwerer Nachteil des Quelltextes ist, daß immer mindestens zwei Durchläufe durch den **gesamten** Quelltext notwendig sind, bevor ein neues Maschinenprogramm fertig assembliert ist. Trotz der Geschwindigkeit des Assemblers selbst wirkt sich diese Tatsache bei langen Programmen schließlich äußerst negativ aus – die im Vergleich mit Laufwerken anderer Computer immer noch recht behäbigen Floppies des C-128 lassen jede marginale Änderung in einem Programm zu einer ausgedehnten Kaffeepause werden.

Aus den oben geschilderten Überlegungen heraus – und schließlich, um mir selbst bei der weiteren Programmierung auf dem C-128 einen Gefallen zu erweisen – entstanden die Konzepte der relokatiblen Module und des Linkers. Um nicht so zu tun, als seien diese Konzepte vollkommen neu und aus der Luft gegriffen, sei hier erwähnt, daß sie auf anderen Rechnern mit Standardbetriebssystemen wie CP/M, MS-DOS oder auch TOS und AMIGA-DOS gang und gäbe sind. Allerdings tragen die dort verwendeten einfachen Überlegungen nicht mehr in einem System wie dem C-128. Warum dies so ist, das wird in den späteren Beschreibungen noch klar werden.

Zuerst zur Frage: Wie bekommt man Vor- und Nachteile des Quelltextes und des Maschinenprogramms unter einen Hut, so daß man möglichst nur in den Genuß der Vorteile kommt?

Diese Frage dürfte sich schon fast aus der Überschrift des Kapitels beantworten: Man erzeuge einen Zwischenkode aus dem Quelltext, der noch kein fertiges Maschinenprogramm ist, aber einen Teil der Vorteile des Quelltextes und einen Teil der Vorteile des Maschinenprogramms besitzt. Einen solchen Zwischenkode nennt man ein relokatibles Modul.

Welche Eigenschaften besitzt ein relokatibles Modul:

- Seine Länge liegt nur wenig über der des fertigen Maschinenprogramms.
- Durch Spezialprogramme kann aus dem Modul in **einem** Durchlauf ein Objektprogramm hergestellt werden. Zusammen mit der Tatsache, daß das Modul gleichzeitig sehr kurz ist, ergibt sich also ein erheblicher Geschwindigkeitsvorteil gegenüber der Assemblierung des entsprechenden Quelltextes.
- Die Startadresse des aus dem Modul hergestellten Maschinenprogramms kann wie bei der normalen Assemblierung frei gewählt werden.

Dies ist der erste Schritt in Richtung auf das angestrebte Ziel; lediglich die leichte Änderung des Zwischenkodes ist noch nicht enthalten – zu diesem Punkt kommen wir aber anschließend noch.

Sehen wir uns als nächstes an, wie wir ein relokatibles Modul mit dem ASE erzeugen:

3.3.11.2 Erzeugung und Aufbau relokatibler Module

Der ASE macht die Erzeugung relokatibler Module so einfach, wie man sich dies nur wünschen kann: Sie verwenden lediglich den Pseudoop

```
.modul "filename",u
```

anstelle der sonst verwendeten Pseudoops ».code« oder ».object« im Quelltext. Der Assembler schaltet daraufhin einen dritten Assemblerpass hinzu, der die Erzeugung

eines Moduls statt eines Maschinenprogramms unternimmt. Im Prinzip sind für die Modulerzeugung wie bei einer normalen Assemblierung nur zwei Durchläufe notwendig; daß der ASE trotzdem einen dritten Pass benötigt, liegt einfach an der Tatsache, daß er sowohl die normale Assemblierung als auch die Modulerzeugung beherrscht. Um einen Vergleich mit den bekannten CP/M-Assemblern zu bieten: ASE entspräche einem Programm, das sowohl die Fähigkeiten von MAC als auch die von RMAC in einem Programm zusammenfaßt. Die Verwendung dreier Passes anstelle von zweien ist wegen der hohen Assemblierungsgeschwindigkeit des ASE nicht weiter tragisch und fällt in der Regel kaum ins Gewicht.

Das weitere wird am besten deutlich, wenn wir erst einmal ein relokatibles Modul erzeugt haben. Geben Sie hierzu den folgenden Quelltext ein:

```
100 -.base $c00
110 -;
120 -.modul "test.rel"
130 -;
140 -speicher       .byte >(start)
150 -               .word uprog
150 -;
160 -start          lda #0
170 -               lda #>(speicher)
180 -               jsr uprog
190 -uprog          rts
```

Das ist sicher kein sinnvolles Programm, enthält aber alles, was benötigt wird, um den Aufbau der relokatiblen Module zu zeigen. Starten Sie die Assemblierung mit »run«. Sie erhalten daraufhin das Modul unter dem Namen »test.rel« auf der Diskette im Laufwerk mit der Gerätenummer 8. Der Längenunterschied zwischen Quelltext, Modul und endgültigem Maschinenprogramm wird in unserem Beispiel noch nicht recht deutlich, weil es relativ kurz ist. Sie gewinnen jedoch einen Eindruck von den Proportionen, wenn Sie auf der Masterdisk die Files mit der Erweiterung ».rel« mit denen gleichen Namens und dem Zusatz »2000« vergleichen. Die ersteren sind die Modulfassungen derselben Programme, die in der endgültigen Form in den »2000er«-Files vorliegen. Rechnet man die Längenverhältnisse einmal aus, erhält man: Ein Quelltext, der voll auskommentiert ist, ist ca. zehnmal bis zwanzigmal so lang wie das fertige Maschinenprogramm – dieser Wert ist natürlich stark davon abhängig, wie Sie kommentieren. Das Modul desselben Programms ist dagegen nur ca. 13 bis 14 Prozent größer als das Maschinenprogramm selbst.

Auch Unterschiede in der Assemblierungsgeschwindigkeit sind in unserem Beispiel noch nicht festzustellen. Es kann sogar geschehen, daß die Modulerzeugung etwas schneller abläuft als die Erzeugung eines Maschinenprogramms mit ».object«, obwohl ein dritter Assemblerlauf durchgeführt wird. Dieses seltsame Verhalten ist von der Floppy bedingt, woran man sieht, wieviel Einfluß der relativ langsame IEC-

Bus auf die Verarbeitungsgeschwindigkeit von Programmen ausübt und wie wichtig es daher ist, die zu verarbeitenden Daten – hier das Modul – möglichst klein zu halten.

Zurück zu unserem Beispielmodul: Wir wollen uns jetzt einmal ansehen, was der Assembler an Zwischenkode erzeugt hat. Gehen Sie dazu in den »Tedmon« des C-128 (drücken Sie hierzu einfach die Taste F8, Tastenbelegung »monitor«). Laden Sie nun das erzeugte Modul in den Speicher:

```
l "test.rel",8,12000
```

lädt das Modul an die Adresse $2000 in der RAM-Bank 1. Dort schauen Sie sich das Modul mit dem Befehl

```
m 12000
```

an. Hier folgt, was Sie auf dem Bildschirm sehen werden, noch einmal abgedruckt:

```
ready.

monitor
    pc  sr ac xr yr sp
; fb000 00 00 00 00 f8

>12000 00 00 00 0c 0b 0c 00 00 00 53 54 41 52 54 00 03:
>12010 0c 00 00 00 00 00 53 50 45 49 43 48 45 52 00 00:
>12020 0c 00 00 5f 0f 1f 00 6f 00 00 3e 28 53 54 41 52:
>12030 54 29 00 1f 00 2f 02 0a 0c a9 00 0f 0f 0f 00 00:
>12040 3e 28 53 50 45 49 43 48 45 52 29 00 a9 0c 20 0a:
>12050 0c 60 0b 42 59 20 20 c7 45 52 44 20 cd 4f 45 4c:
>12060 4c 4d 41 4e 4e 0d 1b 51 23 05 28 43 29 31 39 38:
>12070 36 20 cd 41 52 4b 54 20 26 20 d4 45 43 48 4e 49:
>12080 4b 20 d6 45 52 4c 41 47 0d 1b 51 23 0b c1 4b 54:
>12090 49 45 4e 47 45 53 45 4c 4c 53 43 48 41 46 54 1b:
>120a0 51 0d 1b 51 0d 00 a9 20 20 cd 35 ca d0 fa 60 a0:
>120b0 00 b9 2e 20 f0 1d c9 23 d0 13 24 d7 10 05 a2 14:
>120c0 20 a6 20 c8 be 2e 20 20 a6 20 c8 d0 e4 20 cd 35:
>120d0 c8 d0 de 60 20 1f 21 a9 3e 20 cd 35 20 18 36 a0:
>120e0 00 f0 05 a9 20 20 cd 35 20 55 22 20 2c 36 c8 cc:
>120f0 4d 21 90 ef c0 08 90 22 a9 3a 20 cd 35 a9 12 20:
>12100 aa 22 a0 00 20 55 22 48 29 7f c9 20 68 b0 02 a9:
>12110 2e 20 cd 35 c8 cc 4d 21 90 ea a9 0d 4c cd 35 a5:
>12120 e7 38 e5 e6 c5 ee d0 03 a9 ff 2c a9 00 8d 4f 21:
>12130 aa e8 24 d7 10 02 e8 e8 bd 45 21 8d 4d 21 bd 49:
>12140 21 8d 4e 21 60 08 04 10 08 20 10 40 20 00 00 00:
>12150 20 1f 21 a9 3c 20 cd 35 20 18 36 a9 12 20 aa 22:
>12160 a0 00 20 55 22 48 29 7f c9 20 68 b0 02 a9 2e 20:
>12170 cd 35 c8 cc 4e 21 90 ea 4c 1a 21 20 8b 23 1b 51:
>12180 00 a9 2e 20 cd 35 20 18 36 20 1f 21 20 80 26 24:
>12190 d7 30 05 2c 4f 21 10 22 a0 00 ae b7 25 a9 20 20:
>121a0 cd 35 b9 ba 25 20 2c 36 ca f0 03 c8 d0 ef c0 03:
>121b0 f0 08 a2 03 20 a6 20 c8 d0 f4 a9 20 20 cd 35 4c:
>121c0 d7 27 85 06 86 07 84 08 ad 00 ff 8d de 02 a9 0f:
>121d0 85 02 60 a4 08 a6 07 a5 05 48 a5 06 28 60 20 c2:
>121e0 21 a9 e9 a2 b8 85 04 86 03 20 cd 02 4c d3 21 20:
>121f0 c2 21 a9 e7 a2 b8 85 04 86 03 20 cd 02 4c d3 21:
>12200 20 c2 21 a9 2a a2 b1 85 04 86 03 20 cd 02 4c d3:
>12210 21 20 c2 21 a9 58 a2 cb 85 04 86 03 20 cd 02 4c:
>12220 d3 21 20 c2 21 a9 ce a2 b7 85 04 86 03 20 cd 02:
>12230 4c d3 21 20 c2 21 a9 5c a2 c1 85 04 86 03 20 cd:
>12240 02 4c d3 21 20 c2 21 a9 01 a2 b9 85 04 86 03 20:
```

Ich darf Ihnen schon einmal verraten, daß der Inhalt des Moduls nur bis zur Speicherstelle $2051 einschließlich reicht; den Rest des Ausdrucks können Sie also ignorieren; hier steht noch irgend etwas aus vorangegangenen Arbeitsgängen.

Die ersten beiden Bytes des Moduls ab $2000 sind Nullbytes, wie Sie sehen. Hinzu kommen zwei weitere Nullbytes, die Sie nicht sehen können, die beim Laden des Moduls durch den Monitor als Startadresse des PRG-Files interpretiert wurden. Dies ist bei jedem Modul der Fall und dient einem reibungslosen Kopieren der Module, das als Folge der insgesamt vier Nullbytes durch ein einfaches »dload« und anschließendes »dsave« erfolgen kann. Sie können Module also auf die einfachste aller denkbaren Art und Weisen von einer Diskette auf eine andere bringen; das ist schon einmal eine wichtige Eigenschaft.

Die vier auf die Nullbytes folgenden Bytes stellen zwei Adressen im Format Low-/Highbyte dar: Die erste Adresse bezeichnet die Startadresse der Assemblierung bei der Modulerzeugung, die zweite Adresse enthält entsprechend die bei der Assemblierung erreichte Endadresse. In unserem Beispiel enthalten die Adressen die Werte $c00 und $c0b – als Bytefolgen »00 0c« und »0b 0c«.

Schauen Sie jetzt einmal auf die rechte Seite des Monitorausdrucks mit den ASCII-Äquivalenten der links stehenden Bytewerte. Sie sehen dort, daß der String »start« folgt. »start« aber ist eine Variable aus unserem Beispielprogramm: Der Assembler hat die Variable samt ihrem Wert und den anderen Komponenten, die der ASE für die Beschreibung einer Variablen benutzt, mit in das Modul aufgenommen! Die Frage nach dem Warum ist leicht erklärbar: Dazu muß man sich lediglich fragen, welche Informationen benötigt werden, um aus dem Modul ein Maschinenprogramm mit beliebiger Startadresse erzeugen zu können. Betrachten Sie die Quelltextzeile

```
140    -speicher        .byte >(start)
```

und beachten Sie gleichzeitig, daß »start« ein interner Label ist, dessen Wert sich mit der Startadresse des Programms ändern muß, so wissen Sie bereits den Grund, warum die Variable in das Modul aufgenommen wurde.

Ein Variablendeskriptor innerhalb eines Moduls ist folgendermaßen aufgebaut: Ein führendes Nullbyte zeigt an, daß überhaupt ein Variablendeskriptor folgt. Sobald an der ersten Stelle kein Nullbyte mehr folgt, zeigt dies an, daß dort der Modulrumpf mit dem eigentlichen Modulinhalt folgt. Hinter diesem Nullbyte folgen zwei Bytes, die die sogenannte Ordnungszahl der Variablen enthalten. Diese Ordnungszahl gibt an, in welchem lokalen Block die Variable oder der Label angelegt wurde – es ist ja durchaus denkbar, daß zwei lokale Label des Namens »start« für eine Verschiebung der Startadresse notwendig sind. Genauso könnte übrigens auch der Fall auftreten, daß eine mit ».define« definierte Variable für die Verschiebung benötigt wird. Ein Beispiel hierfür wäre eine Quelltextzeile

```
speicher                .byte kons+>(start)
```

In einem solchen Fall würde auch die Variable »kons« mit in das Variablenfeld des Moduls aufgenommen werden. In unserem Beispiel ist die Ordnungszahl der Variablen »start« gleich Null – wir haben in unserem Beispiel ja von lokalen Blöcken keinen Gebrauch gemacht.

Hinter der Ordnungszahl folgt der Variablenname selbst, dessen Ende durch ein Nullbyte gekennzeichnet ist. Dahinter wiederum folgt der Variablenwert, der der Variablen während der Modulerzeugung zugewiesen wurde – wieder ist die Reihenfolge Low-/Highbyte. Im Beispiel erhielt die Variable »start« während der Assemblierung den Wert $c03, wie man auch nach der Assemblierung überprüfen kann.

Hinter Variablennamen und -wert folgen zwei Bytes, die intern durch den ASE benutzt werden: Das erste Byte ist der Variablenstatus, der unter anderem die wichtige Information enthält, ob die Variable eventuell global ist. Dahinter wieder folgt die Modulnummer, die im Zusammenhang mit den einfachen Modulen, die hier besprochen werden, keine weitere Bedeutung besitzt. Beide Bytes sind in unserem Beispiel gleich Null.

Bei $2013 finden Sie das gleiche Spiel noch einmal für die Variable »speicher«, denn auch diese Variable bzw. dieser Label wird für die Verschiebung benötigt – man betrachte die Quelltextzeile »170 – lda #>(speicher)«. Der Label »uprog« erscheint nicht im Variablenfeld des Moduls. Der Grund hierfür ist, daß »uprog« im Quelltext so verwendet wird, daß die Befehle, in denen »uprog« erscheint, auch ohne die Kenntnis der Variablen an eine neue Startadresse angepaßt werden können. Beispiel hierfür ist der Befehl »jsr uprog«: Die Adressierung des Befehls »jsr« ergibt sich bei einer anderen Startadresse aus der Summe der bei der Modulerzeugung erstellten Adressierung und der Differenz zwischen alter und neuer Startadresse. Ein solch einfacher Zusammenhang läßt sich in den Befehlen ».byte >« und »lda #>« nicht herstellen, deswegen muß zur Relozierung dieser Adressierungen der Wert der verwendeten Variablen bekannt sein.

Zurück zum Beispielmodul: An der Adresse $2022 finden Sie das letzte Byte des Variablendekriptors der Variablen »speicher«. Dahinter folgt nun aber kein weiteres Nullbyte mehr, das auf einen weiteren folgenden Deskriptor hinweisen würde, sondern die Bytes »$5f« und »$0f«. Diese beiden Bytes sind reservierte Bytes, die den Beginn zweier weiterer Tabellen innerhalb des Modulkopfes kennzeichnen. Die Tabellen spielen allerdings in diesem Zusammenhang keine Rolle, sie sind leer. Hinter dem »$0f« folgt deswegen direkt der Modulrumpf.

Der Modulrumpf enthält den eigentlichen Programmkode, besser gesagt, die Informationen, die benötigt werden, um den Kode des Maschinenprogramms für eine beliebige Startadresse innerhalb eines Passes herzustellen.

Das Ganze beginnt wieder mit einem reservierten Byte: Das Byte »$1f« zeigt an, daß an dieser Stelle ein Tabellenbereich mit Bytewerten beginnt – wenn Sie einmal in den Quelltext schauen, sehen Sie, der Text beginnt mit dem Befehl ».byte«. Das hinter »$1f« folgende Byte gibt an, wieviele Bytes der Tabellenbereich umfaßt. Die folgende Null zeigt also hier an, daß der Tabellenbereich leer ist – ein Spezialfall, der sich dann ergibt, wenn die Tabelle nur aus einem Wert besteht, der bei veränderter Startadresse angepaßt werden muß.

In einem solchen Fall tritt das reservierte Byte »$6f« auf den Plan: Hinter »$6f« folgt ein Byteausdruck, ein Ausdruck, der einen Wert zwischen 0 und 255 besitzt und erst während der Relozierung berechnet wird. Die ersten beiden Bytes hinter »$6f« sind die Ordnungszahl während der Berechnung des Ausdrucks bei der Modulerzeugung, dahinter folgt der Ausdruck selbst als ASCII-String. Schauen Sie wieder auf die rechte Seite des Monitorprotokolls – hier sehen Sie den Ausdruck »>(start)«, genau

jenen arithmetischen Ausdruck, der während der Modulerzeugung im Pseudoop ».byte« erscheint. Dieser Ausdruck wird während der Relozierung bewertet und als einzelnes Byte in den resultierenden Maschinenkode übernommen. Bei »normalen« Bytetabellen mit dem Kennzeichen »$1f« genügt es dagegen, die hinter der Angabe der Anzahl der Tabellenbytes aufgeführten Bytewerte unverändert zu übernehmen.

Tabellen von Adreßwerten haben einen ganz entsprechenden Aufbau: Das reservierte Byte »$2f« leitet einen Bereich von konstanten Adreßwerten ein, die während der Relozierung durch einfaches Umrechnen auf die neue Startadresse angepaßt werden – der Mechanismus entspricht dem, der auch bei der Relozierung absoluter Adressierungen (»uprog«) angewandt wird. Hinter dem »$2f« folgt die Anzahl der Bytes in der Worttabelle, dahinter folgen die einzelnen Einträge der Tabelle. Enthält eine solche Tabelle Elemente, die nicht durch einfache Anpassung ermittelt werden können, so erscheint das reservierte Byte »$7f«, dahinter wieder der während der Relozierung zu berechnende Ausdruck, dessen Wert in das resultierende Maschinenprogramm übernommen wird.

Damit sind die Tabellenbereiche abgeschlossen. In unserem Beispiel folgt auf die Zeile mit dem ».word« der erste Prozessorbefehl – »lda #0«. Diesen Befehl finden wir in normal assemblierter Form auch im Modul wieder: $2039 enthält den Maschinenkode »a9 00«. Dieser Kode wird unverändert auch in das resultierende Programm übernommen.

Dahinter folgen wiederum reservierte Bytes; diesmal sind es gleich drei: die drei »$0f«'s an den Adressen $20ab bis $20ac. Diese Bytes leiten wieder einen Ausdruck ein; diesmal handelt es sich um einen Ausdruck, der in die Adressierung eines Prozessorbefehls eingefügt werden muß. Der Ausdruck wird wie sonst auch während der Relozierung bewertet, anschließend werden die dem Ausdruck folgenden Bytes gelesen, die den während der Modulerzeugung erstellten Maschinenkode enthalten. In unserem Beispiel folgen hier die beiden Bytes »a9 0c«; für das Byte »0c« wird während der Relozierung der Wert des vorstehenden Ausdrucks eingefügt.

Der Rest des Moduls besteht entweder aus konstantem Kode, der unabhängig von der Startadresse des Maschinenprogramms ist, oder er kann durch einfache Berechnung an eine Startadresse angepaßt werden, die nicht der bei der Modulerzeugung entspricht.

Einschränkungen:

Eine Einschränkung ergibt sich aus dem Prinzip, nach dem relokatible Module hergestellt werden, für die Erzeugung der Module; diese Einschränkung tritt nur dann in Kraft, wenn Sie – wie es auf dem C-128 fast unvermeidlich ist – Routinen aus den ROMs in ihren Maschinenprogrammen verwenden wollen.

Nehmen wir einmal den Fall an, Sie wollen die fiktive ROM-Routine $4123 aus dem unteren Basic-ROM benutzen, gleichzeitig soll Ihr Programm aber teilweise unterhalb des unteren Basic-ROMs liegen, so daß die Adresse $4123 innerhalb Ihres Maschinenprogramms liegt. Der Aufruf der ROM-Routine soll weiterhin von dem Teil Ihres Programms erfolgen, der nicht unter dem ROM liegt, so daß für den Aufruf ein »jsr $4123« benutzt würde – sicher ein an den Haaren herbeigezogener Fall, der allerdings erwähnt sein sollte. Erzeugen Sie nun ein Modul aus diesem Programm, so nimmt der Assembler beim Aufbau des Moduls an, die Adresse $4123 sei eine interne Adresse. Interne Adressen werden aber während der Relozierung an die neue Startadresse angepaßt (s. »jsr uprog« im obigen Beispiel). Somit wird auch der Aufruf der ROM-Routine verändert – der Assembler ist grundsätzlich nicht in der Lage, zwischen ROM-Routinen und internen Adressen zu unterscheiden. Ausweg aus diesem Dilemma bietet die Wahl einer geeigneten Startadresse bei der Modulerzeugung – dies ist eigentlich der einzige Grund, warum bei der Modulerzeugung die mit ».base« festgelegte Startadresse überhaupt eine Rolle spielt. Die Startadresse muß so gewählt werden, daß Fälle wie der oben beschriebene gar nicht erst auftreten können. Ich empfehle deshalb die Wahl einer konstanten Startadresse $100 während der Modulerzeugung, diese Empfehlung ist aber wieder in keiner Weise verbindlich, sonst hätte ich ».base« gleich bei der Modulerzeugung verboten. Vielmehr gehe ich davon aus, daß die Top-Ass-Programme auch von »highly sophisticated« (Deutsch: ausgefuchsten) Usern verwendet werden, die eventuell noch einige interessante Effekte in den Grenzbereichen der Programme entdecken könnten.

Ansonsten sind keinerlei weitere Einschränkungen zu beachten, die Erzeugung reiner relokatibler Module kann so vor sich gehen, wie jede ganz normale Assemblierung. Das »rein« muß hier betont werden, denn – wie später noch zu sehen – gibt es auch andere Modularten.

3.3.11.3 Laden von relokatiblen Modulen

Nachdem wir nun den Aufbau und die Erzeugung relokatibler Module kennengelernt haben, stellt sich folgerichtig als nächstes die Frage, mit welchen Programmen die Relozierung der Module denn vorgenommen werden kann.

Hierfür stehen Ihnen im Rahmen der Top-Ass-Programme zwei Möglichkeiten zur Verfügung:

- der Lader für Module namens VLO
- der ASE-Linker

3.3.11.3.1 Laden mit dem Loader VLO

Der Loader für relokatible Module »VLO« ist ein eigenständiges Programm, das unabhängig vom ASE-Assembler betrieben werden kann. Es wird im Rahmen der

Beschreibung der ASE-Bestandteile beschrieben, da der VLO im Grunde ein aus dem Assembler herausgeführter Spezialfall des ASE-Linkers ist. Ein weiterer Grund ist natürlich, daß der VLO thematisch hierher gehört.

Der Loader liegt Ihnen auf der Masterdisk in Form zweier Files vor, die die Namen »vlo2000« und »vlo.rel« tragen. »vlo2000« ist ein Maschinenprogramm mit der sedezimalen Startadresse $2000, »vlo.rel« ist das gleiche Programm als relokatibles Modul. Der VLO ist so programmiert, daß er in beiden RAM-Bänken gestartet werden kann, dies ist keine Eigenschaft der relokatiblen Module, sondern ergibt sich einfach aus der Tatsache, daß diese Eigenschaft schon bei der Programmierung so geplant wurde. Eine Einschränkung muß für die Startadresse des Laders innerhalb der RAM-Bänke gemacht werden: Diese darf nicht oberhalb von $afcd in der Bank 0 liegen, da das Programm ansonsten teilweise unter dem Kernal-ROM liegen würde. Dieser Fall wurde bei der Programmierung nicht berücksichtigt, da außerhalb des Bereiches des Kernal-ROMs ja noch genügend anderer Platz vorhanden ist. In der RAM-Bank 1 darf die Startadresse nicht über $6fda liegen; in der Bank 1 unterhält der VLO ein Variablenfeld wie der ASE-Assembler; liegt er oberhalb $6fda, so ragt ein Teil des VLO in das Variablenfeld, was mit großer Wahrscheinlichkeit zum Programmabsturz führen wird. Verwenden Sie eine Startadresse innerhalb der Bank 1, die oberhalb $59cd liegt, und halten gleichzeitig den Assembler aktiv, so müssen Sie berücksichtigen, daß die Userstacks des Assemblers bei einer folgenden Assemblierung den VLO wenigstens teilweise überschreiben werden. Nebenher fallen die Adressen in der RAM-Bank 0, die der Assembler einnimmt, sofern er eben aktiv ist, natürlich ebenfalls als potentielle Standorte für den VLO fort.

Die Bedienung des Laders werden Sie am besten kennenlernen, wenn Sie das Programm einmal starten. Verlassen Sie hierzu als erstes den Assembler durch den Befehl ».x«. Danach laden Sie den Loader entweder durch den Basicbefehl

```
boot "vlo2000"
```

falls Sie eines der neuen schnellen Laufwerke besitzen, ansonsten bewirken Sie das gleiche auch mit der Befehlsfolge

```
bload "vlo2000"
bank 0: sys dec("2000")
```

Daraufhin erscheint die Copyright-Meldung des VLO auf dem Bildschirm:

```
-----  VLO v2.1  Module Loader  -----
(C) 1986 Gerd Moellmann, M & T Verlag
          Aktiengesellschaft
        Alle Rechte vorbehalten
```

Darunter erscheint schon gleich eine Frage:

```
load from unit number:
```

Geben Sie hier die Gerätenummer des Floppylaufwerks an, auf der sich das zu ladende Modul befindet, also beispielsweise die 8. Schließen Sie die Eingabe mit »Return« ab, sodann wird unter der letzten Eingabezeile der Name des zu ladenden Moduls erfragt:

```
module name:
```

Als Beispiel könnten Sie hier einmal »vlo.rel« eingeben, ein Modul, das sich mit Sicherheit auf der Masterdisk befindet und bei dem wir leicht nachprüfen können, ob das Ganze auch funktioniert. Nachdem Sie den Modulnamen eingegeben haben, prüft der VLO zunächst, ob sich ein File dieses Namens auf der Diskette befindet. Ist dies der Fall, erscheint die Meldung des Floppyfehlerkanals auf dem Bildschirm:

```
00, ok, 00, 00
```

und Sie müssen die Startadresse für das zu ladende Programm eingeben:

```
new start address (no bank):
```

Der Zusatz »no bank« soll darauf hinweisen, daß an dieser Stelle die Eingabe der reinen Adresse ohne irgendwelche Banknummern usw. erwartet wird. Geben Sie zum Ausprobieren einmal $a000 ein. Nun werden Sie aufgefordert, die RAM-Bank anzugeben, in der diese Adresse liegt:

```
RAM bank (0,1,...):
```

Wie aus der Ausgabezeile schon ersichtlich, wird hier die reine Banknummer angegeben, kein Konfigurationsindex. Für unsere Beispieladresse $a000 bleibt nur die RAM-Bank 0 zu wählen, vergleichen Sie die Angaben weiter oben. Jetzt beginnt der VLO mit dem Relozieren des angegebenen Moduls; es erscheint die Meldung

```
loading...
```

Sobald der VLO seine Arbeit beendet hat, erhalten Sie dann eine weitere Meldung, die einige Informationen enthält:

```
old origin was $2000
code now from $a000 to $b033 in bank 0
```

Die erste Angabe enthält die Startadresse, die bei der Modulerzeugung verwendet wurde, darunter ersehen Sie, wohin der VLO das Modul geladen hat – besser gesagt, wo der VLO den aus dem Modul gewonnenen Maschinenkode plaziert hat.

Unterhalb der Endmeldung des VLO sehen Sie die Frage

```
start module (y/n)?
```

Drücken Sie »y« für »yes«, wird das eben erzeugte Maschinenprogramm an der Startadresse des Maschinenkodes angesprungen, also gestartet, falls sich der Startpunkt des Maschinenprogramms am Kodestart befindet. In diesem Fall müssen Sie die Frage nach dem Index der Konfiguration beantworten, unter der das Programm angesprungen werden soll. Soll beim Start nur RAM eingeschaltet sein, wählen Sie die 0 oder die 1 – je nachdem, in welche Bank Sie geladen haben. In diesen Konfigurationen werden auch DBM und VLO gestartet, wenn Sie deren Module geladen haben. Drücken Sie »n« für »no« erhalten Sie die Möglichkeit, weitere Module zu laden:

```
new run (y/n)?
```

Betätigen Sie »y«, wird der VLO neu gestartet, wählen Sie die Taste »n«, ist das Programm beendet. Sie befinden sich dann entweder wieder im Basic 7.0 oder im Assembler. Spielen Sie das Ganze einmal durch. Wenn Sie jeweils den gerade geladenen Loader bei der entsprechenden Frage des VLO starten lassen, können Sie wunderbar verfolgen, daß der VLO erstens in beiden Bänken lauffähig ist, und zweitens, daß das Relozieren tatsächlich funktioniert.

Auch den im nächsten Kapitel behandelten DBM-Monitor können Sie relozieren; der Filename ist »dbm.rel«.

Fehlerbehandlung:

Beim Laden von Modulen können selbstverständlich Fehlersituationen auftreten; hierbei meine ich nicht das Relozieren selbst, sondern andere Gegebenheiten:

Als erstes: Was geschieht, wenn Sie einen falschen Modulnamen eingegeben haben und der VLO das File nicht finden kann, das er laden soll? In diesem Fall wird die Fehlermeldung des Fehlerkanals der Floppy angezeigt, beispielsweise

```
62, file not found, 00, 00
```

und Sie landen gleich bei der Frage »new run (y/n)?«. Sie können dann den Filenamen in einem neuen Anlauf richtig eingeben oder das Programm verlassen, um sich erst einmal das Directory anzusehen.

Die weiteren Fehlermeldungen des VLO:

(1) »use of external references«. Der Fehler wird während des Ladens eines Moduls ausgegeben, falls das Modul Verweise auf externe Referenzen aufweist. Was externe Referenzen sind, wird erst in den folgenden Kapiteln erläutert, deshalb soll die Fehlermeldung erst einmal so stehenbleiben.

(2) »mixed modules«. Dieser Fehler würde auftreten, wenn das zu ladende Modul aus mehreren aneinandergehängten Einzelmodulen bestünde. Mit den gegenwärtigen Versionen der Top-Ass-Programme können Sie solche Module nicht erzeugen, allerdings wäre bei späteren Versionen eine Erweiterung denkbar, auf die sich diese Fehlermeldung schon einmal einrichtet.

(3) »modul format«. Es wurde versucht, entweder einen File zu laden, der überhaupt kein Modul ist, oder es handelt sich beispielsweise um ein Modul, das mit einem ASE-Assembler der Version 1.0 erzeugt wurde. Die Version 2.0 verwendet ein anderes – allgemeineres – Format für die relokatiblen Module. Dieses Format ist schon auf künftige Erweiterungen ausgerichtet.

(4) »resulting jmp ($xxff)«: Bei der Relozierung des Moduls ergab sich ein indirekter Sprungbefehl über einen Vektor, dessen beide Bytes in unterschiedlichen Speicherseiten liegen. Wie Sie in Kapitel 1 erfahren haben werden, kann der Prozessor infolge eines Fehlers in seiner Architektur solche Sprünge nicht korrekt ausführen, deshalb gibt der VLO in diesem Fall einen Fehler aus.

(5) »unknown label«. Bei der Bewertung eines Ausdrucks innerhalb des Moduls war ein Label oder eine Variable nicht bekannt. Er müßte also folglich im Modulkopf nicht aufgeführt sein. Hierbei handelt es sich eigentlich um einen theoretischen Fehler – ich kann mir keine Situation vorstellen, in der dies der Fall sein könnte, es sei denn, Sie erzeugen ein Modul im ASE-Format mit einem anderen Programm als mit dem Assembler.

(6) »too many labels«. Der Variablenraum des VLO – über 2000 Label oder Variablen wie beim Assembler – ist voll. Auch dies ist mehr ein theoretischer Fehler. Um bei der Relozierung ca. 2000 Label benutzen zu müssen, müßte das relozierte Programm wohl länger als 64 KByte sein.

(7) »i/o error«. Bei der Übertragung zwischen Rechner und Floppy ist ein Fehler aufgetreten. Dazu müßte beispielsweise ein von der Floppy gelesener Sektor beschädigt sein oder ähnliches.

(8) »suggested i/o or wrong input«. Entweder haben Sie eine ungültige Eingabe am Bildschirm gemacht, oder es handelt sich wieder um einen Übertragungsfehler. Bei Eingabefehlern sollte es sich in den meisten Fällen um die falsche Eingabe eines arithmetischen Ausdrucks handeln, der in einzelnen Eingabefeldern erwartet wird. Bei diesen arithmetischen Ausdrücken handelt es sich um Ausdrücke, wie Sie auch der Assembler annimmt, also: Zahlen dezimal oder sedezimal, Label, sofern definiert, alles verknüpft mit den bekannten Operatoren.

3.3.11.3.2 Relozieren mit dem ASE-Linker

Falls der Assembler gerade aktiviert ist, wenn Sie ein File in den Speicher relozieren wollen, brauchen Sie den VLO nicht erst zu laden. Wie bereits weiter oben gesagt wurde, ist der VLO lediglich ein Subset des sowieso in den ASE integrierten Linkers. Also kann man den ASE-Linker ebensogut – in einigen Fällen auch besser – für die Relozierung einsetzen wie den VLO.

Die Bedienung des ASE-Linkers ist grundsätzlich verschieden von der des Laders. Die Steuerung des Linkers erfolgt nicht dadurch, daß sich das Programm die benötigten Angaben am Bildschirm erfragt, sondern über einen **Quelltext**, der alle Angaben enthält! Das Linken wird wie eine normale Assemblierung mit »run« gestartet – obwohl dort natürlich zum Teil völlig andersartige Aktionen ausgelöst werden. Der Quelltext zur Steuerung des Linkers unterscheidet sich grundsätzlich nicht von normalen Quelltexten. Sämtliche Befehle, die während der normalen Assemblierung eingesetzt werden können, können auch im Link-Quelltext benutzt werden. Diese Eigenschaft ist für die reine Relozierung von Modulen noch nicht so wichtig, wird es jedoch, wenn Sie die weiterführenden Möglichkeiten des Linkers und der Module kennengelernt haben.

Im hier interessierenden Fall sind für einen ersten Test nur die folgenden Pseudoops von Bedeutung:

- ».base« – legt fest, welche Startadresse der bei der Relozierung erzeugte Maschinenkode haben soll.
- ».code« – schreibt vor, wohin der Kode zu schreiben ist – wie bei der Assemblierung.
- ».object« – kann den Kode direkt zur Floppy leiten. Diese Möglichkeit haben Sie beim VLO nicht. Sie können also aus einem Modul ein Maschinenprogramm auf Diskette mit einer festen Startadresse machen, falls Sie ein Programm mit dieser Startadresse häufiger benötigen. Auf diese Weise könnten Sie sich beispielsweise verschiedene Versionen des DBM-Monitors herstellen.
- ».link "modulname",u« – gibt dem Linker bekannt, welches Modul reloziert werden soll. Das »,u« ist wie immer die Gerätenummer der Floppy und kann fehlen, wenn vom Gerät 8 geladen werden soll.

Starten wir den Linker für einen ersten Versuch; geben Sie hierzu den folgenden Quelltext ein:

```
100   -.base $2100
110   -;
120   -.object "vlo2100.obj"
130   -;
140   -.link "vlo.rel"
```

Durch diese wenigen Zeilen wird aus dem Modul »vlo.rel« ein Maschinenprogramm namens »vlo2100.obj« hergestellt, das auf der Floppy #8 abgelegt wird.

Starten Sie nun den ASE durch »run«. Sie hören unmittelbar darauf, daß der Assembler/Linker auf die Diskette zugreift, ferner sehen Sie nacheinander die beiden Meldungen

```
phase 1 ---- vlo.rel
phase 2 ---- vlo.rel linkage address = $2100
```

Anschließend erscheint die bekannte Endmeldung des Assemblers:

```
end of assembly at 0:19.7

origin is $2000
code from $2100 to $3133 in bank 0
4147 Bytes total, 10 Labels used
```

Die ersten beiden Meldungen zeigen, daß der Linker anders als der VLO in zwei Passes vorgeht – im ersten Durchlauf lädt er die Modulköpfe, falls mehrere Module zusammengeladen werden sollen, was bei der reinen Relozierung nicht der Fall ist. Beim Laden der Modulköpfe werden die in den Köpfen verwahrten Label und Variablen, die für die Relozierung benötigt werden, in das Variablenfeld des Assemblers gebracht. Sie können sich diese Label – hier sind es nur zehn – nach Abschluß des Linkens durch den Dumpbefehl ».d« ansehen. In der zweiten Phase werden die Modulrümpfe – hier ist es nur einer – geladen und der resultierende Koder erzeugt. Start- und Endadresse des erzeugten Maschinenkodes sowie die Anzahl der erzeugten Bytes können Sie der Endmeldung nach der Assemblierung entnehmen.

Als Ergebnis der Relozierung erhalten Sie nach ca. 20 Sekunden ein File auf Diskette, das Sie anschließend weiterverwenden können.

Der folgende Quelltext liegt auch auf der Masterdiskette (»vlodbm.mke«) vor und dient dazu, aus den VLO- und DBM-Modulen Programmversionen mit einer frei wählbaren Startadresse zu machen:

```
100 -;=============================================
110 -; Erzeugung von VLO- und DBM-Versionen aus den
120 -; relokatiblen Modulen
130 -;=============================================
140 -;
150 -.define d     = 1
160 -.define v     = 2
170 -;
171 -.cond p1
180 -         .print "VLO- und DBM-Relozierung"
190 -         .print "========================"
```

```
200 -          .print
210 -.econd
210 -.request "VLO oder DBM (v/d): ",choice
211 -.cond (choice.lt.d).or.(choice.gt.v)
212 -          .print "falsche Eingabe!"
213 -          .print "try again..."
214 -          .break
215 -.econd
220 -;
230 -.request "Startadresse: ",org
240 -;
250 -.control choice
260 -. = d
270 -          .object "dbm xxxx.obj"
280 -          .link "dbm.rel"
290 -. = v
300 -          .object "vlo xxxx.obj"
310 -          .link "vlo.rel"
320 -. -
```

Beachten Sie besonders die Möglichkeiten, die z. B. der Einsatz der bedingten Assemblierung auch innerhalb von Link-Quelltexten bietet. Prinzipiell ist im Link-Quelltext alles erlaubt, was auch in anderen Quelltexten erlaubt ist. Ausnahme bildet hier der Pseudoop ».modul« – es ist nicht möglich, aus einem Modul ein anderes Modul herzustellen, der Assembler quittiert den Versuch mit einer Fehlermeldung. Insbesondere ist auch eine Kodeerzeugung vor oder nach der Relozierung möglich. Diese Möglichkeit habe ich erst einmal offengehalten, da sie sich vielleicht einmal als nützlich erweisen könnte, obwohl ich zur Zeit keine konkrete Anwendungsmöglichkeit dafür angeben könnte. Hier ist also wieder einmal die Phantasie des Anwenders gefragt.

3.3.11.4 Entries und externe Referenzen

Die bisher geschilderten Eigenschaften relokatibler Module bringen noch nicht allzu viel für die eigentliche Programmierarbeit, wie dies weiter vorn versprochen wurde. Sicher ist es nützlich, Programme schreiben zu können, die vom Lader oder vom Linker an beliebige Adressen verschoben werden können, da man sich die Herstellung vieler verschiedener Programmversionen mit verschiedenen Startadressen sparen kann, allerdings ist dies doch noch eher eine Spielerei, die bei 90 Prozent der Programme nichts an Arbeit spart.

Eine wirkliche Erleichterung der Programmierung und Zeitersparnis erhält man erst, wenn man die reinen relokatiblen Module als Grundlage für einen weiteren Ausbau

des Prinzips benutzt. Diese Erweiterung stellen die externen Referenzen und die Entries dar.

Um die Bedeutung der beiden Begriffe zu verstehen, vollziehen wir doch einmal nach, wie man überhaupt auf die Idee einer Erweiterung der relokatiblen Module kommen kann:

Was haben wir bisher durch die Module zur Verfügung? Wir können Programme schreiben, die sich durch Linker oder Lader an beliebige Adressen relozieren lassen. Eigentlich ist es dann doch naheliegend, diese Eigenschaft auszunützen, um sich ein Gesamtprogramm aus Modulbausteinen zusammenzubauen, die einfach hintereinander geladen werden – die Endadresse des jeweils vorangehenden Moduls wird als Ladeadresse für das nächste Modul benutzt. Diese Vorgehensweise bringt Vorteile mit sich: Die einzelnen Bausteine des Gesamtprogramms können unabhängig voneinander entwickelt, assembliert und getestet werden – dies folgt dem Prinzip der sogenannten »modularen Programmierung«, die ich ein wenig näher beschreiben möchte:

Sobald Sie Programme schreiben, die eine gewisse Größe überschreiten, werden Sie feststellen, daß Sie mit der Formulierung des Programms in Form von Flußdiagrammen oder Nassi-Schneidermann-Diagrammen nicht mehr sehr weit kommen werden. Die im Programm zu lösenden Aufgaben sind meist zu umfang-reich und komplex, um in Form eines einzelnen Diagramms aufgezeichnet werden zu können. In einem solchen Fall hilft nur noch, die Aufgabe des Programms in mehrere Teilaufgaben (Programm-Module) zu teilen, die unabhängig voneinander formuliert werden – ein Modul beispielsweise für die anstehenden Programm-Eingaben, ein anderes Modul für die Ausgaben usw.

Die einzelnen Module können dann wieder in Form von Flußdiagrammen oder mit vergleichbaren Methoden angegangen werden. Da die Module kleiner sind als das Gesamtprogramm, lassen Sie sich leichter schreiben und schneller assemblieren und austesten als das Gesamtprogramm. Ein ganz anderer Punkt kommt noch hinzu: Ist ein Modul allgemein genug formuliert, besteht die Chance, es auch in anderen Programmen einsetzen zu können.

Insgesamt eine Menge von Vorteilen, die für eine modulare Programmierung sprechen – nicht umsonst hat sich dieses Prinzip auf anderen Rechnern schon lang auch auf Assemblerebene durchgesetzt.

Betrachten wir jetzt allerdings unsere bis dato zur Verfügung stehenden reinen relokatiblen Module, stellen wir fest, daß sie zur Erfüllung der für sie vorgesehenen Aufgabe noch nicht optimal gestaltet sind. Wie sollen Routinen beispielsweise aus einem fertigen Ausgabe-Modul von anderen Modulen angesprochen werden? Das fertige Modul liegt ja in seiner endgültigen Form als Maschinenprogramm im Verbund mit den anderen Modulen im Speicher, die Routinen des Moduls liegen an

Adressen, die den anderen Modulen mitgeteilt werden müssen, damit sie in den entsprechenden »jsr«-Befehlen der Module verwendet werden können.

Nun, diese Forderung ist zu erfüllen: Erinnern Sie sich an die beiden Tabellen, die bei der Beschreibung des Modulaufbaus erwähnt wurden? Diese beiden Tabellen waren im obigen Beispiel noch unbenutzt, jetzt werden sie verwendet, um zwei Arten von Labeln festzuhalten:

(1) **Entries**: Entries sind Label aus dem Modul, die während der Modulerzeugung besonders gekennzeichnet wurden. Die Liste der Entries im Modul enthält sowohl die Labelnamen als auch die Adressen der Label, die während der Assemblierung ermittelt wurden. Andere Module können auf die Einsprungpunkte – so könnte man Entries übersetzen – zugreifen, indem sie den gleichen Labelnamen verwenden und diesen Namen gleichzeitig als externe Referenz (Bezugspunkt aus einem anderen Modul) deklarieren.

(2) **Externe Referenzen**: Dies sind Label, die während der Assemblierung eines Moduls noch nicht bekannt sind. Indem man solche Label als externe Referenzen kennzeichnet, bewirkt man, daß der Labelname in eine Liste externer Referenzen aufgenommen wird, die Teil des Modulkopfes ist.

Der Linker ist in der Lage, mehrere Module mit Entries und externen Referenzen zusammenzubinden. Daher stammt auch der Name »Linker«, der etwa »Verbinder« bedeutet. In der ersten Phase des Linkens liest der Linker sämtliche Modulköpfe und sammelt alle Entries auf, die in den Modulen angegeben sind. In der zweiten Phase liest der Linker dann die Modulrümpfe und ist jetzt in der Lage, die Werte der Entries in alle Module einzutragen, in denen sie als externe Referenzen deklariert wurden.

Es wird wieder das beste sein, das Ganze an einem Beispiel nachzuvollziehen:

```
100   -.base $100
110   -;
111   -.extern speicher
112   -;
120   -.modul "modul1.mod"
121   -;
122   -.define _cr        = $ff00
123   -.define _bsout     = $ffd2
130   -;
140   -#uprog         ldx #0
150   -               stx _cr
160   -m1             lda speicher,x
170   -               beq m2
180   -               jsr _bsout
190   -               inx
```

```
200   -               bne m1
210   -m2             rts
```

Dieser Quelltext soll das erste Modul erzeugen; er dient zur Ausgabe eines Textes, der in einem zweiten Modul stehen soll. Der Pseudoop ».extern« deklariert die hinter dem Befehl stehenden Label als externe Referenzen, also Labelwerte, die erst während des Linkens durch den Linker eingefügt werden sollen. Mehrere Label in der Parameterliste werden wie üblich durch Kommata voneinander getrennt. In Zeile 140 sehen Sie, wie ein Label zum Entry gemacht wird, wie also dafür gesorgt wird, daß ein Label samt Wert im Modulkopf erscheint. Dazu genügt es, vor den Label ein Doppelkreuz zu schreiben. Für die Deklaration als Einsprungpunkt existiert nebenher auch ein zu ».extern« entsprechender Befehl ».public«, der aus einem vorher definierten Label in seiner Parameterliste einen Einsprungpunkt macht. Dieser Pseudoop wird benutzt, um mit ».define« definierte Label und Variablen zu Entries machen zu können. Sowohl die Deklaration als externe Referenz als auch die als Einsprungpunkt bewirkt automatisch, daß der Label gleichzeitig global wird. Erzeugen Sie nun das Modul dieses Quelltextes und geben Sie den zweiten Text ein:

```
100   -.base $100
110   -;
120   -.extern uprog
130   -;
140   -.modul "mod2.mod"
150   -;
160   -#speicher     .byte "Linken von Modulen",0
170   -;
180   -start         jmp uprog
```

Das zweite Modul enthält den auszugebenden Text sowie den Startpunkt für das Gesamtprogramm, der einfach aus einem Sprung zum Ausgabeprogramm besteht. Diesmal erscheint der Label »speicher« als Entry im Modulkopf und »uprog« wird erst während des Linkens eingefügt. Im übrigen brauchen die Verweise auf externe Referenzen nicht so einfach auszusehen, wie dies in den beiden Beispielquelltexten der Fall ist, wo der arithmetische Ausdruck lediglich aus dem Labelnamen der Referenz besteht – es können vielmehr beliebige Verknüpfungen verwendet werden! Das Zusammenbinden der beiden Module zu einem Gesamtprogramm erfolgt nun über einen einfachen Link-Quelltext:

```
100 -.base $c00
110 -;
120 -.link "mod1.mod"
130 -.link "mod2.mod"
```

Starten Sie das Linken mit diesem Quelltext, so erhalten Sie das erzeugte Gesamtprogramm an der Adresse $c00 in der RAM-Bank 0. Mit dem Editorbefehl ».d« oder

mit »_« können Sie den aktuellen Wert von »start« abfragen und das Beispielprogramm an der ausgegebenen Adresse starten, um es auszuprobieren.

3.3.11.5 Einschränkungen bei der Modulerzeugung mit externen Referenzen

Während bei der Erzeugung reiner relokatibler Module dem Programmierer keinerlei einschränkende Vorschriften gemacht werden mußten, unterliegt die Verwendung externer Referenzen und Einsprungpunkte einigen Regeln, auf die hier eingegangen werden soll:

(1) Die Reihenfolge der Deklarationen mit ».extern« und ».public« bei der Modulerzeugung ist festgelegt und unterliegt der Überwachung des Assemblers, der eine Fehlermeldung ausgibt, wenn sie nicht eingehalten wird. Halten Sie sich deshalb bitte an das folgende Schema:

```
.base xxxx          ;   Startadresse

.extern ...         ;   Deklarationen
.extern ...         ;      externer
.extern ...         ;      Referenzen

.modul ...          ;   Modulerzeugung...

.define ...         ;   Definitionen
.define ...         ;      von Labeln
.define ...         ;      und Variablen

.public ...         ;   Deklarationen von
.public ...         ;      Entries, die mit
.public ...         ;      .define def. wurden

...                 ;   Kodeerzeugende
...                 ;      Anweisungen
```

(2) Externe Referenzen können nicht als Makroparameter verwendet werden! Der Grund hierfür ergibt sich aus der Art und Weise, wie Parameter an ein Makro übergeben werden: Nicht der Name eines Parameterlabels wird an das Makro weitergereicht, sondern lediglich der Wert. Sehen Sie hierzu noch einmal die Beschreibung der Makros. Demzufolge kann der Assembler innerhalb eines Makros auch keine Ausdrücke im Modulkode erzeugen, die die externe Referenz beinhalten und somit wiederum ist der Linker nicht in der Lage, die Referenz durch Einsetzen des Labelwertes während des Zusammenbaus des Gesamtprogramms aufzulösen. Glücklicherweise ist diese Einschränkung meines Erachtens nicht sehr einschneidend, denn sowohl Einsprungpunkte als auch externe Referenzen sind durch die entsprechende Deklaration automatisch global. Dadurch können sie aus jedem lokalen Block – also auch aus den Makrodefinitionen heraus – angesprochen werden. Eine Erweiterung des Assemblers in

der Hinsicht, diese Einschränkung aufzuheben, ist bei genügender Nachfrage eventuell für die Version 3.0 vorgesehen.

(3) Auch die Verwendung externer Referenzen als Minimacparameter ist eingeschränkt: Referenzen dürfen nur in den folgenden Minimacs verwendet werden:

```
.liax, .liay, .lixy, .mvi, .adi, .sbi und .cpi
```

Sofern diese Minimacs zwei Parameter erfordern, darf nur einer eine externe Referenz sein und **muß** der erste Parameter sein – derjenige, der bei der Kodeerzeugung der Minimacs unmittelbare Adressierungen erhält. Die Minimacs stellen für den Linker ein echtes Problem dar, da der Assembler sie praktisch als eine Befehlserweiterung des Prozessorbefehlsvorrats behandelt. Nur unter relativ großem Aufwand ist es möglich, externe Referenzen in von Minimacs erzeugtem Kode aufzulösen. Ich habe hier entschieden, die obige Einschränkung zu machen, da der Linker ansonsten um ca. 30 Prozent größer würde. Andererseits schien mir der zu betreibende Aufwand zu groß für den erzielten Nutzeffekt. Bei entsprechender Nachfrage ist auch diese Einschränkung in späteren Versionen des ASE-Linkers zu beseitigen.

(4) Zeropageadressen können nicht nachträglich gelinkt werden. Es ist also nicht möglich, einen Label in einem Modul als externe Referenz zu deklarieren und in einem weiteren Modul dann als Zeropageadresse und Entry anzugeben. Diese Einschränkung ist eigentlich nicht wesentlich und soll nur der Vollständigkeit halber erwähnt sein. Es wird stillschweigend davon ausgegangen, daß die durch die Module verwendeten Zeropageadressen bekannt und konstant sind.

Wichtig! Die letzten drei Regeln werden vom Assembler in der augenblicklichen Version nicht überwacht! Der Programmierer ist selbst dafür verantwortlich, die Regeln einzuhalten.

Für die Programmierarbeit bedeutet dies, daß Sie gegebenenfalls überprüfen sollten, ob die Regeln in Ihren Quelltexten eingehalten wurden, falls etwas nicht so klappt, wie Sie es sich vorgestellt haben.

(5) Das schon bei der Relozierung der reinen Module – der Module ohne externe Referenzen – aufgetretene Problem bei der Verwendung von ROM-Routinen tritt auch beim Linken von mehreren Modulen zu einem Gesamtprogramm wieder unverändert auf. Allerdings steht uns jetzt mit den externen Referenzen – zusammen mit der Verwendung eines Link-Quelltextes – eine Möglichkeit zur Verfügung, Komplikationen sicher zu vermeiden. Das Problem sei noch einmal kurz an folgendem Quelltext aufgezeigt:

```
100   -.base $3ff0
110   -;
120   -.modul "beispiel.mod"
130   -;
```

```
140   -start          jsr $4123 ; ROM-Routine
150   -               ...
160   -               ...

end of assembly at x:xx.x

origin is $3ff0
code from $3ff0 to $4200 in bank 0
xxxx bytes total, yyyy labels used
```

Die gezeigte Endmeldung des Assemblers soll darauf hinweisen, daß der erzeugte Maschinenkode die im Quelltext verwendete Adresse $4123 umfaßt. Bei der Modulerzeugung geschieht nun folgendes: Der Assembler hält die Adresse $4123 für die Adresse eines Unterprogramms innerhalb des gerade assemblierten Quelltextes. Er hat keine Möglichkeit, zu erkennen, ob es sich eventuell um eine ROM-Adresse handelt. Ebenso der Linker, der anschließend das erzeugte Modul beim Zusammenbau mit anderen Modulen relozieren muß: Er nimmt automatisch an, die Adresse $4123 müsse wie alle anderen absoluten Adressen, die innerhalb des Moduls liegen, an die neue Startadresse angepaßt werden. Dadurch wird die Adresse im »jsr«-Befehl beim Linken verändert, was natürlich während der Laufzeit des Programms ins Abseits führen muß, wenn es sich um die Adresse einer ROM-Routine handelt.

Konflikte dieser Art können zuverlässig vermieden werden, wenn Sie alle in Frage kommenden ROM-Routinen bei der Modulerzeugung als externe Referenzen deklarieren. Die Definition der Referenzen erfolgt in gar keinem Modul, sondern im Link-Quelltext! Der Quelltext des obigen Beispiels und der zugehörige Link-Quelltext sehen dann etwa wie folgt aus:

```
100   -.base $3ff0
110   -;
111   -.extern  romup
112   -;
120   -.modul "beispiel.mod"
130   -;
140   -start          jsr romup   ; ROM-Routine
150   -               ...
160   -               ...
```

und

```
100   -.base xxxx
110   -;
120   -.define romup := $4123
130   -.public romup
140   -;
```

```
150   -.link "beispiel.mod"
160   -          ...
```

Sie sehen das Verfahren. Gleichzeitig ist dies das Beispiel, das meiner Meinung nach zeigt, warum sich bisher keine Assembler mit Linker auf den bekanntesten Rechnern mit 6502-CPU finden ließen: Diese Rechner besitzen samt und sonders ROM-Bausteine, die einen Teil des vorhandenen RAMs überlappen, womit die obigen Kollisionen zwangsläufig auftreten. Wie Sie sehen, ist eine vernünftige Lösung dieses Problems nur dann gesichert, wenn eine Möglichkeit besteht, den Linker sehr weitgehend zu steuern – hier durch einen Quelltext. Diese umfangreiche Steuerung wirkt sich allerdings entschieden auf die Komplexität des Assemblers und Linkers aus. Bei Standard-Betriebssystemen wie CP/M tritt dieser Problemkreis dagegen überhaupt nicht erst in Erscheinung. CP/M wird von einer Disk in den Rechner geladen, liegt also selbst im RAM; Überlappungen treten prinzipiell nicht auf, wodurch sowohl Assembler als auch Linker einfach gehalten werden können.

3.3.11.6 Abschließende Bemerkungen zum Linker

Seitdem der ASE soweit gediehen war, daß eine Assemblierung mit Modulerzeugung und anschließendem Linken möglich war, bin ich persönlich auch auf dem C-128 voll auf diese Arbeitsweise umgestiegen. Ich wende sie an, sobald die zu entwickelnden Maschinenprogramme eine gewisse Länge von etwa zwei bis drei KByte überschreiten. Der dadurch gewonnene Zeitvorteil gegenüber herkömmlicher Assemblierung ist nicht zu unterschätzen. Als Beispiel kann ich die Entwicklung des ASE selbst anführen, der in einem »Bootstrap« hochgezogen wurde – der ASE wurde mit sich selbst assembliert, wobei jede Erweiterung des Programms zuerst dem ASE selbst zugute kam; gleichzeitig war eine weitgehende Fehlerkontrolle möglich, indem gleich geprüft werden konnte, ob sich der ASE selbst reproduzieren konnte. Als erster Schritt bei der Entwicklung diente dabei ein in Basic formulierter Assembler. Bei einer »normalen« Assemblierung mit Verkettungen und Einbindungen wurde stets – auch unter Verwendung mehrerer Laufwerke – eine Assemblierzeit von ca. 17 Minuten benötigt, die jedesmal anfiel, wenn auch nur ein einziger Befehl im Quelltext geändert wurde – mithin der Hauptgrund, warum ich eine andere Lösung suchte. Nachdem der Assembler in mehrere Module aufgeteilt war, sah die Sache schon ganz anders aus: Für die Assemblierung eines Moduls, dessen Größe ich jeweils so gewählt hatte, daß der Quelltext eines Moduls ganz in den Speicher paßte, brauchte ich lediglich noch ungefähr 30 Sekunden. Das Zusammenbinden der Module zum Assembler dauerte dann noch einmal 50 Sekunden, wobei der Kode ins RAM geschrieben wurde – wenn das kein Gewinn ist!

Der Link-Quelltext bietet zudem noch einige bislang unerforschte Möglichkeiten. Wie bereits vorstehend erwähnt wurde, ist innerhalb des Link-Quelltextes eine ganz normale Assemblierung möglich, insbesondere fällt hierunter auch die bedingte

Assemblierung. Bisher wurden zwar immer nur sehr einfache Link-Quelltexte verwendet, welche Möglichkeiten aber letztendlich im Link-Text stecken, läßt sich zur Zeit noch nicht recht absehen – denken Sie etwa an Verkettungen, Einbindungen, Makros usw. Hier ist sicher noch einiges an Pionierarbeit möglich – vielleicht finden Sie noch die eine oder andere Möglichkeit, die Programmierarbeit entscheidend zu erleichtern.

Zählen Sie zu allem anderen auch noch die Unterstützung der modularen Programmierweise hinzu, ohne die sich kein größeres Programm einigermaßen fehlerfrei schreiben lassen wird, glaube ich, daß die Mehrzahl der Argumente für die Verwendung des Linkers spricht.

Auch in anderer Hinsicht ist die Wahl der relokatiblen Module in Verbindung mit einem anschließenden Linken für Sie von Vorteil: Die Mehrzahl der größeren Rechner mit Standardbetriebssystemen und auch die neuen Rechner mit der CPU MC68000 – genannt seien hier der AMIGA und die ST-Rechner – verwenden ausschließlich Assembler, die nach den gleichen Prinzipien wie der ASE arbeiten: Programme werden in Modulen geschrieben, die von einem Linker zusammengebunden werden müssen, bevor daraus lauffähiger Kode entsteht. Zwar sind die Modulformate dieser Assembler natürlich verschieden vom Format des ASE, das Arbeitsprinzip bleibt jedoch das gleiche. Es ist daher sicher von Vorteil, dieses Prinzip zu beherrschen.

3.3.12 Fehlermeldungen des Assemblers

Bislang wurden die in den verschiedenen Situationen möglichen Fehler und die Art, in der der Assembler die Fehler meldet, nur sporadisch angesprochen. Dafür ist dieses eigene Kapitel vorgesehen, in dem die Fehlermeldungen in alphabetischer Reihenfolge besprochen werden sollen.

Fehlermeldungen, die während der Assemblierung eines Quelltextes ausgegeben werden, werden grundsätzlich zusammen mit der Quelltextzeile angezeigt, in der der Fehler gefunden wurde. Dadurch ist im Fall, daß die Fehlerzeile in einem Quelltext im Speicher steht, eine sofortige Änderung möglich. Falls Fehler in einem eingebundenen Quelltext, im Quelltext einer Makrobibliothek oder schließlich in einer Makrobibliothek auftreten, die in einem eingebundenen Quelltext verwendet wird, werden mehrere Quelltextzeilen gelistet. Die drei Fälle in Beispielen:

```
syntax error in
20   -          .source "beisp.src"
bound line:
100  -          lda '0
```

im Fall einer einfachen Einbindung – »bound line« heißt »eingebundene« Zeile.

```
syntax error in
20  -           ...mac
library line:
100 -           lda '0
```

im Fall, daß der Fehler in einer Makrobibliothek auftritt.

```
syntax error in
20  -           .source "beisp.src"
library line:
200 -           lda '0
bound line:
110 -           .source "beisp.src"
```

im Fall, daß der Quelltext »beisp.src« in den assemblierten Quelltext eingebunden ist und in diesem eingebundenen Quelltext die Makrobibliothek aufgerufen wird, in der schließlich der Fehler gefunden wird. Im Fall eingebundener Texte und im Fall der Bibliotheken sollte man besser nicht versuchen, den Fehler direkt in der angezeigten Fehlerzeile zu verbessern – die korrigierte Zeile würde an den falschen Ort übernommen werden, falls das noch ausdrücklich erwähnt werden muß.

Die Meldungen im einzelnen –

(1) »**balance of structures**«: Nach Ende des ersten Passes der Assemblierung untersucht der Assembler, ob noch irgendwelche »offenen« Strukturen übriggeblieben sind, d. h. ob beispielsweise das zu einem ».repeat« gehörende ».until« oder das zu einem ».cond« passende ».econd« fehlen. Ist dies der Fall, gibt er obige Fehlermeldung aus.

(2) »**branch too far**«: Verzweigungen mittels der Branchbefehle des Prozessors (»beq, bne, bcc...«) können nur über eine maximale Distanz von 128 Byte rückwärts oder 127 Byte vorwärts ausgeführt werden. Falls der Label, zu dem der Branch führen soll, zu weit entfernt ist, gibt der Assembler deshalb einen Fehler aus.

(3) »**cpu failure jmp ($xxff)**«: Der »alte« 6502-Prozessor, dessen Nachfolger der im C-128 verwendete 8502 ist, besitzt einen Konstruktionsfehler. Dieser Fehler wirkt sich ausschließlich dann aus, wenn ein indirekter Sprung ausgeführt werden soll, der über einen Vektor führen soll, dessen beiden Hälften in verschiedenen Speicherseiten stehen, wenn der Vektor also eine Adresse »$xxff«

besitzt. In diesem Fall holt sich der Prozessor das Highbyte der Sprungadresse nicht von der Adresse »$(xx+1)00«, wie man es erwarten dürfte, sondern von »$xx00«. Da einige kommerzielle Programme für den C-64 von dieser fehlerhaften Ausführung des Befehls im Rahmen des Softwareschutzes Gebrauch machen und da der C-128 im 64er-Modus möglichst kompatibel zum C-64 sein sollte, wurde der Fehler wissentlich in die neue CPU übernommen.

(4) »**don't use .ext behind .modul**«: Bei der Beschreibung der Erzeugung relokatibler Module wurde bereits darauf hingewiesen, daß die Reihenfolge der Befehle ».extern«, ».modul« und ».public« festgelegt ist und vom Assembler überwacht wird. Bitte lesen Sie diese Reihenfolge in Kapitel 3.11.5 noch einmal nach, wenn Sie die obige Fehlermeldung erhalten haben.

(5) »**don't use .pub ahead of .modul** «: Dieser Fehler entspricht dem Fehler Nummer vier. Lesen Sie auch hier das Kapitel 3.11.5 noch einmal durch, um die korrekte Reihenfolge der Befehle zu sehen.

(6) »**end of line expected**«: Der Assembler arbeitet jede Quelltextzeile sequentiell ab, bis er der Meinung ist, alle relevanten Informationen abgearbeitet zu haben. In diesem Fall muß der vom Assembler verwendete Textzeiger entweder auf einem Semikolon stehen, das einen Kommentar einleitet oder auf einem Nullbyte, das das Zeilenende anzeigt. Ist keins von beidem der Fall, liegt mit Sicherheit ein Fehler vor.

(7) »**generic in/out**«: Die allgemeine Fehlermeldung des Assemblers, sobald während der In/Out-Prozeduren ein Fehler auftritt. Beispiele wären etwa, daß ein File nicht gefunden wurde, schon existiert oder auch, daß ein Sektor beschädigt ist und von der Floppy nicht gelesen werden kann.

(8) »**illegal address**«: Die Adressierung eines Prozessorbefehls ist nicht korrekt. Ein einfaches Beispiel wäre der Befehl »pla $4e,x«.

(9) »**illegal expression**«: Ein arithmetischer Ausdruck kann nicht ausgewertet werden; wahrscheinlich haben Sie eine Operation falsch abgekürzt, dies ist jedenfalls die häufigste Fehlerursache.

(10) »**illegal label**«: Ein Label- oder ein Variablenname folgt nicht den vom Assembler auferlegten Konventionen. Sie müssen den Label also umbenennen. Sehen Sie auch die Beschreibung der Integerarithmetik, um sich über den korrekten Aufbau von Variablennamen zu informieren.

(11) »**illegal register**«: Dies ist ebenfalls ein Fehler in der Adressierung; diesmal existiert der Befehl nicht mit dem gewählten Register – beispielsweise »ldx $4e,x«. Diese Fehlermeldung habe ich gesondert herausgeführt, weil ich ein paar Mal diesen jetzt so offensichtlichen Fehler bei der Ausgabe von »illegal address« einfach nicht gesehen habe.

(12) »**illegal structure**«: Sie haben Pseudoops zur strukturierten Programmierung in falschem Zusammenhang verwendet. Der Fehler tritt beispielsweise auf, wenn Sie ».do« mit ».until« verwenden oder einen falschen Schleifenausgang benutzen, also z. B. ».exloop« in einer Repeatschleife.

(13) »**label declared twice**«: Ein Labelname wird doppelt verwendet. Abhilfe schafft entweder eine Umbenennung oder ein lokaler Block mit ».begin« und ».end«.

(14) »**library: unknown macro or missing macend**«: Dieser Fehler tritt nur bei der Verwendung von Makrobibliotheken auf. Entweder befindet sich das aufgerufene Makro überhaupt nicht in der Bibliothek oder aber die Makrodefinition ist nicht durch ».macend« abgeschlossen.

(15) »**macro parameter**«: Die Zahl der Parameter in einem Makroaufruf stimmt nicht mit der Zahl der Parameter in der Makrodefinition überein oder aber Sie haben in der Parameterliste eines Makro einen Fehler gemacht, wie z. B. eine eckige statt einer runden Klammer zu verwenden, um die Parameterliste einzuleiten oder zu beenden.

(16) »**missing .end**«: Das ».end« zu einem ».begin« wurde nicht gefunden.

(17) »**missing .macend**«: Das Ende einer Makrodefinition wurde nicht gefunden. Diese Fehlermeldung erscheint nur im Fall der RAM-Makros, bei Bibliotheksmakros wird Fehler Nummer 14 ausgegeben.

(18) »**missing quotations**«: Es fehlen Anführungszeichen in der angezeigten Quelltextzeile.

(19) »**not a condition**«: Die Fehlermeldung besagt, daß eine Bedingung in einem der Pseudoops zur strukturierten Programmierung falsch angegeben ist. Bedingungen bestehen insbesondere auch aus einem führenden Doppelkreuz; die dahinter folgenden Abkürzungen entnehmen Sie bitte der Beschreibung der ASE-Pseudos zur strukturierten Programmierung.

(20) »**not a mnemonic**«: Diese Fehlermeldung erscheint ausschließlich im Direktmodus, wenn Sie Quelltextzeilen eingeben. Sie besagt, daß ein Mnemonic nicht zum Befehlsvorrat des Prozessors gehört. Denken Sie daran, daß der ASE die herstellerseitig nicht unterstützten illegalen Opkodes nicht annimmt.

(21) »**not implemented**«: Dies ist der gleiche Fehler wie Nummer 20 für den Fall, daß Sie einen Pseudoops eingegeben haben, den der ASE nicht kennt.

(22) »**no .modul allowed**«: Die Fehlermeldung tritt während des Linkens auf, wenn Sie versuchen, aus einem oder mehreren Modulen ein resultierendes Modul herzustellen, was aus Prinzip nicht möglich ist.

(23) »**number out of range**«: Bei der Berechnung eines arithmetischen Ausdrucks wurde der zulässige Zahlenbereich zwischen Null und $ffff = 65535 über- oder

unterschritten. Auch in einzelnen Termen eines Ausdrucks muß der Zahlenbereich eingehalten werden, so daß eine Rechnung wie

```
-1+4
```

falsch ist, obwohl das Ergebnis der Berechnung im zulässigen Bereich liegt (der Term »-1« unterschreitet den Zahlenbereich).

(24) »**string too long**« Ist dazu noch etwas zu sagen? Sie erhalten diese Fehlermeldung beispielsweise, wenn Sie im Editor einen zu langen Suchstring eingeben.

(25) »**structure too complex**«: Diese Fehlermeldung kann zwei Ursachen haben: entweder haben Sie Makros zu tief verschachtelt oder Sie haben das gleiche mit Pseudoops zur strukturierten Programmierung unternommen. Da die maximalen Schachtelungstiefen sehr groß sind, dürfte es sich bei diesem Fehler mehr um einen theoretischen Fall handeln.

(26) »**syntax error**«: Eine allgemeine Fehlermeldung, wenn irgend etwas mit der Syntax eines Befehls nicht stimmt. Es handelt sich hierbei praktisch um die gleiche allgemeine Fehlermeldung, wie sie Basic zuweilen ausgibt.

(27) »**too many labels**«: Der Variablenraum des Assemblers ist während der Assemblierung übergelaufen. Abhilfe ist auf zwei Arten möglich: Sie können die Append-Verkettung einsetzen und Ihren Quelltext in mehrere Teile aufspalten, zwischen denen jeweils der Variablenraum zumindest teilweise gelöscht wird. Besser wäre es vielleicht, den Quelltext zu spalten und von den einzelnen Teilen Module herzustellen, die Sie mit dem Linker zu einem Gesamtprogramm verbinden können.

(28) »**too many names**«: Der Namensstack ist während der Assemblierung übergelaufen. Abhilfe schaffen Sie durch den Einsatz des Befehls ».names«, mit dem Sie den Stack verlegen können.

(29) »**too many structures**«: Die Gesamtzahl der verwendeten strukturierten Programmkonstruktionen ist zu hoch, so daß ein assemblerinterner Stack überläuft. Die beste Abhilfe stellt wieder die Erzeugung einzelner Module dar.

(30) »**type conflict public/extern**«: Ein Label wurde sowohl als externe Referenz in einem ».extern« aufgeführt als auch als Einsprungpunkt deklariert. Eins von beiden kann er nur sein.

(31) »**unknown label**«: Ein in einem arithmetischen Ausdruck verwendeter Label oder eine Variable ist nicht definiert, beispielsweise, weil Sie sich verschrieben haben. Fehler dieser Art werden nur im zweiten oder gegebenenfalls im dritten Durchlauf der Assemblierung entdeckt. Der hier eingetretene Fehler unterliegt einer Spezialbehandlung: Zusätzlich zu den sonstigen Ausgaben im Fehlerfall

erscheint der Labelname in eckigen Klammern auf dem Bildschirm und Sie werden gefragt:

```
break or definition (b/d)?
```

Drücken Sie die Taste »b«, wird die Assemblierung auf der Stelle abgebrochen; drücken Sie »d«, erscheint der Cursor auf dem Bildschirm und Sie können einen Ausdruck eingeben, der für den unbekannten Label verwendet werden soll:

```
? =
```

Der eingegebene Ausdruck wird nur für die vorliegende fehlerhafte Zeile als Ersatz für den Labelwert genommen, er definiert keinen Label. Tritt der unbekannte Label ein zweites Mal auf, wiederholt sich die Prozedur. Zeropagelabel können grundsätzlich nicht nachgetragen werden, da der Assembler im ersten Pass alle unbekannten Label als nicht in der Zeropage liegend annimmt.

(32) »**unknown macro**«: Ein Makro des im Aufruf angegebenen Namens ist nicht bekannt, es handelt sich also wahrscheinlich um einen Schreibfehler im Makronamen.

(33) »**wrong module format**«: Ein zu linkender File entspricht nicht dem vom Linker erwarteten Modulformat. Dieser Fehler tritt auch auf, wenn Sie versuchen, Module im Format des ASE v1.0 mit dem Linker v2.0 zu linken, da die späteren Versionen ein anderes Format benutzen.

Kapitel 4

Der DBM-Monitor/Debugger

Sicher werden Sie sich fragen, warum ein eigenes Monitorprogramm zu den Top-Ass-Programmen gehört, wo doch der C-128 einen fest in die ROMs eingebauten Monitor besitzt. Nun - wenn Ihnen die Fähigkeiten dieses TEDMON des C-128 ausreichen, sollten Sie das vorliegende Kapitel erst einmal links liegen lassen. Sie können immer noch auf den DBM zurückgreifen, wenn Sie feststellen, daß Sie einige besondere Features dieses Programms benötigen. Über die Möglichkeiten des DBM können Sie sich ansatzweise im Anhang orientieren, der eine Befehlsübersicht über die DBM-Befehle enthält.

Für die jetzt noch übriggebliebenen Leser erst einmal Allgemeines zum DBM: Der DBM liegt auf der Masterdiskette in zwei Versionen vor: Das File »dbm2000« ist ein fertiges Maschinenprogramm mit Startadresse $2000, das File »dbm.rel« ist ein relokatibles Modul ohne externe Referenzen, das mit Hilfe des Loaders VLO an eine andere Startadresse geladen werden kann oder mittels des Linkers in ein File auf Diskette verwandelt werden kann, das ebenfalls eine andere Startadresse als $2000 aufweisen kann. Für diesen Zweck finden Sie auf der Masterdiskette einen speziellen Link-Quelltext »vlodbm.mke«, dessen Beschreibung Sie in Kapitel 3.11.3.2 nachlesen können. Der DBM ist so programmiert, daß allein die Startadresse für seine Funktion von Bedeutung ist; er kann in beide RAM-Bänke geladen und dort gestartet werden. Die Version »dbm2000« ist speziell dazu gedacht, vom Assembler aus in die RAM-Bank 1 geladen zu werden, wo zu diesem Zweck ein Freiraum durch den Assembler gelassen wurde. Die Startadresse des Monitors unterliegt Beschränkungen: Sie darf nicht so gewählt sein, daß Teile des Monitors unter dem Kernal-Bereich $c000 bis $ffff zu liegen kommen. Ob dies bei einer Relozierung der Fall ist, können Sie den entsprechenden Endmeldungen des VLO und des Linkers entnehmen.

Der Start des Monitors erfolgt in der einfachsten Weise vom Assembler aus, indem sie die Tastenkombination »SHIFT + RUN/STOP« drücken. Ist der Assembler nicht aktiviert, können Sie alternativ eine der beiden folgenden Befehlskombinationen verwenden:

```
boot "dbm2000"
```

oder bei Verwendung der Floppy 1541

```
bload "dbm2000": bank 0: sys dec("2000")
```

Daraufhin erscheint die Copyright-Meldung auf dem Bildschirm. Falls Sie einen 80-Zeichen-Monitor verwenden, ist es ratsam, vor dem Start des Monitors mit »fast« die höhere Taktfrequenz zu wählen, denn die Bildschirmausgaben auf dem 80-Zeichen-Schirm laufen nur in diesem Modus in annehmbarer Geschwindigkeit ab. Der DBM wird verlassen durch Drücken der Taste »STOP«.

Der DBM-Monitor benutzt für seine Arbeit die gleichen Speicherzellen in der Zeropage wie der TEDMON des C-128. Welche Speicherzellen dies sind, bitte ich den entsprechenden Beschreibungen beispielsweise des C-128-Handbuches oder auch eines ROM-Listings zu entnehmen.

4.1 Bildschirmeditor des Monitors

Unter dem Begriff »Bildschirmeditor« verbirgt sich hier eigentlich die Benutzeroberfläche des Monitors, die deswegen besonders angesprochen werden muß, da sie sich in einigen Punkten von den gewohnten Oberflächen etwa des Basic oder auch des Assemblers unterscheidet.

Nachdem Sie den Monitor gestartet haben, erscheint zunächst das gewohnte Bild – von den angekündigten Veränderungen ist noch nichts zu bemerken. Sie können mit dem Cursor an beliebige Stellen des Bildschirms fahren, dort Befehle eingeben – die Sie bisher noch nicht kennen – oder Änderungen vornehmen. Sämtliche Sonderfunktionen des normalen Bildschirmeditors – auch die Escape-Sequenzen – sind weiter verwendbar.

Eine Änderung gegenüber dem normalen Bildschirmeditor stellen Sie erst dann fest, wenn Sie einmal die Taste »Pfeil nach rechts« (Cursor-Einzeltaste) betätigen: Auf diesen Tastendruck erscheint der Cursor in der linken oberen Ecke der rechten Bildschirmhälfte, er hat seine Farbe von weiß nach gelb geändert. Was Sie gemacht haben, ist, von einer Sonderfunktion des Bildschirmeditors des DBM Gebrauch zu machen – Sie haben statt des Gesamtbildschirms ein Bildschirmfenster aktiviert, das aus der rechten Hälfte des Bildschirms besteht. Alle weiteren Eingaben beziehen sich von nun ab nur noch auf dieses Bildschirmfenster – als wenn Sie ein Fenster unter Basic aktivieren, allerdings beim DBM auf Tastendruck hervorgerufen. Die linke Bildschirmhälfte bleibt von Veränderungen der rechten Hälfte unberührt.

Drücken Sie nun die Taste »Pfeil nach links«. Der Cursor ist nun weiß und erscheint in der linken oberen Ecke des Bildschirms. Genauer müßte man in diesem Fall

sagen, in der linken oberen Ecke der linken Bildschirmhälfte, denn nun haben Sie auf Tastendruck die linke Schirmhälfte als Fenster aktiviert. Auch in der linken Hälfte kann vollkommen unabhängig gearbeitet werden – die rechte Hälfte bleibt unberührt.

Sie können nun jederzeit zwischen den beiden Bildschirmfenstern hin- und herschalten, der Monitor paßt alle seine Ausgaben an die gerade gewählte Bildschirmbreite an und merkt sich auch, was zuletzt ausgegeben wurde, so daß Sie nach einer Umschaltung in ein anderes Fenster sofort dort weitermachen können, wo Sie vor der ersten Umschaltung aufgehört haben – dazu später, bei den Dumps – mehr.

Von den Fenstern kommen Sie durch zweimaliges Drücken der Taste »HOME« in den Gesamtschirm zurück. Das gleiche geht natürlich auch auf dem 40-Zeichen-Bildschirm: Falls Sie einen Monitor besitzen, der auf 40-Zeichen-Darstellung umgeschaltet werden kann, probieren Sie dies einmal aus (mit »ESC x« schalten Sie zwischen 40- und 80-Zeichen-Darstellung um).

Die Möglichkeit, den Bildschirm in Fenster teilen zu können, ist eine nicht zu unterschätzende Erleichterung der Arbeit mit dem Monitor, wie Sie sicher im Laufe der Zeit selbst feststellen werden. Stellen Sie sich zum Beispiel den einfachen Fall vor, daß Sie ein Programmstück disassemblieren und zwischenzeitlich ein im Programmkode aufgerufenes Unterprogramm ebenfalls nebenher disassemblieren wollen. Mit nur einem zur Verfügung stehenden Bildschirm ist dies ein mühsames Unterfangen, mit Fenstern, die sich auf Tastendruck wählen lassen, ist die Sache dagegen mühelos und bequem.

4.2 Parametereingaben/Workspaces/ Zahlenkonvertierungen

Die meisten der DBM-Befehle erwarten in ihrer Parameterliste Angaben von Adressen oder ähnliches, also Zahlenangaben. Zahleneingaben können im DBM in zwei verschiedenen Zahlensystemen vorgenommen werden – sedezimal oder dezimal. Die sedezimale Schreibweise wird vom Monitor bevorzugt, da dieses Zahlensystem sich allgemein unter »Maschinisten« durchgesetzt hat; folglich werden alle Zahlen als sedezimal angesehen, sofern sie nicht durch ein führendes Pluszeichen ausdrücklich als dezimal gekennzeichnet sind:

```
af00        sedezimal
+18         dezimal
```

Adressenangaben, die eine Konfiguration beinhalten müssen, werden grundsätzlich nur sedezimal angenommen, wobei hier eine fünfte sedezimale Ziffer den Index der Konfiguration angibt, in der die Adresse liegen soll. Die Bedeutung der Konfigura-

tionsindizes bitte ich, in Kapitel 2 noch einmal nachzuschlagen, falls Sie mit ihnen noch nicht vertraut sein sollten. Beispiele:

```
0a000      a000   unter Index 0    (RAM-Bank 0)
10c00      c00    unter Index 1    (RAM-Bank 1)
f0c00      c00    unter Index 15   (RAM-Bank 0 + ROMs)
```

Es ist immer nützlich, verschiedene Zahlensysteme nicht von Hand ineinander umrechnen zu müssen, deshalb enthält auch der DBM die Möglichkeit, die Umrechnungen vom Monitor vornehmen zu lassen. Hierzu geben Sie lediglich die umzurechnende Zahl hinter dem Zeichen »_« (Pfeil nach links in der ASCII-Belegung) ein und drücken »RETURN«. Daraufhin wird die Zahl in den drei Zahlensystemen zur Basis 16, 10 und 2 ausgegeben:

```
_12
$0012                          (sedezimal)
+18                            (dezimal)
%0000000000010010              (binär)
```

Binär- und Sedezimalzahlen werden in der Ausgabe durch die beiden Zeichen »%« und »$« kenntlich gemacht, wie Sie es vom Assembler her kennen. Verwenden Sie diese Zeichen als Parameter in den Monitorbefehlen, erhalten Sie eine Fehlermeldung des Monitors. Diese besteht lediglich aus einem Fragezeichen, das hinter das letzte Zeichen der Eingabezeile geschrieben wird – weitere Fehlermeldungen kennt der DBM, wie der TEDMON, nicht.

Viele DBM-Befehle erwarten nicht nur eine einzelne Zahleneingabe, sondern die Eingabe von Adreßbereichen. Bei der Arbeit mit dem Monitor werden Sie im Laufe der Zeit feststellen, daß solche Bereichsangaben sehr oft in verschiedenen Befehlen in gleicher Weise benötigt werden. Es liegt also recht nahe, einen Bereich einmal zu definieren und unter einer Nummer abzulegen, so daß die Eingabe der Bereichsnummer in den Befehlen ausreicht, um den Adreßbereich zu definieren. Diese Idee liegt den sogenannten Workspaces (Arbeitsbereichen) zugrunde. Durch den Befehl

```
w   <nr>   <adr1>   <adr2>
```

geben Sie einen Arbeitsbereich ein; maximal können zehn Workspaces verwendet werden, die die Nummern »<nr>« von 0 bis 9 tragen. »<adr1>« bezeichnet die erste Adresse des Workspace, »<adr2>« bezeichnet die Endadresse. Verwenden Sie Adreßangaben mit Konfigurationsindizes, müssen die beiden Adressen natürlich den gleichen Index aufweisen, damit nicht das Fragezeichen als Fehlermeldung ausgegeben wird. Beispiel für eine Workspace-Eingabe:

```
w   1   10c00   10ca1
```

Der Arbeitsbereich 1 reicht von $c00 in Bank 1 bis $ca1.

4.3 Memory-Dumps

Unter dem Begriff »Memory-Dumps« versteht man allgemein gesprochen die Anzeige eines Teils des Speicherinhalts in einer bestimmten Form. Die einzelnen Bytes können beispielsweise als sedezimale Zahlen oder auch als ASCII-Zeichen interpretiert und am Bildschirm angezeigt werden. Auch eine Interpretation als Maschinenkode ist möglich – in einem solchen Fall spricht man von einer Disassemblierung.

Alle drei erwähnten Arten von Dumps unterstützt der DBM; zuständig hierfür sind die drei Befehle:

```
m <adr>
a <adr>
d <adr>
```

»m« gibt eine Zeile eines sogenannten Hexdumps aus, die einzelnen Bytes werden sedezimal am Bildschirm ausgegeben. »a« gibt eine Zeile eines ASCII-Dumps aus, in dem die Bytes als ASCII-Zeichen interpretiert werden. Nicht druckbare Zeichen werden in der ASCII-Interpretation als Punkte am Bildschirm gezeigt. »d« schließlich disassembliert den an der Adresse »<adr>« stehenden Prozessorbefehl – sofern dort einer steht. Kann das bei »<adr>« stehende Byte nicht als Operationskode interpretiert werden, wird eine Zeile ».by xx« angezeigt; »xx« ist das bei »<adr>« gefundene Byte. Am besten, Sie probieren die drei Befehle einmal aus, damit Sie einen Eindruck vom Aussehen der einzelnen Dumparten erhalten können.

Sicher fragen Sie sich jetzt schon, warum nur eine Zeile des jeweiligen Dumps ausgegeben wird. Jetzt kommen aber die beiden bisher noch nicht verwendeten Cursor-Einzeltasten »Pfeil nach oben« und »Pfeil nach unten« zum Einsatz:

Bringen Sie als Beispiel einmal eine Zeile eines Hexdumps auf den Bildschirm. Geben Sie hierzu beispielsweise den Befehl

```
.m 1000
```

ein. Falls Sie den 80-Zeichen-Bildschirm in seiner vollen Breite benutzen, sehen Sie nun 16 Byte der RAM-Bank 0 ab der Adresse $1000 in sedezimaler Schreibweise, dahinter folgen die ASCII-Entsprechungen der 16 Bytes. Verwenden Sie eines der 40 Zeichen breiten Fenster, werden nur 8 Byte angezeigt. Das gleiche Ergebnis erhalten Sie, wenn Sie den 40-Zeichen-Schirm benutzen. Verwenden Sie die nur 20 Zeichen breiten Fenster des 40-Zeichen-Bildschirms, sehen Sie schließlich nur noch 4 Byte dargestellt, die ASCII-Entsprechungen entfallen wegen der geringen Fensterbreite. Drücken Sie nun – wenn die Dumpzeile auf dem Bildschirm steht – die Taste »Pfeil nach oben«. Sie sehen eine neue Dumpzeile, die die Bytes zeigt, die auf die der ersten Zeile folgen. Halten Sie die Taste gedrückt, wird auch klar, warum diese

Funktion auf die Taste »Pfeil nach *oben*« gelegt wurde: Paßt die neue Dumpzeile nicht mehr auf den Bildschirm, wird der Fensterinhalt nach *oben* weggescrollt.

Das gleiche geht auch in der anderen Richtung. Drücken Sie nun die Taste »Pfeil nach unten«: Der Dump wird in der anderen Richtung gescrollt.

Auf diese Weise können Sie den Speicher ziemlich mühelos vor- und rückwärts in Augenschein nehmen, auf jeden Fall aber bequemer als dies mit dem TEDMON möglich ist. Eine andere Sache kommt noch hinzu: Aktivieren Sie nun einmal eines der beiden Fenster des DBM und lassen Sie dort einen Dump anzeigen, schalten Sie danach in das andere Fenster und sehen sich dort einen anderen Dump an. Sie werden feststellen, daß Sie nach der Umschaltung zwischen den beiden Fenstern sofort den Dump des gerade aktiven Fensters fortführen können, ohne neuerliche Eingaben – ein Druck auf eine der beiden Cursor-Einzeltasten genügt. Das halte ich schon für einen erheblichen Vorteil gegenüber einem normalen Monitor.

Das gleiche Prinzip wird auch bei den beiden im Beispiel nicht verwendeten Dumparten des ASCII-Dumps und der Disassemblierung angewandt. Lediglich die Disassemblierung unterscheidet sich in einem Punkt: Während die beiden anderen Dumparten mühelos von einer gewählten Startadresse rückwärts fortgesetzt werden können, ist dies bei der Disassemblierung aufgrund der nicht konstanten Befehlslängen nicht ohne weiteres möglich. Der Monitor geht deshalb so vor: Die mit dem Befehl »d« eingegebene Adresse wird als untere Startadresse angenommen. Von dieser Adresse aus kann vorwärts disassembliert werden, aber sie kann nicht nach unten überschritten werden. Haben Sie ein Stück vorwärts disassembliert und drücken nun die Taste »Pfeil nach unten«, so disassembliert der Monitor ausgehend von der gesetzten Startadresse, bis er die letzte angezeigte Zeile erreicht. Die Adresse, die bei der vorhergehenden Disassemblierung erreicht wurde, wird als Fortsetzung des Dumps nach unten genommen. Durch diese Vorgehensweise ist gewährleistet, daß Sie bei der Rückwärts- und der Vorwärts-Disassemblierung immer die gleichen Befehle am Bildschirm ausgegeben erhalten.

Änderungen der Speicherinhalte können durch einfaches Überschreiben der ausgegebenen Dumpzeilen durchgeführt werden: Überschreiben Sie dazu beispielsweise in einer Hexdumpzeile einfach eines der angezeigten Bytes mit dem neuen Wert, den es erhalten soll, und drücken Sie »RETURN«. Die Änderung wird sofort in den Speicher übernommen, und die als zusätzliche Information angezeigten ASCII-Äquivalente ändern sich entsprechend. Änderungen in den ASCII-Ausgaben einer Hexdumpzeile haben keine Auswirkungen auf den Speicher. Stattdessen können Sie sich eine Zeile eines ASCII-Dumps anzeigen lassen und diese entsprechend mit dem neuen Zeichen beschreiben. Änderungen der Speicherinhalte setzen naturgemäß voraus, daß Sie sich im klaren sind, was Sie da gerade ändern und daß es sich dabei möglichst nicht um lebenswichtige Adressen für das System handelt.

Im Fall der Änderung einer disassemblierten Zeile entspricht der Vorgang der Änderung einer schon früher einmal angesprochenen Line-, Zeilen- oder Direkt-

assemblierung – alle drei Begriffe bezeichnen ein und dasselbe. Der Monitor macht also im Prinzip das gleiche wie der Assembler – er übersetzt Mnemonic und Adressierung in Operationskodes. Dabei können natürlich keine Label verwendet werden – der Monitor kennt keine Symboltabelle. Änderungen in disassemblierten Zeilen können nur im Mnemonic und in der Adressierung vorgenommen werden. Die Änderung der vorher angezeigten sedezimalen Bytes des Operationskodes bewirkt nichts.

Die Assemblierung kann selbstverständlich auch ohne die Ausgabe einer disassemblierten Zeile erfolgen. Hierzu schreiben Sie einfach die Startadresse der Assemblierung hinter den Punkt, der eine Disassemblerzeile einleitet. Hinter die Adresse wiederum schreiben Sie Mnemonic und Adressierung. Durch »RETURN« wird die Zeile assembliert und zusätzlich die nächstfolgende Adresse ausgegeben, wodurch sich ein Effekt ähnlich dem der automatischen Zeilennumerierung ergibt.

Ganze Workspaces können auf einmal am Bildschirm angezeigt werden, sofern sie ganz auf den Bildschirm passen, ansonsten wird die Ausgabe abgebrochen, sobald der Bildschirm voll ist. Die Befehle hierzu lauten:

```
wm  <nr> -  Hexdump des Workspace »<nr>«
wa  <nr> -  ASCII-Dump des Workspace »<nr>«
wd  <nr> -  Disassemblierung des Workspace »<nr>«
```

Diese Möglichkeit der Dumpanzeige ist besonders nützlich, wenn man einen bestimmten Speicherbereich eingehender bearbeitet – daher leitet sich auch die Bezeichnung »Workspace« her: Auch nachdem der Bildschirminhalt durch weitere Befehlseingaben zerstört wurde, kann man sich mit Hilfe dieser Befehle schnell wieder einen Überblick verschaffen.

4.4 Blockoperationen

4.4.1 Blockverschiebung (transfer)

Blockverschiebungen innerhalb einer RAM-Bank oder von RAM-Bank zu RAM-Bank können mit dem Befehl

```
t <nr>   <adr>
```

durchgeführt werden. »<nr>« bezeichnet die Nummer des Adreßbereiches, der verschoben werden soll, »<adr>« ist die Zieladresse der Verschiebung, also die Adresse, die das erste Byte des zu verschiebenden Bereiches nach der Verschiebung innehaben soll.

4.4.2 Adreßanpassung (convert)

Verschiebt man ein Maschinenprogramm von seiner ursprünglichen Adresse an eine andere Stelle, so wird es in der Regel nicht mehr lauffähig sein, da alle absoluten Adreßangaben in den Adressierungen, die sich auf den verschobenen Bereich beziehen, mit an die neue Lage des Programms angepaßt werden müßten.

Diese Aufgabe übernimmt der Befehl

```
cv <nr> <adr1> <adr2> <adr3>
```

Die Erläuterung der Parameterliste liest sich vielleicht beim ersten Mal etwas verwirrend, ist aber recht einfach zu verstehen: »cv« ändert im Arbeitsbereich »<nr>« alle absoluten Adressierungen, die sich auch den Adreßbereich zwischen »<adr1>« und »<adr2>« beziehen, so, als wenn der Bereich »<adr1>« bis »<adr2>« nach »<adr3>« verschoben worden wäre – was nicht notwendigerweise tatsächlich der Fall sein muß. Mögliches Einsatzgebiet des Befehls: Sie verschieben mit dem Befehl »t« ein Unterprogramm eines Maschinenprogramms an eine andere Stelle. Anschließend müssen Sie natürlich alle Adressen des übriggebliebenen Hauptprogramms ändern, die sich auf den Bereich beziehen, wo das Unterprogramm vorher stand. Ebenso müssen Sie alle Adressen innerhalb des Unterprogramms ändern, die sich auf den alten Standort des Unterprogramms beziehen – es sind also insgesamt zwei Befehle »cv« mindestens notwendig, bevor das gesamte Programm wieder läuft.

An einem einfachen Beispiel; geben Sie folgende Befehle ein:

```
w0   b00  c00
w1   c00  d00
```

Damit sehen wir uns später das Ergebnis unserer Bemühungen an. Nun schreiben wir einen Befehl mit einer absoluten Adressierung in den ersten Bereich:

```
.b00  jmp  b10
```

und stellen uns dabei vor, der ganze Bereich »b00« bis »c00« sei unser zu verschiebendes Programm, wobei »b10« beispielsweise die Adresse eines Unterprogramms sei. Als nächstes verschieben wir das Programm an eine andere Stelle:

```
t    0   c00
```

Mit »wd 1« können Sie sich davon überzeugen, daß die Verschiebung stattgefunden hat. Die Adressierung im Befehl »jmp $0b10« muß allerdings noch an die neue Lage angepaßt werden, was wir mit dem Befehl vornehmen:

```
cv   1   b00   c00   c00
```

Schauen Sie sich den zweiten Bereich noch einmal mit »wd 1« an – die Anpassung hat geklappt. Die Anpassung des Hauptprogramms an die neue Lage des Unterprogramms entfällt hier in Ermangelung eines solchen.

Alle Anpassungen sind davon abhängig, daß der zu verändernde Adreßbereich nur aus Maschinenkode besteht. Befinden sich innerhalb des Bereiches Tabellen oder Texte, so kann keine 100prozentig sichere Disassemblierung vorgenommen werden, somit kann nicht sicher festgestellt werden, wo sich absolute Adressierungen verbergen und damit wiederum kann man nicht erwarten, daß die Anpassung korrekt verläuft und das verschobene Programm lauffähig ist.

4.4.3 Blockvergleiche (block comparison)

In bestimmten Situationen kann es nützlich sein, überprüfen zu können, ob zwei Speicherbereiche den gleichen Inhalt haben oder – wenn nicht – an welchen Adressen sie sich unterscheiden. Diesen Vergleich übernimmt der Befehl

```
bcp <nr> <adr>
```

»<nr>« bezeichnet wieder die Nummer des Arbeitsbereiches, der als Grundlage für den Vergleich dienen soll, »<adr>« ist die Anfangsadresse des zweiten Bereiches, mit dem der Arbeitsbereich verglichen wird.

Als Beispiel könnten Sie einmal einen Vergleich der beiden Workspaces aus dem vorigen Kapitel durchführen:

```
bcp  0    c00
```

Sie erhalten als Ausgabe zwei Adressen – einmal die Adresse, an der die Anpassung erfolgte, zum anderen eine Adresse, die sich aus der Überschneidung der beiden Bereiche ergibt.

4.4.4 Füllen eines Bereiches (fill)

Zu diesem Befehl muß nicht viel gesagt werden, er füllt einen Adreßbereich mit einem vorgegebenen Wert:

```
f     <nr> <byte>
```

»<nr>« ist die Workspace-Nummer, »<byte>« ist der Füllwert, mit dem der Bereich beschrieben wird.

4.5 Disk-Betrieb

Sämtliche Befehle, die zum Diskbetrieb zu zählen sind – Laden, Verifizieren, Abspeichern, Inhaltsverzeichnisse usw. – werden ohne Angabe einer Gerätenummer eingegeben. Statt dessen wird die Gerätenummer einmal mit dem Befehl

```
dv <nr>  (device)
```

gesetzt. Voreingestellt ist Gerät Nummer 8, da dies wohl für die meisten Anwender die Normaleinstellung sein dürfte.

4.5.1 Laden, Verifizieren und Abspeichern

Beim Laden und Verifizieren sind zwei Fälle zu unterscheiden: Zum einen kann man ein Maschinenprogramm an die Adresse laden, die als Startadresse in der Datei selbst angegeben ist (absolutes Laden); zum anderen kann man ein Maschinenprogramm oder andere Daten auch an eine beliebige andere Adresse laden wollen (relatives Laden – was allerdings nichts mit relokatiblen Modulen zu tun hat). Das Verifizieren entsprechend dem Basicbefehl »verify« muß die gleichen Fallunterscheidungen wie das Laden berücksichtigen.

Die vier interessierenden Lade- und Verifikationsbefehle sind damit:

```
l "file"  - absolutes Laden von »file«
v "file"  - Verifizieren eines absolut geladenen Files
l "file",<adr>  - relatives Laden an die Adresse <adr>
v "file",<adr>  - das Verifizieren dazu
```

Bei der Abspeicherung von Speicherbereichen entfällt die obige Unterscheidung selbstverständlich. Es gibt nur einen Befehl zum Speichern:

```
s <nr> "file"
```

»<nr>« bezeichnet die Nummer des Arbeitsbereiches, der gespeichert werden soll.

4.5.2 DOS-Support

Unter der Bezeichnung DOS-Support versteht man im allgemeinen die Möglichkeit, Inhaltsverzeichnisse von Disketten anzeigen zu lassen, den Fehlerkanal der Floppy abzufragen sowie Befehle an die Floppy senden zu können. Alle drei Dinge sind vom DBM aus möglich.

Die Anzeige von Fehlerkanal und Directory ist auf Funktionstasten gelegt worden:

```
F3   -  Directory
F2   -  Fehlerkanal
```

Das Format der Directory-Ausgabe entspricht nicht dem, das Sie von Basic her gewohnt sind – diese Änderung ergibt sich aus der Tatsache, daß das Directory auch auf den nur 20 Spalten breiten Fenstern des 40-Zeichen-Schirms lesbar sein soll. Mit dem Befehl

```
§ <dos-string>
```

können Befehle zur Floppy gesendet werden (das Paragraphenzeichen entspricht dem At-Sign – »Klammeraffe« – im ASCII-Zeichensatz). Der Parameter »<dos-string>« ist ein für die Floppy verständlicher Befehlsstring. Das Format dieser Strings erfahren Sie aus dem Floppy-Handbuch. Beispiel:

```
§ s:file1
```

löscht »file1« von der Diskette. Beachten Sie bitte, daß die DOS-Befehlsstrings nicht in Anführungszeichen eingeschlossen sind.

4.5.3 Disk-Monitor

Der DBM-Monitor enthält einen rudimentären Disk-Monitor, dessen Fähigkeiten sich darauf beschränken, Sektoren von einer Diskette lesen und auf eine Diskette schreiben zu können. Diese beiden Fähigkeiten reichen für 99 Prozent der Anwendungen aus – man denke etwa an das Aufbringen eines Boot-Sektors auf eine Diskette oder das Wiederherstellen eines irrtümlich gelöschten Files. Beide hier genannten Einsatzgebiete erfordern ausgiebige Kenntnisse der Software des C-128, die an dieser Stelle nicht vermittelt werden können. Sehen Sie zu diesem Thema bitte in weiterführenden Büchern nach.

Die Befehle:

```
b-r  nr> <track> <sector>    - Block lesen
b-w  nr> <track> <sector>    - Block schreiben
```

»<nr>« ist die Nummer des Workspaces, in den der gelesene Sektor geschrieben werden soll bzw. der als Sektor auf Disk geschrieben werden soll. »<track>« und »<sector>« sind Spur- und Sektornummer des angezielten Sektors.

Beispiel:

```
w0  b00  c00
b-r  0  +18  1
```

lädt den ersten Directory-Sektor in den Bereich ab $b00. Entsprechend schreibt der Befehl

```
b-w  0  +18  0
```

den Sektor wieder auf Diskette zurück, nachdem Sie eventuell ein File restauriert haben.

4.6 Ausgaben auf den Drucker

Zu einem guten Monitor gehört auch die Möglichkeit, die Bildschirmausgaben auf dem Drucker protokollieren lassen zu können. Der TEDMON des C-128 läßt sich nur über Umwege dazu bringen, etwas auszudrucken.

Beim DBM erfolgt das Ein- und Ausschalten des Druckerprotokolls auf Tastendruck:

```
CBM  +  F1
```

Durch das erste Drücken der Commodore-Taste zusammen mit der Taste »F1« schalten Sie das Protokoll ein – alle Ausgaben, beispielsweise von Dumps – erfolgen nun außer auf den Bildschirm auch über den vorher eingestellten Druckkanal zum Drucker. Das zweite Drücken der Tastenkombination schließt den Druckkanal wieder, so daß die Ausgaben wieder allein auf den Bildschirm erfolgen. Die beiden gerade aktiven Zustände werden bei der Betätigung der Tasten in Form zweier Meldungen – »printer on« und »printer off« – angezeigt. Der zu verwendende Druckkanal wird durch den Befehl

```
pr  <unit>  <secadr>
```

festgelegt. »<unit>« muß die Gerätenummer des Druckers enthalten, »<secadr>« die beim Druck zu verwendende Sekundäradresse. Beide Angaben entnehmen Sie für Ihren Drucker am besten dem Handbuch entweder des Druckers selbst oder des Interfaces, das Sie verwenden. Voreingestellt ist die Gerätenummer 4 mit der Sekundäradresse 7, was auf das vielfach verwendete Wiesemann-Interface zugeschnitten ist.

Um das Druckbild zu beeinflussen, haben Sie zwei zusätzliche Tastenfunktionen zur Verfügung, sobald der Druckkanal durch »CBM + F1« geöffnet ist:

(1) Jedes Drücken der Taste »RETURN« erzeugt auch einen Zeilenvorschub auf dem Drucker.

(2) Durch die Tastenkombination »CBM + F3« können Sie eine Trennzeile aus Sternen ausgeben, deren Länge sich nach der jeweiligen Fensterbreite auf dem Bildschirm richtet.

4.7 Suchen (hunt)

4.7.1 Suchen von Zeichenketten und Bytewerten

Ähnlich wie im TEDMON des C-128 können Zeichenketten und Folgen von Bytewerten gesucht werden. Hierzu dienen die Befehle

```
ha <nr> 'string'
```

und

```
ha <nr> <byte 1> <byte 2> ... <byte n>
```

Wie immer bezeichnet »<nr>« die Nummer des Workspace – desjenigen, der durchsucht werden soll; »string« ist eine Folge von ASCII-Zeichen, die in Hochkommata eingeschlossen werden muß. Im Fall der ersten Befehlsvariante wird jedes Vorkommnis des Strings durch eine ASCII-Dumpzeile angezeigt. Die zweite Befehlsvariante sucht nach einer Folge von Bytewerten, die hinter der Workspacenummer sedezimal eingegeben wird. In diesem Fall wird jedes Vorkommnis durch eine Hexdumpzeile angezeigt. Beispiele für die beiden Varianten:

```
ha 0 'abcdef'
ha 0 12 34 56 78
```

4.7.2 Suchen von Prozessorbefehlen

Bei der Analyse fremder Programme und auch bei der Fehlersuche in eigenen Programmen werden Sie häufig einen Befehl benötigen, der es ermöglicht, bestimmte Opcodes und Adressierungen zu finden. Diese Möglichkeit bietet der Befehl

```
h <nr> <dis-string>
```

Wieder bezeichnet »<nr>« die Workspacenummer des Adreßbereiches, der durchsucht werden soll. »<dis-string>« ist eine Folge von ASCII-Zeichen, die den zu suchenden Prozessorbefehl in disassemblierter Form zeigt – so, wie der DBM den gesuchten Befehl disassembliert zeigen würde. Innerhalb des Strings sind zwei Arten von Jokern erlaubt, die ganz entsprechend zu den beiden Jokern innerhalb von DOS-

Strings wirken: Der Punkt bezeichnet eine Zeichenposition, die beliebig ist; der Stern bezeichnet ein vorzeitiges Ende des Suchstrings.

Einige Beispiele:

```
h  0  lda $a473      Sucht alle Befehle »lda $a473« im
                     Arbeitsbereich Nummer Null.

h  0  ld. $a473      Sucht alle Befehle »lda $a473«, »ldx
                     $a473« oder »ldy $a473« in Space 0.

h  0  lda $a...      Sucht alle nicht-indizierten Befehle
                     »lda« in Space 0, deren Adressierung im
                     Bereich von $a000 bis $afff liegt.

h  0  ld. $..        Sucht alle nicht-indizierten Load-Befeh-
                     le in Space 0, deren Adressierung in der
                     Zeropage liegt.

h  0  ... $..,*      Sucht alle Befehle mit Adressierung
                     »Zeropage X-indiziert« und »Zeropage Y-
                     indiziert«.

h  0  ... ....),y    Sucht alle Befehle mit Adressierungsart
                     »indirekt, nachindiziert«.

h  0  .by*           Sucht alle nicht zu disassemblierenden
                     Stellen in Space 0.

h  0  beq $8a*       Sucht alle Branches »beq«, die in den
                     Bereich $8a00 bis $8aff verzweigen.
```

Diese Art zu suchen ist sehr flexibel gehalten, so daß sie praktisch alle benötigten Fälle erfaßt. Beachten Sie bitte, daß das Aussehen des Suchstrings dem Aussehen des Befehls in disassemblierter Form entsprechen muß, insbesondere gilt dies auch für das Leerzeichen zwischen Mnemonic und Adressierung.

4.8 Debuggerfunktionen

4.8.1 Das Prinzip der Breakpoints

Um ein Austesten von Maschinenprogrammen zu ermöglichen, bietet der DBM eine sogenannte Breakpointbehandlung. Das Prinzip, das hinter diesem Testverfahren steht, soll Gegenstand dieses Kapitels sein.

Wie Sie sicher wissen, enthält der Befehlssatz des 6502 einen Befehl namens »brk«, dessen Sinn in der täglichen Programmierung eigentlich nicht recht einzusehen ist. Dieser Befehl läßt sich aber hervorragend bei der Fehlersuche verwenden, sicher ein Grund dafür, daß er mit in den Befehlsvorrat aufgenommen wurde.

Man geht dazu folgendermaßen vor: An strategisch wichtigen Stellen des zu testenden Maschinenprogramms wird der Befehl »brk« eingesetzt, die vorher an diesen Stellen vorhandenen Originalbefehle merkt man sich in einer Liste. Anschließend wird das Testprogramm gestartet. Sobald nun eines der Breaks erreicht wird, führt der durch das »brk« ausgelöste Interrupt in den Debugger, der daraufhin anzeigt, an welcher Adresse die Unterbrechung aufgetreten ist, welche Inhalte aktuell in den Registern stehen und welche Werte vorher ausgesuchte Speicherbereiche bei der Unterbrechung aufweisen. Als letztes wartet der Debugger schließlich auf eine Reaktion des Testers: Dieser hat zu prüfen, ob die von ihm gesuchte Fehlerbedingung erfüllt ist oder nicht – entsprechend bricht er das Testen ab oder startet das Maschinenprogramm am Ort der Unterbrechung neu. In diesem Fall setzt der Debugger den ursprünglichen Opcode an die Stelle des »brk«, führt den Befehl aus, setzt das »brk« wieder ein und startet schließlich das Programm erneut hinter dem »brk«. Das Testprogramm läuft daraufhin wieder an, bis der nächste Breakpoint getroffen wird.

Die eigentliche Schwierigkeit bei diesem Testverfahren – wie beim Austesten von Programmen überhaupt – ist nicht die technische Realisierung der Hilfsmittel zum Austesten – hier der Breakpointbehandlung. Vielmehr liegt die Hürde darin, daß sich einfach keine allgemeinen Regeln aufstellen lassen, nach denen sich Programme austesten ließen. Praktisch kann man sich nur an die Murphy'schen Gesetze halten: »Was falsch sein kann, wird auch falsch gemacht werden« und »der Fehler steckt immer in der Routine, von der Sie sicher sind, daß sie absolut fehlerfrei ist«. Folgerichtig werden sich die nachfolgenden Beschreibungen auf eine reine Beschreibung der zur Verfügung stehenden Hilfsmittel konzentrieren müssen, die der DBM dem Programmierer zur Verfügung stellt.

4.8.2 Registeranzeige/Starten von Testprogrammen

Die Registeranzeige des DBM erfolgt auf Drücken der Taste »F5« oder durch Eingabe des Befehls »r«. Daraufhin erscheint folgendes Bild am oberen Bildschirmrand:

```
*pc b000  cr 00
ac xr yr sp nv|bdizc
00 00 00 fa --|-----
0b000 00       brk
```

Das Zeichen »|« entspricht hier dem Pfeil nach oben, den ich mit der verwendeten Textverarbeitung leider nicht darstellen konnte.

Die Registeranzeige erfüllt zwei Aufgaben:

(1) Während der Breakpoint-Behandlung kann der Tester der Registeranzeige die aktuellen Werte des Programmzählers, der Register usw. entnehmen.

(2) Beim Start des Testprogramms werden die vorher in die Registeranzeige übernommenen Werte beispielsweise in die Prozessorregister übernommen, so daß das Testprogramm unter definierten Bedingungen gestartet werden kann.

Die Beschreibung der Registeranzeige: Der Stern in der ersten Zeile der Registeranzeige dient lediglich internen Zwecken; die dahinter folgenden zwei Buchstaben »pc« bezeichnen den Programmzähler (program counter). Zusammen mit der Angabe der Konfiguration »cr« kann man der ersten Zeile entnehmen, an welchem Ort ein Breakpoint angelaufen wurde bzw. – beim Start eines Testprogrammes – bestimmen, an welcher Adresse und unter welcher Konfiguration das Programm gestartet werden soll. Die dritte Zeile auf dem Bildschirm gibt zusammen mit der fünften Zeile die Inhalte der Prozessorregister bei der Unterbrechung bzw. beim Start an; der Inhalt des Statusregisters wird nicht als sedezimale Zahl, sondern als Folge von Bits ein- und ausgegeben, das Minuszeichen entspricht einem gelöschten Bit, der Pfeil nach oben bezeichnet ein gesetztes Bit. Die Bezeichnung der einzelnen Flags entspricht dem üblichen Standard, den Sie Kapitel 1 entnehmen können. Als letzte Zeile schließlich finden Sie den Befehl disassembliert, der unter der angezeigten Adresse und Konfiguration beim nächsten Start als erster ausgeführt werden wird.

Änderungen der Inhalte der Registeranzeige werden in einer von der normalen Prozedur etwas abweichenden Art vorgenommen, da sich die Registeranzeige über mehrere Bildschirmzeilen erstreckt: Sie überschreiben zunächst die dargestellten Werte durch Werte Ihrer Wahl und betätigen dann die Tastenkombination »CBM + F5«. Änderungen in der disassemlierten Zeile haben keine Auswirkungen. Die eingesetzten Werte bleiben auch dann erhalten, wenn Sie anschließend andere Operationen vornehmen. Probieren Sie die Änderung der Registeranzeige einmal aus; beschreiben Sie beispielsweise den Wert für den Programmzähler und drücken Sie dann »CBM + F5«. Sie sehen, wie sich die disassemblierte Zeile entsprechend ändert. Die Anzeige des Prozessorstatus wird nicht mit einer sedezimalen Zahl, sondern durch Minuszeichen und Pfeile nach oben beschrieben, entsprechend zum Ausgabeformat.

Der Start eines Testprogramms kann nach Setzen der Registeranzeige auf die gewünschten Werte durch zwei Befehle erfolgen:

```
g [<adr>]
j <adr>
```

Der Befehl »g« (goto) ohne Angabe einer Startadresse startet das Testprogramm an der Startadresse und unter der Konfiguration, die durch die Registeranzeige festgelegt sind. Die Inhalte der Registeranzeige werden in die Prozessorregister übernommen und das Testprogramm wird gestartet. Zurück in die Registeranzeige kommt man ausschließlich, indem das Testprogramm auf ein »brk« trifft. Wird »g« von einer Startadresse gefolgt, wird der Programmzähler der Registeranzeige ignoriert und statt dessen die angegebene Startadresse verwendet. Die Konfiguration einer explizit angegebenen Startadresse kann natürlich nicht so allgemein angegeben werden, wie dies durch die Registeranzeige möglich ist, da die Startadresse lediglich einen Konfigurationsindex enthalten kann, die Registeranzeige jedoch die direkte Eingabe eines Konfigurationswertes erlaubt. Sehr wichtig: Der Befehl »g« setzt vor dem Start des Testprogramms eventuell gesetzte Breakpoints in das Testprogramm ein! Wie Breakpoints eingegeben werden, wird in den folgenden Kapiteln beschrieben.

Hauptsächlich in diesem letzten Punkt unterscheidet sich der Befehl »j« von »g«: »j« fügt **keine** Breakpoints ein. Weiter wird »j« immer von einer Startadresse gefolgt, dies soll die Notation der Syntax ohne eckige Klammern andeuten. Anders als beim TEDMON führt auch beim DBM-Befehl »j« nur ein »brk« zurück in die Registeranzeige.

Beispiel:

Geben Sie mittels des Zeilenassemblers das folgende kurze Programmstück ein:

```
.00c00 a9 10    lda #$10
.00c02 00       brk
```

Starten Sie nun dieses kleine Programm mit »j c00«. Sie sehen das Prinzip. Versuchen Sie gleiches mit dem Befehl »g« (unter der Voraussetzung, daß Sie bisher noch keine Breakpoints eingegeben haben).

4.8.3 Hot Spots

Bei der Einführung der »hot spots« habe ich mich von den Möglichkeiten der Debugger größerer Rechner beeinflussen lassen. Die Grundidee ist einfach: In den allermeisten Fällen enthält die reine Registeranzeige zu wenige Informationen, als daß der Tester viel daraus entnehmen könnte. Viel wichtiger als der aktuelle Inhalt der Register ist in vielen Fällen der Zustand bestimmter Speicherzellen, die das Testprogramm benutzt. Solche Speicherzellen und Speicherbereiche lassen sich durch die Hot Spots zusätzlich zur Registeranzeige darstellen.

Ein Hot Spot wird festgelegt durch den Befehl

```
sp <nr> <adr>
```

»<nr>« kann einen Wert von 0 bis 9 annehmen und bezeichnet die Nummer des gerade eingegebenen Spots – es sind also maximal 10 Spots möglich. »<adr>« ist eine Adresse (inklusive Konfigurationsindex); Diese Adresse und die folgenden Bytes werden in Form einer Hexdumpzeile parallel zur Registeranzeige auf dem Bildschirm gezeigt.

Geben Sie beispielsweise folgenden Befehl ein:

```
sp 0 12345
```

und lassen Sie sich anschließend die Registeranzeige neu zeigen. Sie sehen, daß nun unterhalb der eigentlichen Registeranzeige eine Dumpzeile erscheint. Diese Zeile (und maximal 9 weitere) wird bei jedem folgenden »brk« während der Breakpointbehandlung angezeigt; damit haben Sie eine Möglichkeit, insbesondere den Inhalt flüchtiger Speicherzellen bequem festzuhalten und auch während des Austestens zu ändern.

Alle Spots werden durch den Befehl

```
csp
```

wieder gelöscht, so daß nun wieder allein die Registeranzeige erscheint.

4.8.4 Breakpoints

Der DBM differenziert zwischen drei Arten von Breakpoints. Die drei Breakpointarten unterscheiden sich nicht im verwendeten Prinzip – Verwendung des Software-Interrupts »brk« -, sondern lediglich in der technischen Behandlung durch den Debugger:

(1) Der *unbedingte* Breakpoint: Das an der Breakpointadresse eingefügte »brk« führt in jedem Fall – unbedingt – zur Anzeige der Registerinhalte und der Hot Spots.

(2) Der *bedingte* Breakpoint: Der Debugger zählt, wie oft das an der Adresse des Breakpoints eingefügte »brk« angelaufen wurde, und bricht nach einer vorher festgelegten Anzahl das Testprogramm ab und zeigt die Registerinhalte und Spots.

(3) Der *Userbreakpoint*: Der Tester schreibt ein Maschinenprogramm, das jedesmal ausgeführt wird, wenn das Testprogramm auf einen Userbreakpoint trifft. In diesem Maschinenprogramm wird entschieden, ob das Testprogramm weiterlaufen soll oder ob die Registerinhalte angezeigt werden sollen. Dies ist die leistungsfähigste Art von Breakpoints.

4.8.4.1 Bedingte und unbedingte Breakpoints

Bedingte und unbedingte Breakpoints werden durch einen gemeinsamen Befehl eingegeben:

```
br <nr> <adr> [<n>]
```

»<nr>« ist die Nummer des Breakpoints und kann Werte von 0 bis 9 einnehmen, wodurch maximal 10 bedingte oder unbedingte Breakpoints möglich sind. »<adr>« gibt die Breakpointadresse an, die Adresse (einschließlich Konfigurationsindex), an der ein »brk« in das Testprogramm einzufügen ist, wenn der Testlauf durch »g« gestartet wird; den Originalbefehl, der durch das »brk« ersetzt wird, merkt sich der Debugger in einer Liste, um ihn jederzeit wieder einsetzen zu können. »<n>« schließlich gibt an, wie oft der Breakpoint angelaufen werden muß, bevor der Testlauf unterbrochen wird und die Register angezeigt werden. Fehlt »<n>«, handelt es sich um einen unbedingten Breakpoint.

Beispiel:

Geben Sie wieder ein kleines Maschinenprogramm mit dem Zeilenassembler ein:

```
.00c00 a2 00        ldx #$00
.00c02 a9 01        lda #$01
.00c04 9d 00 0d     sta $0d00,x
.00c07 e8           inx
.00c08 4c 02 0c     jmp $0c02
```

Eine mehr oder weniger sinnlose Schleife, die jedoch als Demonstrationsobjekt zu gebrauchen ist. Geben Sie jetzt einen Breakpoint ein:

```
br 0 c07
```

Mit dem Befehl

```
sb      (show break points)
```

können Sie sich eine Liste der aktuell eingegebenen Breakpoints anzeigen lassen. Der Breakpoint läßt sich durch Eingabe von »br 0 -« wieder aus der Liste entfernen. Mit

```
cbr     (clear break points)
```

wird die gesamte Liste gelöscht.

Mit dem Befehl

```
pn      (points)
```

können Sie ein Einsetzen aller Breakpoints der Liste in das Testprogramm erzwingen. Geben Sie diesen Befehl einmal ein, und sehen Sie sich unser Beispielprogramm disassembliert an:

```
pn
w0 c00 c10
wd 0
```

Sie sehen, daß ein »brk« in das Testprogramm eingefügt wurde. Umgekehrt können Sie auch wieder den Originalbefehl einsetzen lassen:

```
so          (set opcodes)
```

Disassemblieren Sie das Testprogramm noch einmal, und Sie werden sehen, daß das »inx« wieder eingesetzt ist.

Entfernen Sie den Breakpoint wieder aus dem Testprogramm (»so«), bevor Sie das weitere Beispiel nachvollziehen. Starten Sie anschließend das Programm mit

```
g c00
```

Praktisch ohne Zeitverzug wird die Registeranzeige erscheinen. Der Programmzähler steht auf der Adresse des Breakpoints ($c07); der Akkumulator enthält eine Eins, das X-Register ist zum Zeitpunkt des »brk« noch Null, das Testprogramm hat also noch keine Schleife ausgeführt. In der disassemblierten Zeile unterhalb der eigentlichen Registeranzeige sehen Sie den nächsten Befehl, der bei einem Neustart ausgeführt werden wird – das »inx«, das inzwischen durch den Debugger wieder an seine alte Position zurück geschrieben worden ist – der Befehl »g« hatte es durch ein »brk« ersetzt. Sie können sich noch einmal überzeugen, daß der Breakpoint aktuell entfernt ist, indem Sie das Testprogramm noch einmal disassemblieren lassen:

```
wd 0
```

Der bedingte Breakpoint läßt sich auf die gleiche Weise ausprobieren: Ändern Sie den unbedingten Breakpoint in einen bedingten um, indem Sie eingeben:

```
br 0 c07 +20
```

Dies bedeutet, daß erst das zwanzigste Auftreffen auf den Breakpoint zur Registeranzeige führen soll. Starten Sie das Testprogramm erneut mit

```
g c00
```

Die nun erscheinende Registeranzeige zeigt die gleiche Adresse wie vorher, allerdings enthält das X-Register diesmal den Wert $14 (dezimal 20) – folglich ist der Breakpoint 20mal angelaufen worden, ohne eine Reaktion auszulösen.

4.8.4.2 Userbreakpoints

Die bedingten Breakpoints sind zwar schon recht gut, allerdings taugen sie noch nicht dazu, kompliziertere Sachverhalte zu überprüfen; diese Aufgabe übernehmen die Userbreakpoints. Userbreakpoints werden gesetzt durch den Befehl:

```
us <nr> <adr> <up>
```

»<nr>« ist die Nummer des Userbreakpoints. Sie kann zwischen 0 und 4 liegen, es sind also 5 Userbreakpoints möglich. »<adr>« ist die Adresse des Breakpoints wie bei den anderen Breakpointarten. »<up>« bezeichnet die Adresse eines vom Tester geschriebenen Maschinenprogramms, das jedesmal dann ausgeführt wird, wenn das Testprogramm auf einen Userbreakpoint trifft. Auch die Adresse des Userprogramms wird inklusive Konfigurationsindex angegeben, falls es wichtig ist, daß das Programm in einer bestimmten Konfiguration abläuft.

Trifft das Testprogramm auf einen Userbreakpoint, führt das eingefügte »brk« zunächst wie bei den anderen Breakpointarten zum Debugger. Dieser vergleicht die Breakadresse mit der Liste der Breakpoints. Ist der Breakpoint ein Userbreakpoint, startet der Debugger das angegebene Userprogramm. Bei diesem Start enthalten die Prozessorregister A,X, Y und ST die Werte, mit denen das Testprogramm abgebrochen wurde; weitere Informationen befinden sich auf dem Prozessorstack in der hier angegebenen Reihenfolge:

```
PC + 1                          (+11)
PC                              (+10)
Prozessorstatus                 (+9)
Akkumulator                     (+8)
X-Register                      (+7)
Y-Register                      (+6)
Konfiguration beim »brk«        (+5)
4 durch den Debugger reservierte Bytes
```

Die in Klammern angegebenen Werte beziehen sich auf die Art, wie das Userprogramm diese Angaben erreichen kann. Hierzu kann folgende Befehlsfolge benutzt werden:

```
tsx
lda $100+n,x
```

wobei »n« den oben angegebenen Offset bezeichnet.

Das Userprogramm kann nun die komplexesten Abfragen durchführen, beispielsweise: »Wann ist die Summe aus X-Register, Y-Register und Akkumulator gleich einer bestimmten Zahl?« oder »Enthält die Speicherzelle XY den Wert Z, wenn gleichzeitig AB den Wert U enthält und das Y-Register gleich 10 ist?«. Die Art der

Abfrage hängt natürlich weitgehend davon ab, welchem Fehler Sie auf der Spur sind, aber ich denke, das Prinzip ist deutlich. Wichtig ist jedoch, zu beachten – und das ist die große Stärke des Breakpointverfahrens -, daß das auszutestende Programm während des Tests praktisch ohne Zeitverzögerung abläuft. Nur so ist es praktisch möglich, große Programme mit vertretbarem Zeitaufwand zu testen. Verwenden Sie Userbreakpoints, so wird das Testprogramm nur dann abgebrochen werden, wenn eine genau spezifizierte Bedingung oder auch eine Reihe von Bedingungen erfüllt ist.

Das Userprogramm muß mit einem »rts« abgeschlossen werden. Dieser Rücksprung führt wieder in den Debugger. Der Debugger entnimmt dem C-Flag beim Wiedereintritt die Information, ob er das Testprogramm weiterlaufen lassen soll oder ob die Register und Hot Spots anzeigt werden sollen: Ist das C-Flag gelöscht, läuft das Testprogramm weiter, ist es gesetzt, wird die Registeranzeige durchgeführt.

Das Ganze an einem Beispiel; als Testprogramm nehmen wir das im vorigen Kapitel schon eingetippte Beispiel. Das hier folgende Programm soll unser Userprogramm sein:

```
.00b00 ba          tsx
.00b01 bd 08 01    lda $0108,x
.00b04 c9 01       cmp #$01
.00b06 f0 02       beq $0b0a
.00b08 18          clc
.00b09 60          rts
.00b0a 38          sec
.00b0b 60          rts
```

Keine sehr sinnvolle Abfrage, die hier gemacht wird – es wird abgefragt, ob der Akkumulator des Testprogramms eine Eins enthält -, aber, um das Prinzip zu zeigen, reicht das Beispiel aus.

Löschen Sie nun die Breakpointliste mit

```
cbr
```

und geben Sie den Userbreakpoint an mit

```
us 0 c07 b00
```

Nun können Sie das Testprogramm starten. Sie werden feststellen, daß die Registeranzeige gleich im ersten Schleifenlauf eingeschaltet wird.

Es ist sehr schwierig, gerade zum Thema Userbreakpoints ein sinnvolles Beispiel zu finden, das nicht gleich den Umfang mehrerer Seiten annimmt. Ich glaube aber, das Prinzip ist einfach genug, um so im Raume stehenbleiben zu können.

Kapitel 5

REASS – der Reassembler

Immer wieder passiert es, daß man beim Disassemblieren eines in Maschinensprache geschriebenen Programmes feststellt, daß einige der Routinen gut in eigenen Programmen verwendet werden könnten. Doch den gesamten Quelltext von Hand wieder mit dem Editor einzugeben, dazu fehlt meistens die Lust. Hier hilft ein Reassembler, der aus einem Maschinenspracheprogramm wieder ein editierbares Textfile erzeugt.

5.1 Allgemeine Hinweise

REASS erzeugt vollkompatible Quellkodefiles für TOP-ASS, die im Editormodus weiter be- und verarbeitet werden können.

Doch nun zu den besonderen Fähigkeiten des REASS und seiner Bedienung.

REASS ist ein 2-Pass-Reassembler, d. h., er benötigt zwei Durchgänge, um ein Maschinenspracheprogramm komplett rückzuübersetzen. Es können bis zu 2500 Label verwaltet werden. Im 2. Pass wird auch eine Referenztabelle aller Zugriffe auf die Label erzeugt.

Der Maschinenkode kann aus dem ROM-/RAM-Bereich des C128, von einer Diskette oder aus dem ROM-/RAM-Bereich der Floppylaufwerke gelesen werden. Verarbeitet wird Kode der Prozessoren 6502/8502 (auch mit den illegalen Opcodes) und der Prozessoren 65C02 und 65SC02. Da TOPASS weder die illegalen Opcodes noch die Befehlscodes der CMOS-Prozessoren kennt, werden in diesen Betriebsmodi die Kodes als HEX-Werte in ».BYTE«-Pseudobefehle und die Mnemonics in das Kommentarfeld geschrieben. In Kapitel 1.21 sind die illegalen Opcodes und in Kapitel 1.22 die zusätzlichen Befehle der CMOS-Versionen beschrieben. Im Anhang sind alle Befehle mit ihren Adressierungsarten zur schnellen Referenz aufgelistet.

Der erzeugte Quelltext wird auf den Bildschirm ausgegeben. Zusätzlich kann die Ausgabe auf den Drucker, selbstverständlich mit Titelkopf und Seitennummer, und auf Diskette geleitet werden.

REASS fragt über Menüs und Einzelfragen alle benötigten Betriebsparameter ab. Im folgenden werden die Abfragen und ihre Beantwortung behandelt. Es müssen nicht immer alle der hier behandelten Menüs und Fragen erscheinen. Sie sind jeweils von den vorher eingegebenen Daten abhängig.

Alle Eingaben, die ungleich den vorgegebenen Wahlmöglichkeiten sind, werden ignoriert. Soweit möglich, werden alle Eingaben auf Korrektheit geprüft.

Fehlerzustände, die während des Laufes auftreten, werden von REASS aufgefangen, angezeigt und entweder automatisch oder durch interaktive Eingaben korrigiert.

Durch Eingabe eines »Klammeraffen«, das Zeichen im ASCII-Zeichensatz oder des Paragraphenzeichens im DIN-Zeichensatz, kann bei fast allen Eingaben und während der Passes 1 und 2 der Programmlauf abgebrochen werden.

5.2 Bedienungsanleitung

Die Bedienung des REASS ist sehr einfach. Im folgenden werden alle Abfragen, die Sie zu beantworten haben, und alle Wahlmöglichkeiten beschrieben.

5.2.1 Programmstart

Das Programm wird mit

```
RUN »REASS« <RETURN>
```

geladen und automatisch gestartet.

Nach Programmabbruch oder -ende kann REASS nur mehr durch Neuladen wieder gestartet werden.

Als erstes haben Sie die Möglichkeit, die Zeichen- und die Hintergrundfarbe Ihrem Geschmack entsprechend zu wählen. Dann prüft REASS, ob der 80-Zeichen-Bildschirm eingeschaltet ist. Ist er es nicht, schaltet REASS automatisch nach einem Hinweis um.

5.2.2 Maschinenkode lesen von:

Mit diesem ersten Menü kann die Quelle, von der der Maschinenkode gelesen werden soll, festgelegt werden. Die möglichen Eingabekanäle sind die ROM- und die RAM-Bereiche des C128, die Diskette oder der ROM-/RAM-Bereich eines der Floppylaufwerke.

– C128 ROM-/RAM-Bereich

Soll das Programm aus dem C128-ROM/RAM gelesen werden, möchte REASS die Bank wissen, aus der gelesen werden soll.

– Programmfile von der Diskette

Ist die Quelle die Floppy, müssen Programmname, Geräte- und die Laufwerknummer eingegeben werden. Letztere ist nötig, falls Sie mit einem Doppellaufwerk arbeiten. REASS prüft, ob die Floppy eingeschaltet und ansprechbar ist und sich das angegebene Programmfile auf der eingelegten Diskette befindet.

– Floppy ROM-/RAM-Bereich

Soll aus dem RAM/ROM der Floppy gelesen werden, wird nur nach Geräte- und Laufwerknummer gefragt.

5.2.3 Maschinenkode ist für:

Im zweiten Menü wird festgelegt, für welchen Prozessor das Maschinensprache-programm geschrieben ist. REASS kann Kode für die Prozessoren 65xx, 85xx und die CMOS-Versionen 65C02 und 65SC02 verarbeiten. Da 65xx und 85xx einige nicht dokumentierte Zusatzbefehle haben, kann REASS auch diese übersetzen.

Da TOP-ASS nur über Makros in der Lage ist, diese Mnemonics zu übersetzen, wird der Kode in die Assembler-Anweisung ».BYTE« geschrieben und die Mnemonics in das Kommentarfeld.

5.2.4 Quelltext ablegen?

REASS möchte nun wissen, ob der erzeugte Quelltext auf Diskette abgelegt werden soll. Wenn ja, dann muß der Filename, unter dem abgespeichert werden soll, mit der Floppy- und der Laufwerksnummer eingegeben werden. REASS hängt an den eingegebenen Filenamen zur Kennzeichnung ein ».ASM« an.

Zusätzlich braucht REASS die Angabe, wieviel Zeilen er in das File schreiben soll. Wird diese Anzahl überschritten, erzeugt REASS automatisch ein neues File, bei dessen Namen eine fortlaufende Nummer zusätzlich angehängt wird. In diesem Fall

werden die von TOP-ASS benötigten Assembleranweisungen ».CHAIN«, ».CONTINUED« und ».CHAINEND« ebenfalls miterzeugt.

5.2.5 Quelltext drucken?

Jetzt kann man sich entscheiden, ob der erzeugte Quelltext ausgedruckt werden soll. Wenn ja, benötigt REASS den Titelkopf und die erste Seitennummer.

5.2.6 Erste Zeilennummer?

Die nächste Eingabe dient zur Festlegung der ersten Zeilennummer und des Inkrements, mit dem die Zeilennummern im Quelltextfile hochgezählt werden sollen. Erlaubte Eingaben liegen im Bereich des vom BASIC-Interpreter vorgegebenen Zeilennummernbereiches.

Tritt beim Reassemblieren eine Bereichsüberschreitung (Zeilennummer > 65535) auf, verlangt REASS von Ihnen einen neuen Zeilennummernbereich.

5.2.7 Eingabe der Blöcke

Jetzt kommt eine besondere Eingabe, die die weitere Bearbeitung des Quelltextes bedeutend erleichtern kann:

Es müssen die Adreßblöcke mit der Beschreibung der Kodeart definiert werden. Leicht hat man es, wenn das Maschinenspracheprogramm nur aus reinem Befehlskode besteht, denn dann muß nur ein Block der Art »C« eingegeben werden.

Bei langen Programmen ist aber meist Befehlskode mit Textfeldern, Adreß- und Bytetabellen vermischt.

Stößt man bei einer ersten Überprüfung des Maschinenspracheprogrammes mit einem Disassembler – z. B. mit dem im C128 fest implementierten Disassembler des TEDMON oder mit dem des DBM-Monitors/Debuggers – auf solche Blöcke, müssen die Start-, die Endadressen und die Kodeart notiert werden. Diese Adressen werden jetzt hexadezimal eingegeben. Bis zu 40 Adreßblöcke können festgelegt werden, die je nach Eingabedefinition als ».BYTE«-Felder, Befehlskode, ».Text«-Felder, ».WORD«-Tabellen oder Dual-Felder interpretiert werden. Zusätzlich können Blöcke zum Überlesen markiert werden. Diese Funktion ist für Programme, die von Diskette gelesen werden und von denen nicht der gesamte Kode reassembliert werden soll, wertvoll.

Eine Besonderheit sind die Dualfelder. REASS zerlegt jedes Byte in seine Dualzahl und schreibt diese mit einem Stern für jedes gesetzte Bit und einem Punkt für jedes

nicht gesetzte Bit in das Kommentarfeld. Auf diese Weise hat man die Möglichkeit, Character-ROMs auszulesen. Die Darstellung ist etwas gewöhnungsbedürftig, da die Zeichen spiegelbildlich ausgegeben werden.

– Korrektur:

Sollte man sich bei der Eingabe in der vorhergehenden Zeile vertippt haben, so ist das weiter nicht schlimm. Mit der Taste **<Pfeil aufwärts>** kann eine Eingabezeile zurückgesprungen und die Eingabe wiederholt werden.

– Eingabeende:

Beendet wird die Eingabe durch eine Leereingabe (= nur Drücken der Taste <RETURN>) oder nach der Eingabe für den 40. Adreßblock.

5.2.8 Ecklabel

Als weiteres Bonbon können »Ecklabel« eingegeben werden. Damit besteht die Möglichkeit, vor Beginn der Reassemblierung bekannte Label und Adressen im Programm, die von außerhalb (extern) angesprungen werden, in die Labeltabelle einzutragen (wichtig z. B. zum Erkennen des berühmten »BIT«-Tricks).

– Korrektur:

Fehleingaben können durch Neueingabe der Adresse und des Labels korrigiert werden.

– Eingabeende:

Beendet wird diese Funktion durch eine Leereingabe, d. h., nur durch Drücken der Taste <RETURN>.

5.2.9 Pass 1

Im 1. Pass werden alle Sprungziele gesucht und mit automatisch generierten Labeln (Ausnahme: die Ecklabel) versehen. Die Label werden aus dem vorangestellten Buchstaben J (für Sprung- und Branchziele) oder einem L für alle anderen Label mit angehängter Sedezimaladresse gebildet. Der erzeugte Text wird auf dem Bildschirm gezeigt.

5.2.10 Label einlesen?

Nach dem 1. Pass kann ein sequentielles File mit z. B. den Commodore-Systemlabeln eingelesen werden. Dieses Labelfile muß folgenden Aufbau besitzen:

Adresse in dezimal; CHR$(13); Labelname; CHR$(13); Kommentar

Die Einträge in diesem File müssen nach aufsteigenden Adressen sortiert sein. REASS übernimmt dann alle Labelnamen, deren Adressen innerhalb der eingegebenen Adreßblöcke des Programmes liegen und überschreibt dabei die automatisch generierten Label (leider auch evtl. die Ecklabel).

Nach dem Einlesen des ersten Labelfiles können bei Beantwortung der Frage »Weitere Label einlesen ?« mit »J« = ja weitere Files zum Einlesen angegeben werden.

5.2.11 Label eingeben?

Zusätzlich kann danach die Labelliste von Hand editiert werden. Alle Label werden mit ihren Adressen auf dem Bildschirm, sortiert nach Adressen, angezeigt. Der Labelname kann überschrieben werden oder nur durch Eingabe eines <RETURN> unverändert übernommen werden.

– Korrektur:

Wieder besteht durch Drücken der Taste <Pfeil aufwärts> die Möglichkeit, Fehleingaben zu korrigieren.

– Eingabeende:

Das Editieren wird durch Drücken der Taste < > (= Klammeraffe, bzw. beim DIN-Zeichensatz die Taste <Paragraph>) vorzeitig oder nach Erreichen des letzten Eintrags in der Labelliste beendet.

5.2.12 Pass 2 machen?

Wenn man die Frage »2. Pass machen ?« mit N (= Nein) beantwortet, kann man die erstellte Labelliste ausdrucken und auf Floppy abspeichern (siehe 5.2.13).

Doch wir wollen weiter machen.

Im 2. Pass wird der erzeugte Quelltext, je nach der am Programmanfang getätigten Wahl, auch auf dem Drucker und/oder der Diskette ausgegeben. Als erstes wird der Labelvorspann mit allen externen Labeln und mit den von TOP-ASS benötigten Pseudobefehlen ».BASE« und evtl. ».CHAIN« erzeugt. ».CHAIN« wird zur Verkettung von Files benötigt, wenn der erzeugte Text länger als die beim Programmstart angegebene Anzahl Zeilen/File ist. REASS legt dann automatisch weitere Files mit »NAME«+Zählnummer an. Alle Quelltextfiles haben die Namensextension ».ASM« zur Unterscheidung von anderen Files. In diesem Pass wird der schon oben erwähnte Programmiertrick mit dem Befehl BIT dann erkannt,

wenn die Adresse nach dem BIT-Befehl ein Sprungziel ist. Mit dem Befehl BIT können ein oder zwei Byte übersprungen werden. Ein weiterer Trick ist in der Übersetzung von ».WORD«-Tabellen eingebaut. REASS sucht für diese Adresse ein Label. Wird kein Label gefunden, sucht REASS bei der Adresse+1. Bei positivem Suchergebnis wird »LABEL-1« übernommen, bei negativem wird bei Adresse-1 gesucht und evtl. »Label+1« in die Labelliste eingetragen. Sind die Suchoperationen in dieser Reihenfolge alle ergebnislos geblieben, legt REASS automatisch ein Label an. Am Ende des 2. Laufes wird die Anzahl der übersetzten Bytes, die Anzahl der Einträge in der Labelliste (muß nicht mit der Anzahl der ausgedruckten Label übereinstimmen, da alle »-1« und »+1« Label bei der Ausgabe unterdrückt werden) und Anzahl und Namen der evtl. erzeugten Quelltextfiles ausgegeben.

5.2.13 Label speichern?

Wird diese Frage mit »J« = ja beantwortet, möchte REASS die Floppy- und die Laufwerknummer wissen, auf die das Labelfile geschrieben werden soll. Das File wird unter dem Quelltextnamen mit der Extension ».LAB« abgespeichert. Ist das Quelltextfile nicht auf Diskette geschrieben worden, wird zusätzlich noch nach dem Filenamen gefragt.

Dieses File kann mit einem geeigneten Dateiprogramm für sequentielle Files weiter bearbeitet werden. Auf diese Art und Weise erhält man Labelfiles mit allen wichtigen Adressen des C128.

5.2.14 Label drucken?

Bei Beantwortung mit »J« = ja werden die Label, sortiert nach Adressen, mit allen Referenzen, Befehlen und Adressierungen aufgelistet. Diese Tabelle hat folgenden Aufbau:

Ziel- Ziel- : von Cmd Adress.- von Cmd Adress.- adresse

label : Adresse art Adresse art

»Zieladresse« und »Ziellabel« ist das Ziel einer Operation, »von Adresse« ist die Adresse des Befehls, von dem aus der Labelzugriff erfolgt. »Cmd« ist das jeweilige Befehlsmnemonic und unter »Adressierungsart« ist das Symbol entsprechend den Kennzeichnungen im Kapitel 1.6 – natürlich ohne Operand – zu finden.

Anschließend werden die Label aufsteigend nach Namen sortiert und dann ausgegeben.

5.2.15 Programmende: Neustart?

Es besteht die Möglichkeit, einen neuen Reassemblierlauf, mit oder ohne Verwendung der im alten Lauf generierten Label, zu starten. Ansonsten wird das Programm endgültig beendet. Zum Wiederstarten muß es dann, wie in Kapitel 5.2.1 beschrie-ben, neu von der Diskette geladen und gestartet werden.

5.3 Die Fehlermeldungen des REASS

REASS versucht, alle Eingaben auf Plausibilität und Korrektheit zu überprüfen, um damit Fehlfunktionen von vornherein zu verhindern.

Trotzdem kann es beim Programmlauf zu Störungen kommen. REASS fängt sie ab und meldet den Fehler. Sie haben dann bei den meisten Fehlern die Möglichkeit, interaktiv einzugreifen, um ihn zu beheben.

Im folgenden werden die Fehlermeldungen und die mögliche Behebung beschrieben:

Fehlermeldung	Beschreibung und Behebung
Drucker nicht eingeschaltet	REASS fordert zum Einschalten auf, wartet auf einen Tastendruck und überprüft das Gerät erneut.
Floppy nicht eingeschaltet	siehe »Drucker nicht eingeschaltet«
Floppyfehler 21,26,74	REASS druckt die Fehlermeldung mit einem Hinweis aus, wartet auf einen Tastendruck, initialisiert die Diskette neu und versucht die Operation nocheinmal. Drücken der Taste " " oder Paragraph beendet eine Endlos-schleife durch Programmabbruch.
File nicht gefunden	Man kann die Diskette wechseln oder den richtigen Namen eingeben
File existiert	Hier hat man 3 Möglichkeiten auf diese Fehlermeldung zu reagieren: 1. das File auf Diskette löschen, 2. das File auf Disk umbenennen, 3. den Namen des abzuspeichernden Files ändern.

Diskette ist voll	Es kann eine neue Diskette eingelegt und auf Wunsch formatiert werden.
Fataler Diskfehler	Hier kann REASS leider nicht mehr helfen, das Programm wird mit einer Meldung abgebrochen.
Labelüberlauf	Die Anzahl der Label ist zu groß geworden. Beim ersten Auftreten wird diese Fehlermeldung ausgegeben. Nach der Bestätigung arbeitet das Programm weiter, es werden aber keine neuen Label mehr in die Labelliste eingetragen. REASS setzt anstelle von Label dann die Adressen ein.
Speicherüberlauf	Der Speicher ist für die Label zu klein (siehe Labelüberlauf).
Zeilennummernüberlauf	Besteht die Gefahr, daß ein Zeilennummernüberlauf auftritt, verlangt REASS von Ihnen die Eingabe eines neuen Bereiches.

Damit sind wir am Ende der Bedienungsanleitung des REASS angelangt. Wir hoffen, Sie werden das gesamte Assembler-, Debugger- und Reassembler-Paket als komfortable Programmierumgebung mit Gewinn einsetzen können.

Anhang

Alphabetische Tabelle der Prozessorbefehle und Opcodes

Mnem.	–	zp	zx	zy	ab	ax	ay	ix	iy	im
abc		65	35		6d	7d	79	61	71	69
and		29	35		2d	3d	39	21	31	29
asl	0a	06	16		0e	1e				
bcc	90									
bcs	b0									
beq	f0									
bit		24			2c					
bmi	30									
bne	d0									
bpl	10									
brk	00									
bvc	50									
bvs	70									
clc	18									
cld	d8									
cli	58									
clv	b8									
cmp		c5	d5		cd	dd	d9	c1	d1	c9
cpx		e4			ec					e0

Mnem.	–	zp	zx	zy	ab	ax	ay	ix	iy	im
cpy		c4			cc					c0
dec		c6	d6		ce	de				
dex	ca									
dey	88									
eor		45	55		4d	5d	59	41	51	49
inc		e6	f6		ee	fe				
inx	e8									
iny	c8									
jmp					4c			6c		
jsr					20					
lda		a5	b5		ad	bd	b9	a1	b1	a9
ldx		a6		b6	ae		be			a2
ldy		a4	b4		ac	bc				a0
lsr	4a	46	56		4e	5e				
nop	ea									
ora		05	15		0d	1d	19	01	11	09
pha	48									
php	08									
pla	68									
plp	28									
rol	2a	26	36		2e	3e				

Mnem.	–	zp	zx	zy	ab	ax	ay	ix	iy	im
ror	6a	66	76		6e	7e				
rti	40									
rts	60									
sbc		e5	f5		ed	fd	f9	e1	f1	f9
sec	38									
sed	f8									
sei	78									
sta		85	95		8d	9d	99	81	91	
stx		86		96	8e					
sty		84	94		8c					
tax	aa									
tay	a8									
tsx	ba									
txa	8a									
txs	9a									
tya	98									

Adressierungsarten:

– = implizit oder Akkumulator, bei Branches entsprechend eine Adreßdistanz
zp = Zeropage absolut
zx = Zeropage X-indiziert
zy = Zeropage Y-indiziert
ab = absolut
ax = absolut X-indiziert
ay = absolut Y-indiziert
ix = indirekt, vorindiziert; bei Jump indirekter Sprung
iy = indirekt, nachindiziert
im = unmittelbar (immediate)

Alphabetische Tabelle der illegalen Opcodes

Mnem.	–	zp	zx	zy	ab	ax	ay	ix	iy	im
a11						9c	9e			
aax		87		97	8f					8b
asr										6b
arr										4b
axs										cb
dcp		c7	d7		cf	df	db	c3	d3	
dop	*)									
isc		e7	f7		ef	ff	fb	e3	f3	
kil	*)									
lar							bb			
lax		a7	b7		af		bf	a3	b3	
nop	*)									
rla		27	37		2f	3f	3b	23	33	
rra		67	77		6f	7f	7b	63	73	
slo		07	17		0f	1f	1b	13	03	
sre		47	57		4f	5f	5b	43	53	
top	*)									

Keiner der illegalen Kodes wird durch den ASE-Assembler oder den Monitor direkt unterstützt. Eine Einführung in einen Quelltext über Makrobibliotheken ist jedoch möglich.

Die mit dem Stern gekennzeichneten Befehle haben mehrere gleichwertige Opcodes:

dop: 04, 14, 34, 44, 54, 64, 74, d4, f4, 80, 89, 93

kil: 02, 12, 22, 32, 42, 52, 62, 72, 92, b2, d2, f2

nop: 1a, 3a, 5a, 7a, da, fa

top: 0c, 1c, 3c, 5c, 7c, dc, fc

(alle Zahlenangaben sind sedezimal)

Nach Wert sortierte Übersicht über die Prozessorbefehle inklusive illegaler Opcodes und 65C02

Opcode hex.	Opcode dez.	6502	illegal	65C02
00	0	brk		
01	1	ora ix		
02	2		kil	
03	3		slo iy	
04	4		dop	tsb zp
05	5	ora zp		
06	6	asl zp		
07	7		slo zp	rmb 0,zp
08	8	php		
09	9	ora im		
0a	10	asl a		
0b	11		top	
0c	12			tsb ab
0d	13	ora ab		
0e	14	asl ab		
0f	15		slo ab	bbr 0,rel
10	16	bpl rel		
11	17	ora iy		
12	18		kil	ora ()
13	19		slo ix	
14	20		dop	trb zp
15	21	ora zx		
16	22	asl zx		
17	23		slo zx	rmb 1,zp
18	24	clc		
19	25	ora ay		
1a	26		nop	inc a
1b	27		slo ay	
1c	28		top	trb ab
1d	29	ora ax		
1e	30	asl ax		
1f	31		slo ax	bbr 1,rel
20	32	jsr ab		

Opcode hex.	Opcode dez.	6502	illegal	65C02
21	33	and ix		
22	34		kil	
23	35		rla ix	
24	36	bit zp		
25	37	and zp		
26	38	rol zp		
27	39		rla zp	rmb 2,zp
28	40	plp		
29	41	and im		
2a	42	rol a		
2b	43			
2c	44	bit ab		
2d	45	and ab		
2e	46	rol ab		
2f	47		rla ab	bbr 2,rel
30	48	bmi rel		
31	49	and ix		
32	50		kil	and ()
33	51		rla iy	
34	52		dop	bit zx
35	53	and zx		
36	54	rol zx		
37	55		rla zx	rmb 3,zp
38	56	sec		
39	57	and ay		
3a	58		nop	dec a
3b	59		rla ay	
3c	60		top	bit ax
3d	61	and ax		
3e	62	rol ax		
3f	63		rla ax	bbr 3,rel
40	64	rti		
41	65	eor ix		
42	66		kil	
43	67		sre ix	
44	68		dop	
45	69	eor zp		
46	70	lsr zp		

Opcode hex.	Opcode dez.	6502	illegal	65C02
47	71		sre zp	rmb 4,zp
48	72	pha		
49	73	eor im		
4a	74	lsr a		
4b	75		arr im	
4c	76	jmp ab		
4d	77	eor ab		
4e	78	lsr ab		
4f	79		sre ab	bbr 4,rel
50	80	bvc rel		
51	81	eor iy		
52	82		kil	eor ()
53	83		sre iy	
54	84		dop	
55	85	eor zx		
56	86	lsr zx		
57	87		sre zx	rmb 5,zp
58	88	cli		
59	89	eor ay		
5a	90		nop	phy
5b	91		sre ay	
5c	92		top	
5d	93	eor ax		
5e	94	lsr ax		
5f	95		sre ax	bbr 5,rel
60	96	rts		
61	97	adc ix		
62	98		kil	
63	99		rra ix	
64	100		dop	stz zp
65	101	adc zp		
66	102	ror zp		
67	103		rra zp	rmb 6,zp
68	104	pla		
69	105	adc im		
6a	106	ror a		
6b	107		asr im	
6c	108	jmp ()		

Opcode hex.	Opcode dez.	6502	illegal	65C02
6d	109	adc ab		
6e	110	ror ab		
6f	111		rra ab	bbr 6,rel
70	112	bvs rel		
71	113	adc iy		
72	114		kil	adc ()
73	115		rra iy	
74	116		dop	stz zx
75	117	adc zx		
76	118	ror zx		
77	119		rra zx	rmb 7,zp
78	120	sei		
79	121	adc ay		
7a	122		nop	ply
7b	123		rra ay	
7c	124		top	jmp ix
7d	125	adc ax		
7e	126	ror ax		
7f	127		rra ax	bbr 7,rel
80	128		dop	bra rel
81	129	sta ix		
82	130			
83	131		aax ix	
84	132	sty zp		
85	133	sta zp		
86	134	stx zp		
87	135		aax zp	smb 0,zp
88	136	dey		
89	137		dop	bit im
8a	138	txa		
8b	139		aax im	
8c	140	sty ab		
8d	141	sta ab		
8e	142	stx ab		
8f	143		aax ab	bbs 0,rel
90	144	bcc rel		
91	145	sta iy		
92	146		kil	sta ()

Opcode hex.	Opcode dez.	6502	illegal	65C02
93	147		dop	
94	148	sty zx		
95	149	sta zx		
96	150	stx zy		
97	151		aax zy	smb 1,zp
98	152	tya		
99	153	sta ay		
9a	154	txs		
9b	155			
9c	156		a11 ax	stz ab
9d	157	sta ax		
9e	158		a11 ay	stz ax
9f	159			bbs 1,rel
a0	160	ldy im		
a1	161	lda ix		
a2	162	ldx im		
a3	163		lax ix	
a4	164	ldy zp		
a5	165	lda zp		
a6	166	ldx zp		
a7	167		lax zp	smb 2,zp
a8	168	tay		
a9	169	lda im		
aa	170	tax		
ab	171			
ac	172	ldy ab		
ad	173	lda ab		
ae	174	ldx ab		
af	175		lax ab	bbs 2,rel
b0	176	bcs rel		
b1	177	lda iy		
b2	178		kil	lda ()
b3	179		lax iy	
b4	180	ldy zx		
b5	181	lda zx		
b6	182	ldx zy		
b7	183		lax zy	smb 3,zp
b8	184	clv		

Opcode hex.	Opcode dez.	6502	illegal	65C02
b9	185	lda ay		
ba	186	tsx		
bb	187		lar ay	
bc	188	ldy ax		
bd	189	lda ax		
be	190	ldx ay		
bf	191		lax ay	
c0	192	cpy im		
c1	193	cmp ix		
c2	194			
c3	195		dcp ix	
c4	196	cpy zp		
c5	197	cmp zp		
c6	198	dec zp		
c7	199		dcp zp	
c8	200	iny		
c9	201	cmp im		
ca	202	dex		
cb	203		axs im	
cc	204	cpy ab		
cd	205	cmp ab		
ce	206	dec ab		
cf	207		dcp ab	bbs 4,rel
d0	208	bne rel		
d1	209	cmp iy		
d2	210		kil	cmp ()
d3	211		dcp iy	
d4	212		dop	
d5	213	cmp zx		
d6	214	dec zx		
d7	215		dcp zx	smb 5,zp
d8	216	cld		
d9	217	cmp ay		
da	218		nop	phx
db	219		dcp ay	
dc	220		top	
dd	221	cmp ax		
de	222	dec ax		

Opcode hex.	dez.	6502	illegal	65C02
df	223		dcp ax	bbs 5,rel
e0	224	cpx im		
e1	225	sbc ix		
e2	226			
e3	227		isc ix	
e4	228	cpx zp		
e5	229	sbc zp		
e6	230	inc zp		
e7	231		isc zp	smb 6,zp
e8	232	inx		
e9	233	sbc im		
ea	234	nop		
eb	235			
ec	236	cpx ab		
ed	237	sbc ab		
ee	238	inc ab		
ef	239		isc ab	bbs 6,rel
f0	240	beq rel		
f1	241	sbc iy		
f2	242		kil	sbc ()
f3	243		isc iy	
f4	244		dop	
f5	245	sbc zx		
f6	246	inc zx		
f7	247		isc zx	smb 7,zp
f8	248	sed		
f9	249	sbc ay		
fa	250		nop	plx
fb	251		isc ay	
fc	252		top	
fd	253	sbc ax		
fe	254	inc ax		
ff	255		isc ax	bbs 7,rel

Beeinflussung der Prozessorflags

Befehl	N	V	B	D	I	Z	C
adc	x	x				x	
and	x					x	
asl	x					x	x
bcc							
bcs							
beq							
bit	7	6				x	
bmi							
bne							
bpl							
brk			1		1		
bvc							
bvs							
clc							0
cld				0			
cli					0		
clv		0					
cmp	x					x	x
cpx	x					x	x
cpy	x					x	x
dec	x					x	
dex	x					x	
dey	x					x	
eor	x					x	
inc	x					x	
inx	x					x	
iny	x					x	
jmp							
jsr							
lda	x					x	
ldx	x					x	
ldy	x					x	
lsr	0					x	x
nop							
ora	x					x	
pha							
php							

Befehl	N	V	B	D	I	Z	C
pla	x					x	
plp	x	x	x	x	x	x	x
rol	x					x	x
ror	x					x	x
rti	x	x	x	x	x	x	x
rts							
sbc	x	x				x	x
sec							1
sed				1			
sei					1		
sta							
stx							
sty							
tax	x					x	
tay	x					x	
tsx	x					x	
txa	x					x	
txs							
tya	x					x	

Es bedeutet:

x – Beeinflussung je nach Operation
1 – Flag wird gesetzt
0 – Flag wird gelöscht
7 – Bit 7 des Operanden erscheint im Statusregister
6 – Bit 6 des Operanden erscheint im Statusregister

Alphabetische Tabelle der ASE-Befehle

.alter	– Alternative zu ».cond«.
.append "file",u	– Quelltextverkettung mit Löschen der Variablen.
.base <adr>	– Startadresse der Assemblierung.
.begin	– Start eines lokalen Blocks.
.break	– Unbedingter Abbruch der Assemblierung.
.byte "abc",1,2	– Tabelle von Bytewerten, ASCII-Zeichen und Bildkodewerten.
.cdlins	– Einschalten der Ausgabe bedingt assemblierter Zeilen bei Ausdruck eines Quelltextes.
.chain	– Einleitung der Chain-Verkettung
.chainend "file",u	– Ende der Chain-Verkettung.
.clist	– Vorzeitiges Ende der Druckausgabe mit ».list«.
.code <n>,<adr>	– Verschiebung der Kodeablage nach Bank »<n>«, Adresse »<adr>«.
.common <v1>,<v2>..	– Labelübergabe der Label »<v1>«, »<v2>« usw.; lokaler Fall.
.common <v1>:<v2>:	– Gleiches für globale Übergabe.
.cond <expr>	– Einleitung der bedingten Assemblierung.
.cond p1	– »Falls Pass 1«.
.cond p2	– »Falls Pass 2«.
.cond p3	– »Falls Pass 3«.
.continued "file",u	– Fortsetzung der Chain-Verkettung.
.control <expr>	– Bedingte Assemblierung mit mehreren Alternativen.
.define <v> = <exp>	– Wertzuweisung lokal.
.define <v>:= <exp>	– Wertzuweisung global.
.econd	– Ende der mit ».cond« eingeleiteten bedingten Assemblierung.
.end	– Ende eines mit ».begin« eingeleiteten lokalen Blocks.
.extern <v>,<v1>..	– Deklaration externer Referenzen.
.link "file",u	– Anweisung zum Linken eines Moduls.
.list <opnpar>	– Befehl zur Ausgabe eines Listings.
.lprint "ab",0	– Senden von Steuerzeichen über den ».list«- Kanal.
.macro m(a,b)	– Einleitung einer Makrodefinition.
.macend	– Ende einer Makrodefinition.
.maclib "file",u	– Setzen der aktuellen Makrobibliothek.
.malins	– Anweisung zum Ausdruck der durchlaufenen Zeilen der Makrodefinitionen beim Ausdruck des Quelltextes.
.modul "file",u	– Erzeugung eines relokatiblen Moduls.
.names <adr>	– Verlegung des Namensstacks nach »<adr>« in RAM-Bank 1.

.nonums – Keine Zeilennummern ausgeben beim Ausdruck des Quelltextes.
.nops – Keine Opcodes und Adressen ausgeben.
.object "file",u – Erzeugten Maschinenkode zur Floppy in ein File namens »file« senden.
.objend – Vorzeitiges Ende des Sendens des Objektkodes.
.page – Erzwungene neue Seite beim Ausdruck.
.print "abc" – Ausgabe einer Meldung auf den Bildschirm.
.public <v>,<v1>.. – Deklaration eines vorher mit ».define« definierten Labels als Entry.
.request "ab",<v> – Interaktive Eingabe eines lokalen Labels / einer lokalen Variablen in Pass 1.
.request "ab":<v> – Globaler Fall.
.revers "abc" – Tabelle aus reversem Bildkode.
.screen "abc" – Tabelle aus »normalem« Bildkode.
.set <v> – Redefiniton eines internen Labels.
.skip <n>,<m> – Einstellung der Druckseitenlänge auf »<n>« auszugebende Quelltextzeilen; »<m>« Zeilenvorschübe zum Überspringen der Perforation.
.source "file",u – Einbindung eines Quelltextes.
.space of <n>,<b> – Reservierung eines Tabellenbereiches von »<n>« Bytes, vorbesetzt mit dem Wert »<b>«.
.symbols <a>,<opnp> – Ausgabe der Symboltabelle, nach Kriterium »<a>« sortiert.
.syntax – Assemblierung ohne Kodeerzeugung (Testlauf).
.wait – Warten auf Drücken der Taste »Return«.
.word <adr>,<adr>.. – Tabelle aus Adreßwerten (2 Byte).
..m (a,b) – Aufruf eines RAM-Makros.
...m (a,b) – Aufruf eines Makros aus einer Makrobibliothek.
.= – Eine Alternative der mit ».control« eigeleiteten bedingten Assemblierung.
.– – Ende der mit ».control« eingeleiteten bedingten Assemblierung.

Minimacs:

.lax <adr> – Laden des Akkumulators und des X-Registers mit dem Inhalt zweier aufeinanderfolgender Speicherstellen.
.lay <adr> – dto. mit Akku und Y-Register.
.lxy <adr> – dto. mit X- und Y-Register.
.liax <adr> – Laden des Akkumulators und des X-Registers mit Low- und Highbyte eines Adreßwertes.
.liay <adr> – dto. mit Akku und Y-Register.
.lixy <adr> – dto. mit X- und Y-Register.

.sax <adr> – Abspeichern von Akku und X-Register in zwei aufeinanderfolgenden Speicherzellen.
.say <adr> – dto. für Akku und Y-Register.
.sxy <adr> – dto. für X- und Y-Register.
.mov <adr1>,<adr2> – Inhalt der Speicherzellen <adr1, adr1+1> nach <adr2, adr2+1> bringen.
.mvi <adr1>,<adr2> – Low- und Highbyte von <adr1> in die Speicherzellen <adr2,adr2+1> bringen.
.shl <adr> – Aufeinanderfolgende Speicherzellen um eine Stelle nach links verschieben, Nullbit nachschieben.
.slc <adr> – dto. und Inhalt des C-Flags nachschieben.
.shr <adr> – wie ».shl« mit Rechtsverschiebung.
.src <adr> – wie ».slc« mit Rechtsverschiebung.
.add <adr1>,<adr2> – Addition zweier Pointer, Ergebnis in »<adr2>«.
.adl <adr> – Addition des Akkumulators zum Pointer »<adr>«.
.adi <adr1>,<adr2> – Addition der Konstante »<adr1>« zum Pointer »<adr2>«.
.sub <adr1>,<adr2> – Wie ».add« zur Subtraktion.
.sbi <adr1>,<adr2> – Wie ».adi« zur Subtraktion.
.cpr <adr1>,<adr2> – Vergleich zweier Pointer, Ergebnis im Prozessorstatus.
.cpi <adr1>,<adr2> – Vergleich des Pointers »<adr2>« mit der Konstanten »<adr1>«.
.jeq <adr> – Bedingter Sprung als Kombination aus Branch und »jmp«; Springe, wenn Z-Flag.
.jne <adr> – dto. als Ersatz für »bne«.
.jcc <adr> – dto. als Ersatz für »bcc«.
.jcs <adr> – dto. als Ersatz für »bcs«.
.jmi <adr> – dto. als Ersatz für »bmi«.
.jpl <adr> – dto. als Ersatz für »bpl«.
.jvc <adr> – dto. als Ersatz für »bvc«.
.jvs <adr> – dto. als Ersatz für »bvs«.

Strukturen:

.case of <bytexp> – Einleitung der mehrfachen Fallunterscheidung.
.caseend – Ende dieser Fallunterscheidung.
.do – Einleitung der Do-Schleife.
.during #<b> – Ausgang aus der Repeatschleife.
.else – Alternative in der ».if«-Struktur.
.enif – Ende der ».if«-Struktur.
.excase #<b> – Ausgang aus der Case-Struktur.
.exloop #<b> – Ausgang aus der Do-Schleife.
.if #<b> – Einleitung der einfachen Fallunterscheidung.

.loop	– Ende der Do-Schleife.
.of <bytexp>	– Alternative in der Case-Struktur.
.otherwise	– Default-Alternative in der Case-Struktur.
.repeat	– Beginn der Repeatschleife.
.until #<b>	– Ende der Repeatschleife.
.unless #<b>	– Ausgang aus der Repeatschleife.
.while #<b>	– Ausgang aus der Do-Schleife.

Abkürzungen der Pseudobefehle

.alter	.aL
.append	.A
.base	.bA
.begin	.B
.break	.bR
.byte	.bY
.case of	.C
.caseend	.caseE
.cdlins	.cD
.chainend	.cH
.clist	.cL
.code	.coD
.common	.cO
.cond	.coN
.continued	.contI
.control	.conT
.define	.dE
.do	.D
.during	.dU
.econd	.eC
.else	.eL
.end	.eN
.enif	.enI
.excase	.exC
.exloop	.E
.extern	.exT
.link	.liN
.list	.lI
.lprint	.lP
.macend	.macE
.maclib	.macL
.macro	.M
.malins	.maL
.modul	.moD
.names	.N
.nonum	.noN
.nops	.nO
.object	.oB
.objend	.objeN
.of	.O
.otherwise	.oT
.page	.pA
.print	.P
.public	.pU
.repeat	.R
.request	.reQ
.revers	.reV
.screen	.sC
.set	.sE
.skip	.sK
.source	.S
.space of	.SP
.symbols	.sY
.syntax	.syN
.unless	.unL
.until	.U
.wait	.wA
.while	.W
.word	.wO

ASE-Token

Hex.	Mnemonic	Pseudoop	Hex	Mnemonic	Pseudoop
81	lda	. repeat	a6	sec	. revers
82	sta	. until	a7	cli	. space of
83	ldx	. unless	a8	sei	. request
84	ldy	. do	a9	clv	. code
85	cmp	. loop	aa	cld	. syntax
86	adc	. while	ab	sed	. control
87	and	. case of	ac	rti	. =
88	stx	. of	ad	plp	. –
89	sty	. otherw.	ae	pha	. excase
8a	inc	. caseend	af	pla	. lax
8b	cpx	. during	b0	inx	. lay
8c	asl	. exloop	b1	dey	. lxy
8d	bit	. begin	b2	rts	. sax
8e	lsr	. end	b3	nop	. say
8f	ora	. define	b4	clc	. sxy
90	rol	. base	b5	sec	. shl
91	ror	. byte	b6	cli	. shr
92	sbc	. word	b7	sei	. slc
93	cpy	. object	b8	clv	. src
94	dec	. objend	b9	cld	. icr
95	eor	. macro	ba	sed	. dcr
96	jmp	. ..	bb	rti	. liax
97	jsr	. .	bc	php	. liay
98	txa	. macend	bd	dex	. lixy
99	tya	. common	be	brk	. mov
9a	tay	. append	bf	bpl	. mvi
9b	tsx	. adl	c0	bmi	. add
9c	txs	. clist	c1	bvc	. adi
9d	tay	. econd	c2	bvs	. sub
9e	plp	. alter	c3	bcc	. sbi
9f	pha	. cond	c4	bcs	. cpr
a0	pla	. maclib	c5	bne	. cpi
a1	inx	. names	c6	beq	. lprint
a2	dey	. source	c7		. jeq
a3	rts	. list	c8		. jne
a4	nop	. symbols	c9		. jcc
a5	clc	. screen	ca		. jcs

Hex.	Mnemonic	Pseudoop	Hex	Mnemonic	Pseudoop
cb		. jvc			
cc		. jvs			
cd		. jmi			
ce		. jpl			
cf		. skip			
d0		. modul			
d1		. continued			
d2		. chainend			
d3		. chain			
d4		. if			
d5		. else			
d6		. enif			
d7		. print			
d8		. wait			
d9		. break			
da		. set			
db		. extern			
dc		. public			
dd		. link			
de		. malins			
df		. cdlins			
e0		. nops			
e1		. nonum			
e2		. page			

Der ASE verwendet Token des jeweils angegebenen Wertes für die Darstellung der Mnemonics und Pseudoops innerhalb des Quelltextes. Den Token der Pseudoops geht jeweils ein Punkt voraus, dadurch können sie von den Token der Mnemonics unterschieden werden. Zwischen vorangehendem Punkt und dem Token sind Leerzeichen erlaubt.

Alphabetische Übersicht über die DBM-Befehle

<kadr>	= Adresse mit optionaler Eingabe eines Konfigurationsindex.
<adr>	= Adresse ohne Konfigurationsindex.

a <kadr>	– ASCII-Dumpzeile beginnend bei Adresse »<kadr>« ausgeben.
bcp <nr> <kadr>	– Workspace »<nr>« mit Bereich ab Adresse »<kadr>« vergleichen.
br <n> <kadr> [<m>]	– Breakpoint Nummer »<n>« auf Adresse »<kadr>« setzen; optionale Eingabe einer Bedingung »<m>«.
b-r <nr> <t> <s>	– Track »<t>«, Sektor »<s>« in Workspace »<nr>« lesen.
b-w <nr> <t> <s>	– Workspace »<nr>« als Track »<t>«, Sektor »<s>« auf Diskette schreiben.
cbr	– Alle Breakpoints löschen.
csp	– Alle Hot Spots löschen.
cv <nr> <adr1> <adr2> <adr3>	– Adreßangleichung nach Blockverschiebung in Workspace »<nr>«; betroffen sind absolute Adressen im Bereich »<adr1>« bis »<adr2>«; »<adr1>« wird in »<adr3>« umgerechnet, die folgenden Adressen »<adr1 + n>« in »<adr3 +n>«.
»CBM + F1«	– Druckerprotokoll ein/aus.
»CBM + F3«	– Trennzeile aus Sternen ausgeben.
»CBM + F5«	– Übernahme der Werte aus der Registeranzeige in die Speicher des Monitors.
»Cursor down«	– (Einzeltaste) Dump nach unten scrollen.
»Cursor left«	– (Einzeltaste) Linkes Teilfenster einschalten.
»Cursor right«	– (Einzeltaste) Rechtes Teilfenster einschalten.
»Cursor up«	– (Einzeltaste) Dump nach oben scrollen.
d <kadr>	– Ausgabe einer disassemblierten Zeile.
dv <n>	– Floppy mit Gerätenummer »<n>« als Default-Gerät wählen.
f <nr> <byte>	– Füllen des Workspace »<nr>« mit dem Bytewert »<byte>«.
F1	– Fehlerkanal der Floppy anzeigen.
F3	– Directory anzeigen.
F5	– Registeranzeige zeigen.

g [<kadr>]	– Starten eines Testprogrammes mit fakultativer Eingabe einer Startadresse.
h <nr> <disline>	– Suchen der disassemblierten Zeile »<disline>« in Workspace »<nr>« (mit Jokern).
ha <nr> 'string'	– ASCII-String »string« in Workspace »<nr>« suchen.
ha <nr> <b> {<b>}	– Kette von Bytewerten »<b>« in Workspace »<nr>« suchen.
»Home Home«	– Aufheben eines Fensters.
i	– Directory anzeigen.
j <kadr>	– Testprogramm starten.
l "file" [,<kadr>]	– Laden eines Files von der Floppy mit optionaler Eingabe einer Ladeadresse.
m <kadr>	– Ausgabe einer Hexdumpzeile.
pn	– Alle Breakpoints in ein Testprogramm einsetzen.
pr <u> <s>	– Druckkanal mit Gerätenummer »<u>« und Sekundäradresse »<s>« wählen.
r	– Registeranzeige.
s <nr> "file"	– Workspace »<nr>« abspeichern.
sb	– Breakpoints anzeigen.
so	– Alle Breakpoints durch Originalbefehle ersetzen.
sp <nr> <kadr>	– Hot Spot Nummer »<nr>« auf Adresse »<kadr>« richten.
t <nr> <kadr>	– Blockverschiebung des Workspace »<nr>« nach Adresse »<kadr>«.
us <n> <kadr1> <kadr2>	– Userbreakpoint Nummer »<n>« auf Adresse »<kadr1>« setzen; das Userprogramm soll bei Adresse »<kadr2>« liegen.
v "file" [,<kadr>]	– Verifizieren eines geladenen Files mit optionaler Eingabe einer Ladeadresse.
w <n> <kadr1> <kadr2>	– Eingabe des Workspace »<nr>« mit Startadresse <kadr1>« und Endadresse »<kadr2>«.
wa <nr>	– Bildschirm mit ASCII-Dumpzeilen des Workspace »<nr>« füllen.
wd <nr>	– Dto. mit disassemblierten Zeilen.
wm <nr>	– Dto. mit Hexdumpzeilen.
§ <dos-string>	– Befehl zur Floppy senden.
_ <adr>	– Zahlenkonvertierungen

Der Inhalt der beiliegenden Diskette

»ase v2.0«	– Makroassembler mit integriertem Editor und Linker.
»dbm2000«	– Monitor-/Debugger mit Startadresse $2000 (Version 2.0).
»dbm.rel«	– Relokatibles Modul des Monitors.
»vlo2000«	– Loader für relokatible Module mit Startadresse $2000 (Version 2.1).
»vlo.rel«	– Relokatibles Modul des Loaders.
»vlodbm.mke«	– Link-Quelltext zur Herstellung von DBM- und VLO-Objektprogrammen aus den Modulen.
»reass«	– Symbolischer Reassembler (Autor: Michael Bauer).
»info!«	– Informationsprogramm. Bitte laden und starten Sie dieses Programm unbedingt; es enthält eine Beschreibung des nächsten Programms.
»copy.protext«	– Kopierprogramm zum Duplizieren der gesamten Diskette oder einzelner Programme. Die Beschreibung entnehmen Sie bitte dem File »info!«.

Stichwortverzeichnis

Spitzen-Software für Commodore 128/128 D

WordStar 3.0 mit MailMerge

Der Bestseller unter den Textverarbeitungsprogrammen für PCs bietet Ihnen bildschirmorientierte Formatierung, deutschen Zeichensatz und DIN-Tastatur sowie integrierte Hilfstexte. Mit MailMerge können Sie Serienbriefe mit persönlicher Anrede an eine beliebige Anzahl von Adressen schreiben und auch die Adreßaufkleber drucken.

WordStar/MailMerge für den Commodore 128 PC
Bestell-Nr. MS 103 (5¼"-Diskette)

Hardware-Anforderungen: Commodore 128 PC, Diskettenlaufwerk, 80-Zeichen-Monitor, beliebiger Commodore-Drucker oder ein Drucker mit Centronics-Schnittstelle

Für nur DM 199,–* (sFr. 178,–/öS 1890,–)

***inkl. MwSt. Unverbindliche Preisempfehlung**

Markt&Technik
128er-Software

MicroPro
WordStar 3.0
mit MailMerge für den
Commodore 128 PC

5¼"-Diskette
im Floppy 1541-Format

Und dazu die weiterführende Literatur:

Mit diesem Buch haben Sie eine wertvolle Ergänzung zum WordStar-Handbuch: Anhand vieler Beispiele steigen Sie mühelos in die Praxis der Textverarbeitung mit **WordStar** ein. Angefangen beim einfachen Brief bis hin zur umfangreichen Manuskripterstellung zeigt Ihnen dieses Buch auch, wie Sie mit Hilfe von MailMerge Serienbriefe an eine beliebige Anzahl von Adressen mit persönlicher Anrede senden können.

Best.-Nr. 90181
ISBN 3-89090-181-6
DM 49,– (sFr. 45,10/öS 382,20)

Erhältlich bei Ihrem Buchhändler.

Markt&Technik-Produkte erhalten Sie in den Fachabteilungen der Warenhäuser, im Versandhandel, in Computerfachgeschäften oder bei Ihrem Buchhändler.

701321

Markt&Technik
Zeitschriften · Bücher
Software · Schulung

Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 4613-0

Spitzen-Software für Commodore 128/128 D

dBASE II, Version 2.41

dBASE II, das meistverkaufte Programm unter den Datenbanksystemen, eröffnet Ihnen optimale Möglichkeiten der Daten- u. Dateihandhabung. Einfach u. schnell können Datenstrukturen definiert, benutzt und geändert werden. Der Datenzugriff erfolgt sequentiell oder nach frei wählbaren Kriterien, die integrierte Kommandosprache ermöglicht den Aufbau kompletter Anwendungen wie Finanzbuchhaltung, Lagerverwaltung, Betriebsabrechnung usw.

dBASE II für den Commodore 128 PC
Bestell-Nr. MS 303 (5¼"-Diskette)

Hardware-Anforderungen: Commodore 128 PC, Diskettenlaufwerk, 80-Zeichen-Monitor, beliebiger Commodore-Drucker oder ein Drucker mit Centronics-Schnittstelle

Für nur DM 199,–* (sFr. 178,–/öS 1890,–*)

***inkl. MwSt. Unverbindliche Preisempfehlung**

Markt&Technik
128er-Software

dBASE™ II
ASHTON-TATE
für den
Commodore 128 PC

5¼"-Diskette
im Floppy 1541-Format

Und dazu die weiterführende Literatur:

Zu einem Weltbestseller unter den Datenbanksystemen gehört auch ein klassisches Einführungs- und Nachschlagewerk! Dieses Buch von dem deutschen Erfolgsautor Dr. Peter Albrecht begleitet Sie mit nützlichen Hinweisen bei Ihrer täglichen Arbeit mit **dBASE II.** Schon nach Beherrschung weniger Befehle ist der Einsteiger in der Lage, Dateien zu erstellen, mit Informationen zu laden und auszuwerten.

Best.-Nr. 90189
ISBN 3-89090-189-1
DM 49,– (sFr. 45,10/öS 382,20)

Erhältlich bei Ihrem Buchhändler.

Markt&Technik-Produkte erhalten Sie in den Fachabteilungen der Warenhäuser, im Versandhandel, in Computerfachgeschäften oder bei Ihrem Buchhändler.

701322

Markt&Technik
Zeitschriften · Bücher
Software · Schulung

Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 4613-0