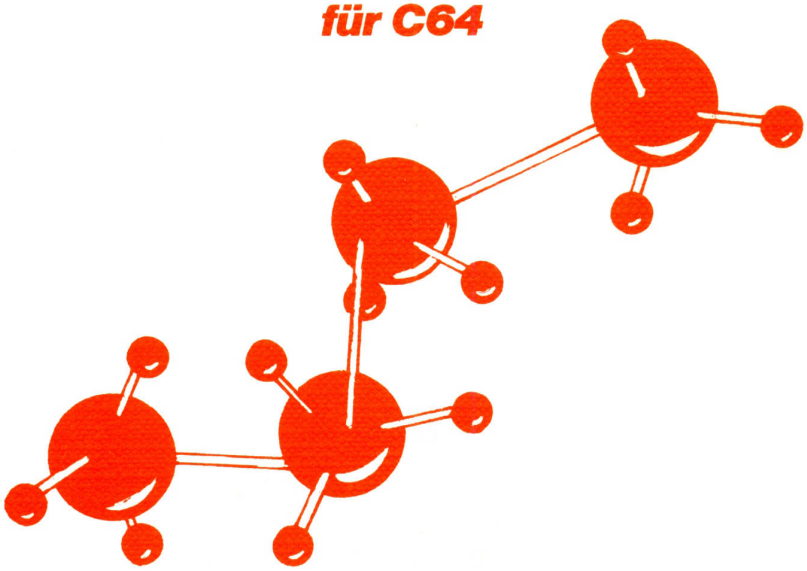
Schmidt

DAS ASSEMBLER TRAININGSBUCH

ZU
PROFI-ASS · SM MAE · T.EX.AS.
für C64

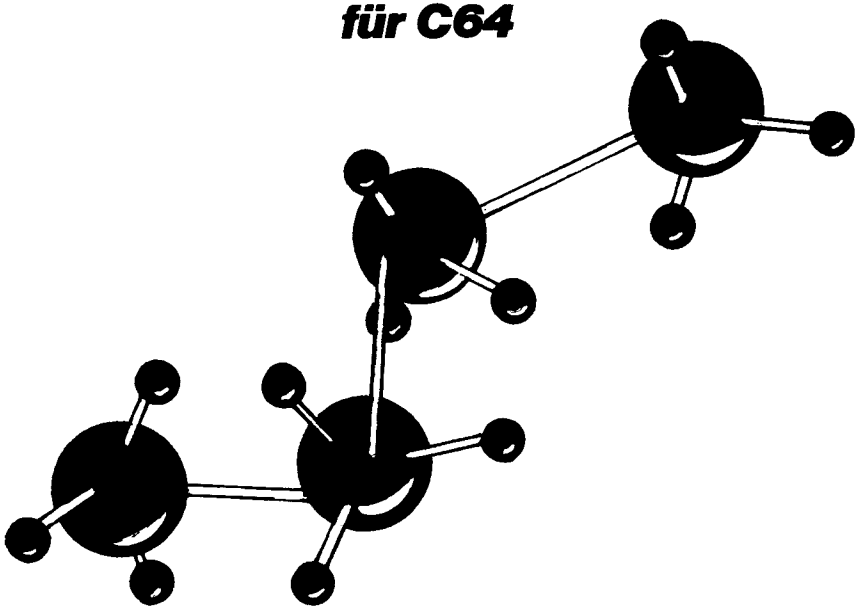


EIN DATA BECKER BUCH

Schmidt

DAS ASSEMBLER TRAININGSBUCH

ZU
PROFI-ASS · SM MAE · T.EX.AS.
für C64



EIN DATA BECKER BUCH

INHALTSVERZEICHNIS

ASSEMBLER

KAPITEL 1: ASSEMBLER - AUFGABEN UND EINSATZ

KAPITEL 2: PROGRAMMIERUNG IN ASSEMBLER

ANHANG

Das vorliegende Buch ist wie die meisten Programmpakete zur Maschinensprache in zwei Teile aufgegliedert. Um die Arbeit von dem verwendeten Programmpaket möglichst unabhängig zu machen, sind alle Beispiele für die Assembler: PROFI MON von DATA BECKER, MAE 64 von SM Software und T.EX.AS. von INTERFACE AGE ausgearbeitet. Auch wenn Sie einen anderen Assembler als einen der aufgeführten benutzen, können die in der Regel rein formalen Unterschiede zwischen den einzelnen Assemblern leicht mit Hilfe des Handbuches korrigiert werden. Das Buch vermittelt im Wesentlichen die Grundlagen und die praktische Anwendung der Assembler- Programmierung möglichst unabhängig von dem verwendeten Assembler oder Monitor.
Im ersten Teil wird der Maschinensprache Monitor erklärt und anhand von Beispielen der Umgang mit diesem eingeübt. Jeweils am Ende eines Absatzes werden Fragen zu dem behandelten Stoff gestellt. Sie sollten diese Fragen schriftlich beantworten, um Ihre Lösungen besser mit den abgedruckten vergleichen zu können. Wenn Sie eine Frage nicht beantworten können, sollten Sie im Text und nicht bei den Antworten nachschlagen. Auf diese Weise werden Sie Ihr Lernziel sicher erreichen.
Im zweiten Teil des Buches wird der Assembler in derselben Weise besprochen. Die letzten Kapitel trainieren den Umgang mit dem Monitor und dem Assembler anhand einiger allgemeingültiger Beispiele.
Im letzten Kapitel wird auch ein umfassend ausgearbeitetes System zur strukturierten Programmierung vorgestellt und der Umgang damit an einem Beispiel eingeübt.
Grundlegende Kenntnisse der Maschinensprache - Programmierung (Kenntnis der Befehle) sind zur Bearbeitung des Stoffes von Vorteil, aber keineswegs Voraussetzung. Alle 6502 (6510) Befehle und Adressierungsarten werden besprochen und mit Beispielen veranschaulicht.
Der Anhang macht das Buch zu einem Nachschlagewerk, daß Sie auch als Profi immer wieder zur Hand nehmen werden. In einem eigenen Kapitel werden dann Arbeitshinweise zur strukturierten Assembler - Programmierung gegeben und diese

anhand von Beispielen, im letzten Kapitel, in die Praxis umgesetzt.

Nachdem Sie das Buch durchgearbeitet haben, wird Ihnen Ihr Assembler ein willkommenes und unentbehrliches Werkzeug zur Maschinensprache - Programmierung sein. Dazu wünsche ich Ihnen viel Freude und Erfolg.

Programme in Maschinensprache zu programmieren bietet einige Vorteile. Sie können alle technischen Möglichkeiten Ihres Rechners nutzen, die Programme sind kürzer und laufen erheblich schneller.
Es gibt Anforderungen an die Laufzeit eines Programmes, die überhaupt nur in Maschinensprache zu erfüllen sind (Grafik, Sound, Echtzeitanwendungen usw.).
Maschinensprache ist allerdings ein Begriff welcher mehrere Spezifizierungen besitzt. Unter der unmittelbaren Maschinensprache ist das zu verstehen, was tatsächlich von der Maschine (dem Prozessor) aus dem Speicher gelesen wird. Wie Ihnen bekannt ist, ist das in unserem Fall ein aus acht Bit zusammengesetztes Byte. Jedes dieser acht Bit kann die Information 1 oder 0 beinhalten. Wir können dieses Byte also als eine Binäre Zahl betrachten. Damit haben wir aber schon die erste Vereinfachung, weg von der Maschine, hin zum Menschen, gemacht. Denn der Prozessor betrachtet bei der Befehlserkennung jedes Bit für sich.
Um den Prozessor also beispielsweise dazu zu veranlassen einer Variablen einen bestimmten Wert zuzuordnen, muß die Bitfolge: 10101001 in den Speicher gebracht werden. Diese Bitfolge ist für uns völlig nichtssagend und dementsprechend schwer zu merken.
Ein ähnlicher Befehl in einer höheren Programmiersprache ist wesentlich aussagekräftiger und somit gut einprägsam. Beispielsweise in BASIC: LET A = 5.
Eine weitere Vereinfachung von der Bitfolge zu einprägsamerem Code ist die Umwandlung der Binärzahlen in Hexadezimalzahlen (kurz Hexzahlen). Das Hexadezimalsystem wurde deshalb gewählt, weil sich das Binärsystem sehr leicht in das Hexadezimalsystem umrechnen läßt.

Für diejenigen unter Ihnen, die noch nicht mit den verschiedenen Zahlensystemen vertraut sind, sei hier kurz das Notwendige erklärt: Ein Zahlensystem kennen Sie bereits sehr genau. Das Zahlensystem mit der Basis 10 (Dezimalsystem). Die Ziffern einer Zahl geben uns nur einen Wert an, wenn zuvor vereinbart wurde, welche Stelle einer Zahl welchen Wert

repräsentiert. Im Dezimalsystem kennen wir in der Reihenfolge von rechts nach links die Stellen: Einer, Zehner, Hunderter, Tausender usw. Wir können die Zahl 2548 beispielsweise auch in einer Tabelle darstellen.

T	H	Z	E
2	5	4	8

Der Wert der Stellen ergibt sich aus der Basis des Zahlensystems potenziert mit der wievielten Stelle von rechts, wobei mit 0 für den Exponenten begonnen wird. Die obige Tabelle ließe sich also auch anders beschriften:

10^3	10^2	10^1	10^0
2	5	4	8

Anstelle der Basis 10 läßt sich jede andere Basis für andere Zahlensysteme festlegen. Wir haben bereits festgestellt, daß wir die Bytes mit denen der Prozessor arbeitet als Binärzahlen betrachten können. Binärzahlen sind nichts anderes als Zahlen zur Basis 2. Es gibt daher im Binär- oder Dual- System nur zwei Ziffern: 1 und 0. Wir wollen einmal die Darstellung einer 8- stelligen Binärzahl in einer Tabelle wie oben betrachten. Die Wertigkeit der Stellen schreiben wir als Potenzen und als ausgerechnete Dezimalzahlen.

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1
1	0	1	0	1	0	0	1

Um diese Binärzahl in das Dezimalsystem umzurechnen, müssen wir für jede Stelle die Wertigkeit mit der Ziffer dieser Stelle multiplizieren. Also wie folgt:

```
128 * 1 = 128
 64 * 0 =   0
 32 * 1 =  32
 16 * 0 =   0
  8 * 1 =   8
  4 * 0 =   0
  2 * 0 =   0
  1 * 1 =   1
```

Die Ergebnisse sind dann zu addieren. Das ergibt:

```
  128
   32
    8
+   1
  ---
  169
```

Sie sehen, das ist sehr mühsam und langwierig.
Der umgekehrte Weg ist noch umständlicher. Wir wollen einmal die Dezimalzahl 87 in das Binärsystem umwandeln. Zu diesem Zweck müssen wir zunächst die höchste Binärstelle finden, die noch einmal in die 87 paßt. Das ist die 64 er Stelle. Für diese Stelle ist also eine 1 einzutragen. Die Differenz zwischen 64 und 87 ist 23. Die Binärstelle 32 ist demnach 0. Die Binärstelle 16 ist mit 1 einzutragen, da die 16 einmal in die 23 paßt. Es verbleiben 7. Die folgende Binärstelle 8 ist somit 0. Auf diese Weise sind dann auch die letzten drei Stellen auf 1 zu setzen. Dezimal 87 ergibt also Binär 1010111. Damit dürfte die Unhandlichkeit des Binärsystem für uns, die wir im Dezimalsystem zu rechnen gewohnt sind, hinlänglich bewiesen sein. Der Prozessor arbeitet aber nur mit Bits.
Eine Lösung bietet hier das Hexadezimalsystem (auch sedezimal). Dieses Zahlensystem bezieht sich auf die Basis 16. In diesem Zahlensystem sind somit 16 Ziffern erforderlich. Diese sind: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. Wir wollen wieder eine Hexadezimalzahl in einer Tabelle darstellen:

16^3	16^2	16^1	16^0
4069	256	16	1
5	0	A	F

In das Dezimalsystem rechnen wir wieder wie oben um:

```
4096 *  5 = 20480
 256 *  0 =     0
  16 * 10 =   160
   1 * 15 =    15
```

Diese Werte sind noch zu addieren. Das ergibt:

```
   20480
     160
+     15
   -----
   20655
```

Die Umrechnungen zwischen Binär- und Hexadezimalsystem sind wesentlich einfacher: Es lassen sich immer vier Binärstellen zu einer Hexadezimalstelle zusammenfassen. Für die obige Bitfolge sähe das so aus:

```
Binär:          1 0 1 0  1 0 0 1

Hexadezimal:      A        9
```

Zur Darstellung der Speicherinhalte verwenden wir also in der Regel das Hexadezimalsystem. Es sei denn, der Wert einzelner Bits ist von Bedeutung. In diesem Fall wird das Byte im Binärsystem dargestellt. Zur Unterscheidung steht vor einer Hexzahl das $ Zeichen und vor einer Binärzahl das % Zeichen.

Zurück zu unserem erstem Befehl. Der Ausdruck: "A9" läßt sich schon erheblich besser einprägen als die Bitfolge: 10101001. Wenn wir also dem Prozessor die Befehle in Hexformat geben können und die Befehle, also das Programm, in Hexformat ausgegeben werden, so sind wir schon ein gutes Stück

weitergekommen.
Der Prozessor benötigt die Befehle aber nach wie vor in den entsprechenden Bitmustern. Es muß demnach eine Umwandlung stattfinden. Das erledigt natürlich ein Programm für uns - der Monitor.
Das Programm: "Monitor" erlaubt es uns Einträge in den Rechnerspeicher vorzunehmen, ohne daß wir uns um die Umrechnung: Binär - Hex zu kümmern brauchen. Auch der umgekehrte Weg, also das Auslesen der Binärwerte und deren Darstellung in Hexform auf dem Bildschirm, erledigt der Monitor für uns.
Außer diesen beschriebenen Aufgaben bietet der Monitor noch weitergehende Leistungen zur Maschinensprache- Programmierung an, z.B. Speicherbereiche vergleichen oder übertragen, ASCII Darstellung, Einzelschritt Ablauf usw. Wir werden diese Punkte im Einzelnen noch genauer besprechen.
Auf eine Funktion wollen wir in diesem Zusammenhang aber noch eingehen: das Disassemblieren.
Den bisher als Beispiel besprochenen Befehl mit der Bitfolge: 10101001 haben wir durch Betrachtung als Binärzahl und weitere Umwandlung in die Hexzahl "A9" bereits wesentlich "handlicher" gemacht. Diese Hexzahl bietet allerdings immer noch keinerlei Bezug zu der Bedeutung des Befehls. Als Lösung bietet sich hier eine weitere Umcodierung der Befehls-Bitfolge an. Anstelle der Bitfolgen treten symbolische Buchstabenfolgen die einen Bezug zur Bedeutung der Befehle haben.
Diese Darstellung ist die mnemotechnische Darstellung und die Buchstabenfolgen sind Mnemonic´s. Für unsere Bitfolge: 10101001 steht das Mnemonic "LDA" als Abkürzung für "load Accumulator" (lade den Accumulator). Eine solche Schreibweise ist wesentlich leichter zu merken.
Im Anhang finden Sie alle 65xx Befehle mit Mnemonic, Adressierung und Hexwert erklärt. Auf die verschiedenen Adressierungen kommen wir noch zu sprechen.
Die Darstellung von Maschinenbefehlen in Mnemonic´s bildet die Assemblersprache. Assemblieren heißt das Mnemonic in die Bitfolge übersetzen, disassemblieren bezeichnet den umgekehrten Vorgang. Der Monitor bietet die Möglichkeit zum Disassemblieren. Sie können sich also ein im Speicher befindliches Maschinenprogramm in der Assemblersprache auf

den Bildschirm ausgeben lassen.

Bei diesen Erleichterungen durch die symbolische Darstellung darf man nicht vergessen, daß die Befehle dadurch in keiner Weise verändert wurden. Letztlich bewegen wir uns immer noch ausschließlich auf der Maschinenebene. Das heißt wir haben es nur mit der "Intelligenz" des Prozessors zu tun und diese ist bekanntlich nicht mit der höherer Programmiersprachen, wie etwa BASIC, zu vergleichen.
Das bedeutet unter anderem, daß wir uns um jede Kleinigkeit selbst kümmern müssen. Wir müssen genau festlegen wo das Programm im Speicher stehen soll, wo genau Variablen stehen, Zahlensystemumrechnungen vornehmen, sogar die Ein- und Ausgabe ist erheblich komplizierter als in BASIC beispielsweise. Der Prozessor arbeitet eben nur mit Bits.
Eine unverzichtbares Werkzeug stellt hier der Assembler dar. Mit dem Assembler ist ein weiterer Schritt zur Erleichterung der Maschinensprache- Programmierung getan, ohne einen Vorteil damit aufzugeben.
Welche Funktionen ein Assembler im Einzelnen erfüllt, werden wir im zweiten Teil dieses Buches besprechen. Hier sei nur kurz umrissen, daß der Assembler einiges der fehlenden "Intelligenz" der reinen Maschinensprache ersetzt, wenn man es versteht ihn richtig einzusetzen und dazu werden Sie nach Studium des zweiten Teiles sicher in der Lage sein.
Um hier allerdings Mißverständnissen vorzubeugen, sei noch vermerkt, daß der Assembler keineswegs den Monitor ersetzt. Der Assembler dient zur Erstellung von Maschinensprache-Programmen und der Monitor zum Starten und Austesten der Programme.
Im folgenden Teil werden wir alle Funktionen des Monitors im Einzelnen besprechen.

Fragen zu Monitor / Assembler

1.) **Was verstehen Sie unter unmittelbarer Maschinensprache ?**

2.) **Beschreiben Sie die erste Vereinfachung der unmittelbaren Maschinensprache ?**

3.) **Was ist Assemblersprache ?**

4.) **Warum werden die Speicherinhalte am einfachsten im Hexformat dargestellt ?**

5.) **Wird die "Intelligenz" der Maschinensprache durch die Assemblersprache erweitert ?**

6.) **Kann der Assembler den Monitor ersetzen ?**

Antworten zu Monitor / Assembler

zu 1.) Die unmittelbare Maschinensprache ist ein Maschinenprogramm bestehend aus den Befehlen als Bitfolgen, so wie diese tatsächlich im Speicher stehen.

zu 2.) Genau genommen stellt bereits die Betrachtung der Bitfolgen als Binärzahlen eine erste Vereinfachung der Maschinensprache dar.

zu 3.) In der Assemblersprache werden die Befehle der Maschinensprache nicht in Bitfolgen sondern in Mnemonic´s (Symbole - meist Abkürzungen der Funktion) ein- und ausgegeben.

zu 4.) Weil sich Zahlen im Binärsystem sehr leicht in das Hexadezimalsystem umrechnen lassen und so übersichtlicher und einprägsamer werden.

zu 5.) Keineswegs! Die Maschinensprache bleibt durch die Verwendung eines Assemblers unverändert. Die Handhabung derselben wird allerdings wesentlich erleichtert.

zu 6.) Nein. Der Monitor erfüllt andere Aufgaben als der Assembler. Zur Erstellung von Maschinensprache-Programmen werden Monitor und Assembler benötigt.

KAPITEL 1: DIE FUNKTIONEN IM EINZELNEN

1.1. Register Display (Register anzeigen)

Im Anhang finden Sie ein sogenanntes Programmiermodell zu 65xx Prozessoren. Dieses Modell beinhaltet alle für den Programmierer wichtigen "Innereien" eines Mikroprozessors der Serie 65xx.
Wir wollen dieses Modell hier besprechen, da wir uns in den folgenden Kapiteln gelegentlich darauf beziehen werden.

Adress- und Datenbus
Zunächst sind dort zwei Elemente als Bus bezeichnet. Der Datenbus und der Adressbus. Ein Bus ist ein Informationsträger, der Informationen von einem Teilnehmer zu einem anderen oder zwischen Teilnehmern hin und her transportiert. Ein Bus welcher Informationen von einem Teilnehmer zu weiteren Teilnehmern transportiert, aber nicht in umgekehrter Richtung, wird unidirektionaler Bus genannt. Das Gegenstück hierzu, der bidirektionale Bus, kann Informationen von und zu jedem Teilnehmer transportieren.
Beim unidirektionalen Bus ist also nur ein Teilnehmer in der Lage eine Information auf den Bus zu legen. Der bidirektionale Bus erlaubt jedem Teilnehmer eine Information auf den Bus zu legen. Gleichzeitig kann jedoch nur eine Information transportiert werden. Die momentan auf dem Bus liegende Information kann jeder Teilnehmer lesen.
Ob es sich bei einem Bus um einen uni- oder bidirektionalen Bus handelt, wird in einer Zeichnung durch die Pfeile an den Anschlußstellen kenntlich gemacht. Sind an beiden Anschlußstellen Pfeile, so handelt es sich um einen bidirektionalen Bus. Beim unidirektionalen Bus ist die Seite des Informations- Gebers glatt ohne Pfeil gezeichnet, während die empfangenden Teilnehmer durch einen Pfeil kenntlich gemacht werden.
In unserem Programmiermodell finden wir je einen Bus von jeder Art. Der Datenbus ist ein bidirektionaler Bus. Jeder

Teilnehmer auf diesem Bus kann Daten an die anderen Teilnehmer senden oder Daten von diesen empfangen. Der Adressbus ist ein unidirektionaler Bus. Das Programmiermodell zeigt zwei zum Senden berechtigte Teilnehmer. Das ist eine Ausnahme die hier rein zeichnerisch zum besseren Verständniss so dargestellt wurde. Tatsächlich darf ein unidirektionaler Bus wie der Adressbus nur einen zum Senden berechtigten Teilnehmer haben. Durch eine Steuerung kann in bestimmten Fällen aber diese Berechtigung umgeschaltet werden. Das ist hier der Fall.
Was wir bisher Daten genannt haben, sind einfach Binärzahlen. Diese Binärzahlen setzen sich aus Bits zusammen. Beim Datenbus aus acht Bit, also einem Byte. Beim Adressbus aus 16 Bit, also zwei Bytes.

Der ACCU

An den Datenbus sind 6 Register als Teilnehmer angeschlossen. Wir beginnen mit dem ersten Register ganz rechts, dem Accumulator (ACCU). Zunächst ist der ACCU wie alle Register in der Lage ein Byte vom Datenbus zu holen und dieses zu speichern. Auf dem Datenbus können dann andere Bytes transportiert werden, das gespeicherte Byte bleibt im ACCU erhalten.

Die ALU

Unter dem ACCU finden Sie die ALU (Arithmetisch logische Einheit) gezeichnet. Wie der Name schon sagt, ist dieses Element des Prozessors in der Lage zu rechnen. Die ALU kann addieren, subtrahieren und die logischen Funktionen AND, OR, EOR ausführen. Division und Multiplikation habe ich hier nicht vergessen. Diese Rechenarten kennt die ALU leider nicht. Die V-förmige Zeichnung resultiert aus den zwei Ein- und dem einen Ausgang der ALU.
Wir wollen zum besseren Verständniss eine Addition in dem Programmiermodell durchspielen. Es soll 2+3 gerechnet werden. Zunächst wird die 2 (binär 00000010) auf dem Datenbus erwartet. Diese 2 wird dann in den ACCU geladen. Dann wird die 3 (binär 00000011) auf den Datenbus gelegt und der ALU der Befehl zum addieren gegeben. Die addiert daraufhin die beiden Bytes an ihren Eingängen, vom Datenbus und vom ACCU, und schickt das Ergebniss der Addition in den ACCU. Von dort

kann das Ergebnis dann auf den Datenbus gelegt und weiterverarbeitet werden. Die richtige Reihenfolge dieser Abläufe steuert die Befehlserkennung.
Der ACCU kann also Bytes vom Datenbus speichern und enthält das Ergebnis arithmetischer oder logischer Operationen.

Das Statusregister
Im nach links folgenden Statusregister besitzt jedes der 8 Bits eine eigene Bedeutung. Die Bedeutung der einzelnen Bits, in diesem Fall Flags genannt, ist im Anhang aufgeführt. Wir kommen später hierauf zurück.

Das X- und Y- Register
Danach folgen das X und Y Register. Die Funktionen dieser Register sind gleich. Hier können Bytes vom Datenbus gespeichert und wieder ausgegeben werden. Außerdem dienen diese Register zur Unterstützung bestimmter Adressierungsarten. Alle Adressierungsarten sind im Anhang aufgeführt. Was Adressierung bedeutet, wollen wir im folgenden besprechen.

Der Adresszähler
Um arbeiten zu können benötigt der Prozessor ein Programm und mehr oder weniger Arbeitsspeicher. Das Programm kann im ROM (nur lese Speicher) oder im RAM (schreib / lese Speicher) stehen. Den Arbeitsspeicher stellt das RAM dar. Den Speicher bildet im Ganzen, also RAM und ROM zusammen, eine Folge von 65536 Bytes. Jedes dieser Bytes kann einzeln durch den Adressbus angesprochen werden. Die kleinste Adresse ist 0 - die größte Adresse ist 65535. In Binärzahlen ausgedrückt: 0000000000000000 - kleinste Adresse / 1111111111111111 - größte Adresse. Wenn Sie nachzählen werden Sie feststellen, daß für die größte Adresse genau 16 Stellen erforderlich sind. Das sind genau zwei Byte. Das niederwertige dieser zwei Byte nennen wir Low- und das höherwertige Highbyte.
Wie der Inhalt eines Bytes aus dem Speicher ausgelesen wird, machen wir uns wieder anhand des Programmiermodells deutlich: Die 16 Bit Adresse des Bytes im Speicher, wird in die beiden Register - Adresszähler low und high - eingelesen. Damit wird diese Adresse auf den Adressbus gelegt. Der externe Speicher legt daraufhin den Inhalt des adressierten Bytes auf den

Datenbus. Von dort können wir das Byte in eines der Register übernehmen und weiterverarbeiten.
Auf dem gleichen Weg können wir auch ein Byte aus einem der Register in das RAM schreiben. In diesem Fall wird wieder der Speicherplatz über den Adressbus adressiert und dann das Byte aus dem Register auf den Datenbus gelegt. Zu erwähnen ist in diesem Zusammenhang noch, daß die Inhalte der Speicherstelle oder des Registers durch die Ausgabe auf den Datenbus nicht verändert werden. Wenn also das X- Register im RAM abgespeichert wird, so wird dadurch der Inhalt des X-Registers nicht verändert.
Den richtigen Ablauf der Vorgänge steuert hier wieder die Befehls- Erkennung. Beim Ablauf eines Programmes wird ein Befehl nach dem anderen abgearbeitet. Das Programm, also die Folge von Befehlen, steht im Speicher. Im normalen Programmablauf muß der Adresszähler somit immer erhöht werden, um einen Befehl nach dem anderen zu adressieren. Ein Sprung im Programm wird durch entsprechendes setzen des Adresszählers erreicht. Auch hierfür ist die Befehlserkennung zuständig.

Der Stapelzeiger

Damit kommen wir auch schon zum letzten Register, dem Stapelzeiger. Der Stapelzeiger hat dieselbe Funktion wie der Adresszähler, nur daß der Stapelzeiger nicht 16 sondern 8 Bit umfaßt. Das bedeutet, daß durch den Stapelzeiger nur 256 verschiedene Bytes im Speicher adressiert werden können (binär 11111111 = dezimal 255). Ist der Stapelzeiger für die augegebene Adresse zuständig, so ist das High- Byte der ausgegebenen Adresse immer 1. Dadurch können die Bytes 256 (binär 0000000100000000) bis 511 (binär 0000000111111111) adressiert werden.
Der Stapelzeiger wird allerdings nicht zur Adressierung während des Programmablaufes benutzt, sondern dient dem Aufbau eines Stapelspeichers. In einem Stapelspeicher wird der Speicherinhalt nicht durch Angabe einer bestimmten Adresse eingeschrieben oder ausgelesen. In einem solchen Speicher werden die Inhalte in einem Stapel abgelegt und zum Auslesen wieder von dem Stapel abgehoben. Das bedeutet beispielsweise, daß wir drei Bytes mit dem Wert 1, 2 und 3 ohne Adressangabe auf den Stapel ablegen. Zuerst 1 dann 2

dann 3. Lesen wir den Stapelspeicher wieder aus, so werden wir zuerst 3 dann 2 dann 1 erhalten. Was wir zuerst abgespeichert haben, lesen wir zuletzt aus.
Wir wollen uns dieses Beispiel einmal genauer am Programmiermodell ansehen: Ist der Stapelspeicher leer, so enthält der Stapelzeiger den Wert 255. Durch den entsprechenden Befehl veranlassen wir die Befehlserkennung den Wert 1 aus dem ACCU auf den Stapelspeicher abzulegen. Daraufhin wird der Stapelzeiger auf den Adressbus gelegt. Die sich aus High- Byte = 00000001 und Low- Byte = Stapelzeiger = 11111111 ergebende Adresse ist 511 (binär 0000000111111111). Die auf dem Datenbus aus dem ACCU liegende 1 wird also an der Adresse 511 im Speicher abgelegt. Die Befehls- Erkennung vermindert dann den Wert des Stapelzeigers um 1 auf 254. Speichern wir nun die 2 im Stapelspeicher ab. Der Vorgang ist selbstverständlich derselbe wie bei der 1. Nur durch den um eins verminderten Stapelzeiger wird die 2 an der Adresse 510 abgelegt. Nachdem die 2 abgelegt wurde, enthält der Stapelzeiger den Wert 253. Ist dann noch die 3 im Stapelspeicher an Adresse 509 abgelegt, so zeigt der Stapelzeiger den Wert 252.
Sehen wir jetzt einmal den umgekehrten Vorgang, das Auslesen aus dem Stapelspeicher an: Die Befehls- Erkennung wird wieder durch den entsprechenden Befehl dazu veranlasst ein Byte vom Stapelspeicher in den ACCU zu holen. Dazu wird zunächst der Stapelzeiger um 1 erhöht und enthält damit in unserem Beispiel den Wert 253. Daraus resultiert die Adresse aus Low- und High- Byte von 509. Diese Adresse wird auf den Adressbus, und damit der Inhalt der Speicherzelle 509 auf den Datenbus gelegt. Dieser Inhalt war 3. Auf dieselbe Weise werden wir als nächstes Byte vom Stapelspeicher die 2 aus Adresse 510 und dann die 1 aus Adresse 511 holen.
So lassen sich sehr schnell und einfach bis zu 256 Bytes ohne Adressangabe abspeichern. Wir werden mit dem Stapelspeicher noch öfter zu tun haben.

Wie Sie sehen, besteht die ganze "Intelligenz" des Mikroprozessors aus der richtigen Verwaltung dieser wenigen Register. Was aus den wenigen Elementen stundenlangen Spielspaß oder Stunden gesparter Arbeitszeit macht, ist Ihre Intelligenz, die des Progammierers.

Die Register des Prozessors besitzen also elementare Bedeutung für die Programmierung in Maschinensprache. Es ist deshalb wichtig, daß der Monitor die momentanen Inhalte aller Prozessorregister anzeigen kann. Besonders im Einzelschrittmodus oder Unterbrechungsbetrieb sollten unbedingt die Register angezeigt werden. Wir werden auf diese Betriebsarten noch zu sprechen kommen.

PROFI MON

Mit dem Befehl - R - können die Register jederzeit angezeigt werden. Die Register werden sowohl im Einzelschritt- wie im Unterbrechungs- Betrieb bei jedem Ablaufstop angezeigt. Außerdem ist es möglich die Register vor einem Programmstart, durch einfaches Überschreiben der Anzeigezeile, auf bestimmte Werte zu setzen. Das kann ein sogenanntes Startprogramm zum Testen ersparen. Da vom Monitor gestartete Programme mit einem Unterbrechungsbefehl (BRK) abgeschlossen sein müssen, werden auch nach Beendigung jedes Programmes alle Register angezeigt.

T.EX.ASS.

Dieser Monitor zeigt die Register leider nicht an. Sie müssen sich mit Hilfsprogrammen diese Möglichkeit selbst schaffen. Wir werden auf solche Programme noch zu sprechen kommen.

Fragen zu 1.1.

1.) Was ist ein Bus ? Welche zwei grundsätzlichen Arten von Bussen gibt es ?

2.) Welche Funktion können der Accumulator, das X- und das Y-Register in gleicher Weise erfüllen ?

3.) Welche besonderen Funktionen hat der Accumulator ?

4.) Welchen Adressbereich kann der Adresszähler adressieren ?

5.) Welchen Adressbereich belegt der Stapelspeicher ?

6.) Welcher Unterschied besteht zwischen dem "normalen" Abspeichern eines Bytes im RAM und dem Abspeichern im Stapelspeicher ?

Antworten zu 1.1.

zu 1.) Ein Bus ist ein Informationsträger. Es gibt uni- und bi- direktionale Busse.

zu 2.) Diese drei Register können ein Byte zur weiteren Verwendung speichern.

zu 3.) Der Accumulator enthält immer das eine Argument und dann das Ergebnis einer arithmetischen oder logischen Berechnung. Der Stapelspeicher wird immer über den Accumulator beschrieben oder ausgelesen.

zu 4.) Der Adresszähler bildet ein 16 Bit breites Register. Aufgeteilt in High- und Low- Byte. Die höchste darstellbare Zahl mit 16 Bit ist 1111111111111111 (dezimal 65535). Einschließlich der Adresse 0000000000000000 sind demnach 65536 Bytes über den Adressbus adressierbar. Das sind 64K Byte.

zu 5.) Der Stapelspeicher wird durch den Stapelzeiger adressiert. Der Stapelzeiger ist 8 Bit breit. Die höchste darstellbare Zahl mit 8 Bit ist 11111111 (dezimal 255). Einschließlich 00000000 sind demnach 256 Bytes im Stapelspeicher abzuspeichern. Da das High- Byte des Adressbus bei Adressierung durch den Stapelzeiger immer 00000001 ist, belegt der Stapelspeicher den Bereich von 0000000100000000 (dezimal 256) bis 0000000111111111 (dezimal 511).

zu 6.) Gewöhnlich wird ein Byte unter Angabe der Adresse im RAM abgespeichert. Wird ein Byte im Stapelspeicher abgelegt, so ist keine Adressangabe erforderlich. Es muß allerdings berücksichtigt werden, daß das zuletzt abgelegte Byte zuerst wieder ausgelesen wird.

1.2. Memory Display (Speicherinhalt zeigen)

Es wurde bereits ausgeführt, daß die Speicherinhalte in der Regel immer im Hexformat ausgegeben werden. Der Speicherinhalt %01010110 wird also von den meisten Maschinensprache- Monitoren im Hexformat als $56 auf dem Bildschirm ausgegeben.
Bevor der Monitor jedoch einen Speicherinhalt zeigen kann, müssen wir die Adresse des gewünschten Speicherplatzes angeben. Diese Angabe wird meist auch im Hexformat erwartet. Oft ist es möglich einen sogenannten Hexdump des Speicherinhaltes darzustellen. Ein Hexdump zeigt einen Speicherbereich in aufeinanderfolgenden Hexzahlen. Sie sehen unten einen solchen Hexdump des Bereiches $A000 - $A023

```
A000  94 E3 7B E3 43 42 4D 42 41 53 49 43
A00C  30 A8 41 A7 1D AD F7 A8 A4 AB BE AB
A018  80 B0 05 AC A4 A9 9F A8 70 A8 27 A9
```

Die vierstellige Hexzahl zu Beginn jeder Zeile ist die Adresse des ersten abgebildeten Bytes der Zeile. Der Speicherinhalt an Adresse $A000 ist also $94 - an Adresse $A001 = $E3 usw. Um einen solchen Hexdump zu erhalten ist natürlich die Anfangs- und Endadresse anzugeben. Meist in der Form: M A000 A023.
Einige Monitore, so PROFI MON und T.EX.ASS., stellen neben den Hexwerten noch die entsprechenden ASCII Zeichen dar. Da der Prozessor bekanntlich nur in Binärzahlen "denkt", müssen alle Zeichen (Buchstaben, Zahlen usw.) auch in Form von Binärzahlen verarbeitet werden. Man hat sich hier glücklicherweise auf einen Standard geeinigt. Dieser Standard ist der ASCII Code. In der Regel wird Text nach diesem Code im Speicher abgelegt. Sie finden im Anhang eine ASCII Code Tabelle abgedruckt. Wenn der durch den Hexdump dargestellte Speicherbereich also Text in Form von ASCII Zeichen enhält, wird dieser Text sofort lesbar dargestellt.
Wir kommen nun zu den einzelnen Monitoren. Wenn Sie einen anderen Monitor als einen der beiden vorgestellten benutzen, so entnehmen Sie die entsprechenden Befehle bitte Ihrem Handbuch.

Der Befehl M (Memory Display) hat das Format: M XXXX YYYY. Wobei XXXX die vierstellige Startadresse und YYYY die vierstellige Endadresse jeweils im Hexformat sind. Um den obigen Hexdump auf dem Bildschirm zu erhalten, ist folgender Befehl zu geben: M A000 A023 - Zwischen den Angaben muß jeweils ein Leerzeichen (Spacetaste) stehen. Rechts neben dem Hexdump werden die ASCII Zeichen dargestellt. Da in diesem Fall nur in den ersten beiden Zeilen Text (CBM BASIC) enthalten ist, ergeben die anderen ASCII Zeichen keinen Sinn. Probieren Sie das einmal aus.

T.EX.ASS.

Schalten Sie den Monitor zuvor mit dem Befehl: .HE - auf Hexadezimalbetrieb. Wir gehen in diesem Buch im Weiteren immer davon aus, daß im Hexadezimalbetrieb gearbeitet wird.
Der Monitor bietet Ihnen drei Möglichkeiten zur Anzeige eines Speicherinhaltes.
.DB $XXXX - Das Dollarzeichen vor der vierstelligen Startadresse muß unbedingt eingegeben werden. Das gilt für alle Eingaben im Hexformat, da sich der Befehl .HE (Hexadezimalbetrieb) oder .DE (Dezimalbetrieb) nur auf die Ausgabe bezieht. Der Speicherinhalt wird jetzt mit einem Byte und dem entsprechenden ASCII Zeichen pro Zeile ausgegeben.
.DD $XXXX - Ab der Startadresse $XXXX wird der Speicherinhalt mit zwei Byte pro Zeile ausgegeben.
.DC $XXXX - Ab der Startadresse $XXXX wird der Speicherinhalt auf dem ganzen Bildschirm in ASCII Zeichen dargestellt.
Bei allen Funktionen können Sie durch einfache Cursorbewegung die Bereiche oberhalb und unterhalb des bisher dargestellten Bereiches ansehen. Ein Hexdump wie oben gezeigt, läßt sich nicht ausgeben.
Probieren Sie die drei Befehle einmal mit der Adresse $A000 durch.
Mit den Befehlen .PB - .PD - .PC lassen sich die Speicherinhalte auch in der oben gezeigten Form auf den Drucker ausgeben. In diesem Fall ist dann zu der Startadresse noch die Endadresse anzugeben. Beispielsweise: .PB $A000 $A023

Fragen zu 1.2.

1.) Was ist ein Hexdump ?

2.) Wozu dient der ASCII Code ?

3.) Warum werden zur Eingabe der Start- oder Endadresse meist zwei Bytes benötigt ?

Antworten zu 1.2.

zu 1.) Mit einem Hexdump wird ein Speicherbereich zeilenweise in zweistelligen Hexzahlen dargestellt.

zu 2.) Der ASCII - Code ist eine Zuweisung von Buchstaben zu bestimmten Zahlen. Das A ist beispielsweise der 64 zugewiesen. Das ist erforderlich weil der Prozessor nur mit Binärzahlen arbeitet.

zu 3.) Außer für den Fall, daß eine Adresse kleiner $0100 ist, sind zur Darstellung jeder Adresse immer zwei Byte erforderlich. Der Adressraum umfasst den Bereich von $0000 bis $FFFF. Die meisten Monitore erwarten für eine Adresse immer eine vierstellige Angabe. Auch wenn die führenden Stellen 0 sind. Das ist durch interne Umrechnungsroutinen bedingt.

1.3. Go (Programm starten)

Wir haben bisher viel Theorie bearbeiten müssen. Ich hoffe dennoch, daß Sie jetzt noch aufmerksam dabei sind, denn wir wollen gleich das erste kleine Maschinenprogramm eingeben und starten.
Das Programm soll die einfache Aufgabe lösen, das Byte $AA an die Adresse $5000 im RAM zu transportieren. Aber auch hier haben die "Götter" die Theorie vor die Praxis gesetzt: Sie wissen bereits, daß der ACCU, das X- und das Y- Register in der Lage sind Bytes an einer bestimmten Speicheradresse abzulegen. Zuerst muß das gewünschte Byte hierzu allerdings in das entsprechende Register eingeladen werden. Sie finden drei Befehle dieser Art zu Beginn der 65xx Befehle im Anhang dieses Buches. Sehen Sie dort einmal nach. Die Befehle lauten: LDA (lade den ACCU), LDX (lade das X- Register), LDY (lade das Y- Register). Wir wollen das Byte $AA in den ACCU laden. Der passende Befehl lautet also "LDA", das Byte $AA. Nur den Ausdruck "das Byte $AA" kann der Prozessor so nicht verstehen. Wir müssen dem Prozessor also mitteilen woher er das zu ladende Byte holen soll. Das geschieht mit den verschiedenen Adressierungsarten. Sie finden diese wieder im Anhang.
An dieser Stelle eine kurze Anmerkung zum Anhang: Im Moment ist es sicher manchmal lästig von diesen Seiten zurück in den Anhang zu blättern. Natürlich wäre es möglich, die entsprechenden Anhangseiten direkt bei den behandelnden Kapiteln abzudrucken. Dem steht folgende Überlegung gegenüber: Voraussichtlich werden Sie die meiste Zeit dieses Buch gebrauchen, wenn Sie den Grundlagen und Übungsteil bereits durchgearbeitet haben. Wenn Sie also eigenständig programmieren. Sie werden dann feststellen, daß Sie bestimmte Tabellen und ähnliche Informationen auch als Profi immer wieder benötigen. Die in der Praxis am häufigsten gebrauchten Tabellen finden Sie dann übersichtlich gesammelt am Ende dieses Buches. Dadurch daß wir schon jetzt häufig den Anhang gebrauchen, lernen Sie diesen sehr gut kennen. Er wird so zu einem immer wieder nützlichen Arbeitsmittel.
Jetzt aber zurück zu unserem ersten Programm. Sehen Sie sich also einmal die 65xx Adressierungsarten im Anhang an. Einige

der dort aufgeführten Möglichkeiten sind sicher nicht sofort verständlich. Wir werden alle Arten mit Beispielen trainieren. Für unser gestelltes Problem ist die zweite aufgeführte Adressierungsart zweckmäßig: die unmittelbare Adressierung. Diese Adressierung teilt dem Prozessor mit, daß das unmittelbar auf das Befehlsbyte folgende Byte in den ACCU zu laden ist. Das wollen wir genauer betrachten: Das Kürzel für die unmittelbare Adressierung ist IM. Bei den 65xx Befehlen finden Sie jeweils unter der Erklärung des Befehls eine vierspaltige Tabelle. In der ersten Spalte stehen die Kürzel der Adressierung. "IM" finden Sie bei dem Befehl LDA in der ersten Zeile der Tabelle. Die zweite Spalte zeigt den Zusatz zur Erkennung der Adressierung für die Assemblersprache. Die dritte Spalte gibt den Code für den Befehl und die Adressierungsart in einem Byte zusammengefasst an. Dieser hier als Hexzahl ($A9) dargestellte Code, steht so als entsprechende Bitfolge im Speicher. Wenn der Prozessor beim Abarbeiten eines Programmes auf diese Bitfolge stößt, erkennt die Befehlserkennung hieraus, daß der ACCU mit dem nächsten Byte im Speicher zu laden ist. Die Informationen über die Art des Befehls und die gewünschte Adressierung (also woher/wohin das Datenbyte kommen soll) ist somit immer in einem Byte, dem Befehlsbyte, enthalten. Es ist wichtig sich das zu vergegenwärtigen.
Die vierte Spalte gibt die zur Ausführung benötigten Zyklen an. Damit sind die Taktzyklen des Prozessors gemeint. Im C 64 wird der 6510 Prozessor mit etwa einem Megaherz getaktet. Das bedeutet, daß ein Taktzyklus etwa eine Mikrosekunde dauert. Der LDA Befehl benötigt also etwa 2 Mikrosekunden zur Ausführung. In zeitkritischen Programmen kann diese Information sehr wichtig werden.
Die auf das Befehlsbyte folgenden Bytes sind die Operanden. Je nach Befehl können 0, 1 oder 2 Operanden erforderlich sein. Die unmittelbare Adressierung erfordert einen Operanden, eben das Datenbyte. Im Speicher steht der von uns benötigte Befehl somit folgendermaßen:

Befehlsbyte	$A9
Operand	$AA

Damit ist der erste Schritt, den ACCU mit dem Byte $AA zu

laden, erfolgt. Es gilt jetzt den Akkumulator- Inhalt an der Adresse $5000 abzulegen. In den 65xx Befehlen finden Sie die Befehle: STA (speichere den ACCU), STX (speichere das X-Register) und STY (speichere das Y- Register). Diese Befehle dienen dazu, einen Registerinhalt an einer bestimmten Speicher- Adresse abzulegen. Auch hier ist es erforderlich zusammen mit dem Befehl die Adressierung festzulegen. Versuchen Sie einmal unter den ersten fünf 65xx Adressierungen im Anhang die herauszufinden, welche die Speicherstelle $5000 adressieren kann.
Die absolute Adressierung kommt als Einzige in Frage. Das Befehlsbyte ist also unter STA AB zu suchen. Sie finden hier das Byte $8D. Diesem Befehlsbyte folgen die Operanden, welche die Adresse enthalten. Beachten müssen Sie dabei, daß zwei-Byte Adressen immer zuerst mit dem Low- und dann mit dem High- Byte im Speicher abgelegt werden müssen. Die Bytefolge im Speicher für unser Programm sieht bis jetzt so aus:

Befehlsbyte	$A9	LDA - IM
Operand	$AA	
Befehlsbyte	$8D	STA - AB
1. Operand	$00	Low- Byte
2. Operand	$50	High- Byte

Sehen wir uns das Programm einmal genau an und verfolgen die Vorgänge am Programmiermodell: Der Adresszähler wird mit der Startadresse des Programmes geladen. Diese Adresse wird auf den Adressbus gelegt. Das erste Byte des Programmes ($A9) Wird auf den Datenbus gelegt. Da unmittelbar vorher das Programm gestartet wurde, weiß die Befehlserkennung, daß dieses Byte ein Befehlsbyte sein muß. Entsprechend erkennt die Befehlserkennung den Befehl LDA - IM. Die Befehlserkennung erhöht daraufhin den Adresszähler um eins. Diese Adresse auf den Adressbus ausgegeben bewirkt, daß das Byte $AA auf dem Datenbus liegt. Die Befehlserkennung veranlaßt den ACCU dieses Byte vom Datenbus zu laden. Daraufhin wird der Adresszähler wieder um eins erhöht. Die Befehlserkennung weiß, daß das folgende Byte wieder ein Befehlsbyte sein muß. Das nach der Adresszähler- Erhöhung auf dem Datenbus liegende Byte $8D, wird somit als der Befehl STA - AB interpretiert. Die gewünschte absolute Adressierung

veranlaßt die Befehlserkennung dazu, den Adresszähler um eins zu erhöhen, den Operanden $00 zwischenzuspeichern, den Adresszähler nochmal um eins zu erhöhen, den Operanden $50 zwischenzupeichern, den Adresszähler Low- und High zwischenzuspeichern und schließlich die Bytes $00 und $50 in den Adresszähler zu laden. Auf dem Adressbus liegt somit die Adresse $5000. Dann wird noch der Accumulator- Inhalt $AA auf den Datenbus gelegt und das RAM veranlaßt den Datenbusinhalt an der adressierten Speicherstelle abzulegen. Damit ist die gewünschte Operation ausgeführt. Der zwischengespeicherte Adresszähler- Inhalt kann wieder zurückgeschrieben und das Programm mit dem nächsten Befehl fortgesetzt werden. Es wird Ihnen bei Ihrer späteren Programmiertätigkeit sehr dienlich sein, wenn Ihnen diese Grundlagen völlig geläufig sind.
Wie wir oben gesehen haben, fährt der Prozessor mit dem nächsten Befehl fort. Da die von uns gewünschte Operation bereits ausgeführt ist, muß dieser Befehl den Prozessor dazu veranlassen das Programm zu beenden und zum Monitor zurückzukehren. Zu diesem Zweck stehen uns zwei Befehle zur Verfügung. Der BRK und der RTS Befehl. Die genaue Bedeutung dieser Befehle werden wir in einem späteren Kapitel besprechen. Mit welchem Befehl Sie bei dem von Ihnen verwendeten Monitor von einem gestarteten Programm zum Monitor zurückkehren, müssen Sie ausprobieren oder dem Handbuch entnehmen. In den meisten Fällen wird mit dem BRK Befehl zum Monitor zurückgekehrt (so auch bei PROFIMON und T.EX.AS). Die Befehlsbytes finden Sie bei den 65xx Befehlen im Anhang (vorletzten zwei Befehle). Beide Befehle kennen nur die implizierte Adressierung. Da keine Adress- oder Datenangaben gemacht werden müssen, ist nur das Befehlsbyte ohne Operanden erforderlich (BRK - $00 / RTS - $60).
Das gesamte Programm um das Byte $AA an der Adresse $5000 abzulegen sieht wie folgt aus:

Speicheradresse	Inhalt	Bedeutung
$5000	xx	Nach PGM Start = $AA
$5001	$A9	LDA - IM
$5002	$AA	Operand
$5003	$8D	STA - AB
$5004	$00	1. Operand
$5005	$50	2. Operand
$5006	$00	BRK
	($60)	(RTS)

Dieser Speicherbereich wurde so gewählt, weil in diesem Bereich in der Regel unser kleines Programm mit dem Monitorprogramm nicht kollidiert, also nicht dieselben Adressen belegt. Das Programm, die Bytes von Adresse $5001 - $5006, können an jede andere RAM - Adresse gebracht werden, falls Ihr Monitor den obigen Speicherbereich belegt. Natürlich muß die Abspeicheradresse für das Byte $AA von $5000 dann entsprechend abgeändert werden.
Verwenden Sie einen anderen als die besprochenen Monitore, so wird Ihnen Ihr Handbuch die Informationen zur Eingabe der Bytes unseres Programmes und dem Programmstart geben.

PROFIMON
Geben Sie den Befehl M 5000. Daraufhin erscheint eine Zeile mit den Speicherinhalten der Speicherplättze $5000 - $5007. Der Inhalt dieser Bytes interessiert uns jetzt nicht. Fahren Sie mit dem Cursor auf das zweite Byte der Zeile, das ist das Byte mit der Adresse $5001, und überschreiben den Inhalt mit A9. Auf dieselbe Weise sind die anderen Programmbytes einzugeben. Folgende Zeile muß dann auf Ihrem Bildschirm stehen:

>: 5000 xx A9 AA 8D 00 50 00 xx ASCII Zeichen

Steht die Zeile so auf Ihrem Bildschirm, so betätigen Sie die RETURN Taste noch während der Cursor in der Zeile steht. Geben Sie dann noch einmal den Befehl M 5000. Daraufhin muß die Zeile wie oben gezeigt erscheinen. Unser Programm steht dann richtig im Speicher. Andernfalls müssen Sie unbedingt die Zeile entsprechend korrigieren.

Starten Sie dann das Programm mit dem Befehl G 5001. Achten Sie darauf, 5001 und nicht 5000 einzugeben. Ist alles richtig verlaufen, so erscheint sofort die Statuszeile mit den Registerinhalten. Über dieser Zeile sehen Sie noch die Zeichen B*. Das ist ein Hinweis darauf, daß mit dem BRK Befehl abgebrochen wurde. Geben Sie jetzt wieder den Befehl M 5000. Folgende Zeile muß erscheinen:

>: 5000 AA A9 AA 8D 00 50 00 xx ASCII Zeichen

T.EX.AS.
Geben Sie den Befehl .HE (Hexadezimalbetrieb) und dann den Befehl .DB $5000. Sie sehen den Speicherbereich $5000 - $5018 wie in 1.2. beschrieben dargestellt. Fahren Sie mit dem Cursor in die zweite Bildschirmzeile an die Adresse $5001. Tragen Sie ab da unser Programm ein. Sie müssen hierzu einfach das ausgegebene Byte mit den neuen Werten überschreiben und die RETURN - Taste betätigen. Sind alle Programmbytes eingetragen, so geben Sie noch einmal den Befehl .DB $5000. Sie können diesen Befehl in der Zeile geben in der Sie sich gerade befinden. Überzeugen Sie sich an den erneut dargestellten Speicherinhalten, daß das Programm richtig eingegeben wurde. Ist das der Fall, so starten Sie das Programm mit dem Befehl .EX $5001. Duch den erneuten Befehl .DB $5000 können Sie sich überzeugen, daß an Adresse $5000 das Byte $AA eingetragen wurde.

Was ist zu tun wenn´s nicht stimmt ?
Meldet sich der Monitor überhaupt nicht zurück oder enthält die Adresse $5000 nach Programmablauf nicht den Wert $AA, so stimmt die von Ihnen eingetragene Bytefolge nicht oder Sie haben eine falsche Startadresse angegeben. Meldet sich der Monitor nicht zurück, müssen Sie den Computer abschalten und dann den Monitor neu laden.

Fragen zu 1.3.

1.) Welche anderen Prozessor- Register könnten die Aufgabe des ACCU in unserem Programm übernehmen ? Stellen Sie die Bytefolgen für diese beiden Möglichkeiten auf und probieren Sie sie aus.

2.) Warum ließe sich für unser Beispiel nicht die nullseitige Adressierung verwenden ? Schlagen Sie hierzu im Anhang bei den 65xx Adressierungen nach.

3.) Stellen Sie ein Programm auf, welches das Byte $AA an der Speicheradresse $05 ablegt. Das Programm soll an der Adresse $5000 starten. Vorsicht, die nullseitige Adressierung ist zu verwenden. Diese Adressierung benötigt nur einen Operanden. Probieren Sie das Programm aus.

4.) Welche Vorteile hat die nullseitige Adressierung offensichtlich ?

5.) Was würde geschehen, wenn Sie für den Befehl STA - AB $5000 die Bytefolge $8D $50 $00 in den Speicher bringen und das Programm starten ?

6.) Warum ist die unmittelbare Adressierung für den STA Befehl nicht möglich ?

Antworten zu 1.3.

zu 1.) Das X- und das Y- Register können den ACCU ersetzen. Die entsprechende Bytefolge für das X- Register wäre: $A2 $AA $8E $00 $50 $00 - für das Y- Register: $A0 $AA $8C $00 $50 $00

zu 2.) Dadurch, daß bei der nullseitigen Adressierung das High- Byte der Adresse automatisch auf null gesetzt wird, läßt sich mit dieser Adressierungsart nur der Bereich von $00 - $FF adressieren. Die Adresse $5000 läßt sich somit über die nullseitige Adressierung nicht erreichen.

zu 3.) Die Bytefolge des Programms muß wie folgt lauten: $A9 $AA $85 $05 $00

zu 4.) Das Programm wird um ein Byte kürzer und es muß nur ein Adressbyte aus dem Speicher gelesen werden, wodurch das Programm auch schneller wird. Bei zeitkritischen Anwendungen kann das sehr wichtig werden.

zu 5.) Durch die vertauschten Adressbytes wird das Byte $AA an Adresse $0050 abgelegt.

zu 6.) Da Maschinenprogramme in der Regel nach dem Austesten im ROM (nur lese Speicher) stehen, wäre es dann nicht mehr möglich das Byte unmittelbar hinter dem Befehlsbyte zu verändern.

1.4. Load / Save (Programm laden / speichern)

Die Befehle LOAD und SAVE kennen Sie bereits von BASIC her. Wird in BASIC ein Programm abgespeichert, so müssen nur der Name des Programmes und das Speichergerät durch die Geräteadresse angegeben werden. Den Programmstart und das Programmende entnimmt das Betriebssystem selbständig den entsprechenden Zeigern im Speicher. Um ein Maschinenprogramm abzuspeichern, sind dagegen zusätzlich noch die Angaben über die Anfangs- und Endadresse zu geben. Die Startadresse wird mit dem Programm auf der Diskette abgespeichert. Soll das Programm beim Einlesen an dieselbe Adresse von der es gespeichert wurde geladen werden, so muß zu dem Ladebefehl keine Startadresse angegeben werden. Die meisten Monitore bieten aber auch die Möglichkeit ein Programm an eine andere Adresse zu laden als die von welcher es abgespeichert wurde. In diesem Fall muß die neue Startadresse natürlich zusammen mit dem Ladebefehl angegeben werden.
Zur Übung dieser Befehle wollen wir einmal das im letzten Abschnitt erstellte Programm abspeichern und wieder laden.

PROFIMON
Der SAVE Befehl lautet:

```
S "name",Geräteadresse,Anfangsadresse,Endadresse + 1
```

Dabei gilt die Geräteadresse 01 für das Kassettengerät und die Geräteadresse 08 für die Floppy Disk. Die Anfangs- und Endadresse ist immer als vierstellige Hexzahl anzugeben. Somit ist folgende Befehlszeile einzugeben um das Programm auf Diskette zu speichern:

```
S "EINS",08,5001,5007
```

Nachdem Sie das Programm auf diese Weise abgespeichert haben, löschen Sie das Programm indem Sie mit Hilfe des M Befehls alle Bytes der Adressen $5001 - $5006 auf den Wert $FF setzen. Kontrollieren Sie die erfolgte Löschung durch den Befehl: M 5000. Jetzt soll das Programm wieder an die alte Adresse geladen werden. Der Ladebefehl lautet:

L "name",Geräteadresse,Anfangsadresse

Wenn Sie die Geräteadresse und die Anfangsadresse nicht angeben, so wird das Programm an die ursprüngliche Adresse von der Floppy mit der Geräteadresse 8 geladen. Durch Angabe der Anfangsadresse können Sie das Laden ab einer bestimmten Adresse erzwingen. Geben Sie jetzt folgende Befehlszeile ein:

L "EINS"

Da das Programm von der Floppy an die ursprüngliche Adresse geladen werden soll, sind keine weiteren Angaben erforderlich. Überprüfen Sie mit M 5000 ob das Programm wieder korrekt eingelesen wurde. Wir wollen das Programm auch einmal an eine andere Adresse laden. Geben Sie hierzu folgende Befehlszeile ein:

L "EINS",08,4000

Überprüfen Sie wieder mit M 4000 ob das Programm korrekt eingelesen wurde.

T.EX.AS.
Dieser Monitor erwartet durch seine Workspace Konzeption eine etwas andere Handhabung des SAVE Befehls. Der Monitor bietet die Möglichkeit Workspaces (Arbeitsbereiche) festzulegen. Wir kennen bereits drei Ausgabearten für die sich je ein Workspace festlegen läßt. Ich will das an dem Befehl .DB erläutern: Der Befehl .DB gibt wie Sie wissen den Speicherinhalt in Byteform wieder. Mit dem Befehl:

.WB Anfangsadresse Endadresse

läßt sich für die Wiedergabe in Byteform ein bestimmter Arbeitsbereich festlegen. Das kann die Arbeit mit dem Monitor erleichtern. Ein Beispiel hierfür ist die Möglichkeit den Befehl .DB durch einfaches betätigen der Taste f3 auszuführen. Es wird dann der Bereich ab .WB Anfangsadresse dargestellt. Auf diese Weise gehören auch die Befehle .DD - .WD und die Taste f4 sowie die Befehle .DC - .WC und die

Taste f2 zusammen. Der Befehl .WA gibt Ihnen einen Überblick über alle Workspaces. Probieren Sie die verschiedenen Möglichkeiten einmal aus. Die anderen Workspaces werden wir noch besprechen.
Der SAVE Befehl bezieht sich immer auf den Workspace .WS. Bevor wir unser Programm abspeichern können, ist dieser Bereich mit dem Befehl:

```
.WS $5001 $5007
```

festzulegen. Kontrollieren Sie mit dem Befehl .WA die Ausführung des .WS Befehls. Das Abspeichern erfolgt dann einfach durch die Zeile:

```
.SA"EINS"
```

Nachdem Sie das Programm auf diese Weise abgespeichert haben, löschen Sie das Programm indem Sie mit Hilfe des .DB Befehls alle Bytes der Adressen $5001 - $5006 auf den Wert $FF setzen. Kontrollieren Sie die erfolgte Löschung durch den Befehl: .DB $5000. Jetzt soll das Programm wieder an die alte Adresse geladen werden. Der Ladebefehl lautet:

```
.LO Anfangsadresse "name"
```

Wenn Sie die Anfangsadresse nicht angeben, so wird das Programm an die ursprüngliche Adresse von der Floppy geladen. Durch Angabe der Anfangsadresse können Sie das Laden ab einer bestimmten Adresse erzwingen. Geben Sie jetzt folgende Befehlszeile ein:

```
.LO "EINS"
```

Da das Programm von der Floppy an die ursprüngliche Adresse geladen werden soll, sind keine weiteren Angaben erforderlich. Überprüfen Sie mit .DB $5000 ob das Programm wieder korrekt eingelesen wurde. Wir wollen das Programm auch einmal an eine andere Adresse laden. Geben Sie hierzu folgende Befehlszeile ein:

```
.LO $4000 "EINS"
```

Überprüfen Sie wieder mit .DB $4000 ob das Programm korrekt eingelesen wurde.

Fragen zu 1.4.

1.) Welche Angaben sind erforderlich um ein Maschinenprogramm abzuspeichern ?

2.) Welche Angaben sind erforderlich um ein Maschinenprogramm zu laden ?

3.) Wie starten Sie das Beispielprogramm wenn Sie es wie beschrieben ab Adresse $4000 geladen haben ?

Antworten zu 1.4.

zu 1.) Name - Startadresse - Endadresse - Geräteadresse

zu 2.) Name - Geräteadresse - eventuell Startadresse

zu 3.) PROFIMON: G 4000 / T.EX.ASS. .EX $4000 Zu beachten ist, daß das Byte $AA weiterhin an Adresse $5000 abgelegt wird.

1.5. Disassemble (Programm in Mnemonics)

Diese Monitorfunktion haben wir bereits in Monitor / Assembler angesprochen. So wie wir den Code %10101001 zu der Hexadezimalzahl $A9 vereinfacht haben, können wir den Code noch übersichtlicher und einprägsamer machen, indem wir ihn durch ein Symbol in Form einer Buchstabenfolge ersetzen. Wir haben das bei unserem Beispielprogramm auch bereits getan. Sie erinnern sich: Die Hexzahl $A9 steht für den Befehl LDA und die unmittelbare Adressierung. Im Anhang bei den 65xx Befehlen steht in der zweiten Spalte der Tabelle unter jedem Befehl das in der Assemblersprache verwendete Zeichen zur Kenntlichmachung der verwendeten Adressierung. Steht in dieser Spalte kein Zeichen, so erübrigt sich die besondere Kennzeichnung der Adressierung. Das trifft für drei Adressierungsarten zu:

Die implizierte Adressierung. Alle Befehle mit dieser Adressierung beziehen sich schon durch den Befehl auf ein bestimmtes Register oder benötigen keine Daten zu ihrer Ausführung. Eine Adressierung ist dadurch überflüssig. Beispiel: BRK
Die absolute Adressierung wird an den nachfolgenden zwei Operanden erkannt die eine 16 Bit Adresse bilden. Beispiel: STA $5000
Die nullseitige Adressierung wird an dem nachfolgenden einen Operanden erkannt, welcher die Adresse einer Speicherzelle in der Zeropage (nullten Seite) bildet. Beispiel: STA $05

An dem Beispiel für die absolute Adressierung wird Ihnen aufgefallen sein, daß die Adresse nicht in der Reihenfolge Low- Highbyte sondern in gewöhnlicher Schreibweise angegeben wurde. Das ist einer der Vorteile der Assemblersprache. 16 Bit Adressen werden in der üblichen Schreibweise ein- oder ausgegeben. Im Speicher muß für das Maschinenprogramm selbstverständlich die vom Prozessor geforderte Reihenfolge eingehalten werden.

Das Beispielprogramm in Assembler sieht damit so aus:

```
$5001 LDA # $AA
$5003 STA   $5000
$5006 BRK
```

Die vierstellige Hexzahl vor jedem Befehl ist die Adresse des Befehlsbytes.
Die meisten Monitore bieten die Möglichkeit ein Maschinenprogramm zu disassemblieren. Diese Funktion ist ähnlich der Funktion Memory Display in 1.2., nur daß der Speicherinhalt in Assemblersprache ausgegeben wird. Zu diesem Zweck ist dann der entsprechende Befehl einschließlich der Anfangs- und Endadresse des darzustellenden Bereiches zu geben. Enthält dieser Bereich Daten oder Text, also kein Maschinenprogramm, so kann die Disassemblierung natürlich kein sinnvolles Ergebnis liefern.
Manche Monitore bieten zusätzlich noch die Möglichkeit der direkten Assemblierung. Es ist dann möglich die Befehle als Mnemonics einzugeben. Der Monitor setzt die symbolische Eingabe sofort in den richtigen Code um und speichert diesen an der aktuellen Speicheradresse ab. Das ist ein Direktassembler und ist nicht zu verwechseln mit dem Assembler welchen wir im zweiten Teil dieses Buches besprechen werden.

PROFIMON
Das Befehlsformat zur Disassemblierung ist:

```
D XXXX YYYY
```

XXXX ist die Anfangs- und YYYY die jeweils vierstellige Endadresse in Hexformat. Geben Sie zur Disassemblierung des Beispiel- Programmes einmal folgenden Befehl:

```
D 5001 5007
```

Folgendes muß auf Ihrem Bildschirm erscheinen:

```
>, 5001 A9 AA      LDA # $AA
>, 5003 8D 00 50   STA $5000
>, 5006 00         BRK
```

Diese Ausgabe hat 5 Spalten. In der ersten Spalte steht die Adresse des Befehlsbytes. Das Befehlsbyte steht in der zweiten Spalte. In der dritten und vierten Spalte stehen der 1. und 2. Operand. Die vierte Spalte zeigt dann das Mnemonic mit Adressierung und Operanden. Die Bytes der ersten drei Spalten können Sie durch einfaches Überschreiben ändern. Das disassemblierte Mnemonic ändert sich dann entsprechend nach betätigen der RETURN Taste. Probieren Sie das einmal aus.
Sie verlassen diesen Modus, indem Sie mit dem Cursor nach unten in eine leere Zeile fahren und dann die RETURN Taste betätigen.

T.EX.AS.
Der Disassembler wird hier mit folgendem Befehl eingeschaltet:

```
.DL $XXXX
```

XXXX stellt eine vierstellige Hexzahl dar. Das Beispielprogramm sehen Sie sich mit folgender Befehlszeile an:

```
.DL $5001
```

Der Bildschirm muß das Programm in folgender Form zeigen:

```
$5001 LDA  #$AA
$5003 STA $5000
$5006 BRK
```

Sie können das Programm ändern, indem Sie einfach mit dem Cursor in die entsprechende Zeile fahren und das Mnemonic ändern oder ersetzen. Nach betätigen der RETURN Taste wird das von Ihnen eingegebene Mnemonic übersetzt und der Befehlscode gefolgt von den Operanden sofort im Speicher abgelegt. Sie können danach sofort den nächsten Befehl eingeben. Probieren Sie das einmal aus. Sehen Sie sich den Speicherbereich auch mit dem .DB Befehl an, nachdem Sie einige Mnemonics eingegeben haben.
Der Workspace für diese Funktion wird mit dem .WL Befehl

festgelegt. Die Druckerausgabe erreichen Sie über den .PL Befehl.

Fragen zu 1.5.

1.) Welche Adressierungsarten werden in der Assemblersprache nicht eigens kenntlich gemacht ?

2.) Muß in der Assemblersprache eine 16 Bit- Adresse auch in der Reihenfolge Low- Highbyte angegeben werden ? Begründen Sie Ihre Antwort.

3.) Warum wird die Adresse der Befehlsbytes in einem Assemblerlisting nicht von Zeile zu Zeile nur um eins erhöht ?

4.) Nehmen wir an, das Beispielprogramm soll wie folgt erweitert werden:

```
LDA # $AA
LDX # $00
STA $5000
STX $4FFF
BRK
```

Führen Sie diese Erweiterung durch, testen diese und stellen dann wieder das ursprüngliche Programm her. Welcher Nachteil im Bezug auf die schon eingegebenen Befehle fällt im Vergleich mit einer ähnlichen Aufgabe bei einem BASIC Programm auf ?

Antworten zu 1.5.

zu 1.) Die implizierte, absolute und die nullseitige.

zu 2.) Nein, 16 Bit- Adressen können in der Assemblersprache in gewöhnlicher Schreibweise eingegeben werden. Der Assembler speichert diese Adressen dann automatisch in der vom Prozessor geforderten Reihenfolge als Operanden ab.

zu 3.) Im Speicher folgen auf die Befehlsbytes, außer bei Einbyte- Befehlen, immer ein oder zwei Operanden. So steht das nächste Befehlsbyte auch nicht an der Adresse unmittelbar nach dem vorherigen Befehlsbyte sondern, abhängig von der Anzahl Operanden, ein oder zwei Adressen später.

zu 4.) Da in einem BASIC Programm gewöhnlich ein Zeilennummern- Abstand größer als 1 gewählt wird, ist es leicht möglich zwischen bestehende Zeilen weitere einzufügen. Das wird dadurch möglich, daß die BASIC Zeilennummern keine absoluten Speicheradressen darstellen, sondern nur die Reihenfolge der Abarbeitung vorschreiben. Die Adressen der Befehle eines Maschinen- Programmes stellen dagegen die tatsächlichen Speicheradressen dar. Werden hier Befehle eingefügt, so müssen alle nachfolgenden Befehle entsprechend von Hand verschoben werden. Das gilt umgekehrt für das Löschen eines Befehls innerhalb eines Maschinen- Programmes. Es läßt sich leicht einsehen, daß dies zu Beginn eines langen Maschinen-Programmes immens viel Arbeit bedeutet. Abhilfe schafft hier ein Assembler wie wir ihn im zweiten Teil des Buches besprechen.

1.6. Compare (Speicherbereiche vergleichen)

Diese oft sehr hilfreiche Funktion bieten viele Monitore. Bei Aufruf der Funktion werden zwei Speicherbereiche Byte für Byte miteinander verglichen. Werden zwei unterschiedliche Bytes gefunden, so wird die Adresse des unterschiedlichen Bytes ausgegeben und die Suche fortgesetzt, bis das nächste unterschiedliche Byte oder das Bereichsende erreicht wird.
Um die beiden Speicherbereiche festzulegen, sind mindestens drei Angaben erforderlich. Ein Bereich ist mit Anfangs- und Endadresse anzugeben. Für den zweiten Bereich genügt die Anfangsadresse. Die Endadresse des zweiten Bereiches ergibt sich dann aus der Länge des ersten Bereiches und muß somit nicht angegeben werden. Die ausgegebenen Adressen für ungleiche Speicherplätze beziehen sich bei den meisten Monitoren auf den zweiten Bereich.
Probieren Sie diese Funktion aus, indem Sie unser Beispielprogramm von der Diskette einmal ab Adresse $4000 und einmal ab Adresse $5000 laden und dann den Vergleichsbefehl für diese Bereiche geben. Im zweiten Teil des Buches werden wir die praktische Anwendung des Befehls noch an einem Beispiel kennenlernen.

PROFIMON
Das Befehlsformat ist hier folgendermaßen vorgeschrieben:

```
C XXXX YYYY ZZZZ
```

XXXX und YYYY müssen die Anfangs- und Endadresse des ersten Bereiches als vierstellige Hexzahl enthalten. ZZZZ enthält die Anfangsadresse des zweiten Bereiches.
Nachdem Sie das Beispielprogramm einmal ab Adresse $4000 (L "EINS",08,4000) und einmal ab Adresse $5000 (L "EINS",08,5000) eingeladen haben, geben Sie folgenden Befehl:

```
C 4000 4005 5000
```

Wenn Sie alles richtig gemacht haben, so meldet sich der Monitor ohne weitere Mitteilung zurück. Da die beiden Bereiche identisch sind, findet die Funktion auch keine

unterschiedlichen Bytes. Ändern Sie jetzt ein Byte in einem der beiden Bereiche und geben erneut den "Compare" Befehl. Die Adresse des unterschiedlichen Bytes aus dem zweiten Bereich muß jetzt ausgegeben werden.
Sie können genauso wie mit dem BASIC Editor einen Befehl zweimal geben, indem Sie das zweitemal einfach mit dem Cursor in die Befehlszeile fahren und die RETURN Taste betätigen. Bei gleichen Befehlen oder nur kleinen Änderungen kann das einige "Tipperei" sparen.

T.EX.AS.
Der Monitor kennt diese Funktion nicht. Wenn Sie die Funktion öfter benötigen, lohnt es sich eine entsprechende Hilfsroutine zu programmieren. Sie werden dazu nach Durcharbeiten dieses Buches leicht in der Lage sein.

Fragen zu 1.6.

1.) Warum muß für den zweiten Vergleichsbereich nur die Anfangsadresse angegeben werden ?

2.) Lassen sich mit der Compare- Funktion auch Bereiche mit Text vergleichen ?

3.) Wieviel ungleiche Bytes können maximal gefunden werden ?

Antworten zu 1.6.

zu 1.) Der Monitor berechnet die Endadresse des zweiten Bereiches selbsttätig aus der Anfangs- und Endadresse des ersten Bereiches.

zu 2.) Selbstverständlich. Text wird in Byteform, meist nach dem ASCII Code, im Speicher abgelegt. Sind die Adressen der zu vergleichenden Texte im Speicher bekannt, so ist ein Vergleich mit der Compare-Funktion ohne weiteres möglich. Für die Compare-Funktion ist es völlig unwichtig, ob die zu vergleichenden Bytes Text, Befehle oder Daten enthalten. Verglichen wird nur der Wert der Bytes.

zu 3.) Wenn kein Byte der beiden Bereiche in der richtigen Reihenfolge miteinander übereinstimmt, so werden alle Bytes eines Bereiches als nicht übereinstimmend gefunden.

1.7. Transfer (Blöcke kopieren)

Diese Monitorfunktion werden Sie noch sehr schätzen lernen. Wie die Bezeichnung bereits vermuten läßt, können Sie mit Hilfe der Transfer- Funktion einen beliebigen Speicherbereich in einen RAM Bereich übertragen. Bei einem solchen Transfer eines Maschinenprogrammes, taucht eine Problematik auf, die es erforderlich macht, daß wir uns zunächst mit den 65xx Sprungbefehlen auseinandersetzen.
Die beiden einfachsten Sprungbefehle sind der JSR und der JMP Befehl. Der JSR Befehl ist direkt mit dem BASIC Befehl GOSUB und der JMP Befehl mit dem BASIC Befehl GOTO vergleichbar.
Mit dem JSR Befehl wird also ein Unterprogramm aufgerufen. Wir wollen uns wieder anhand des Programmiermodells genau ansehen wie der Befehl arbeitet: Sobald die Befehlserkennung den Befehlscode für den JSR Befehl liest, holt diese die beiden auf das Befehlsbyte folgenden Operanden durch zweimaliges erhöhen des Adresszählers aus dem Speicher und speichert sie zwischen. Daraufhin wird der Adresszählerinhalt, der jetzt auf den letzten Operanden zeigt, im Stapelspeicher abgelegt. Zuerst das High- und dann das Lowbyte. Dann werden die zwischengespeicherten Operanden des JSR Befehls in den Adresszähler geladen und das erste Byte auf welches der Adresszähler jetzt zeigt, als Befehlsbyte interpretiert. Damit ist der Sprung in das Unterprogramm vollzogen. Die Rückkehr aus einem BASIC Unterprogramm wird mit dem RETURN Befehl erzwungen. Das BASIC Programm arbeitet nach dem RETURN Befehl den Befehl unmittelbar hinter dem GOTO Befehl ab. Auch das ist in der Maschinensprache genauso. Hier bewirkt der Befehl RTS, daß die beiden obersten Bytes des Stapelspeicher in den Adresszähler geladen werden, der Adresszähler um eins erhöht wird und das erste Byte auf welches der Adresszähler nun zeigt als Befehlsbyte interpretiert wird. Wenn das Unterprogramm den Stapelzeiger nicht verändert hat, so zeigt der Adresszähler genau auf das Byte nach dem letzten Operanden des JSR Befehl. Damit ist der Rücksprung von dem Unterprogramm vollzogen.
Der JMP Befehl kennt zwei verschiedene Adressierungen. Zunächst die absolute Adressierung: Der JMP Befehl mit

absoluter Adressierung veranlaßt die Befehlserkennung dazu, einfach die Operanden des JMP Befehls in den Adresszähler zu laden und das Byte auf welches der Adresszähler jetzt zeigt als das nächste Befehlsbyte zu interpretieren. Damit ist der Sprung vollzogen.
Der JMP Befehl kennt als einziger 65xx Befehl die indirekte Adressierung. Auf den Code des JMP Befehl mit indirekter Adressierung reagiert die Befehlserkennung wie folgt: Die Operanden des JMP Befehl werden in den Adresszähler geladen. Das Byte auf welches der Adresszähler dann zeigt, wird zwischengespeichert. Der Adresszähler wird um eins erhöht und das jetzt aktuelle Byte ebenfalls zwischengespeichert. Diese beiden zwischengespeicherten Bytes werden dann in den Adresszähler geladen und das Byte auf welches der Adresszähler damit zeigt wird als der nächste Befehl interpretiert. Damit ist der indirekte Sprung vollzogen.
Wir werden für alle Befehle noch Beispiele erarbeiten. Die oben beschriebenen Befehle sehen in Assembler so aus:

```
20 LB HB JSR $XXXX    ;springe in Unterroutine
60       RTS          ;kehre aus Unterroutine zurück
4C LB HB JMP $XXXX    ;springe an Adresse
6C LB HB JMP ($XXXX) ;springe an die Adresse auf welche die
                       Bytes an Adresse $XXXX zeigen
```

Diese bisher besprochenen Sprungbefehle stellen die unbedingten Sprungbefehle. Der befohlene Sprung wird also unbedingt ausgeführt. Hiervon unterscheiden sich die bedingten Sprungbefehle. Bei diesen Befehlen wird ein Sprung nur unter bestimmten Bedingungen ausgeführt. Diese bedingten Sprungbefehle stellen die eigentliche Intelligenz des Prozessors dar. Denn ohne einen bestimmten Vorgang von veränderlichen Zuständen abhängig zu machen, ist ein flexibler Programmablauf kaum möglich.
Die bedingten Sprungbefehle, auch Branchbefehle, machen die Ausführung des Sprunges von dem Zustand (gesetzt / nicht gesetzt) eines bestimmten Bit (Flag) des Statusregister abhängig. Für vier Flags gibt es je einen Branchbefehl der auf das gesetzte Bit und einen Branchbefehl der auf das gelöschte Bit reagiert. Das ergibt 8 Branchbefehle die auf 4 verschiedene Flags reagieren. Die Bedeutung der einzelnen

Flags finden Sie im Anhang erklärt (65xx Flags). Sehen Sie dort jetzt einmal nach. Die Branchbefehle finden Sie bei den 65xx Befehlen (bedingte Sprünge) ebenfalls im Anhang. Sie sehen dort, daß die Branchbefehle nur eine Adressierung kennen: Die relative Adressierung. Diese Adressierung bedarf einer genaueren Erläuterung. Wir wollen uns die Vorgänge wieder mit Hilfe des Programmiermodells erklären. Im Speicher stehen folgende Befehle:

```
$5000 D0 02     BNE $5004
$5002 F0 FB     BEQ $4FFF
```

Die Befehlserkennung liest das Byte $D0 an Adresse $5000 und interpretiert es als Befehl. Zuerst wird der Befehl als Branchbefehl, der ausgeführt werden soll wenn das Z- Flag gelöscht ist, interpretiert. Es wird also im Statusregister geprüft ob das Z- Flag gelöscht ist oder nicht. Ist das Z-Flag gesetzt, so wird der Adresszähler um 2 erhöht und der nächste Befehl ausgeführt. Ist die Sprungbedingung (Z- Flag = 0) erfüllt, so wird der Adresszähler um 1 erhöht und der Operand des Branchbefehls gelesen. Bei diesem Operanden wird zunächst das 8. Bit getestet. Dieses Bit bestimmt ob der Sprung vorwärts oder rückwärts ausgeführt werden soll. Bei gesetztem 8. Bit des Operanden wird der Sprung rückwärts, bei gelöschtem 8. Bit vorwärts ausgeführt. Der Operand hat in obigem Beispiel den Wert 2 oder Binär 00000010. Das 8. Bit ist also nicht gesetzt, es soll ein Vorwärts- Sprung ausgeführt werden. In diesem Fall wird der Wert des Operanden zu dem momentanen Adresszähler- Inhalt addiert. Der Adresszähler zeigte zuletzt auf den Operanden, also auf die Adresse $5001. Das Ergebnis der Addition lautet somit $5001 + $02 = $5003. Dieser Wert wird in den Adresszähler geladen, der Adresszähler um 1 erhöht und das Programm an der Adresse $5004 fortgesetzt. Der Sprung ist somit ausgeführt.
Nehmen wir einmal an, das Z- Flag wäre gesetzt und die Sprungbedingung damit nicht erfüllt gewesen. In diesem Fall würde das Programm ab Adresse $5002 fortgesetzt. Das Befehlsbyte $F0 wird als Branchbefehl erkannt, die Sprungbedingung (Z- Flag = 1) ist erfüllt, der Sprung wird ausgeführt. Der Operand enthält den Wert $FB oder Binär 11111011. Das 8. Bit ist gesetzt, es wird ein Rückwärts-

Sprung erwartet. Der Operand wird dann negiert. Das bedeutet, daß alle gesetzten Bit gelöscht und alle gelöschten Bit gesetzt werden. Das ergibt in unserem Beispiel %00000100 - $04. Dieser Wert wird vom momentanen Adresszähler- Inhalt (ein Byte nach dem Operanden) subtrahiert: $5003 - $04 = $4FFF. Das Ergebnis wird in den Adresszähler geladen und das Programm an dieser Stelle fortgesetzt. Der Sprung ist damit ausgeführt.

Die relative Adressierung gibt also eine Adresse nicht absolut als 16 Bit Adresse an, sondern ist immer relativ zu der Adresse des Branchbefehls zu sehen. Die maximale Sprungweite beträgt vorwärts 127 Bytes und rückwärts 128 Bytes ab der Adresse des Befehlsbytes. Nur die Branchbefehle kennen die relative Adressierung. Welche Vorteile diese Adressierung bietet werden wir noch besprechen.

Um diese Befehle in einem Beispielprogramm anzuwenden wollen wir noch zwei 65xx Befehlsarten betrachten. Die Incrementier- und Decrementier- Befehle. Incrementieren heißt hier um 1 erhöhen, decrementieren um 1 vermindern. Sie finden diese Befehle unter den arithmetischen Befehlen mit den Symbolen: DEC - DEX - DEY - INC - INX - INY im Anhang. Wie Sie dort sehen, sind für den DEC und den INC Befehl mehrere Adressierungsarten möglich. Die Befehle DEX, DEY, INX und INY beziehen sich nur auf das X- oder Y- Register. Daher ist für diese Befehle nur die implizierte Adressierung möglich. Der Befehl DEX ohne Operanden bewirkt also einfach den momentanen Inhalt des X- Registers um 1 zu vermindern. Enthielt das X- Register zuvor den Wert $23, so enthält das X- Register nach dem Befehl DEX den Wert $22. Wichtig ist dabei noch zu wissen was geschieht, wenn das zu decrementierende Byte bereits den Wert $00 hat. Das Register oder die adressierte Speicherstelle enthält in diesem Fall nach der Decrementierung den Wert $FF. Wird dieses Byte wieder incrementiert, so ergibt sich der Wert $00.

Damit können wir endlich wieder zur Praxis übergehen und das Gelernte an einem Beispielprogramm ausprobieren. Das Programm soll eine sogenannte Verzögerungsschleife ergeben. Solche Verzögerungsschleifen werden sehr oft benötigt. Beispielsweise um eine Schrift für eine bestimmte Zeit auf dem Bildschirm erscheinen zu lassen oder um auf ein langsameres Peripherie- Gerät zu warten.

Hier das Assembler Listing:

```
$5000 A0 00     LDY # $00 ;Y- Register = $7F
$5002 20 0B 50  JSR $500B ;Sprung in Unterroutine
$5005 C8        INY       ;Y- Register + 1
$5006 30 09     BMI $5011 ;Sprung wenn N- Flag = 1
$5008 4C 02 50  JMP $5002 ;Sprung nach $5002
$500B A2 FF     LDX # $FF ;X- Register = $FF
$500D CA        DEX       ;X- Register - 1
$500E D0 FD     BNE $500D ;Sprung wenn Z- Flag = 0
$5010 60        RTS       ;Rückkehr aus Unterroutine
$5011 00        BRK       ;Rückkehr zum Monitor
```

Gehen wir einmal einen Programmablauf durch: In der ersten Zeile wird das Y- Register mit dem Wert $00 geladen. Die zweite Zeile bewirkt den Sprung in ein Unterprogramm ab Adresse $500B. Der erste Befehl des Unterprogrammes lädt das X- Register mit dem Wert $FF. Dieser Wert wird dann in der zweiten Unterprogrammzeile um eins vermindert. Die dritte Unterprogrammzeile prüft, ob das Ergebnis der Decrementierung = 0 war (siehe auch Anhang: 65xx Flags). Da das X- Register auf den Wert 255 ($FF) gesetzt wurde, ergab die Decrementierung den Wert 254 ($FE). Der Sprung wird also ausgeführt. Gesprungen wird an den Decrementier- Befehl an Adresse $500D. Das X- Register wird dort wieder decrementiert; auf 253 ($FD). Insgesamt werden so genau 255 Decrementierungen vorgenommen bis die Sprungbedingung des Branchbefehls an Adresse $500E nicht mehr erfüllt ist. Das Unterprogramm fährt dann mit dem Befehl an Adresse $5010 fort. Dieser Befehl veranlaßt die Rückkehr aus der Unterroutine an die Adresse $5005. Dort wird das Y- Register incrementiert. Das Y- Register wurde zu Beginn des Programmes auf 0 gesetzt und währenddessen nicht verändert. Nach der Incrementierung enthält das Y- Register also den Wert $01. Der folgende Branchbefehl führt den Sprung aus, wenn das 8. Bit des Y- Register gesetzt ist. Diese Bedingung ist nicht erfüllt, der Sprung an Adresse $5011 wird somit nicht ausgeführt und der nächste Befehl abgearbeitet. Dieser unbedingte Sprungbefehl an Adresse $5008 erzwingt den Sprung an Adresse $5002. Dort wird wieder das Unterprogramm

aufgerufen und nach der Rückkehr aus dem Unterprogramm das Y-Register auf den Wert $02 incrementiert. Die Sprungbedingung des Branchbefehls an Adresse $5006 ist damit nicht erfüllt, an Adresse $5008 wird erneut zurückgesprungen. Das 8. Bit des Y- Registers wird erst bei Erreichen des Wertes %10000000 - $80 gesetzt werden. Der beschriebene Ablauf wird also genau 128 mal ausgeführt, bevor der Branchbefehl - BMI $5011 - ausgeführt wird. Insgesamt wurde damit 128 mal bis 255 gezählt. An Adresse $5011 wird dann mit dem BRK Befehl zum Monitor zurückgekehrt.
Das Programm als Hexdump:

```
$5000 A0 00 20 0B 50 C8 30 09
$5008 4C 02 50 A2 FF CA D0 FD
$5010 60 00
```

Geben Sie das Programm in Hexform mit Hilfe Ihres Monitors ein. Bevor Sie das Programm starten, sollten Sie die Bytes noch einmal genau mit den oben abgebildeten vergleichen. Vergleichen Sie auch mit Hilfe des Disassembler das oben abgebildete Assembler - Listing. Wenn Sie sicher sind, daß alles übereinstimmt, speichern Sie das Programm unter dem Namen "ZWEI" auf Diskette ab. Gewöhnen Sie sich an ein Maschinenprogramm nie zu starten, bevor Sie die aktuelle Version auf Diskette gespeichert haben. Ein Maschinenprogramm kann bei dem kleinsten Fehler abstürzen. Unter Umständen bleibt in einem solchen Fall nur die Möglichkeit den Computer abzuschalten. Dann ist das Programm natürlich komplett gelöscht. Auch ein Resettaster kann eine Zerstörung des Programmes nicht immer verhindern. Also starten Sie das Programm nachdem Sie es abgespeichert haben. Sie können sich darauf verlassen, daß bei einem auftretenden Fehler ein Tippfehler Ihrerseits vorliegt.
Das Programm benötigt zur Ausführung eine sehr kurze Zeit, obwohl einmal bis 32640 gezählt werden mußte. Um die Aufgabe zu lösen, eine Schrift auf dem Bildschirm lesen zu lassen, reicht diese kurze Verzögerung sicher nicht aus. Sie bekommen hier einen ersten Vorgeschmack von der im Vergleich zu BASIC wirklich verblüffenden Arbeitsgeschwindigkeit der Maschinensprache. Wenn Sie wollen können Sie die Laufzeit eines BASIC Programmes mit denselben Aufgaben ja einmal

ausprobieren.
Damit kommen wir nun zu dem Ausgangsproblem. Wenn Sie mit einer Transfer- Funktion das obige Programm beispielsweise an die Adresse $4000 - $4011 übertragen und den ursprünglichen Bereich $5000 - $5011 dann anderweitig benutzen, also das dortige Programm zerstören, werden Sie eine böse Überaschung erleben, wenn Sie das transferierte Programm ab Adresse $4000 dann starten wollen.
Das transferierte Programm steht ab Adresse $4000 wie unten abgebildet:

```
$4000 A0 00     LDY # $00 ;Y- Register = $7F
$4002 20 0B 50  JSR $500B ;Sprung in Unterroutine
$4005 C8        INY       ;Y- Register - 1
$4006 30 09     BMI $4011 ;Sprung wenn N- Flag = 1
$4008 4C 02 50  JMP $5002 ;Sprung nach $5002
$400B A2 FF     LDX # $FF ;X- Register = $FF
$400D CA        DEX       ;X- Register - 1
$400E D0 FD     BNE $400D ;Sprung wenn Z- Flag = 0
$4010 60        RTS       ;Rückkehr aus Unterroutine
$4011 00        BRK       ;Rückkehr zum Monitor
```

Wenn Sie sich die Befehle an Adresse $4002 und $4008 einmal ansehen, stellen Sie fest, daß die Sprungadressen noch in den alten Bereich zeigen. Da das Programm dort aber zerstört ist, würde ein Programmstart ab Adresse $4000 sicher nicht zu dem gewünschten Ergebnis führen. Mit der Transfer- Funktion werden die Bytes unverändert in den Zielbereich übertragen. Absolute Adressen behalten ihre alten Werte. Auch die Bytes der Branchbefehle sind nicht verändert worden. Dennoch zeigen die Mnemonics der Branchbefehle die richtigen Zieladressen. Das liegt wie Sie sicher bereits vermuten an der relativen Adressierung. Die relative Entfernung zu den Sprungzielen hat sich durch den Transfer ja nicht verändert. Es wird hier auch ein weiterer Vorteil eines Assemblers offensichtlich: Die zuvor beschriebene Berechnung des Operanden zur Ermittlung der Zieladresse erledigt der Assembler. Die Zieladresse kann also direkt als 16 Bit Adresse eingegeben werden oder wird als solche vom Disassembler ausgegeben. Sollten Sie doch einmal gezwungen sein den Operanden oder die Zieladresse von Hand zu errechnen, so können Sie das wie zuvor beschrieben

tun.
Soll also ein Maschinenprogramm transferiert werden, so ist zu beachten, daß alle absoluten Adressen die innerhalb des ursprünglichen Bereiches liegen, im transferierten Programm korrigiert werden müssen. Einige Monitore bieten die Möglichkeit die absoluten Adressen eines transferierten Programmes automatisch zu korrigieren.

PROFIMON
Das Befehlsformat für die Transfer- Funktion ist wie folgt vorgeschrieben:

```
T XXXX YYYY ZZZZ
```

XXXX und YYYY sind die Anfangs- und Endadresse des zu übertragenden Bereiches. ZZZZ ist die Zieladresse. Die automatische Korrektur der absoluten Adressen ist nicht möglich.
Probieren Sie diese Funktion praktisch aus.

T.EX.AS.
Das Befehlsformat für die Transfer- Funktion ist wie folgt vorgeschrieben:

```
.TR XXXX YYYY ZZZZ
```

XXXX und YYYY sind die Anfangs- und Endadresse des zu übertragenden Bereiches. ZZZZ ist die Zieladresse. Die automatische Korrektur der absoluten Adressen mit dem .CH Befehl möglich. Um diesen Befehl einzusetzen, ist zunächst ein entsprechender Workspace festzulegen. Das erfolgt mit den Befehlen .UF $XXXX $YYYY (Quellbereich) und .UT $ZZZZ (Zielbereich). Sind diese Bereiche so festgelegt, veranlaßt der Befehl .CH $XXXX $YYYY den Monitor die absoluten Adressen des Zielbereiches zu korrigieren. XXXX und YYYY bezeichnen für den .CH Befehl die Anfangs- und Endadresse des Zielbereiches. Wurden die Bereiche vor dem Transferbefehl (.TR) mit den Befehlen .UF und .UT festgelegt, so kann der Transferbefehl ohne Parameter gegeben werden. Wir wollen das an dem Beispielprogramm einmal ausprobieren. Zuerst die Bereiche festlegen:

```
.UF $5000 $5011
.UT $4000
```

Die Bereiche werden nach jeder Änderung ausgegeben. Sie können so überprüfen, ob die Eingaben richtig erfolgten. Zum transferieren können Sie jetzt einfach den Befehl .TR geben. Mit dem Befehl .DL $4000 sehen Sie das transferierte, unkorrigierte Programm. Zur Korrektur ist jetzt noch der Befehl:

```
.CH $4000 $4011
```

zu geben. Sehen Sie sich jetzt das korrigierte Programm mit .DL $4000 an.
Zu beachten bleibt noch, daß alle absoluten Adressen korrigiert werden. Also auch die der Lade- und Speicherbefehle soweit die Adressen innerhalb des Quellbereiches liegen. Branchbefehle die durch die Korrektur eine unmögliche Sprungweite erhalten würden (größer 127 oder 128), werden durch eine Fehlermeldung kenntlich gemacht. Probieren Sie das einmal aus.

Fragen zu 1.7.

1.) Welche Adressierungen kennt der JSR Befehl ?

2.) Welche Sprungbefehle müssen nach dem Transfer eines Maschinenprogrammes korrigiert werden ?

3.) Wie werden die Zieladressen der Branchbefehle mit Hilfe eines Assemblers angegeben - bzw. mit Hilfe eines Disassemblers ausgegeben ?

4.) Welche 65xx Befehle kennen die relative Adressierung und welche maximale Sprungweite ist mit dieser Adressierung möglich ?

5.) Welche Flags werden durch die Incrementier- und Decrementier- Befehle beeinflußt (Anhang 65xx Befehle) ? Welche Branchbefehle können wie angewendet werden ?

6.) Wie wäre das Beispielprogramm "ZWEI" zu ändern, damit 255 mal bis 255 gezählt wird ? Probieren Sie Ihre Lösung aus.

Antworten zu 1.7.

zu 1.) Der JSR Befehl kennt nur die absolute Adressierung.

zu 2.) Alle JSR und JMP Befehle mit absoluter Adressierung, sofern die Zieladressen im Quellbereich liegen. Der JMP ($XXXX) Befehl (indirekte Adressierung) muß in dem Fall korrigiert werden, daß die Adresse $XXXX im Quellbereich liegt und die Zieladresse nicht mehr an dieser Adresse vorgefunden werden soll.

zu 3.) Die Zieladressen der Branchbefehle werden mit Hilfe eines Assemblers als 16 Bit Adressen angegeben, obwohl der Branchbefehl im Speicher nur einen Operanden mit der Information der relativen Sprungweite und der Sprungrichtung hat. Die erforderliche Umrechnung nimmt der Assembler automatisch vor.

zu 4.) Die relative Adressierung ist nur mit den Branchbefehlen möglich. Die maximale Sprungweite beträgt vorwärts 127 Bytes und rückwärts 128 Bytes.

zu 5.) Das N- und das Z- Flag werden durch die Incrementier- und Decrementier- Befehle beeinflußt. Damit kann getestet werden ob das in- oder decrementierte Byte Null ist oder nicht (Z- Flag, BEQ / BNE) und ob das 8. Bit gesetzt ist oder nicht (N- Flag, BPL / BMI).

zu 6.) Folgende Befehle sind zu ändern:

```
$5000 AO FF     LDY #FF
$5005 88        DEY
$5006 FO 09     BEQ $5011
```

Es gibt durchaus noch andere Möglichkeiten. Wenn Sie eine funktionierende andere Möglichkeit gefunden haben, so ist das genauso gut. Das Beispielprogramm stellt ohnehin keine optimale Lösung des gestellten Problems dar. Es wurde Wert auf die Demonstration aller Befehle gelegt.

1.8. Hunt (Zeichenfolge suchen)

Auch diese Monitorfunktion erleichtert die Arbeit an einem Maschinenprogramm ganz erheblich. Wie sich bereits aus der Überschrift vermuten läßt, dient diese Funktion zum Auffinden einer bestimmten Bytefolge in einem vorher festzulegenden Bereich. Dieser Bereich ist wieder mit Anfangs- und Endadresse anzugeben. In den seltensten Fällen ist es erforderlich, den gesamten Speicherbereich zu durchsuchen. Daher ist der Suchbereich anzugeben.
Wir wollen an unserem Beispielprogramm "ZWEI" einmal diese Funktion ausprobieren.

PROFIMON
Dieser Monitor bietet zwei Möglichkeiten zur Festlegung der gesuchten Bytefolge: Die gesuchten Bytes können einmal als Hexzahlen angegeben werden oder wenn ein Text durchsucht werden soll, können die gesuchten Bytes im Klartext angegeben werden. Die folgenden beiden Befehlsformate sind demnach möglich:

```
H XXXX YYYY BB BB BB bis zu 29 Bytes

H XXXX YYYY "Text mit bis zu 29 Zeichen"
```

XXXX und YYYY geben die Anfangs- und Endadresse des zu durchsuchenden Bereiches an. Geben Sie einmal folgenden Befehl:

```
H 5000 5011 4C 02 50
```

Das Ergebnis muß die Ausgabe der Adresse $5008 sein. Die gesuchte Bytefolge $4C $02 $50 steht ab der Adresse $5008.
Zur Auffindung eines Textes wollen wir das Betriebssystem nach dem Wort "READY" durchsuchen:

```
H A000 AFFF "READY"
```

Das Ergebnis muß die Ausgabe der Adresse $A378 sein. Die gesuchte Bytefolge "READY" steht ab der Adresse $A378. Prüfen

Sie das mit dem Befehl - M A378 - einmal nach.

T.EX.AS.
Dieser Monitor bietet die Möglichkeit nach 65xx Befehlen direkt zu suchen. Der zu suchende Befehl ist also nicht als Bytefolge sondern direkt in Assembler einschließlich Operanden einzugeben. Auch hier ist zuvor wieder ein Workspace mit dem Befehl .WO festzulegen. Der Befehl .FI legt dann den zu suchenden Befehl fest. Bei der mir vorliegenden Version des Monitors müssen die Operanden in Dezimalform gegeben werden. Bei den folgenden Beispielen ist das zu berücksichtigen. Geben Sie folgende Befehle:

```
.WO $5000 $5011
.FI LDY #0
```

Das Ergebnis muß sein: $5000 LDY #$00. Mit dem Befehl .RA können Sie für den Operanden einen Bereich festlegen. Ein Beispiel:

```
.RA $00 $FF
.FI LDX #*
```

Das Ergebnis muß sein: $5008 LDX #$FF. Das * Zeichen ist ein Joker und steht für den durch .RA festgelegten Bereich. Ändern Sie den Befehl LDY an Adresse $5000 in LDX und geben noch einmal den Befehl .FI LDX #*. Das Ergebnis muß sein: LDX #$00 / LDX #$FF. Es werden also alle LDX Befehle mit unmittelbarer Adressierung und einem Operanden im Bereich $00 - $FF gefunden.
Der Monitor kennt noch zwei weitere Joker: Das ! Zeichen kann anstelle beliebiger Zeichen des Befehls stehen. .FI LD! #* findet alle Ladebefehle mit einem Operanden im .RA Bereich und unmittelbarer Adressierung. Das ? Zeichen kann anstelle der Adressierungs- Kennzeichen stehen. Mit .FI JMP ?*? werden also alle JMP Befehle mit indirekter und absoluter Adressierung gefunden deren Operanden im .RA Bereich liegen. Die Joker können beliebig kombiniert werden.
Mit diesen Möglichkeiten dürften sich alle Suchprobleme nach einem bestimmten Befehl lösen lassen. Der Befehl .FB sucht nach einzelnen Bytes. Sie beherschen diese vielfältigen

Möglichkeiten am schnellsten, wenn Sie eine Weile damit experimentieren.

Fragen zu 1.8.

1.) Was ist zu beachten, wenn Sie den Befehl JSR über die Suche nach dem Byte $20 suchen ?

2.) Nach welcher Bytefolge ist zu suchen, wenn Sie den Text "READY" finden wollen und der Monitor die Klartexteingabe nicht erlaubt (siehe ASCII Code Tabelle im Anhang) ?

Antworten zu 1.8.

zu 1.) Das Befehlsbyte $20 für den Befehl JSR kann auch als Operand anderer Befehle vorkommen. z.B. LDA $5020. Die gefundenen Bytes sind also daraufhin zu überprüfen, ob sie nicht Operanden eines anderen Befehls sind.

zu 2.) Für Großbuchstaben ist die Bytefolge: $D2 $C5 $C1 $C4 $D9 und für Kleinbuchstaben die Bytefolge: $52 $45 $41 $44 $59 zu suchen. Zu beachten ist, daß der C 64 im Großschreibmodus alle Buchstaben groß schreibt. In diesem Fall sind die Codes der Kleinbuchstaben maßgebend, da die Codes der Großbuchstaben in diesem Modus als Grafikzeichen wiedergegeben werden.

1.9. Fill (Speicherbereich mit einem Zeichen füllen)

Im Zusammenhang mit dieser Monitorfunktion wollen wir auch die indizierten Adressierungen vorstellen.
Zunächst die absolut X- oder Y- indizierte Adressierung. Im Speicher stehe folgender Befehl:

```
$5000 BD 00 40  LDA $4000,X ;lade den ACCU ab-
                            solut X-indiziert
```

Das X- Register enthalte den Wert $10. Sehen wir uns am Programmiermodell an, wie die Befehlserkennung auf diesen Befehl reagiert: Der momentane Adresszähler- Inhalt wird zwischengespeichert. Zu dem Wert der Operanden wird der Inhalt des X- Register addiert ($4000 + $10 = $4010) und das Ergebnis in den Adresszähler geladen. Das Byte an dieser Adresse ($4010) wird dann in den ACCU geladen, der zwischengespeicherte Adresszähler wieder zurückgeladen und auf den nächsten Befehl gesetzt. Der Befehl ist damit abgearbeitet. Das gilt auf dieselbe Weise für die Ladebefehle LDX, LDY, für den Speicherbefehl STA und für einige weitere Befehle (siehe Anhang 65xx Befehle). Die nullseitige absolut indizierte Adressierung unterscheidet sich lediglich dadurch von der oben beschriebenen Adressierung, daß nur ein Operand erforderlich ist. Das Highbyte wird immer als Null angesehen. Um ein Beispielprogramm für diese Adressierung vorzustellen befassen wir uns noch mit einer weiteren Befehlsart der 65xx Prozessoren: Den Vergleichsbefehlen - CMP, CPX, CPY, BIT. Diese Befehle dienen dem Vergleich zweier Bytes ohne diese Bytes zu verändern. Sehen wir uns folgenden Befehl genauer an:

```
$5000 A9 AA      CMP #$AA ;subtrahiere $AA vom
                          ACCU ohne den ACCU
                          zu ändern
```

Durch die unmittelbare Adressierung bezieht sich der Befehl auf den Operanden. Der Operand wird also vom Accumulator-Inhalt subtrahiert. Das Ergebnis wird nicht weiter verwendet. Es werden nur die Flags beeinflußt.

Folgende Möglichkeiten können durch die Flags erkannt werden:

ACCU kleiner Operand - C- Flag = 0
ACCU größer/gleich Operand - C- Flag = 1
ACCU gleich Operand - Z- Flag = 1
ACCU ungleich Operand - Z- Flag = 0

Durch Auswertung des N- Flags ergeben sich weitere Möglichkeiten, die aber von den Inhalten des Accumulator und des Operanden abhängig sind. Alle diese Möglichkeiten treffen natürlich für alle Adressierungs- Arten des CMP Befehls zu. Die Befehle CPX und CPY arbeiten in derselben Weise, nur steht an der Stelle des ACCU das X- oder Y- Register.
Der BIT Befehl beeinflußt ebenfalls nur die Flags. Anstelle einer Subtraktion wird eine UND Verknüpfung zwischen dem adressierten Byte und dem ACCU durchgeführt.
Die Vergleichsbefehle stellen eine bedeutende Ergänzung zu den Branchbefehlen dar. Es ist daher wichtig, daß Sie sich deren Funktion genau verdeutlichen.
Damit kommen wir wieder zur Praxis, zu einem weiteren Beispielprogramm. Mit Hilfe dieses Beispielprogramm soll ein festgelegter Speicherbereich an einen anderen Bereich übertragen werden. Das Ende des zu übertragenden Bereiches soll an dem Byte $FF erkannt werden. Dadurch sind nur Start- und Zieladresse festzulegen. Das Programm soll wieder ab Adresse $5000, der zu übertragende Bereich ab Adresse $5100 und die Zieladresse ab $4000 liegen. Die Technik das Ende eines Bereiches, beispielsweise einer Tabelle, an einem bestimmten Byte zu erkennen wird häufig angewendet. Das Erkennungsbyte muß dabei selbstverständlich einen Wert haben, den kein Byte des Bereiches annehmen kann. Unser zu übertragender Bereich soll Text in Form von ASCII - Code enthalten. Das Byte $FF ist im ASCII - Code nicht enthalten und eignet sich somit sehr gut als Erkennungsbyte.

Sie sehen nachfolgend das Assemblerlisting unseres dritten Beispielprogramms:

```
$5000 A2 FF     LDX #$FF     ;X = 0
$5002 E8        INX          ;X + 1
$5003 30 0A     BMI $500F    ;X- Reg.= 128 dann Sprung
$5005 BD 00 51  LDA $5100,X  ;Byte laden
$5008 9D 00 40  STA $4000,X  ;Byte übertragen
$500B C9 FF     CMP #$FF     ;Byte = $FF ?
$500D D0 F3     BNE $5002    ;Nein- zurück
$500F 00        BRK          ;Rückkehr Monitor
```

Ab Adresse $5100 sollen folgende Bytes stehen:

$5100 54 45 53 54 FF (das ergibt "TEST")

Gehen wir wieder einen Programmlauf durch: Das X- Register erhält den Wert $FF. Der Befehl INX incrementiert das X-Register auf den Wert $00. An Adresse $5003 wird geprüft ob das X- Register den Wert 128 ($80) erreicht hat. Ist das der Fall, so wird die Übertragung abgebrochen. Dies dient zur Sicherheit falls das Erkennungsbyte $FF nicht im zu übertragenden Bereich steht. Es können somit bis zu 128 Byte übertragen werden. Das X- Register hatte den Wert $00. Die Sprungbedingung ist somit nicht erfüllt, der Befehl LDA $5100,X lädt das Byte an Adresse $5100 + $00 = $5100 in den ACCU. Das Datenbyte $54 wird mit dem folgenden Befehl dann an die Adresse $4000 + $00 = $4000 geschrieben. Der Befehl CMP #$FF subtrahiert von dem Datenbyte den Wert $FF. $54 - $FF = $55. Da ein Überlauf stattgefunden hat ist das C- Flag = 0. Das Ergebnis ist nicht Null also ist das Z- Flag = 0. Das 8. Bit des Ergebnises ist gleich dem N- Flag = 0 (Vergleich auch Anhang 65xx Flags). Der folgende Branchbefehl wird ausgeführt da die Sprungbedingung, Z- Flag = 0, erfüllt ist. An Adresse $5002 wird das X- Register dann auf den Wert $01 incrementiert. Die Sprungbedingung für den folgenden Branchbefehl ist somit noch nicht erfüllt. Der Befehl an Adresse $5005 lädt das Datenbyte aus Adresse $5100 + $01 = 5101 in den ACCU. Dieses Byte ($45) wird dann mit dem nächsten Befehl an Adresse $4000 + $01 = $4001 übertragen. Der Vergleich an Adresse $500B fällt wieder negativ aus, so

daß der Vorgang wiederholt wird bis das Byte $FF übertragen wurde. Nach der Übertragung dieses Bytes wird der Branchbefehl an Adresse $500D nicht mehr ausgeführt. Der folgende Befehl veranlaßt dann die Rückkehr zum Monitor. Der Branchbefehl an Adresse $5003 wurde nicht ausgeführt, da das X- Register nur bis zu dem Wert $04 incrementiert wurde. Nachfolgend steht der Hexdump des Programmes:

```
$5000 A2 FF E8 30 0A BD 00 51
$5008 9D 00 40 C9 FF D0 F3 00
```

Tippen Sie das Programm so ein und kontrollieren die Eingabe wieder genau auf Übereinstimmung. Disassemblieren Sie dann das Programm und vergleichen noch einmal mit dem vorstehenden Assemblerlisting. Wenn Sie sicher sind, daß alles übereinstimmt, speichern Sie das Programm unter dem Namen "DREI" ab. Tragen Sie dann noch die Bytes ab Adresse $5100 ein und starten dann das Programm. Ist alles richtig verlaufen, so müssen die Bytes einschließlich dem Erkennungsbyte nach Adresse $4000 - $4004 übertragen worden sein.
Das obige Programm weist bei genauerer Betrachtung zwei Nachteile auf: Zum ersten sind die Start- und Zieladresse durch das Programm fest vorgegeben. Um diese Adressen zu ändern müssen die Operanden der Lade- und Speicherbefehle geändert werden. Wenn diese Routine in einem Nur-Lese-Speicher steht, ist eine einfache Änderung der Operanden aber nicht möglich. Zum zweiten darf der zu übertragende Bereich nicht länger als 128 Bytes sein. Um diese Probleme zu lösen beschäftigen wir uns mit einer weiteren indizierten Adressierung: Der indirekt Y- indizierten Adressierung. Auch wenn nach diesem Titel ein Seufzer zu hören ist, wollen wir uns nicht davon abschrecken lassen diese Adressierung ebenfalls mit Hilfe des Programmiermodells zu durchleuchten. Im Speicher stehe folgender Befehl:

```
$5000 B1 61     LDA ($61),Y ;Lade den ACCU in-
                            direkt Y- indiziert
```

Das Y- Register habe den Wert $10, an den Adressen $0061 und $0062 stehen die Bytes $00 und $40. Nachdem der Befehl

erkannt wurde speichert die Befehlserkennung den momentanen Adresszähler- Inhalt zwischen, lädt den Operanden des Befehls ($61) in das Lowbyte und $00 in das Highbyte des Adresszählers. Dieser zeigt damit auf die Adresse $0061. Die Bytes an Adresse $0061 und $0062 bilden eine 16 Bit Adresse zu der dann der Wert des Y- Register addiert wird. Das ergibt: $4000 + $10 = $4010. Diese Adresse ($4010) wird in den Adresszähler geladen und das Byte an dieser Adresse in den ACCU geladen. Nachdem der Adresszähler den ursprünglichen Wert aus dem Zwischenspeicher zurückerhalten hat, kann der nächste Befehl abgearbeitet werden.
Die Adresse des Datenbytes wird also vom Inhalt des Y-Register und von der 16 Bit Adresse in der Zeropage (Nullseite), die durch den Operanden gegeben ist, bestimmt. Das hat den Vorteil, daß alle adressbestimmenden Faktoren auch dann änderbar sind, wenn der Befehl im ROM steht.
Damit können wir das Beispielprogramm dahingehend verbessern, daß die Start- und Zieladressen beliebig im RAM (Schreib-Lesespeicher) in der Zeropage vorgegeben werden können und daß der Bereich eine beliebige Länge haben darf.
Sie sehen nachfolgend das entsprechende Assemblerlisting:

```
$5000 A9 00     LDA #$00    ;Startadresse
$5002 A0 51     LDY #$51    ;laden
$5004 85 61     STA $61     ;In Zeiger
$5006 84 62     STY $62     ;bringen
$5008 A9 00     LDA #$00    ;Zieladresse
$500A A0 40     LDY #$40    ;laden
$500C 20 10 50  JSR $5010   ;Programm aufrufen
$500F 00        BRK         ;Rückkehr Monitor
```

```
$5010 85 63     STA $63      ;Zieladresse in
$5012 84 64     STY $64      ;Zeiger bringen
$5014 A0 00     LDY #$00     ;Y = 0
$5016 B1 61     LDA ($61),Y ;Byte lesen
$5018 91 63     STA ($63),Y ;Byte übertragen
$501A C8        INY          ;Y + 1
$501B D0 04     BNE $5021    ;Seite übertragen ?
$501D E6 62     INC $62      ;Zeiger um eine
$501F E6 64     INC $64      ;Seite erhöhen
$5021 C9 FF     CMP #$FF     ;Bereich Ende ?
$5023 D0 F1     BNE $5016    ;Nein- weiter
$5025 60        RTS          ;Rücksprung
```

Verfolgen wir wieder einen Programmablauf: Das eigentliche Programm ist als Unterroutine aufgebaut und beginnt an Adresse $5010. Die Befehle von Adresse $5000 bis $500A setzen den Anfangszeiger an Adresse $0061 $0062 und laden die Zieladresse in den ACCU und das Y- Register. Die Zieladresse wird der Unterroutine so übergeben. Diese Technik der Parameter- Übergabe durch die Prozessor- Register wird häufig eingesetzt. Sie hilft Speicherplatz sparen und vereinfacht die Handhabung einer Unterroutine. An Adresse $500C wird die Routine dann aufgerufen.

Die ersten beiden Befehle bringen die Zieladresse in den Zeiger an Adresse $0063 $0064. Dann wird das Y- Register auf Null gesetzt und durch den folgenden Befehl der ACCU mit dem ersten Datenbyte geladen. Die Adresse des Datenbytes ergibt sich aus der 16 Bit Adresse des Zeiger an Adresse $0061 $0062 plus dem Wert des Y- Register. Also $5100 + $00 = $5100. Der folgende Befehl überträgt das Byte an die Adresse: Zeigerinhalt an Adresse $0063 $0064 plus Y- Register - $4000 + $00 = $4000.

An Adresse $501A wird das Y- Register um 1 erhöht. Der folgende Befehl prüft, ob das Y- Register von $FF auf $00 incrementiert wurde. Ist das der Fall, so wird es erforderlich die Highbytes der Zeiger um 1 zu erhöhen, da ein Überlauf der Lowbytes erfolgt ist. Das wollen wir kurz genauer betrachten. Enthielt das Y- Register vor der Incrementierung den Wert $FF, so wurde das Datenbyte aus Adresse $51FF gelesen und an Adresse $40FF übertragen. Nach der Incrementierung enthält das Y- Register den Wert $00. Ein

Datenbyte würde damit unrichtig aus der Adresse $5100 gelesen und an die Adresse $4000 übertragen. Die richtigen Adressen wären aber $5200 und $5100. Durch ein Incrementieren der Highbytes der beiden Zeiger werden diese auf die richtigen Werte gesetzt. Verdeutlichen Sie sich diesen Zusammenhang. Wir werden bei den arithmetischen Befehlen darauf noch zu sprechen kommen.
Im ersten Durchlauf wurde das Y- Register auf den Wert $01 incrementiert. Die Sprungbedingung für den Branchbefehl an Adresse $501B ist somit erfüllt, der Sprung wird ausgeführt. An Adresse $5021 wird dann geprüft, ob das Erkennungsbyte gefunden wurde. Ist das nicht der Fall, so bewirkt der Branchbefehl an Adresse $5023 die Wiederholung des Vorganges, bis das Erkennungsbyte gefunden wurde. Durch Auffinden dieses Bytes wird der Branchbefehl an Adresse $5023 nicht ausgeführt und mit dem folgenden Befehl die Rückkehr aus der Unterroutine bewirkt. Der BRK Befehl an Adresse $500F veranlaßt dann die Rückkehr zum Monitor.
Nachfolgend finden Sie den Hexdump des Programmes:

```
$5000 A9 00 A0 51 85 61 84 62
$5008 A9 00 A0 40 20 10 50 00
$5010 85 63 84 64 A0 00 B1 61
$5018 91 63 C8 D0 04 E6 62 E6
$5020 64 C9 FF D0 F1 60
```

Geben Sie das Programm in Ihren Computer ein und kontrollieren die Eingabe genau auf Unterschiede. Vergleichen Sie dann wieder das Assemblerlisting. Sind keine Fehler vorhanden, so speichern Sie das Programm auf Diskette unter dem Namen "VIER" ab.
Bevor wir das Programm starten, muß der zu übertragende Bereich gekennzeichnet werden. Da wir mehr als 4 Bytes übertragen wollen und nicht von Hand einen größeren Bereich kontrollieren wollen, lassen wir den Monitor arbeiten. Dabei kommt uns die Funktion "Fill" sehr zur Hilfe. Es ist notwendig in einem größeren Speicherbereich dafür zu sorgen, daß das Erkennungsbyte $FF dort nicht vorkommt. Wir erreichen das, indem wir den Bereich beispielsweise mit dem Byte $00 füllen.

Dazu ist folgender Befehl notwendig:

```
PROFIMON: F 5100 5400 00
T.EX.AS : .NE $5100 $5400 $00
```

Setzen Sie dann noch das Erkennungsbyte $FF an die Adresse $5400. Vergessen Sie das nicht, das Programm könnte sonst abstürzen. Überprüfen Sie immer die richtige Ausführung der Monitor- Befehle. Es ist immer möglich, daß Sie einen unbemekten Eingabefehler gemacht haben. Sie sollten sich vor dem Programmstart davon überzeugen, daß der Zielbereich ($4000 - $4300) nicht bereits mit $00 Bytes gefüllt ist. In diesem Fall können Sie den Bereich mit Hilfe der Fill-Funktion mit einem anderen Byte überschreiben. Starten Sie jetzt das Programm. Nach der Rückkehr zum Monitor muß der Bereich $4000 - $4300 mit $00 Bytes aufgefüllt sein.

Bevor wir an einem weiteren Beispiel- Programm die letzte indizierte Adressierung besprechen können, sind noch einige 65xx Befehle zu erklären.
Beginnen wir mit den einfachsten, den Register Transfer Befehlen. Diese Befehle dienen dazu, sehr schnell den Inhalt eines Prozessorregisters in ein anderes Register zu bringen. Es wird bereits mit dem Befehlsbyte die Adressierung angegeben. Daher benötigen diese Befehle keinen Operanden und sind somit auch sehr kurz. Welcher Befehl welchen Transfer veranlaßt, entnehmen Sie bitte dem Anhang. Besonders zu beachten sind die Befehle TXS und TSX. Mit diesen Befehlen können Sie durch verändern des Stapelzeigers in die Verwaltung des Stapelspeichers eingreifen. Das ist beispielsweise unmittelbar nach dem Einschalten erforderlich. Der Stapelzeiger muß dann auf den Anfang ($FF) gesetzt werden.
Die ALU der 65xx Prozessoren kann drei logische Verknüpfungen vornehmen. AND (UND), OR (ODER) und EOR (Exclusiv ODER). Bei diesen Verknüpfungen werden die Bytes nicht als Binärzahlen sondern die Bits einzeln betrachtet. Dabei werden die Bits der beiden zu verknüpfenden Bytes einzeln bearbeitet.

Anhand einer sogenannten Wahrheitstabelle lassen sich die Verknüpfungsvorschriften am einfachsten darstellen:

B1	B2	=	AND	OR	EOR
0	0	=	0	0	0
0	1	=	0	1	1
1	0	=	0	1	1
1	1	=	1	1	0

Sie sehen oben: das Ergebnis einer AND Verknüpfung ist nur dann = 1, wenn beide verknüpften Bits = 1 sind. Das Ergebnis einer OR Verknüpfung ist dann = 1, wenn mindestens eines der verknüpften Bits = 1 ist. Das Ergebnis einer EOR Verknüpfung ist dann = 1, wenn nur eines der verknüpften Bits = 1 ist.
Diese Definitionen sollten Sie dem Inhalt nach auswendig kennen. Stellen Sie zur Übung einmal eine Wahrheitstabelle für jede Funktion einzeln auf.
Sehen wir uns für diese drei Verknüpfungen einmal an, zu welchem Ergebnis die ALU bei der Verknüpfung der Bytes $AB und $27 kommt:

	%10101011	$AB
	%00100111	$27
AND =	%00100011	$23
OR =	%10101111	$AF
EOR =	%10001100	$8C

Wie Sie am Programmiermodell sehen, wird die Verknüpfung immer zwischen dem adressierten Byte und dem ACCU vollzogen. Das Ergebnis steht immer im ACCU. Die entsprechenden Befehle und ihre Adressierung finden Sie im Anhang bei den 65xx Befehlen. Die Anwendung der Logischen Befehle, werden wir an einigen Beispielen in diesem Buch trainieren.
Nun Sind noch die Verschiebebefehle zu erklären, bevor wir wieder alles in die Praxis umsetzen: diese Befehle ermöglichen es, die Bits eines Bytes um ein Bit nach links oder rechts zu verschieben. Das linke oder rechte Bit, welches durch die Verschiebung aus dem Byte herausgeschoben wird, wird in das Carry- Flag geschoben. Nach einer Rechtsverschiebung enthält das Carry- Flag also das 1. und

nach einer Linksverschiebung das 8. Bit des verschobenen Bytes. Sie finden das im Anhang bei den Verschiebebefehlen durch eine Zeichnung verdeutlicht. Wie Sie dort sehen, kann das nachgeschobene Bit entweder als 0 oder als der Inhalt des Carry- Flags bestimmt werden.
Eine der vielfältigen Anwendungen dieser Befehle wollen wir jetzt benutzen: ein Byte mit beliebigem Wert um ein Byte nach links zu verschieben, ergibt eine Multiplikation mit 2. Ein Byte mit beliebigem Wert um ein Byte nach rechts verschoben, ergibt eine Division durch zwei. Sehen wir uns das am Beispiel an:

```
                %00000110    $06
Linksversch.  = %00001100    $0C
Rechtsversch. = %00000011    $03
```

Das Ergebnis ist natürlich nur dann korrekt, wenn durch die Verschiebung kein Byte nach links oder rechts herausgeschoben wurde. Eine weitere Verschiebung ergibt eine Multiplikation / Division mit 4, 8, 16 usw.
Die Verschiebebefehle benutzen als einzige die Accumulatoradressierung. Diese Adressierung besagt einfach, daß der Accumulator- Inhalt durch den Verschiebebefehl entsprechend verschoben werden soll. Beispiel:

```
ASL A
```

verschiebt den Accumulator- Inhalt um ein Bit nach links, wobei eine Null nachgeschoben wird.

Jetzt bleibt noch die X- indizierte indirekte Adressierung zu behandeln. Dazu verwenden wir wieder das Programmiermodell aus dem Anhang.
Das X- Register habe den Wert $02, in der Zeropage stehen die Werte:

```
$0057 00 50 00 40
```

Folgender Befehl ist abzuarbeiten:

```
$3000 A1 57     LDA ($57,X)
```

Der Adresszähler- Inhalt wird zwischengespeichert, der Inhalt des X- Register wird zu dem Operanden addiert. $02 + $57 = $59. Das Ergebnis ($59) wird in das Lowbyte des Adresszählers geladen und das Highbyte des Adresszähler auf $00 gesetzt. Damit zeigt der Adresszähler auf die Adresse $0059. Das Byte an dieser Adresse ($00) und das folgende Byte ($40) werden in den Adresszähler geladen. Dieser zeigt damit auf die Adresse $4000. Das Byte an dieser Adresse wird dann in den ACCU geladen. Nachdem der Adresszähler aus dem Zwischenspeicher zurückgeladen und auf den nächsten Befehl eingestellt ist, ist der Befehl abgearbeitet. Enthielte das X- Register den Wert $00, so würde das Byte an Adresse $5000 in den ACCU geladen. Das X- Register und der Operand bestimmen also die Adresse der Adresse des Datenbytes. Die Adresse des Datenbytes steht immer in der Zeropage also im RAM.

Das folgende Beispielprogramm hat ebenfalls die Aufgabe, einen Quellbereich an einen Zielbereich zu übertragen. Es soll aber möglich sein, mehrere mögliche Zielbereiche festzulegen und dem Programm den gewünschten Zielbereich über einen Index mitzuteilen. Das Programm zeigt das in diesem Absatz besprochene in der Praxis:

```
$5000 A9 00     LDA #$00    ;Zeiger für Ziel-
$5002 A0 40     LDY #$40    ;adressen setzen
$5004 85 5A     STA $5A
$5006 84 5B     STY $5B
$5008 A9 00     LDA #$00
$500A A0 43     LDY #$43
$500C 85 5C     STA $5C
$500E 84 5D     STY $5D
$5010 A9 00     LDA #$00    ;Quelladresse
$5012 A0 51     LDY #$51    ;laden
$5014 A2 01     LDX #$01    ;Index in X
$5016 20 1A 50  JSR $501A   ;Übertragen
$5019 00        BRK         ;Rückkehr Monitor
```

```
$501A 85 58     STA $58     ;Zeiger für Quell-
$501C 84 59     STA $59     ;adresse setzen
$501E 86 57     STX $57     ;Zieladr.Z.setzen
$5020 06 57     ASL $57     ;Zeiger * 2
$5022 A2 00     LDX #$00    ;X = 0
$5024 A1 58     LDA ($58,X) ;Datenbyte holen
$5026 20 33 50  JSR $5033   ;Zeiger increment.
$5029 81 58     STA ($58,X) ;an Zieladresse
$502B 20 33 50  JSR $5033   ;Zeiger increment.
$502E C9 FF     CMP #$FF    ;Erkennungsbyte ?
$5030 D0 F2     BNE $5024   ;Nein- zurück
$5032 60        RTS         ;Rücksprung

$5033 F6 58     INC $58,X   ;Lowbyte erhöhen
$5035 D0 02     BNE $5039   ;kein Übertrag- Sprung
$5037 F6 59     INC $59,X   ;Highbyte erhöhen
$5039 A8        TAY         ;ACCU zwischenspeichern
$503A 8A        TXA         ;X in ACCU
$503B 45 57     EOR $57     ;Bereich Umschalten
$503D AA        TAX         ;ACCU in X
$503E 98        TYA         ;ACCU aus Zwischensp.
$503F 60        RTS         ;Rücksprung
```

Gehen wir einen Programmlauf durch: Das Programm ist wieder als Unterprogramm aufgebaut. Von Adresse $5000 bis $500E werden die Zieladressen gesetzt. Das eigentliche Programm beginnt an Adresse $501A und erwartet im ACCU und Y- Register die Quelladresse und im X- Register die wievielte Zieladresse gewählt werden soll. An Adresse $501A wird die übergebene Quelladresse in den Zeiger gebracht. Das X- Register wird in den Auswahlzeiger an Adresse $0057 gebracht und dieser Zeiger dann an Adresse $5020 mit 2 multipliziert. Sie werden gleich sehen, warum das erforderlich ist. Daraufhin wird das X-Register auf $00 gesetzt und der ACCU X- indiziert indirekt geladen. Die Adresse des zu ladenden Bytes ergibt sich wie folgt: X- Register ($00) + Operand ($58) = $0058. An dieser Zeropage Adresse befindet sich der Quellzeiger mit der momentanen Adresse $5100. Es wird also das Byte an Adresse $5100 in den ACCU gelesen.
Im Programm wird dann an Adresse $5026 ein Unterprogramm aufgerufen. Dieses Unterprogramm dient der Incrementierung

des Quell- und Zieladressenzeigers und der richtigen Einstellung des X- Registers. Um diese Aufgaben von einem Unterprogramm erledigen zu lassen, habe ich einige kleine Kniffe angewendet. Das X- Register soll immer entweder mit dem Wert $00 auf den Quellbereich oder mit dem mit 2 multiplizierten Index auf die Zeropage- Adresse der Adresse des Zielbereiches zeigen. Der momentane X- Register Inhalt ist $00. Das Register zeigt somit auf den Quellbereich. Das Unterprogramm muß also bei diesem Aufruf den Zeiger für den Quellbereich incrementieren. Verfolgen wir das: an Adresse $5033 wird die nullseitig X- indizierte Speicherzelle $58 incrementiert ($58 + $00 = $58). Da kein Überlauf stattfindet, wird an Adresse $5039 gesprungen. Andernfalls würde das (High) Byte an Adresse $0059 nullseitig X- indiziert ($59 + $00 = $0059) auch incrementiert werden.
An Adresse $5039 wird der ACCU Inhalt im Y- Register zwischengespeichert, da der ACCU ja noch das zu übertragende Byte enthält. Dann wird das X- Register in den ACCU gebracht und mit dem mit 2 multiplizierten Index aus Adresse $0057 exclusiv ODER verknüpft. Das ergibt:

```
X- Register     %00000000  EOR
Index           %00000010   =

                %00000010
```

Dieses Ergebnis wird in das X- Register zurückgebracht. Der ACCU wird wieder mit dem zu Übertragenden Byte aus dem Y- Register geladen. Aus dem Unterprogramm wird zurück an Adresse $5029 gesprungen.
Dort wird das zu übertragende Byte in den Zielbereich gespeichert. Die Zieladresse ergibt sich wie folgt: X- Register ($02) + Operand ($58) = $005A zeigen auf die Adresse $005A in der Zeropage, an welcher die Adresse $4000 des Zielbereiches steht. Das Datenbyte wird also an Adresse $4000 abgelegt.
Daraufhin wird wieder das Unterprogramm zur Incrementierung der Zeiger aufgerufen. Bei diesem Aufruf wird der Zieladressen - Zeiger incrementiert, da das X- Register den Wert $02 enthält ($02 + $58 = $5A). Damit sind dann beide Zeiger einmal incrementiert worden. Die Zeiger enthalten

jetzt folgende Werte:

```
$0058 01 51 01 40 00 50
```

Die exclusiv ODER Vernüpfung an Adresse $503B ergibt folgendes:

```
X- Register     %00000010  EOR
Index           %00000010   =

                %00000000
```

Das X- Register zeigt somit wieder auf den Quellbereich. Sie sehen, daß das X- Register bei jedem Aufruf zwischen Ziel- und Quellbereich umgeschaltet wird.
Nach dem Rücksprung an Adresse $502E, wird dort auf das Erkennungsbyte geprüft und bei nicht gegebenem Bereichsende das folgende Byte übertragen. Andernfalls wird an Adresse $5019 zurückgesprungen und dort zum Monitor zurückgekehrt.
Es ist wichtig, daß Sie dieses Programm genau nachvollzogen haben. Gehen Sie noch mindestens einen Programmlauf durch. Nachfolgend steht der Hexdump zu diesem Beispielprogramm.

```
$5000 A9 00 A0 40 85 5A 84 5B
$5008 A9 00 A0 43 85 5C 84 5D
$5010 A9 00 A0 51 A2 01 20 1A
$5018 50 00 85 58 84 59 86 57
$5020 06 57 A2 00 A1 58 20 33
$5028 50 81 58 20 33 50 C9 FF
$5030 D0 F2 60 F6 58 D0 02 F6
$5038 59 A8 8A 45 57 AA 98 60
```

Geben Sie das Programm in Ihren Computer ein und speichern es nach der Kontrolle unter dem Namen "FUENF" auf Diskette ab. Testen können Sie das Programm in derselben Weise wie das Beispielprogramm "VIER". Sie können auch mehr als 2 Zielbereiche festlegen. Gehen Sie dabei aber nicht über die Zeropage - Adresse $0070 hinaus, da die dortigen Speicherstellen anderweitig benutzt werden. Experimentieren Sie so lange mit dem Programm, bis Sie den Stoff gründlich erarbeitet haben.

Damit haben Sie alle 65xx Adressierungen am Beispiel kennengelernt. Selbstverständlich bedarf es noch einiger Übung bis Sie zu jedem Problem sofort die richtige Adressierung erkennen. Ich empfehle Ihnen an dieser Stelle einmal einige Experimente mit dem bisher vermittelten Stoff und den Beispielprogrammen anzustellen. Sie werden sehen, daß Sie dadurch relativ schnell eine gute Sicherheit im Umgang mit den 65xx Befehlen und Adressierungen erlangen. Im folgenden Absatz werden Sie dann die Arithmetischen Befehle kennenlernen.

Fragen zu 1.9.

1.) Wie heißen die vier Arten der indizierten Adressierung ?

2.) Durch Änderung eines Befehles im Beispielprogramm "DREI" kann erreicht werden, daß das Programm einen Bereich von 128 Bytes mit einem bestimmten Byte füllt (ähnlich der Fill- Funktion). Welcher Befehl ist wie zu ändern ? Was ist bezüglich der Adressen der nachfolgenden Befehle zu beachten ? Probieren Sie Ihre Lösung aus.

3.) Ist es zutreffend, daß die X- indizierte indirekte Adressierung sich nicht wesentlich von der indirekt Y-indizierten Adressierung unterscheidet ?

4.) Ist der Befehl - LDA ($5000),Y - mit 65xx Prozessoren möglich ?

5.) Welche Vorteile bringt es, daß im Programm "VIER" an Adresse $501A das Y- Register und nicht die Lowbytes der Zeiger ($61 / $63) incrementiert werden ?

Antworten zu 1.9.

zu 1.) 1. Absolut X- oder Y- indizierte / 2. Nullseitig X- oder Y- indizierte / 3. indirekt Y- indizierte / 4. X-indiziert indirekte

zu 2.) Durch Ändern des Befehls LDA $5100,X an Adresse $5005 in den Befehl LDA #$AA würde der Bereich $5000 - $507F mit dem Byte $AA gefüllt. Da der neue Befehl durch die unmittelbare Adressierung nur einen Operanden hat, müssen alle folgenden Befehle um ein Byte aufrücken. Der Branchbefehl an der ursprünglichen Adresse $500D muß dann ein Byte weniger überspringen. Der Operand ist dadurch in $F4 abzuändern.

zu 3.) Nein, die X- indizierte indirekte Adressierung bezieht den Index auf die Adresse der indirekten Adresse während die indirekt Y- indizierte Adressierung den Index auf die Adresse des Datenbyte bezieht.

zu 4.) Nein, die Zeiger- Bytes können sich nur in der Zeropage befinden.

zu 5.) Es müßten zwei Incrementier- Befehle gegeben werden. Auch müßte zweimal der eventuelle überlauf getestet werden, um gegebenenfalls das entsprechende Highbyte zu incrementieren. Dadurch würden das Programm und die Ausführungszeit erheblich länger. Wichtig ist auch, daß Abläufe innerhalb des Prozessors wesentlich schneller ablaufen als Abläufe, die das externe ROM oder RAM benötigen. Der Befehl INY wird mehr als doppelt so schnell wie der Befehl INC $61 und dreimal so schnell wie der Befehl INC $5000 ausgeführt (vergl. 65xx Befehle).

1.10. Walk (Einzelschrittmodus)

Wie der Name schon sagt, können Sie mit Hilfe dieser Monitorfunktion ein Maschinenprogramm in Einzelschritten ablaufen lassen. Nach jedem abgearbeiteten Befehl wird das Programm angehalten und der folgende Befehl disassembliert dargestellt. Je nach Monitor werden auch die Inhalte der Prozessorregister ausgegeben. Nach betätigen einer Taste wird dann der folgende Befehl abgearbeitet und das Programm wieder angehalten. Durch Betätigen einer bestimmten Taste, meist der STOP Taste, kann das Programm auch abgebrochen und zum Monitor zurückgekehrt werden. Damit können Sie bei einem Programm, welches nicht in der gewünschten Weise arbeitet leicht den Fehler finden.
Bevor wir diese Funktion in der Praxis ausprobieren, behandeln wir noch einige 65xx Befehle, deren Ausführung sehr interessant mit dem Einzelschrittmodus zu beobachten ist.
Zunächst die Befehle zum gezielten Löschen oder Setzen der einzelnen Flags des Statusregister: Das C- D- I- und V- Flag können so einzeln gelöscht und das C- D- und I- Flag einzeln gesetzt werden. Sie finden die entsprechenden Befehle wieder im Anhang.
Da keine Adressierung erforderlich ist, arbeiten alle Befehle zum setzen der Flags mit der implizierten Adressierung. Eine praktische Anwendung können Sie sich bereits vorstellen:

```
$5000 A9 08     LDA #$08 ;ACCU = $08
$5002 38        SEC      ;setze das Carry Flag
$5003 6A        ROR A    ;ACCU /2 +$80
$5004 00        BRK      ;Rückkehr Monitor
```

Der Ablauf ist folgender: Der ACCU wird unmittelbar mit dem Wert $08 geladen. Der Befehl SEC setzt das Carry Flag. Dieses wird dann durch den Verschiebebefehl ROR A an das 8. Bit des ACCU geschoben (%00001000 ROR A = %10000100). Der Inhalt des ACCU wurde durch die Verschiebung nach rechts durch zwei dividiert. Der gesamte Vorgang entspricht also der Rechnung: $08 / 2 + $80 = $84. Probieren Sie das einmal aus. Um das Ergebnis lesen zu können, müssen Sie dieses natürlich vor der Rückkehr zum Monitor abspeichern. Sie sind jetzt sicher in

der Lage, das Programm entsprechend zu erweitern. Zeigt Ihr Monitor nach der Rückkehr durch den BRK Befehl die Registerinhalte an (z.B. PROFIMON), so können Sie das Ergebnis sofort ablesen.
Auch die Anwendung der anderen Befehle zum setzen der Flags werden Sie noch in der Praxis kennenlernen. Zuvor müssen wir uns jedoch mit der Addition und Subtraktion im Binärsystem befassen.
Die Rechenvorschriften sind in allen Zahlensystemen gleich. Unterschiedlich ist die Ziffer bei deren Erhöhung um 1 ein Übertrag gebildet werden muß. Diese Ziffer ist immer die höchste in dem betreffenden Zahlensystem darstellbare Zahl. Im Binärsystem ist das die 1, im Dezimalsystem die 9 und im Hexadezimalsystem das F. Wir wollen einmal die Zahlen 59 und 154 in drei Zahlensystemen addieren:

```
  Dezimal       Binär      Hexadezimal

    59        %00111011        $3B
+ 154       + %10011010      + $9A
-----       ------------     -----
  213         %11010101        $D5
```

Das gehen wir in Kurzform einmal durch.
Addition im Dezimalsystem: Beginnend an der Einerstelle.

```
9 + 4                = 3 plus Übertrag 1
5 + 5 + Übertrag 1 = 1 plus Übertrag 1
0 + 1 + Übertrag 1 = 2
```

Addition im Binärsystem: Beginnend an der Einerstelle.

```
1 + 0                = 1
1 + 1                = 0 plus Übertrag 1
0 + 0 + Übertrag 1 = 1
1 + 1                = 0 plus Übertrag 1
1 + 1 + Übertrag 1 = 1 plus Übertrag 1
1 + 0 + Übertrag 1 = 0 plus Übertrag 1
0 + 0 + Übertrag 1 = 1
0 + 1                = 1
```

Addition im Hexadezimalsystem: Beginnend an der Einerstelle.

```
B + A               = 5 plus Übertrag 1
3 + 9 + Übertrag 1 = D
```

Probieren Sie das mit mehreren anderen Zahlen aus. Wandeln Sie die Ergebnisse im Binär- und Hexadezimalsystem zur Kontrolle auf ihre Richtigkeit immer in das Dezimalsystem um. Die Subtraktion im Dezimalsystem ist ebenso leicht auf andere Zahlensysteme übertragbar. Es soll 44 von 162 subtrahiert werden:

Dezimal	Binär	Hexadezimal
162	%10100010	$A2
- 44	- %00101100	- $2C
118	%01110110	$76

Im Einzelnen stellt sich das wie folgt dar:

Subtraktion im Dezimalsystem: Beginnend an der Einerstelle.

```
2 - 4               = 8 plus Übertrag 1
6 - 4 - Übertrag 1 = 1
1 - 0               = 1
```

Subtraktion im Binärsystem: Beginnend an der Einerstelle.

```
0 - 0               = 0
1 - 0               = 1
0 - 1               = 1 plus Übertrag 1
0 - 1 - Übertrag 1 = 0 plus Übertrag 1
0 - 0 - Übertrag 1 = 1 plus Übertrag 1
1 - 1 - übertrag 1 = 1 plus Übertrag 1
0 - 0 - Übertrag  1 = 1 plus Übertrag 1
1 - 0 - Übertrag 1 = 0
```

Subtraktion im Hexadezimalsystem: Beginnend an der Einerstelle.

```
2 - C               = 6 plus Übertrag 1
A - 2 - Übertrag 1 = 7
```

Üben Sie auch hier wieder die Subtraktion im Binär- und Hexadezimalsystem. Das Rechnen in diesen Zahlensystemen

werden Sie bei Ihrer späteren Programmierarbeit in Maschinensprache immer wieder benötigen. Es ist daher wichtig, daß Sie zumindest die Addition und Subtraktion im Binär- und Hexadezimalsystem sicher beherrschen. Übung macht hier den Meister.
Die 65xx Prozessoren kennen einen Additions- und einen Subtraktionsbefehl. Die Argumente für diese Befehle können maximal den Wert $FF (255) annehmen. Da diese Begrenzung in der Regel zu klein ist, wird bei jeder Rechnung ein Übertrag berücksichtigt und gebildet. Dazu dient das Carry Flag. Sehen wir uns folgende Befehlssequenz einmal an:

```
$5000 A9 A5     LDA #$A5   ;Summand in
$5002 A0 24     LDY #$24   ;Zeropage
$5004 85 60     STA $60
$5006 84 61     STY $61
$5008 18        CLC        ;Carry Flag löschen
$5009 A5 60     LDA $60    ;Lowbytes
$500B 69 5C     ADC #$5C   ;addieren
$500D 8D 00 40  STA $4000  ;Ergebnis speichern
$5010 A5 61     LDA $61    ;Highbytes
$5012 69 6B     ADC #$6B   ;addieren
$5014 8D 01 40  STA $4001  ;Ergebnis speichern
$5017 00        BRK        ;Rückkehr Monitor
```

An Adressen $5000 - $5006 wird die 16 Bit Zahl $24A5 an die Zeropageadressen $60 und $61 gebracht (übliche Reihenfolge Low- Highbyte). An Adresse $5008 wird das Carry Flag gelöscht. An Adresse $5009 das Lowbyte $A5 der 16 Bit Zahl $24A5 in den ACCU geladen. An Adresse $500B wird eine Addition mit folgenden Argumenten durchgeführt:

```
Accumulatorinhalt   $A5
Operand             $5C
Carry Flag          $00
                    ---
Ergebnis            $01 plus Übertrag
```

Durch den Übertrag wird das Carry Flag gesetzt. An Adresse $500D wird das Ergebnis $01 der Addition an Adresse $4000 abgelegt. Mit den nächsten beiden Befehlen wird das Highbyte

$24 der 16 Bit Zahl $24A5 in den ACCU geladen und eine weitere Addition mit folgenden Argumenten durchgeführt:

Accumulatorinhalt	$24
Operand	$6B
Carry Flag	$01

Ergebnis	$90

Dieses Ergebnis $90 wird dann an Adresse $4001 abgelegt. Folgende 16 Bit Addition wurde durchgeführt:

```
  $24A5
+ $6B5C
-------
  $9001
```

Das Ergebnis $9001 steht an Adressen $4000 (Lowbyte) und $4001 (Highbyte). Nachfolgend finden Sie den Hexdump des Programmes:

```
$5000 A9 A5 A0 24 85 60 84 61
$5008 18 A5 60 69 5C 8D 00 40
$5010 A5 61 69 6B 8D 01 40 00
```

Geben Sie das Programm in Ihren Computer ein und speichern es nach der gewohnten Kontrolle unter dem Namen "SECHS" auf Diskette ab. Wir wollen das Programm im Einzelschrittmodus ablaufen lassen:

PROFIMON: W 5000
Das Programm wird auf diesen Befehl hin wie oben beschrieben abgearbeitet. Brechen Sie die Ausführung durch betätigen der STOP Taste ab, wenn Sie an dem BRK Befehl an Adresse $5017 angelangt sind.

T.EX.AS
Dieser Monitor kennt keinen Einzelschrittmodus in dieser Form. Es gibt allerdings die Möglichkeit, Unterbrechungspunkte zu setzen, an welchen das Programm gestoppt wird. Wir werden diese Möglichkeit im folgenden

Absatz besprechen. Starten Sie das Programm mit dem Befehl .EX $5000.

Wenn Ihr Monitor über einen Einzelschrittmodus verfügt, und bei jedem Stop die Registerinhalte anzeigt, können Sie den Rechenvorgang sehr schön verfolgen. Achten Sie dabei auch auf die Flags.
Das obige Programm hat wieder den Nachteil der mangelnden Flexibilität. Nur der erste Summand liegt im RAM und es werden immer zwei 16 Bit Zahlen addiert. Wir wollen ein Programm entwerfen dessen Summanden beide im RAM liegen und bis zu 255 Bytes haben dürfen. Die Summanden sollen in den Bereichen $5100 - $51FE und $5200 - $52FE stehen. Das Ergebnis soll im Bereich $5300 - $53FF stehen. Die Reihenfolge der Wertigkeit ist die bisher gebrauchte: Das LSB (least signifikant Byte - niedrigst wertiges Byte) an der niedrigsten Adresse, das MSB (most signifikant Byte - höchst wertiges Byte) an der höchsten Adresse. Nachfolgend finden Sie das Assemblerlisting:

```
$5000 A9 02     LDA #$02     ;2 Bytes
$5002 20 06 50  JSR $5006    ;Routine aufr.
$5005 00        BRK          ;Rückkehr Monitor
$5006 85 60     STA $60      ;Anz.Bytes zwschsp.
$5008 A9 00     LDA #$00     ;ACCU = 0
$500A AA        TAX          ;X = 0
$500B 2A        ROL A        ;C- Flag setzen
$500C BD 00 51  LDA $5100,X  ;Summanden
$500F 7D 00 52  ADC $5200,X  ;addieren
$5012 9D 00 53  STA $5300,X  ;Ergebnis speichern
$5015 6A        ROR A        ;C- Flag sichern
$5016 E8        INX          ;X + 1
$5017 E4 60     CPX $60      ;Alle Summanden ?
$5019 D0 F0     BNE $500B    ;Nein- zurück
$501B 2A        ROL A        ;MSB Übertrag ?
$501C 90 05     BCC $5023    ;Nein- Rücksprung
$501E A9 01     LDA #$01     ;Ergebnis MSB
$5020 9D 00 53  STA $5300,X  ;= $01
$5023 60        RTS          ;Rücksprung
```

Gehen wir wieder einen Programmlauf durch: Die Summanden

seien $ABCD und $DCBA. Im Speicher muß demnach stehen:

```
$5100 CD AB
$5200 BA DC
```

Das Unterprogramm ab Adresse $5006 erwartet nach dem Einsprung im ACCU die Anzahl der zu addierenden Bytes. In obigem Listing wird an Adresse $5000 diese Anzahl mit $02 geladen (maximal $FF).
Unmittelbar danach wird die Additionsroutine aufgerufen. Dort wird an Adresse $5006 die Anzahl der zu addierenden Bytes in der Zeropage an Adresse $0060 zwischengespeichert.
An Adresse $5008 wird der Accu auf Null gesetzt. Wir werden gleich sehen warum das erforderlich ist. Durch Transfer des ACCU Inhalt in das X- Register wird dieses ebenfalls auf Null gesetzt. Das X- Register dient als Zeiger und muß daher zu Beginn auf Null gesetzt werden.
Der Befehl ROL A an Adresse $500B dient hier zum Setzen des C- Flags. Da der ACCU den Wert %00000000 enthält wird das C- Flag durch den ROL A Befehl auf Null gesetzt. Sie werden sich sicher fragen, warum hier nicht der Befehl CLC benutzt wurde. Die Begründung hierzu wird gleich offensichtlich.
Die folgenden drei Befehle addieren die Summanden und speichern das Ergebnis ab. Da alle drei Befehle absolut X- indiziert adressiert sind und das X- Register momentan den Wert $00 hat, wird folgende Addition durchgeführt:

```
          Aus Adresse $5100      $CD
          Aus Adresse $5200      $BA
          Carry Flag           + $00
                                 ---
Ergebnis nach Adresse $5300      $87 plus C = 1
```

Der nächste Befehl speichert den Inhalt des C- Flag zwischen, indem das C- Flag an das 8. Bit des ACCU geschoben wird.
Daraufhin wird an Adresse $5016 das X- Register auf den Wert $01 incrementiert. Der folgende Befehl vergleicht daraufhin den Inhalt des X- Register mit der zwischengespeicherten Anzahl zu addierender Bytes $02. Der Vergleich wird mit dem Befehl CPX ausgeführt. Dieser Befehl beeinflußt aber auch das C- Flag. Daher war es an Adresse $5015 erforderlich den

Inhalt des C- Flag im 8. ACCU Bit zwischenzuspeichern. Da das X- Register die Anzahl zu addierender Bytes noch nicht erreicht hat, wird an Adresse $500B zurückgesprungen.
Dort wird mit dem ROL A Befehl der Inhalt des C- Flag zurückgeholt, womit sich auch dieser anfänglich unpassend erscheinende Befehl erklärt.
Die nächsten drei Befehle führen jetzt folgende Addition durch:

```
Aus Adresse $5101        $AB
Aus Adresse $5201        $DC
Carry Flag             + $01
                       -----
Ergebnis nach Adresse $5301    $88 plus C = 1
```

Der Inhalt des C- Flag wird daraufhin wieder zwischengespeichert und das X- Register auf $02 incrementiert. Damit wird der Sprung an Adresse $5019 nicht ausgeführt.
An Adresse $501B wird das C- Flag wieder aus dem Zwischenspeicher geholt. Der Branchbefehl an Adresse $501C wird somit ausgeführt, wenn die letzte Addition keinen Überlauf erzeugt hat. In diesem Beispiel allerdings ist das C- Flag gesetzt und der Sprung wird somit nicht ausgeführt. Durch den vorhandenen Überlauf wird erkannt, daß das MSB des Ergebnis = $01 sein muß. Dem tragen die Befehle an Adresse $501E und $5020 Rechnung, indem dort das MSB (in diesem Fall an Adresse $5302) auf $01 gesetzt wird. Damit steht dann folgendes Ergebnis im Speicher:

```
$5300 87 88 01 - Stellenrichtig: $018887
```

Der Befehl an Adresse $5023 veranlaßt den Rücksprung aus der Routine, worauf an Adresse $5005 zum Monitor zurückgekehrt wird.
Nachfolgend finden Sie den Hexdump des Programmes:

```
$5000 A9 02 20 06 50 00 85 60
$5008 A9 00 AA 2A BD 00 51 7D
$5010 00 52 9D 00 53 6A E8 E4
$5018 60 D0 F0 2A 90 05 A9 01
```

```
$5020 9D 00 53 60
```

Speichern Sie das Programm nach Eingabe und Kontrolle unter dem Namen "SIEBEN" auf Diskette ab.
Lassen Sie dann das Programm wieder im Einzelschrittmodus ablaufen und beobachten dabei genau die Prozessorregister. Ändern Sie auch die Summanden und die Anzahl zu addierender Bytes. Wenn Sie das Programm normal (ohne Einzelschrittmodus) starten und 255 Bytes addieren lassen, werden Sie dennoch keine Rechenzeit bemerken. Versuchen Sie sich einmal vorzustellen wie groß das Ergebnis im Dezimalsystem wäre. Maschinensprache ist sehr schnell - wie gesagt.
Außer der Darstellung von Zahlen in verschiedenen Zahlensystemen gibt es noch zwei weitere für Sie wichtige Darstellungsformen:
Die Fließkommadarstellung. Bei dieser Form werden alle Zahlen als Potenzen dargestellt. Das bringt eine Erleichterung vieler Berechnungen und vor allem können sehr große oder sehr kleine Zahlen mit weniger Stellen dargestellt werden. Alle Berechnungen des BASIC Interpreters Ihres C 64 werden im Fließkommaformat durchgeführt. Ich will hier nicht weiter auf die Fließkommaarithmetik eingehen, da damit der gesteckte Rahmen weit überschritten würde. Im Literaturverzeichnis finden Sie in 1 alle Informationen die Sie zu diesem Thema benötigen.
Die BCD (binary coded dezimal - Binär codierte Dezimalzahlen) Darstellung hingegen werden wir genauer betrachten. Wie Sie bereits festgestellt haben, ist die Umrechnung vom Binär- in das Dezimalsystem und umgekehrt sehr umständlich. Eine mögliche Vereinfachung stellt hier der BCD Code dar.
Die höchste im Dezimalsystem verwendete Ziffer ist 9. Um diese Ziffer binär darzustellen werden 4 Binärstellen benötigt - 9 = %1001. Im BCD Code werden immer 4 Binärstellen zu einer Dezimalstelle zusammengefaßt. Für diese vier Binärstellen beträgt der maximal erlaubte Wert dann %1001. Die Bitfolgen %1010, %1011, %1100, %1101, %1110 und %1111 kommen im BCD Code für jeweils ein Nibble (4 Bit) nicht vor. Wird bei einer Rechnung im BCD Code für ein Nibble der Wert %1001 überschritten, so muß ein Übertrag gebildet werden. Ansonsten gelten innerhalb des Nibble die zuvor vorgestellten Binär- Rechenregeln.

Wir rechnen hierzu ein Beispiel durch:

```
Dezimal      Binär           BCD

   36      %00100100        %0011 0110
+ 157    + %10011101   + %0001 0101 0111
-----    -----------   -----------------
  193      %11000001     %0001 1001 0011
```

Um den Übertrag einer Rechnung im BCD Code festzustellen, muß der Wert Ergebnis- Nibble auf einen größeren Wert als %1001 hin überprüft werden. Ist der Ergebnis- Nibble größer %1001, so muß %1100 von diesem subtrahiert und ein Übertrag gebildet werden.
Das obige Beispiel zeigt den Vorteil der leichten Umrechnung in den BCD Code und den Nachteil des höheren Bedarfs an Stellen zur Darstellung einer Zahl. Wird die häufige Umrechnung von Zahlen in das Dezimalsystem verlangt, so kann insbesondere für kaufmännische Problemstellungen die Arbeit im BCD Code von Vorteil sein.
Die 65xx Prozessoren lassen sich durch Setzen des D- Flags in den Dezimalbetrieb umschalten. Im Dezimalbetrieb arbeiten der ADC und der SBC Befehl im BCD Code. Sie finden nachfolgend das Programm "SIEBEN" leicht abgewandelt für den Dezimalbetrieb:

```
$5000 A9 02     LDA #$02     ;2 Bytes
$5002 F8        SED          ;Dezimal Betrieb ein
$5003 20 08 50  JSR $5008    ;Routine aufr.
$5006 D8        CLD          ;Dezimal Betrieb aus
$5007 00        BRK          ;Rückkehr Monitor
```

```
$5008 85 60     STA $60      ;Anz.Bytes zwschsp.
$500A A9 00     LDA #$00     ;ACCU = 0
$500C AA        TAX          ;X = 0
$500D 2A        ROL A        ;C- Flag setzen
$500E BD 00 51  LDA $5100,X  ;Summanden
$5011 7D 00 52  ADC $5200,X  ;addieren
$5014 9D 00 53  STA $5300,X  ;Ergebnis speichern
$5017 6A        ROR A        ;C- Flag sichern
$5018 E8        INX          ;X + 1
$5019 E4 60     CPX $60      ;Alle Summanden ?
$501B D0 F0     BNE $500D    ;Nein- zurück
$501D 2A        ROL A        ;MSB Übertrag ?
$501E 90 05     BCC $5025    ;Nein- Rücksprung
$5020 A9 01     LDA #$01     ;Ergebnis MSB
$5022 9D 00 53  STA $5300,X  ;= $01
$5025 60        RTS          ;Rücksprung
```

Das Programm ist nur um die beiden Befehle SED und CLD erweitert worden und entspricht ansonsten genau dem Programm "SIEBEN". Nehmen Sie die entsprechenden Änderungen an Ihrem Programm vor und probieren es dann nach dem Abspeichern auf Diskette aus. Nachstehende Befehlsfolge für den Monitor erleichtert Ihnen die Änderung:

```
T 5000 5023 4000
T 4006 4023 5008
```

Sie brauchen dann nur die neuen Befehle und den JSR $5008 Befehl neu einzugeben.

Bis auf die Befehle zur Benutzung des Stapelspeicher und den Interrupt Befehlen kennen Sie nun schon alle 65xx Befehle. Sie haben sicher bemerkt, daß es fast noch wichtiger ist, die verschiedenen Möglichkeiten der Adressierung zu kennen als die Befehle. Die notwendige Sicherheit hierzu werden Sie sich durch Übung schnell aneignen. Wenn Sie beim Lesen dieser Zeilen die Idee haben irgend etwas doch einmal ausprobieren zu wollen, so sollten Sie das unbedingt sofort tun. Nichts bringt Ihnen den Stoff schneller näher als die eigene Kreativität.

Fragen zu 1.10.

1.) Welche Adressierung steht als einzige für die Befehle zur Beeinflußung der Flags zur Verfügung ?

2.) Welche Flags lassen sich direkt setzen oder löschen ?

3.) Was ergibt im Binärsystem 1 - 1 - Übertrag 1 und was ergibt 1 + 1 + Übertrag 1 ?

4.) Welche Befehle sind in den Programmen "SECHS" und "SIEBEN" zu ändern um eine Subtraktion durchzuführen ? Beachten Sie dabei, daß das Carry Flag bei der Subtraktion negiert betrachtet wird (siehe auch Anhang 65xx Befehle / 65xx Flags). Probieren Sie Ihre Lösung aus.

5.) Wieviel Bytes benötigen Sie zur Darstellung der Zahl 248 im BCD Code ? Stellen Sie die Zahl im BCD Code dar.

6.) Schlagen Sie eine Lösung für das Programm "SIEBEN" vor, mit welcher es nicht erforderlich ist das Carry Flag zwischenzuspeichern. Die Adresse und Reihenfolge der Bytes der Summanden und des Ergebnis soll beibehalten werden. Ein Tip: Die De- und Incrementierbefehle verändern das C- Flag nicht (siehe auch 65xx Befehle) - Das Y- Register kann als zweiter Zähler verwendet werden. Dann ist das X- Register weiterhin zu incrementieren und das Y- Register zu decrementieren. Probieren Sie Ihre Lösung aus.

Antworten zu 1.10.

zu 1.) Die implizierte Adressierung, da kein Datenbyte erforderlich ist.

zu 2.) Löschen lassen sich das §C- D- I- und V- Flag. Setzen lassen sich das C- D- und I- Flag.

zu 3.) 1 - 1 - Übertrag 1 = 1 plus Übertrag 1. 1 + 1 + Übertrag 1 = 1 plus Übertrag 1.

zu 4.) Für Programm "SECHS":

```
$5008 38        SEC
$500B E9 5C     SBC #$5C
$5012 E9 1B     SBC #$1B ;Operand geändert da
                         sonst negatives Erg.
```

Für Programm "SIEBEN":

```
$5008 A9 80     LDA #$80
$500A A2 00     LDX #$00    ;alle folgenden Befehle
                             müssen um 1 Byte ver-
                             schoben werden
$5010 FD 00 52  SBC $5200,X
$501C 60        RTS
```

Die Befehle nach Adresse $501C können entfallen, da ein Übertrag nicht erfolgen kann wenn der Minuend größer als der Subtrahend ist.

zu 5.) Es werden zwei Bytes benötigt:

%0000 0010 und %0100 1000.

zu 6.) Nachfolgend das Listing der Lösung:

```
$5000 A0 02     LDY #$02     ;Anz.Bytes
$5002 20 06 50  JSR $5006    ;Routine aufr.
$5005 00        BRK          ;Rückkehr Monitor
$5006 A2 00     LDX #$00     ;Zeiger = 0
$5008 18        CLC          ;C- Flag löschen
$5009 BD 00 51  LDA $5100,X  ;addieren und
$500C 7D 00 52  ADC $5200,X
$500F 9D 00 53  STA $5300,X  ;abspeichern
$5012 E8        INX          ;Zeiger + 1
$5013 88        DEY          ;Zähler - 1
$5014 D0 F3     BNE $5009    ;fertig ?
$5016 90 05     BCC $501D    ;MSB Überlauf ?
$5018 A9 01     LDA #$01     ;MSB setzen
$501A 9D 00 53  STA $5300,X
$501D 60        RTS          ;Rücksprung
```

Wie Sie sehen, ist diese Lösung wesentlich besser als die im Programm SIEBEN besprochene. Es ist in der Maschinensprache- Programmierung die Regel, daß die grundsätzliche Lösung eines Problems immer weiter verbessert werden kann. Die erste Lösung ist um so besser, desto besser die Problembeschreibung ist. Die Problemanalyse kann bei komplexeren Aufgaben aufwendiger sein als die eigendliche Programmierung. Sie werden im zweiten Teil dieses Buches einige Verfahren zur Problemanalyse für Maschinenprogramme kennenlernen.

Wandeln Sie zur Übung auch das obige Programm zur Subtraktion um. Es sind nur drei Befehle zu ändern.

1.11. Quicktrace (Programmablauf mit Unterbrechungspunkten)

Die im vorhergehenden Absatz besprochene Einzelschrittfunktion kann für die Fehlersuche in Maschinenprogrammen gute Dienste leisten. Wenn das Programm aber bereits lauffähig war und nur erweitert wurde, wäre es sinnvoll nur den erweiterten Teil zu testen. Ebenso wenn das Programm eine oder mehrere Verzögerungsschleifen wie im Programm "ZWEI" besprochen enthält, ist der Einzelschrittmodus schlecht anzuwenden. Im Programm "ZWEI" müssten Sie 255 mal eine Taste betätigen, um den Programmteil nach dem Unterprogrammaufruf zu testen.
Die meisten guten Monitore besitzen aus diesem Grund die Möglichkeit vor dem Programmablauf Unterbrechungspunkte festzulegen. Diese Unterbrechungspunkte werden durch Angabe der 16 Bit Adressen, an welchen das Programm anhalten soll, festgelegt. Nach dem Start eines Programmes im Unterbrechungsmodus wird das Maschinenprogramm normal abgearbeitet bis der Adresszähler mit dem Wert eines gesetzten Unterbrechungspunktes übereinstimmt. Liegt eine solche Übereinstimmung vor, so wird das Programm angehalten und bei den meisten Monitoren der nächste Befehl disassembliert und die Registerinhalte angezeigt. Das Programm kann dann durch betätigen einer Taste fortgesetzt, oder durch die STOP Taste abgebrochen werden.
Bevor wir das in der Praxis ausprobieren, sollen Sie noch die Befehle zur Benutzung des Stapelspeicher kennenlernen. Wenn Ihnen die Funktion des Stapelspeicher nicht mehr geläufig ist, so schlagen Sie bitte auf Seite 14 noch einmal nach.
Im Anhang finden Sie bei den 65xx Befehlen vier Befehle zur Benutzung des Stapelspeicher. Wie Sie dort sehen, läßt sich der Inhalt des Accumulator und des Statusregister auf den Stack (Stapelspeicher) ablegen und wieder vom Stack lesen.
Die Möglichkeit zur Ablage des Statusregister auf den Stack wird meist zur Sicherung der Flags vor einer Operation welche die Flags verändert, benutzt. Sie haben dieses Problem bereits in Programm "SIEBEN" kennengelernt. Hier ist also eine andere Lösungsmöglichkeit durch die Ablage der Flags (Statusregister) auf den Stack gegeben.
Die Benutzung des Stack ist sehr einfach und schnell. Es gibt allerdings einige wichtige Punkte zu beachten, da sonst

leicht ein Programmabsturz oder Datenverlust eintreten können.
Wie Sie bereits wissen, benutzt auch die Befehlserkennung bei dem JSR Befehl den Stack zur Abspeicherung der Rücksprungadresse. Diese Tatsache müssen Sie bei der Benutzung des Stack innerhalb eines Unterprogrammes unbedingt im Auge behalten. Ein Beispiel dazu:

```
$5000 20 00 51 JSR $5100

$5100 A9 12    LDA #$12
$5102 48       PHA
$5103 60       RTS
```

An Adresse $5000 werden die Bytes $50 und dann $02 (vergl. Seite 44) auf den Stack abgelegt und dann der Adresszähler mit der Adresse $5100 geladen. An Adresse $5100 wird der ACCU mit dem Byte $12 geladen und dieses Byte an Adresse $5102 auf den Stack abgelegt. Der RTS Befehl an Adresse $5103 bewirkt, daß der Adresszähler mit den obersten beiden Bytes des Stack geladen wird. Diese beiden Bytes bilden die Rücksprungadresse. In diesem Beispiel wird aber nicht die korrekte Rücksprungadresse $5002 geholt, sondern die Adresse $0212 und daraufhin der Adresszähler auf $0213 incrementiert. Das diese Adresse nicht die korrekte Rücksprungadresse ist, kann die Befehlserkennung nicht bemerken. Der Rücksprung an Adresse $0213 wird ausgeführt. Das bedeutet natürlich unweigerlich einen Programmabsturz.
Es ist also sehr wichtig alle innerhalb eines Unterprogrammes auf den Stack abgelegten Bytes vor dem Rücksprung wieder ausgelesen zu haben. Sollte das zu umständlich sein, kann auch der erste Befehl des Unterprogrammes TSX (Stapelzeiger in X- Register) und der letzte Befehl vor dem RTS Befehl TXS (X- Register in Stapelzeiger) sein. Wobei selbstverständlich das X- Register durch das Programm nicht verändert werden darf oder zwischengespeichert sein muß.
Da der Stack mit 256 Bytes Speicherkapazität relativ klein ist und zudem die Rücksprungadressen der Unterprogrammaufrufe speichern muß, ist es wichtig bei der Benutzung des Stack auf den verbleibenden Speicherplatz im Stack zu achten. Wenn der Stackpointer (Stapelzeiger) den Wert $00 enthält und somit

auf die letzte Stackadresse zeigt, bewirkt ein erneutes Ablegen auf dem Stack, daß der Stackpointer auf den Wert $FF decrementiert wird. Dadurch werden dann die Daten oder Rücksprungadressen, die zu Beginn auf den Stack abgelegt wurden, zerstört.
Die Hauptanwendung des Stack liegt für den Programmierer in der Möglichkeit einzelne Bytes schnell und einfach zwischenzuspeichern. Darüberhinaus kann es manchmal von Vorteil sein die Rücksprungadressen im Stack zu manipulieren und so Möglichkeiten zu schaffen, die mit dem normalen 65xx Befehlssatz nicht möglich sind.
Der BASIC Befehl ON X GOSUB XX ist zur Bedienung eines Menues sehr vorteilhaft einzusetzen. Das folgende Programmbeispiel soll eine ähnliche Struktur in Maschinensprache vorstellen.

```
$5000 20 1B 50  JSR $501B    ;Eingabe holen
$5003 29 03     AND #$03     ;maximal 3 mögl.
$5005 0A        ASL A        ;* 2
$5006 AA        TAX          ;Adresszeiger laden
$5007 BD 3B 50  LDA $503B,X  ;Lowbyte Spr.Adresse
$500A 85 60     STA $60      ;holen - setzen
$500C E8        INX          ;Zeiger auf Highbyte
$500D BD 3B 50  LDA $503B,X  ;Highbyte Spr.Adresse
$5010 85 61     STA $61      ;holen - setzen
$5012 A9 4F     LDA #$4F     ;Highbyte Rücksprung
$5014 48        PHA          ;in Stack
$5015 A9 FF     LDA #$FF     ;Lowbyte Rücksprung
$5017 48        PHA          ;in Stack
$5018 6C 60 00  JMP ($0060)  ;Programm ausführen

$501B 20 E4 FF  JSR $FFE4    ;Zeichen von Tastatur
$501E AA        TAX          ;Flag setzen
$501F F0 FA     BEQ $501B    ;Keine Eingabe - zurück
$5021 60        RTS          ;Rücksprung mit Zeichen

$5022 A9 43     LDA #$43     ;Lowbyte "NULL"
$5024 A0 50     LDY #$50     ;Highbyte "NULL"
$5026 20 1E AB  JSR $AB1E    ;Zeile ausgeben
$5029 60        RTS          ;Rücksprung

$502A A9 48     LDA #$48     ;Lowbyte "EINS"
```

```
$502C A0 50     LDY #$50    ;Highbyte "EINS"
$502E 20 1E AB  JSR $AB1E   ;Zeile ausgeben
$5031 60        RTS         ;Rücksprung

$5032 A9 4D     LDA #$4D    ;Lowbyte "ZWEI"
$5034 A0 50     LDY #$50    ;Highbyte "ZWEI"
$5036 20 1E AB  JSR $AB1E   ;Zeile ausgeben
$5039 60        RTS         ;Rücksprung

$503A 00        BRK         ;Rückkehr Monitor
```

Folgende Datenbytes müssen im Speicher stehen:

```
$503B 22 50 2A 50 32 50 3A 50
$5043 4E 55 4C 4C 00 45 49 4E
$504B 53 00 5A 57 45 49 00
```

Das eigendliche Programm steht einschließlich einer Unterroutine von Adresse $5000 bis 5021. Darauf folgen vier Programme die durch den entsprechenden Tastendruck auszuführen sind. Sehen wir uns einmal einen Programmlauf an: An Adresse $5000 wird eine Unterroutine aufgerufen. Der erste Befehl der Unterroutine an Adresse $501B ist wiederum ein Unterprogrammaufruf. Es wird die Betriebssystemroutine an Adresse $FFE4 aufgerufen. Wir wollen uns mit der Arbeitsweise dieser Routine nicht beschäftigen. Wichtig ist nur zu wissen, welche Funktion die Routine erfüllt und welche Parameter an die Routine zu übergeben sind und welche Parameter die Routine nach der Rückkehr ihrerseits übergibt.
Die Betriebssystemroutine an Adresse $FFE4 dient zur Abfrage der Tastatur. Bei Einsprung benötigt die Routine keine Übergabe von Parametern. Die Routine kann also einfach aufgerufen werden. Nach dem Aufruf wird geprüft welche Taste gedrückt ist und der ASCII Code der gedrückten Taste im ACCU übergeben. Ist keine Taste gedrückt, so enthält der ACCU den Wert $00.
Nach Rückkehr aus der Betriebssystemroutine, wird an Adresse $501E geprüft ob die Routine im ACCU den Wert $00 für keine gedrückte Taste übergeben hat. Der Befehl TAX erfüllt hier nur den Zweck die Flags entsprechend zu setzen. Wurde keine Taste betätigt, so wird über den Branchbefehl an Adresse

$501F die Betriebssytemroutine erneut aufgerufen. Andernfalls wird aus der Unterroutine nach Adresse $5003 zurückgekehrt. Das Unterprogramm entspricht in seiner Funktion exakt der BASIC Zeile:

```
10 GET A$:IF A$ = "" THEN GOTO 10
```

Das Programm läßt lediglich die Auswahl zwischen vier verschiedenen auszuführenden Programmen zu (vier Menuepunkte). Der UND Befehl an Adresse $5003 erfüllt zwei Funktionen gleichzeitig: Zum ersten wandelt er den ASCII Code der gedrückten Taste in den Zahlenwert um und zum zweiten begrenzt er den Wert auf maximal $03. Sehen Sie sich dazu die ASCII Code Tabelle im Anhang an. Die Codes der Ziffern 0 bis 9 lauten $30 bis $39. Wenn nun der Code für beispielweise die Ziffer 5 ($35) mit $08 UND verknüpft wird, so ergibt das den Wert 5:

```
ASCII Code      %00110101
UND verknüpft   %00001111

ergibt Wert     %00000101
```

Durch die UND Verknüpfung mit $03 wird die zusätzliche Begrenzung auf den maximalen Wert von $03 erzielt. Dabei ist zu beachten, daß die Betätigung einer anderen Taste als der Tasten 0, 1, 2 oder 3 ein mehr oder weniger zufälliges Ergebnis liefert. Die Taste 5 ergibt:

```
ASCII Code      %00110101
UND verknüpft   %00000011

ergibt Wert     %00000001
```

Der Wert der gedrückten Taste wird dann an Adresse $5005 mit 2 multipliziert und an Adresse $5006 in das X- Register gebracht. Nehmen wir einmal an es wäre die Taste 2 betätigt worden, so enthält das X- Register den Wert $04.
Das X- Register dient als Zeiger auf die Adresse des auszuführenden Programmes. Die Liste der Startadressen der durch das Programm auswählbaren Programme steht von Adresse

$503B bis $5042 im Speicher. Dabei ist wieder die Reihenfolge Lowbyte / Highbyte eingehalten. An Adresse $5007 wird das Lowbyte der Startadresse in den ACCU gelesen. Durch den Wert des X- Register (Zeiger) wird das Byte an Adresse $503F mit dem Wert $32 eingelesen. Das Lowbyte der Startadresse $32 wird dann in der Zeropage an Adresse $0060 abgespeichert. Nachdem der Zeiger (X- Register) incrementiert wurde, wird auf dieselbe Weise das Highbyte der Startadresse $50 aus der Sprungtabelle an die Zeropageadresse $0061 gebracht.
Die Befehle von Adresse $5012 bis $5017 "fälschen" den Stack, indem dort die Rücksprungadresse $5000 in den Stack gebracht wird. Mit dem nächsten RTS Befehl holt die Befehlserkennung die obersten beiden Bytes vom Stack in den Adresszeiger, incrementiert diesen und setzt das Programm an dieser Stelle fort. Beachten Sie, daß durch die Incrementierung des Adresszählers ein Byte vor der Rücksprungadresse angegeben werden muß. In diesem Fall also $4FFF.
Mit dem indirekten Sprungbefehl an Adresse $5018 wird dann das ausgewählte Programm aufgerufen. In diesem Bespiel das Programm an Adresse $5032.
Dieses Programm an Adresse $5032 benutzt wieder eine Betriebssystemroutine. Diese Routine dient zur Ausgabe einer Zeile auf dem Bildschirm an der aktuellen Cursorposition. Der Routine müssen im ACCU (Lowbyte) und im Y- Register (Highbyte) die Adresse der auszugebenden Zeichen im Speicher übergeben werden. Die Zeichen müssen dort im ASCII Code vorliegen. Es sind alle im Anhang aufgeführten Zeichen erlaubt. Also auch Steuerzeichen für Farbe, Cursor, Zeichensatz usw. Die auszugebenden Zeichen müssen mit einem Nullbyte beendet werden. Das Programm erwartet an Adresse $5043 bis $5051 auf diese Weise abgespeichert die Wörter "NULL", "EINS" und "ZWEI".
Die Befehle an Adresse $5032 bis $5034 laden die Adresse $504D für die auszugebende Zeile. An dieser Adresse steht das Wort "ZWEI". Dieses Wort wird durch die an Adresse $5036 aufgerufene Betriebssystemroutine auf den Bildschirm ausgegeben.
Der Rücksprung mit dem RTS Befehl an Adresse $5039 bewirkt durch die "Stackfälschung" den Rücksprung an Adresse $5000. Dort wird wieder auf die Eingabe einer Ziffer gewartet (Menue). Mit den Ziffern 0 und 1 werden dem eben beschribenen

ähnliche Programme aufgerufen. Diese Programme geben nur die Wörter "NULL" oder "EINS" statt "ZWEI" aus. Mit der Ziffer 3 wird über den BRK Befehl an Adresse $503A zum Monitor zurückgekehrt.
Es sei hier noch darauf hingewiesen, daß solche "Stackfälschungen" die Lesbarkeit eines Programmes erschweren. Das heißt, daß für Sie oder einen anderen Programmierer, ein Programm in welchem von der Möglichkeit den Stack direkt zu manipulieren häufig Gebrauch gemacht wurde, schlecht nachvollziehbar ist. Wenn wie im obigen Beispiel allerdings die Problemstellung ein solches Vorgehen erfordert, sollten Sie auch von diesen Möglichkeiten Gebrauch machen. Es muß dann in der Dokumentation zu der entsprechenden Routine nur deutlich auf die Manipulation hingewiesen werden.
Sie finden nachfolgend den Hexdump des Programmes und der erforderlichen Daten:

```
$5000 20 1B 50 29 03 0A AA BD
$5008 3B 50 85 60 E8 BD 3B 50
$5010 85 61 A9 4F 48 A9 FF 48
$5018 6C 60 00 20 E4 FF AA F0
$5020 FA 60 A9 43 A0 50 20 1E
$5028 AB 60 A9 48 A0 50 20 1E
$5030 AB 60 A9 4D A0 50 20 1E
$5038 AB 60 00 22 50 2A 50 32
$5040 50 3A 50 4E 55 4C 4C 00
$5048 45 49 4E 53 00 5A 57 45
$5050 49 00
```

Geben Sie das Programm so in Ihren Computer ein und überprüfen es genau bevor Sie es unter dem Namen "ACHT" auf Diskette abspeichern.
Das Programm soll mit gesetzten Unterbrechungspunkten gestartet werden. Würden Sie das Programm im Einzelschrittmodus abarbeiten lassen, müßten Sie auch die beiden Betriebssystemroutinen im Einzelschrittmodus durchlaufen. Das wäre sehr langwierig und uninteressant.
Beachten Sie, daß bei betätigen der Taste mit der Ziffer 3 in obigem Programm zum Monitor zurückgekehrt wird. Zu diesem Zeitpunkt befindet sich aber noch die "gefälschte"

Rücksprungadresse im Stack. Sollte das bei Ihrem Monitor zu Komplikationen führen, so müssen Sie das Programm erweitern indem Sie einfach vor der Rückkehr zum Monitor zweimal den Befehl PLA geben um so den Stack wieder richtig zu setzen.
Die beiden besprochenen Monitore PROFIMON und T.EX.AS. behandeln die Abarbeitung eines Programmes mit Unterbrechungspunkten sehr unterschiedlich. Sollten Sie einen anderen Monitor benutzen, so setzen Sie einen Unterbrechungspunkt mindestens an Adresse $5003 und $5021.

PROFIMON
Dieser Monitor bietet die Möglichkeit einen Unterbrechungspunkt festzulegen. Bei Erreichen der Adresse des Unterbrechungspunktes wird in den Einzelschrittmodus umgeschaltet. Der Unterbrechungspunkt wird mit folgendem Befehl gesetzt:

```
U XXXX YYYY
```

XXXX gibt die 16 Bit Adresse des Unterbrechungspunktes an. YYYY bietet eine weitere Möglichkeit: Enthält YYYY beispielsweise den Wert 0008, so wird erst in den Einzelschrittmodus umgeschaltet, wenn die Adresse XXXX 8 mal erreicht wurde. Das kann oft sehr nützlich sein. Wird YYYY nicht angegeben, so wird beim ersten Erreichen der Adresse XXXX in den Einzelschrittmodus umgeschaltet. Im Einzelschrittmodus wird das Programm dann, wie im vorherigen Absatz beschrieben, durch betätigen einer Taste in einzelnen Befehlen abgearbeitet. Dabei werden wieder die Registerinhalte angezeigt. Durch die STOP Taste kann das Programm auch hier wieder abgebrochen werden.
Gestartet wird ein Programm im Quicktracemodus (mit Unterbrechungspunkten) durch den Befehl:

```
Q XXXX
```

XXXX stellt hier die 16 Bit Startadresse dar.
Geben Sie folgende Befehlszeilen zum Start des Programmes "ACHT" ein:

```
U 5003
```

```
Q 5000
```

Betätigen Sie dann die Taste mit der Ziffer 0. Darauf wird in den Einzelschrittmodus ab Adresse $5003 geschaltet. Verfolgen Sie das Programm dann bis an Adresse $5026 die Betriebssystemroutine aufgerufen wird. Dort brechen Sie die Abarbeitung mit der STOP Taste ab. Geben Sie dann folgende Befehlszeilen:

```
U 5029
Q 5000
```

Betätigen Sie wieder die Taste mit der Ziffer 0. Wenn der Ablauf korrekt ist, starten Sie das Programm mit dem Befehl: G 5000
Wir werden im Kapitel über Interrupts noch auf den Unterbrechungsmodus von PROFIMON zu sprechen kommen. Dort werden Sie dann erfahren, warum die Abarbeitung im Unterbrechungsmodus langsamer vor sich geht.

T.EX.AS.
Mit diesem Monitor können Sie bis zu 10 verschiedene Unterbrechungspunkte setzen. Bei Erreichen der Adresse eines Unterbrechungspunktes wird zum Monitor zurückgekehrt. Die Unterbrechungspunkte (Breakpoints) werden wie folgt gesetzt:

```
.BP X $YYYY
```

X stellt die Nummer (0 - 9) und $YYYY die 16 Bit Adresse des Breakpoint dar.
Es kann wichtig sein über die Funktion des Unterbrechungsmodus mit T.EX.AS. informiert zu sein: Vor dem Start eines Maschinenprogrammes mit dem Befehl .EX wird an den Adressen, für die ein Breakpoint gesetzt wurde, der dort vorhandene Befehl gegen den BRK Befehl ausgetauscht. Der ausgetauschte Befehl wird zwischengespeichert. Erreicht das Programm den BRK Befehl so wird zum Monitor zurückgekehrt, so als hätten Sie den BRK Befehl an diese Adresse gesetzt. Bevor sich der Monitor wieder meldet, werden die ursprünglichen Befehle natürlich wieder eingesetzt. Das bedeutet allerdings das Breakpoints nur im RAM Bereich gesetzt werden können.

Geben Sie jetzt folgende Befehlszeilen:

```
.BP 0 $5003
.BP 1 $5029
.EX $5000
```

Betätigen Sie dann die Taste mit der Ziffer 0. Darauf muß sich der Monitor sofort wieder melden. Löschen Sie dann den Breakpoint 0 mit folgendem Befehl:

```
.BP 0 0
```

Starten Sie das Programm dann erneut und betätigen wieder die Taste mit der Ziffer 0. Daraufhin muß das Wort "NULL" ausgegeben werden und zum Monitor zurückgekehrt werden. Hat bis hierhin alles korrekt funktioniert, so löschen Sie auch den Breakpoint 1 und starten das Programm erneut. Die Werte aller Breakpoints werden nach .BP ohne Parameter ausgegeben. Sie können durch richtigen Einsatz der Breakpoints testen wieweit ein defektes Programm funktioniert und den Fehler so besser einkreisen.

Fragen zu 1.11.

1.) Welche Register lassen sich direkt auf den Stack ablegen ?

2.) Welchen Vorteil weist der Unterbrechungsmodus gegenüber dem Einzelschrittmodus auf ?

3.) Das Programm "ACHT" läßt sich ebenfalls optimieren: Durch eine weitere Stackfälschung kann das ausgewählte Programm mit dem RTS Befehl aufgerufen werden. Der Befehl JMP ($0060) entfällt damit. Vorsicht ! Die Sprungtabelle $503B - $5042 muß geändert werden. Auch wenn die aufrufbaren Programme an den alten Adressen bleiben können (nach RTS wird der Adresszähler incrementiert). Führen Sie diese Optimierung durch und probieren Sie sie aus.

4.) Ändern Sie das Programm "SIEBEN" so, daß das C- Flag im Stack zwischengespeichert wird (Befehl PHP). Vorsicht ! Der Stackpointer muß bei der Rückkehr aus einem Unterprogramm wieder den Einsprungwert haben. Probieren Sie Ihre Lösung aus.

Antworten zu 1.11.

zu 1.) Nur der Accumulator und das Statusregister lassen sich mit den Befehlen PHA und PHP auf den Stack ablegen.

zu 2.) Die Abarbeitung von Verzögerungsschleifen und Betriebssystemroutinen ist im Einzelschrittmodus sehr langwierig. Durch setzten von Unterbrechungspunkten an geeigneter Stelle können diese Routinen umgangen werden.

zu 3.) Nachfolgend das Listing der Lösung:

```
$5000 A9 4F     LDA #$4F     ;Highbyte Rücksprung
$5002 48        PHA          ;in Stack
$5003 A9 FF     LDA #$FF     ;Lowbyte Rücksprung
$5005 48        PHA          ;in Stack
$5006 20 16 50  JSR $5016    ;Eingabe holen
$5009 29 03     AND #$03     ;maximal 3 mögl.
$500B 0A        ASL A        ;* 2
$500C AA        TAX          ;Adresszeiger laden
$500D BD 3C 50  LDA $503C,X  ;Highbyte Spr.Adresse
$5010 48        PHA          ;in Stack
$5011 BD 3B 50  LDA $503B,X  ;Lowbyte Spr.Adresse
$5014 48        PHA          ;in Stack
$5015 60        RTS          ;Programm ausführen

$5016 20 E4 FF  JSR $FFE4
$5019 AA        TAX
$501A F0 FA     BEQ $5016
$501C 60        RTS
```

Die Sprungtabelle muß wie folgt geändert werden:

```
$503B 21 50 29 50 31 50 39 50
```

Die aufrufbaren Programme müssen bei dieser Lösung an ihren Adressen bleiben.

zu 5.) Nachfolgend das Listing der Lösung:

```
$5000 A9 02     LDA #$02     ;2 Bytes
$5002 20 06 50  JSR $5006    ;Routine aufr.
$5005 00        BRK          ;Rückkehr Monitor
$5006 85 60     STA $60      ;Anz.Bytes zwschsp.
$5008 A9 00     LDA #$00     ;ACCU = 0
$500A AA        TAX          ;X = 0
$500B 48        PHA          ;Flags = 0 vorbereiten
$500C 28        PLP          ;C- Flag setzen
$500D BD 00 51  LDA $5100,X  ;Summanden
$5010 7D 00 52  ADC $5200,X  ;addieren
$5013 9D 00 53  STA $5300,X  ;Ergebnis speichern
$5016 08        PHP          ;C- Flag sichern
$5017 E8        INX          ;X + 1
$5018 E4 60     CPX $60      ;Alle Summanden ?
$501A D0 F0     BNE $500C    ;Nein- zurück
$501C 28        PLP          ;MSB Übertrag ?
$501D 90 05     BCC $5024    ;Nein- Rücksprung
$501F A9 01     LDA #$01     ;Ergebnis MSB
$5021 9D 00 53  STA $5300,X  ;= $01
$5024 60        RTS          ;Rücksprung
```

1.12. Bank (Speicherkonfiguration bestimmen)

Wie Sie wissen, besitzt Ihr C 64 eine RAM Kapazität von 64 K Byte. Das Betriebssystem belegt einen Speicherbereich von 16 K Byte. Wie wir allerdings in Absatz 1.1. besprochen haben, kann der Prozessor 6510 mit 16 Adressbit maximal 64 K Byte adressieren. Dieses Problem wird bei dem C 64 in der Form gelöst, daß das RAM nicht immer in allen Bereichen "eingeschaltet" ist. Das bedeutet, daß für einen bestimmten Adressbereich immer wahlweise das ROM oder das RAM aktiv ist. Wie das genau funktioniert, können Sie bei Bedarf in 2 nachlesen.
Als Programmierer interessiert uns nur, auf welche Weise das RAM oder ROM für einen bestimmten Bereich ein oder ausgeschaltet wird. Zunächst jedoch sehen Sie sich einmal die im Anhang abgebildete Speicherkonfiguration für den Zustand nach dem Einschalten des C 64 an. Wie Sie dort sehen, ist der Bereich von $A000 bis $FFFF durch Betriebssystem und I/O (Input / Output - Peripherie wie Floppy, Drucker, Userport usw.) Bereich belegt. Nur der Bereich von $C000 bis $CFFF ist frei. Sozusagen hinter dem Betriebssystem und dem I/O Bereich "schlummern" 20 K Byte RAM und werden in dieser Speicherkonfiguration nicht genutzt. Die beiden 8 K Byte Blöcke RAM hinter dem Betriebssystem und der 4 K Byte Block hinter dem I/O Bereich lassen sich durch setzen des Bytes an Adresse $0001 ein oder ausschalten. Da alle Operationen des C 64 aber durch die Maschinensprachroutinen des Betriebssystems gesteuert werden, kann das ROM in diesem Bereich nur abgeschaltet werden, wenn die Kontrolle von einem anderen Maschinenprogramm übernommen wurde.
Von dem Byte an Adresse $0001 werden nur die beiden niederwertigsten Bit zur Steuerung der Speicherkonfiguration verwendet. Das 1. Bit wird mit LORAM das 2. Bit mit HIRAM bezeichnet. Eine zusätzliche Möglichkeit zur Steuerung der Speicherkonfiguration bietet sich mit zwei Eingängen am Expansion Slot Ihres C 64. Dort befinden sich die beiden Eingänge EXROM und GAME. Mit Hilfe dieser Eingänge ist es möglich die Speicherkonfiguration durch ein aufgestecktes Modul zu bestimmen.
In welchem Zusammenhang die Speicherkonfiguration mit diesen

Bit steht können Sie der Tabelle im Anhang entnehmen.
Wir wollen dazu ein kleines Beispielprogramm entwerfen. Das Programm soll einige Bytes aus einem bestimmten RAM Bereich in das RAM unter dem ROM ab Adresse $A000 bringen. Eine weitere Routine soll diese Bytes dann wieder zurückholen:

```
$5000 20 24 50  JSR $5024    ;RAM einschalten
$5003 A2 00     LDX #$00     ;Zeiger setzen
$5005 BD 00 51  LDA $5100,X  ;Byte holen -
$5008 9D 00 A0  STA $A000,X  ;und übertragen
$500B CA        DEX          ;Zeiger decrementieren
$500C D0 F7     BNE $5005    ;Zeiger nicht $00- Spr.
$500E 20 2C 50  JSR $502C    ;RAM abschalten
$5011 00        BRK          ;Rückkehr Monitor

$5012 20 24 50  JSR $5024    ;RAM einschalten
$5015 A2 00     LDX #$00     ;Zeiger setzen
$5017 BD 00 A0  LDA $A000,X  ;Byte holen -
$501A 9D 00 51  STA $5100,X  ;und übertragen
$501D CA        DEX          ;Zeiger decrementieren
$501E D0 F7     BNE $5017    ;Zeiger nicht $00- Spr.
$5020 20 2C 50  JSR $502C    ;RAM abschalten
$5023 00        BRK          ;Rückkehr Monitor

$5024 78        SEI          ;Keine Unterbrechung
$5025 A5 01     LDA $01      ;Steuerbyte holen
$5027 29 FC     AND #$FC     ;1. und 2. Bit löschen
$5029 85 01     STA $01      ;Steuerbyte neu setzen
$502B 60        RTS          ;Rücksprung

$502C A5 01     LDA $01      ;Steuerbyte holen
$502E 09 03     ORA #$03     ;1. und 2. Bit setzen
$5030 85 01     STA $01      ;Steuerbyte neu setzen
$5032 58        CLI          ;Unterbrechung erlauben
$5033 60        RTS          ;Rücksprung
```

Da Sie inzwischen alle verwendeten Befehle kennen, ist es nicht mehr erforderlich einen Programmablauf Schritt für Schritt gemeinsam durchzugehen. Sie sollten daß für sich allerdings immer machen. Auch wenn Sie eine Routine selbst programmiert haben, sollten Sie vor dem ersten Start

mindestens einen Programmdurchlauf auf dem Papier nachvollziehen. Dabei werden dann oft Fehler gefunden, die nach einem durch den Fehler verursachten Programmabsturz erst mühsam gesucht werden müssen. Durch das erneute Überdenken der Routine können auch schon Ansätze zur Optimierung gefunden werden.
Das Programm verwendet zur An- und Abschaltung des RAM Bereiches $A000 - $FFFF (siehe Tabelle im Anhang) jeweils ein Unterprogramm. Um diesen RAM Bereich einzuschalten wird das gesamte Betriebssystem abgeschaltet. Mit dem Interrupt (im folgende Kapitel besprechen wir genau was das ist) wird in Ihrem C 64 allerdings jede sechzigstel Sekunde eine Routine des Betriebssystems aufgerufen. Ist das Betriebssystem durch das RAM ersetzt, so steht an der Adresse aufgerufenen Routine irgend eine zufällige Bytefolge. Es erfolgt also unweigerlich ein Absturz. Dieser Aufruf der Betriebssystemroutine kann durch setzen des I- Flags unterbunden werden. Das erfolgt mit dem ersten Befehl der Routine an Adresse $5024 welche das RAM einschaltet.
Wie Sie der Tabelle im Anhang entnehmen können, wird der gewünschte RAM Bereich $A000 - $FFFF durch Löschen der Bit LORAM und HIRAM eingeschaltet. Da die verbleibenden 6 Bit für andere Zwecke benutzt werden, dürfen diese durch die Löschung des 1. und 2. Bit nicht verändert werden. Das wird durch die Verwendung des AND Befehls erreicht. Nehmen wir an das Byte an Adresse $0001 enthält den Wert %01100111. Die UND Verknüpfung ergibt dann folgendes:

```
Byte Wert           %01100111
UND verknüpft mit   %11111100

ergibt              %01100100
```

Sie sehen, daß nur das 1. und 2. Bit gelöscht wurden. Die anderen 6 Bit bleiben unverändert. Das neue Steuerbit wird dann an Adresse $0001 gebracht und damit ist das RAM im Bereich $A000 - $FFFF eingeschaltet.
Die Unterroutine zum Abschalten des RAM an Adresse $502C muß die Bit LORAM und HIRAM setzen. Dabei dürfen wieder die übrigen 6 Bit des Steuerbytes nicht verändert werden. Das wird durch die Verwendung des ORA Befehls an Adresse $502E

erreicht. Sehen wir uns auch die durchgeführte ODER Verknüpfung genau an:

```
Byte Wert            %01100100
ODER verknüpft mit   %00000011

ergibt               %01100111
```

Auch hier wurden die anderen 6 Bit nicht verändert. Nachdem das Steuerbyte an Adresse $0001 gebracht wurde, ist das RAM wieder ab- und das Betriebssystem wieder eingeschaltet. Damit kann auch der Interrupt wieder erlaubt werden. Das erfolgt an Adresse 5032 durch Löschen des I- Flag.
Sie finden nachfolgend den Hexdump der Routinen:

```
$5000 20 24 50 A2 00 BD 00 51
$5008 9D 00 A0 CA D0 F7 20 2C
$5010 50 00 20 24 50 A2 00 BD
$5018 00 A0 9D 00 51 CA D0 F7
$5020 20 2C 50 00 78 A5 01 29
$5028 FC 85 01 60 A5 01 09 03
$5030 85 01 58 60
```

Geben Sie das Programm so in Ihren Computer ein und speichern es dann unter dem Namen "NEUN" auf Diskette ab. Probieren Sie die Routinen aus, indem Sie den Bereich $5100 - $51FF mit dem Byte $11 füllen und die Routine an Adresse $5000 aufrufen. Füllen Sie dann den Bereich $5100 - 51FF mit dem Byte $22 und starten dann die Routine an Adresse $5012. Der Bereich $5100 - $51FF muß daraufhin wieder mit dem Byte $11 gefüllt sein. Sollte nach dem Starten ein Fehler auftreten, so können Sie den Fehler mit Hilfe des Einzelschrittmodus oder durch das Setzen von Unterbrechungspunkten sicher gut finden. Beachten Sie dabei, daß einige Monitore nach dem Setzen des I- Flag oder dem Abschalten des Betriebssystem nicht mehr korrekt im Einzelschritt- oder Unterbrechungsmodus arbeiten. Die Unterbrechungspunkte sind in diesem Fall entsprechend zu setzen.
Zwei Dinge sind noch zu ergänzen: Zum Ersten ist es unter Umständen interessant zu wissen, daß der Bereich $D000 - $DFFF drei verschiedene Belegungen hintereinander besitzt. Je

nach Belegung des Steuerbytes in Adresse $0001 kann sich dort der I/O Bereich, das Character ROM (Zeichenmatrizen) oder RAM befinden. Das Character ROM wird immer durch Löschen des dritten Bit (CHAREN) des Steuerbytes an Adresse $0001 eingeschaltet.
Zum Zweiten wird ein Schreib- und ein Lesezugriff auf die Bereiche $A000 - $BFFF und $E000 - $EFFF unterschiedlich durchgeführt. Ein Lesezugriff in diesen Bereichen liest das ROM aus. Ein Schreibzugriff schreibt in das RAM unter dem ROM. In beiden Fällen gilt das für LORAM und HIRAM des Steuerbytes = 1. In unserem Beispielprogramm wäre es demnach auch möglich gewesen den Bereich $A000 - $A0FF zu beschreiben ohne das ROM abzuschalten. Beachten Sie, daß im Bereich $D000 - $DFFF das RAM vor einem Schreibzugriff in das RAM eingeschaltet werden muß. Andernfalls werden die Register der Peripheriebausteine beschrieben.

Diese Zusammenhänge wurden hier nur insoweit beschrieben, wie diese für Sie als Programmierer interessant sind. Für die Programmierung von Grafik- und Musikprogrammen oder von Hardwareerweiterungen sind weitere Kenntnisse der Zusammenhänge erforderlich. Dieses Grundlagenbuch kann darauf nicht weiter eingehen. Sie finden im Anhang unter den Literaturhinweisen entsprechende weiterführende Literatur.

PROFIMON
Mit diesem Monitor ist es möglich das RAM des gesamten Speicherbereiches auszulesen. Also auch in den Bereichen, in welchen im Normalzustand nur das ROM ausgelesen werden kann. Zu diesem Zweck kann mit einem bestimmten Monitorbefehl festgelegt werden ob in dem Bereich $A000 - $BFFF und $E000 - $FFFF das RAM oder das ROM ausgelesen werden soll. Für den Bereich $D000 - $DFFF kann zwischen den drei Möglichkeiten: RAM, I/O oder Character ROM gewählt werden. Zur Festlegung der Bereiche dienen drei Befehle:

```
BR (normale Konfiguration)
BA (nur RAM eingeschaltet)
BC ($D000 - $DFFF Charakter ROM
    ansonsten RAM eingeschaltet)
```

Das probieren wir einmal aus: Geben Sie den Befehl M A000. Auf Ihrem Bildschirm erscheint:

```
A000 94 E3 7B E3 43 42 4D 42
```

Ändern Sie die Bytes durch Überschreiben mit dem Byte $AA. Auf Ihrem Bildschirm steht also:

```
A000 AA AA AA AA AA AA AA AA
```

Der erneute Befehl M A000 zeigt natürlich die alten Werte da der ROM Bereich angezeigt wird und dieser sich ja nicht ändern kann. Geben Sie jetzt folgende Befehle:

```
BA
M A000
```

Jetzt sehen Sie, daß die Bytes $AA tatsächlich unter das ROM geschrieben wurden. Sie können so auch testen ob das Programm "NEUN" nach dem Start an Adresse $5000 die zu übertragenden Bytes korrekt im Bereich $A000 - $A0FF abgelegt hat.
Die Änderung der Bereiche wirkt sich nur auf folgende Befehle aus:

- M (Speicher ansehen)
- D (Disassembler)
- C (Bereiche vergleichen)
- T (Transfere)
- H (Suchen)
- F (Bereich füllen)

Es können in den umgeschalteten Bereichen also keine Programme gestartet und die Bereiche nicht abgespeichert werden. Abhilfe ist mit der Transferfunktion meist möglich.

Fragen zu 1.12.

1.) Warum ist es nicht möglich, gleichzeitig die 64 K Byte RAM und den 24 K Byte ROM Bereich zu nutzen ?

2.) Welche RAM Bereiche dürfen Sie für Maschinenprogramme oder Daten verwenden ?

3.) Welche Möglichkeiten zur Beeinflußung der Speicherkonfiguration gibt es ?

4.) Was müssen Sie unbedingt beachten bevor Sie das Betriebssystem ROM abschalten ?

5.) Wie unterscheiden sich Schreib- und Lesezugriff auf die Bereiche $A000 - $BFFF und $E000 - $FFFF bei der normalen Speicherkonfiguration ?

Antworten zu 1.12.

zu 1.) Weil mit den zur Verfügung stehenden 16 Adressbit maximal 64 K Byte gleichzeitig adressierbar sind. Deshalb kann in den entsprechenden Adressbereichen nur entweder das RAM oder das ROM oder die Peripherie Bausteine eingeschaltet sein.

zu 2.) Sofern der BASIC Speicher von Adresse $0800 bis $9FFF nicht durch ein BASIC Programm belegt ist, kann dieser Bereich genutzt werden. Der Bereich von $C000 - $CFFF ist in der Regel immer frei. Die RAM Bereiche hinter dem ROM eignen sich gut zur Ablage von Daten. Es muß während der Programmierphase beachtet werden, welchen Bereich der Monitor belegt. Für PROFIMON ist das $C000 - CFFF und für T.EX.AS. ist das $6000 - $9FFF.

zu 3.) Die Speicherkonfiguration des C 64 läßt sich durch die drei Bit LORAM, HIRAM und CHAREN des Steuerbytes an Adresse $0001 und durch die Expansionslot- Eingänge GAME und EXROM beeinflußen.

zu 4.) Bevor das Betriebssystem ROM abgeschaltet wird, muß durch setzen des I- Flag ein Interrupt unterbunden werden, da die aufzurufende Interruptroutine durch die Abschaltung nicht vorhanden ist.

zu 5.) Bei einem Schreibzugriff auf diese Bereiche wird in das RAM geschrieben, bei einem Lesezugriff wird das ROM ausgelesen. Ermöglicht wird das durch eine besondere Steuerung der Speicherbereiche innerhalb Ihres C 64. Sie finden in 2 eine genaue Beschreibung dieser Technik.

2.1. Interrupttechnik

Sie haben bereits im vorhergehenden Absatz davon gehört, daß es eine Möglichkeit zur Unterbrechung des normalen Programmablaufes gibt. Auf gut neudeutsch ist diese Unterbrechung ein Interrupt. Ein Interrupt kann durch die Hard- und Software ausgelöst werden. Zur Auslösung eines Interrupt durch die Hardware sind drei Prozessorpins vorgesehen. Jeweils ein Low- Signal an einem der Pins kann einen der drei möglichen Interrupts auslösen. Wir wollen die drei Möglichkeiten im Einzelnen einmal betrachten:

Reset
Unmittelbar nach dem Einschalten eines Prozessorsystems muß der Prozessor an einer bestimmten Adresse beginnen ein Programm abzuarbeiten. Diese an sich einfache Maßgabe ist dennoch sehr wichtig, da das intelligenteste Betriebssystem nur arbeiten kann, wenn es auch zu Beginn gestartet wird. Zu diesem Zweck wird unmittelbar nach dem Einschalten eines Prozessorsystems der Reset Pin des verwendeten Prozessor kurz auf Low- Pegel gelegt. Dieses Signal veranlaßt die Befehlserkennung, die Bytes an Adresse $FFFC (Lowbyte) und $FFFD (Highbyte) in den Adresszähler zu laden. Das Byte auf welches der Adresszähler damit zeigt, ist das erste Befehlsbyte mit dem das Prozessorsystem die Arbeit nach dem Einschalten aufnimmt.
Die beiden Bytes an Adresse $FFFC und $FFFD stellen den Resetvektor dar. Bei Ihrem C 64 zeigt dieser Reset Vektor auf die Adresse $FCE2. Die Register des Prozessor werden durch den Reset nicht beeinflußt. Die Aktivierung des Reset Pin ist also für den C 64 völlig identisch mit dem Befehl JMP $FCE2.
Ein Reset kann auch zu jedem Zeitpunkt nach dem Einschalten aktiviert werden. Der Prozessor reagiert dann so als ob ein JMP ($FFFC) Befehl erkannt worden wäre. Das läßt sich mit Hilfe eines Reset Tasters erreichen. Der Reset Eingang des 6510 Prozessor Ihres C 64 ist am Userport (Pin 3) und am Expansion Slot (Pin C) erreichbar. Der Reset Taster muß nur

den Reset Eingang mit Masse (Pin 1) verbinden können.
Wenn ein Maschinenprogramm hoffnungslos abgestürzt ist und der Computer auf die Tastatur nicht mehr reagiert, kann der Reset Taster die Rettung sein. Durch den Reset wird der C 64 dann in den Einschaltzustand versetzt. Der Speicherinhalt wird dabei nur in der Zeropage und an den ersten drei Bytes des BASIC Speicher verändert. Ein Maschinenprogramm welches an einer anderen Adresse steht, was ja meist der Fall ist, wird nicht verändert. Wenn der Absturz also nicht das Programm zerstört hat, so brauchen Sie es nicht neu einzugeben. Ein in Maschinensprache geschriebener Monitor muß meist nicht neu geladen werden, sondern kann mit dem SYS Befehl von BASIC aus neu gestartet werden. Etwas Vorsicht ist hier allerdings geboten - Sie wissen nicht was während des Absturzes passiert ist. Es könnte hier der Monitor oder das Programm verändert und damit zerstört worden sein.

NMI (nicht maskierbarer Interrupt)
Während ein Reset eher eine Beendigung denn eine Unterbrechung ist, so stellt der NMI tatsächlich eine Unterbrechung eines laufen Programmes dar. Das heißt, daß das unterbrochene Programm nach Ausführung der Interruptroutine wieder fortgesetzt werden soll.
Um das zu ermöglichen, wird nach der Aktivierung des NMI Pin der gerade bearbeitete Befehl noch ausgeführt, daraufhin das Statusregister und dann der momentane Adresszählerinhalt auf dem Stack abgelegt. Weiter wird dann das I- Flag gesetzt und der NMI Vektor aus Adresse $FFFA und $FFFB in den Adresszähler geladen. Der NMI Vektor zeigt beim C 64 auf die Adresse $FE43. Wurde der NMI Pin aktiviert, so wird also die Routine an Adresse $FE43 aufgerufen.
Die ersten Befehle dieser Routine sichern die Inhalte des ACCU und des X- und Y- Register im Stack, da für eine korrekte Fortsetzung des unterbrochenen Programmes diese Werte wieder in die entsprechenden Register eingeladen werden müssen. Die Routine endet dann mit dem Wiedereinlesen der Registerinhalte und zuletzt mit dem RTI Befehl.
Der RTI Befehl wirkt ähnlich wie der RTS Befehl nur wird zusätzlich zu dem Adresszähler das Statusregister aus dem Stack zurückgeladen und das I- Flag gelöscht. Damit besitzen alle Prozessorregister ihre Inhalte die sie vor der

Unterbrechung hatten. Das Programm kann damit unverändert fortgesetzt werden.
Durch betätigen der RESTORE Taste oder der RESTORE Taste zusammen mit der STOP Taste wird im C 64 ein NMI erzeugt. Da der zweite Befehl der NMI Routine ein indirekter Sprungbefehl ist, kann es sein das der Vektor dieses Sprungbefehls an Adresse $0318 - $0319 durch den Absturz verändert wurde und so ein NMI durch die RESTORE Taste nach einem Absturz nicht korrekt ausgeführt wird.
Ein NMI, durch die RESTORE und STOP Taste aufgerufen, führt normalerweise einen BASIC Warmstart durch. Das heist, daß einige BASIC Parameter in der Zeropage neu gesetzt werden und in den READY Modus gesprungen wird.
Der NMI Eingang des Prozessors ist ebenfalls an den Expansion Slot herausgeführt (Pin D). Intern ist der NMI Eingang des Prozessor noch mit einem Ausgang eines der beiden CIA (Compex Interface Adapter) verbunden. So kann ein NMI durch geeignete Programmierung des CIA ausgelöst werden. Wir können darauf in Rahmen dieses Buches nicht näher eingehen. In 1 wird dieses Thema erschöpfend besprochen.

IRQ (Interrupt Request - Interrupt Anforderung)
Der IRQ arbeitet exakt in derselben Weise wie der NMI. Bis auf einen Unterschied: Wird der IRQ Pin des Prozessors aktiviert, so prüft die Befehlserkennung zunächst das I-Flag. Ist das I- Flag gesetzt, so wird der IRQ ignoriert. Andernfalls wird der IRQ Vector aus Adresse $FFFE und $FFFF geholt. Die weitere Abarbeitung entspricht exakt der eines NMI.
Der IRQ kann als einziger Interrupt auch softwaremäßig erzeugt werden. Das erfolgt durch den Befehl BRK. Wird der BRK Befehl von der Befehlserkennung ausgeführt, so wird zusätzlich zu dem I- Flag noch das B- Flag gesetzt. Der IRQ Vektor gilt auch für den BRK Befehl. Der BRK Befehl läßt sich allerdings nicht durch setzen des I- Flag unterbinden.
Auch der IRQ Eingang des Prozessor ist an den Expansion Slot herausgeführt (Pin 4). Intern ist der IRQ Eingang mit mehreren anderen Bausteinen des C 64 verbunden. Das kann für die Grafik und Sound Programmierung sehr nützlich sein (siehe Literaturverzeichnis 1, 3, 4).
Einer der CIA erzeugt über diese Verbindung jede sechzigstel

Sekunde einen IRQ. Die damit aufgerufene IRQ Routine stellt die interne Uhr des C 64 weiter und fragt die Tastatur nach eventuell gedrückten Tasten ab.
Der IRQ Vektor zeigt auf die Adresse $FF48. Wir wollen uns den ersten Teil der IRQ Routine an dieser Adresse einmal ansehen:

```
$FF48 48        PHA          ;ACCU zwischensp.
$FF49 8A        TXA          ;X- Register
$FF4A 48        PHA          ;zwischenspeichern
$FF4B 98        TYA          ;Y- Register
$FF4C 48        PHA          ;zwischenspeichern
$FF4D BA        TSX          ;Stackpointer in Zeiger
$FF4E BD 04 01  LDA $0104,X  ;Break Flag von Stack
$FF51 29 10     AND #$10     ;holen und testen
$FF53 F0 03     BEQ $FF58    ;kein BRK Befehl
$FF55 6C 16 03  JMP ($0316)  ;BRK Routine
$FF58 6C 14 03  JMP ($0314)  ;Interrupt Routine
```

Zu Beginn der Routine werden die Prozessorregister auf dem Stack zwischengespeichert. Dann wird über einen kleinen Umweg der nach dem IRQ auf den Stack abgelegte Statusregisterinhalt aus dem Stapelspeicher Bereich direkt ausgelesen. Der Test ob das B- Flag nach dem IRQ gesetzt war, entscheidet dann darüber ob der Vektor an Adresse $0316 oder der Vektor an Adresse $0314 zur Fortsetzung der Routine benutzt wird.
Im Normalfall zeigt der Vektor an Adresse $0316 auf die Adresse $FE66. Dort wird ein BASIC Warmstart durchgeführt. Der Vektor an Adresse $0314 zeigt normalerweise auf Adresse $EA31. Mit dieser Routine wird wie oben beschrieben die Uhr weitergestellt und die Tastatur abgefragt.
Der Weg eines IRQ mag auf den ersten Blick unnötig kompliziert erscheinen: Nach akzeptiertem IRQ wird an die Adresse $FF48, auf welche der IRQ Vektor zeigt, gesprungen. Dort wird dann, je nachdem was der Auslöser für den IRQ war, in die Routine auf welche der Vektor an Adresse $0314 oder an Adresse $0316 zeigt gesprungen. Dadurch aber, daß die Vektoren für die indirekten Sprünge im RAM Bereich stehen, wird das System sehr flexibel. Sie erhalten damit als Programmierer die Möglichkeit Ihre eigenen Interruptroutinen aufzurufen ohne Hardwareänderungen vornehmen zu müssen.

Das wollen wir gleich einmal an einem Beispiel ausprobieren:

```
$5000 A9 15     LDA #$15  ;neue Adresse der
$5002 A0 50     LDY #$50  ;der Interruproutine
$5004 78        SEI       ;kein Interrupt mehr
$5005 8D 14 03  STA $0314 ;Interrupt Vektor
$5008 8C 15 03  STY $0315 ;neu setzen
$500B A9 00     LDA #$00  ;Zähler setzen
$500D 8D 00 51  STA $5100 ;Farbenzähler
$5010 8D 01 51  STA $5101 ;Verzögerungszähler
$5013 58        CLI       ;Interrupt erlaubt
$5014 00        BRK       ;Rückkehr Monitor

$5015 CE 01 51  DEC $5101 ;Verz.zähler decrem.
$5018 F0 03     BEQ $501D ;0 ? dann Farbe ändern
$501A 4C 31 EA  JMP $EA31 ;normale Interruptrout.
$501D AD 00 51  LDA $5100 ;neue Farbe holen
$5020 8D 20 D0  STA $D020 ;in VIC Register
$5023 CE 00 51  DEC $5100 ;nächste Farbe
$5026 4C 31 EA  JMP $EA31 ;normale Interruptrout.
```

Die Routine von Adresse $5000 bis $5014 dient nur zur Inizialisierung der eigentlichen Interruptroutine an Adresse $5015. Auf diese Adresse $5015 der neuen Interruptroutine, oder genauer der Erweiterung der bestehenden Interruproutine, wird der Vektor an Adresse $0314 durch die Hilfsroutine gesetzt. Das muß durch eine solche Hilfsroutine erfolgen, um während der Änderung des Low- und Highbytes des Vektors einen Interrupt mit Hilfe des SEI Befehl zu unterbinden. Würden Sie den Vektor mit Hilfe des Monitor ändern, so würde nachdem Sie das Lowbyte des Vektor auf $15 geändert haben, der maximal eine sechzigstel Sekunde später erfolgende Interrupt an Adresse $EA15 springen. Damit kann natürlich nur ein Absturz erfolgen.
Die Hilfsroutine setzt noch zwei Zähler an Adressen $5100 und $5101 auf $00. Diese Zähler werden von der Erweiterung der Interruptroutine benutzt. Die Erweiterung decrementiert zunächst nur den Zähler an Adresse $5101. Solange dieser Zähler noch nicht den Wert $00 erreicht hat, wird sofort nach der Decrementierung in die normale Interruptroutine gesprungen.

Nach 256 Decrementierungen, das sind 256 * 1/60 Sekunden = 4.27 Sekunden, wird ein neuer Wert für eine andere Farbe in das Rahmen- Farbregister des VIC an Adresse $D020 gebracht. Da der C 64 16 verschiedene Farben für den Rahmen ermöglicht, sind nur die vier niederwertigen Bit signifikant. Die neue Farbe ist sofort auf dem Bildschirm sichtbar.
Daraufhin wird der Farbzähler auf eine neue Farbe gesetzt und wieder in die normale Interruptroutine gesprungen. Das Programm bewirkt also das etwa alle 4 Sekunden die Rahmenfarbe geändert wird. Alle BASIC und Maschinensprache Programme die den Interruptvektor an Adresse $0314 nicht verändern, laufen bis auf den Farbwechsel des Rahmen unberührt von der erweiterten Interruptroutine. Das liegt in der Funktion der Interrupttechnik begründet und bildet eine ihrer besonderen Stärken.
Probieren Sie das einmal aus. Sie finden nachfolgend den Hexdump des Programmes:

```
$5000 A9 15 A0 50 78 8D 14 03
$5008 8C 15 03 A9 00 8D 00 51
$5010 8D 01 51 58 00 CE 01 51
$5018 F0 03 4C 31 EA AD 00 51
$5020 8D 20 D0 CE 00 51 4C 31
$5028 EA
```

Tippen Sie das Programm so in Ihren Computer und speichern es nach der Kontrolle unter dem Namen "ZEHN" auf Diskette ab. Starten Sie dann die Hilfsroutine an Adresse $5000. Daraufhin muß nach etwa 4 Sekunden der Rahmen auf Ihrem Bildschirm die Farbe wechseln. Sollte der von Ihnen verwendete Monitor den Interruptvektor zu eigenen Zwecken verändert haben, so kann es zu Schwierigkeiten kommen. Sie können diese einfach umgehen, indem Sie auf die BASIC Ebene zurückkehren:

```
PROFIMON: X
T.EX.AS.: .EN
```

Eventuell schalten Sie den Computer auch einfach aus und wieder ein. Laden Sie das Programm dann als Maschinenprogramm wieder ein (LOAD "ZEHN",8,1) und starten es mit SYS 5*4096. In diesem Fall dürfen Sie nur ein maximal 15 K Byte langes

BASIC Programm nachladen, da die Erweiterung der Interruptroutine sonst durch das nachgeladene BASIC Programm zerstört wird. Wir werden darauf im folgenden Absatz genauer eingehen.

Damit haben Sie auch die Grundzüge der Interrupttechnik kennengelernt. Das vorgestellte Beispiel ist stellvertretend für einen großen Anwendungsbereich der Interrupttechnik. Durch den regelmäßig wiederkehrenden Interrupt werden quasi mehrere Aufgaben gleichzeitig erledigt: 1. unsere Erweiterung - 2. Uhr weiterstellen - 3. Tastaturabfrage - 4. BASIC Programm oder direkt Befehl abarbeiten.
Natürlich wissen Sie inzwischen das diese Aufgaben nicht gleichzeitig abgearbeitet werden. Durch die hohe Geschwindigkeit erscheint dies aber für den Benutzer so.
Eine zweite wichtige Anwendung der Interrupttechnik wird häufig in Verbindung mit Peripheriegeräten wie Floppy Disk usw. angewendet. In diesem Fall kann ein Peripheriegerät eine Interruptanforderung an den Computer senden, indem es eine Interruptleitung aktiviert. Der Vektor des entsprechenden Interrupts zeigt dann auf eine Routine die das Peripheriegerät entsprechend bedient. Das ist dann sehr nützlich, wenn das Peripheriegerät unregelmäßig oder sehr schnell die Verbindung zum Computer benötigt. Dasselbe ist natürlich auch umgekehrt möglich, wenn der Computer ein intelligentes (mit eigenem Prozessor) Peripherigerät durch einen Interrupt zur Bedienung auffordert. Das ist bei der Floppy Diskstation 1541 von Commodore der Fall.
Manche professionelle Systeme kennen auch eine Rettungsroutine bei Stromausfall. Mit dieser Interruptroutine werden innerhalb der verbleibenden Zeit bis die Spannung einen Mindestwert unterschritten hat die wichtigsten Daten und Funktionen gerettet und eventuell ein Alarm ausgelöst.
Sie werden die Interrupttechnik überwiegend in zusammenhang mit Grafik- und Soundprogrammierung benötigen. Der C 64 bietet gerade im Bezug auf die Interrupttechnik Möglichkeiten wie kaum ein anderer Computer dieser Klasse. Welche Möglichkeiten der C 64 bietet und wie Sie diese einsetzen können, konnten wir hier nur umreißen. Im Literaturverzeichnis finden Sie zu diesem Thema Bücher für die wichtigsten Spezialgebiete (insbesondere 1, 3, 4, 5).

Es bleibt noch darauf hinzuweisen, daß die meisten Monitore den BRK Vektor an Adresse $0316 benutzen um den Monitor selbst mit einem Warmstart aufzurufen. Aus diesem Grund konnten wir bisher die Beispielprogramme immer mit dem BRK Befehl abbrechen.

PROFIMON

Dieser Monitor wendet für den Einzelschritt- und den Unterbrechungsmodus die Interrupttechnik an. Die Zeit bis ein Interrupt eintritt wird so kurz eingestellt, daß der Prozessor immer nur einen Befehl abarbeiten kann bis der nächste Interrupt aktiviert wird. Die Interruptroutine stellt dann entweder die Register dar und wartet auf einen Tastendruck (Einzelschrittmodus) oder vergleicht den Inhalt des Adresszähler zum Zeitpunkt des Interrupt mit dem gesetzten Unterbrechungspunkt (Unterbrechungsmodus).
Damit erklärt sich auch das wesentlich langsamere Abarbeiten eines Programmes im Unterbrechungsmodus dadurch, daß nach jedem abgearbeiteten Befehl des Programmes die Interruptroutine abgearbeitet werden muß.

T.EX.AS.

Wenn Sie mit diesem Monitor unsere Hilfsroutine aufgerufen haben, springt der Monitor in den BASIC Modus zurück. Die neue Interruptroutine abeitet aber korrekt solange der Monitor nicht neu gestartet wird.
Das liegt daran, daß der Monitor den Interruptvektor für eigene Zwecke benutzt.

Fragen zu 2.1.

1.) Welche drei Interruptmöglichkeiten für 65xx Prozessoren kennen Sie ?

2.) Welche Funktion hat das I- Flag für die Interrupts ?

3.) Wie führt der C 64 den BRK Befehl und wie den IRQ aus ?

4.) Erklären Sie anhand des Programmiermodell die Funktion des RTI Befehl.

Antworten zu 2.1.

zu 1.) Die 65xx Prozessoren kennen den Reset, den NMI (nicht maskierbarer Interrupt) und den IRQ (Interrupt Request - Interruptanforderung) als Interruptmöglichkeiten. Dabei zählt der Reset nur bedingt als Interrupt, da das laufende Programm nach einem Reset in der Regel nicht fortgesetzt wird.

zu 2.) Das I- Flag wird durch eine Aktivierung des NMI und des IRQ gesetzt. Bei gesetztem I- Flag wird kein IRQ ausgeführt. Der BRK Befehl reagiert dagegen nicht auf das I- Flag. Der RTI Befehl löscht das I- Flag.

zu 3.) Da der BRK Befehl und der IRQ denselben Vektor an Adresse $FFFE benutzen, wird zunächst in beiden Fällen an Adresse $FF48 gesprungen. Nachdem in der dortigen Routine die Prozessorregister zwischengespeichert wurden, wird entweder der Breakvektor an Adresse $0316 oder der IRQ Vektor an Adresse $0314 zur Fortsetzung der Interruptroutine verwendet.

zu 4.) Der RTI Befehl veranlaßt die Befehlserkennung das zwischengespeicherte Statusregister vom Stack zu holen und dann die Rücksprungadresse vom Stack in den Adresszähler zu laden. Nachdem der Adresszähler dann incrementiert wird und somit auf den nächsten abzuarbeitenden Befehl zeigt, wird das I- Flag gelöscht und das unterbrochene Programm fortgesetzt.

2.2. Einsatz der Maschinensprache

Sie haben jetzt bis auf eine Ausnahme alle 65xx Befehle kennengelernt und sind in der Lage, kleinere Maschinenspracheroutinen eigenständig zu schreiben. Bisher sind Sie es allerdings gewohnt, die Routinen unter der Kontrolle eines Monitor zu starten und zu testen. Während der Programmentwicklung ist das auch praktisch und notwendig.
Nachdem eine Routine aber entwickelt, ausgetestet und für gut befunden ist, soll sie ihre Aufgabe ohne den Monitor erfüllen. Sie benötigen also einige Informationen, wie Sie Ihre Routinen zum Einsatz bringen. Zuvor sollen Sie aber noch den letzten Ihnen unbekannten 65xx Befehl kennenlernen: Den NOP Befehl. Das Kürzel steht für - No OPeration - und der Befehl bewirkt schlicht die Incrementierung des Adresszählers um eins. Die Befehlserkennung führt keine weitere Operation außer der Incrementierung durch.
Eine Anwendung für diesen auf den ersten Blick seltsam erscheinenden Befehl ist das Auffüllen überflüssiger Bytes. Dazu ein Beispiel:

```
$5000 A9 00     LDA #$00
$5002 8D 00 40  STA $4000
$5005 20 00 51  JSR $5100
```

In diesem Ausschnitt aus einem Programm soll das Byte $00 nicht mehr an Adresse $4000 abgelegt werden, sondern an der Zeropageadresse $0060. Zu diesem Zweck ist der Befehl an Adresse $5002 entsprechend zu ändern. Durch die bei der Änderung zu verwendende nullseitige Adressierung wird der Operand $40 für die absolute Adressierung überflüssig. Würde das Byte $40 aber so stehengelassen, so würde die Befehlserkennung es als RTI Befehl interpretieren. Andererseits ist es bei einem längeren Programm mit dem Monitor sehr mühsam, alle folgenden Befehle um ein Byte aufzurücken und eventuelle absolute Sprünge entsprechend zu korrigieren. Die einfachste Lösung dieses Problems ist, bei der Programmierung mit einem Monitor das Byte $40 durch das Byte $EA für den NOP Befehl zu ersetzen:

```
$5000 A9 00     LDA #$00
$5002 85 60     STA $60
$5004 EA        NOP
$5005 20 00 51  JSR $5100
```

Sie sehen, daß auf diese Weise der Befehl ohne weitere Programmänderung ausgetauscht werden konnte. Ich habe oben betont, daß diese Lösung bei der Programmierung mit einem Monitor die einfachste sei. Sie werden im zweiten Teil des Buches ein solches Problem mit einem Assembler noch viel einfacher lösen lernen.
Um jetzt Ihre Maschinenspracheroutinen einzusetzen, benötigen Sie folgende Informationen:
Sie müssen wissen, in welchem Speicherbereich Sie die Routinen arbeiten lassen können. Dazu können Sie zunächst die Abbildung im Anhang zu Rate ziehen. Sie sehen dort, daß der Speichérbereich von Adresse $C000 - $CFFF frei ist. In den meisten Fällen werden Maschinenprogramme für den C 64 in diesem Bereich abgelegt. Weiterhin steht Ihnen der gesamte BASIC Speicherplatz zur Verfügung. Dabei ist es allerdings erforderlich, dem Betriebssystem Ihres C 64 mitzuteilen daß der BASIC Speicherbereich eingeschränkt wird. Andernfalls könnte das Maschinenprogramm durch das BASIC Programm oder BASIC Daten zerstört werden. Zu diesem Zweck befindet sich in der Zeropage an Adresse $0037 - $0038 ein 16 Bit Zeiger auf das BASIC RAM Ende. Wollen Sie beispielsweise den Speicherbereich $9000 - $A000 für Maschinensprache reservieren, so sind folgende BASIC Befehle einzugeben:

```
POKE 55,0:POKE 56,9*16:NEW
```

Der NEW Befehl muß unbedingt gegeben werden, da das Betriebssystem durch diesen Befehl veranlaßt wird, weitere Zeiger auf das neue BASIC Ende zu setzen. Die POKE Befehle sind ohne den NEW Befehl unwirksam.
Ein weiterer in den meisten Fällen freier Speicherbereich ist der Cassettenpuffer an Adresse $033C - 03FB. Wenn Sie den Datenrecorder nicht benutzen, können Sie in diesem Bereich ohne weiteres Daten oder Routinen ablegen.
Weiterhin benötigen Sie für Ihre Routinen freie Plätze in der Zeropage. Allerdings wird die gesamte Zeropage bis auf vier

Bytes an Adresse $00FB - $00FE vom Betriebssystem benutzt. Viele der vom Betriebssystem benutzten Zeropageplätze stellen Zeiger dar, die vor der Rückkehr in den BASIC Modus ihren alten Wert zurückerhalten müssen. Andernfalls arbeitet das Betriebssystem im BASIC Modus nicht mehr korrekt.
Solange Sie in Ihren Maschinenspracheroutinen keine Betriebssystemroutinen aufrufen, können Sie den Arithmetikbereich des Betriebssystem an Adresse $0057 - $0070 in der Zeropage benutzen. Wenn Sie Betriebssystemroutinen aufrufen, müssen Sie damit rechenen, daß dieser Bereich verändert wird. Es ist in diesem Fall also nicht sinnvoll, Daten oder feststehende Zeiger in diesem Bereich abzulegen. Eine genaue Auflistung der Zeropagebelegung durch das Betriebssystem finden Sie in 2. der Literaturhinweise.
Ein Maschinenprogramm wird durch Angabe der Sekundäradresse 1 von Diskette geladen: LOAD "name",8,1 oder LOAD "name",1,1 vom Datenrecorder. Die Angabe der Sekundäradresse 1 veranlaßt das Betriebssystem, das Programm ab der Adresse zu laden, von der es abgespeichert wurde. Nach dem Laden eines Maschinenprogrammes muß der NEW oder CLR Befehl gegeben werden. Wird ein Maschinenprogramm innerhalb eines BASIC Programmes nachgeladen, so startet das BASIC Programm nach dem Ladevorgang wieder mit der ersten BASIC Zeile. Sie werden hierzu gleich ein Beispiel finden.
Zum Starten eines Maschinenprogrammes aus dem BASIC Modus gibt es zwei Befehle. Zum ersten den USR (X) Befehl. Dieser Befehl bewirkt den Aufruf einer Maschinenspracheroutine, auf deren Adresse der 16 Bit Zeiger an Adresse $0311 -0312 zeigt. Der Wert des Argumentes X in Klammern nach dem USR Befehl wird zuvor noch in das Fließkommaformat übersetzt und in den Fließkommaaccu des Betriebssystem gebracht. Das Argument kann so durch das Maschinenprogramm weiterverarbeitet werden. Nachdem die Routine abgearbeitet ist, wird das Ergebnis aus dem Fließkommaaccu wie bei den anderen BASIC Funktionen (SQR (X) - SIN (C) usw.) ausgegeben. Mit der entsprechenden Maschinenspracheroutine ist der USR (X) Befehl also genauso wie die anderen BASIC Funktionen einzusetzen. Da wir uns in diesem Buch nicht mit der Fließkommaarithmetik beschäftigen, verzichten wir auf ein Beispiel des USR (C) Befehls. Sie finden dieses Thema in 1 und 2 der Literaturhinweise ausführlich behandelt.

Die zweite Möglichkeit, ein Maschinenprogramm zu starten, ist durch den SYS X Befehl gegeben. X gibt die Adresse des Maschinenprogrammes im Speicher an. Im folgenden Beispiel finden Sie die Anwendung des oben geschilderten:

```
10 IF A = 0 THEN  POKE  55,0:POKE  56,5*16:CLR:A  =  1:LOAD
"ZEHN 2",8,1
20 SYS 5*4096
```

Dieses kleine BASIC Programm setzt das BASIC Ende auf $5000, lädt das Beispielprogramm "ZEHN 2" von der Diskette und startet es mit dem SYS Befehl. Das Programm ZEHN 2 entspricht exakt dem Programm ZEHN, nur wurde der BRK Befehl an Adresse $5014 durch den RTS Befehl ersetzt. Sie müssen diese Änderung durchführen, da nach der Initialisierung mit der Routine ab Adresse $5000 sonst die Break Routine des Betriebssystems aufgerufen würde. Diese Routine setzt den Interruptzeiger aber wieder auf den alten Wert, womit die Wirkung der Routine ab Adresse $5000 aufgehoben ist. Die geänderte Routine ZEHN 2 muß natürlich auch auf Diskette gespeichert werden.
Sie können anstelle von 5 * 16 und 5 * 4096 natürlich auch die ausgerechneten Werte verwenden. Die Angabe in dieser Schreibweise ist allerdings einfacher und übersichtlicher.

Damit sind Sie am Ende des ersten Teiles dieses Buches angelangt. Sie kennen jetzt alle 65xx Befehle und Adressierungsarten, können Zahlensysteme umrechnen und in verschiedenen Zahlensystemen addieren und subtrahieren. Außerdem kennen Sie die für Sie als Programmierer wichtige Struktur des C 64. So ganz nebenbei haben Sie auch alle Monitorfunktionen kennen- und anwendengelernt.
Sie sollten, um dieses Wissen zu festigen, jetzt einige Maschinenspracheroutinen selbständig mit Ihrem Monitor entwickeln und austesten. Wie wäre es beispielsweise mit einem Menüprogramm, welches als Menüpunkte alle bisherigen Beispielprogramme zur Auswahl anbietet ? Dazu sollte ein richtiges Menü auf dem Bildschirm erscheinen (Betriebssystemroutine $AB1E). Die Programme müssen dann natürlich an andere Adressen verlegt werden.
Wenn Sie andere oder weitere Projekte verwirklichen wollen, so lassen Sie sich nicht davon abhalten. Desto mehr Sie sich

eigenständig mit der Materie auseinandersetzen, desto sicherer werden Sie die Maschinensprache beherrschen. Sie sollten allerdings mit dem Monitor keine Programme die länger als etwa 256 Bytes sind, entwickeln. Das ist selbstverständlich möglich, aber mit einem Monitor unnötig mühsam. Dazu lernen Sie im folgenden Teil den Assembler kennen.

Für die Besprechung des Assemblers gehe ich davon aus, daß alle 65xx Befehle und Adressierungen bekannt sind und der Monitor für Sie ein vertrautes Werkzeug darstellt.

Fragen zu 2.2.

1.) Ist es möglich, mehrere NOP Befehle unmittelbar hintereinander zu schreiben ?

2.) Was müssen Sie beachten, wenn Sie den BASIC Speicherbereich für Maschinenspracheprogramme nutzen wollen ?

3.) Welche vier Zeropageadressen werden nicht vom Betriebssystem des C 64 benutzt ?

Antworten zu 2.2.

zu 1.) Das ist ohne weiteres möglich. Für jeden NOP Befehl wird dann einfach durch die Befehlserkennung der Adresszähler erhöht.

zu 2.) Der Bereich, welcher für die Maschinensprache reserviert werden soll, muß vor dem Überschreiben durch BASIC Programme oder Daten geschützt werden. Dies erfolgt durch Ändern des BASIC-Ende-Zeigers an Adresse $0311 - $0312. Nach der Änderung muß der CLR oder NEW Befehl gegeben werden.

zu 3.) Die Zeropageadressen $00FB - $00FE werden nicht vom C 64 Betriebssystem benutzt. Diese Adressen können also für Daten oder feste Zeiger benutzt werden.

KAPITEL 1: ASSEMBLER - AUFGABEN UND EINSATZ

1.1. Editor / Labels / Variable

Die Programmierung in Assembler unterscheidet sich grundlegend von der im ersten Teil des Buches ausgeübten Maschinensprache- Programmierung. Das muß so gesehen werden obwohl das Ergebnis der Programmierung bei beiden Verfahren der Maschinensprache- Programmierung zu mehr oder weniger identischem Code führt.
Der entscheidende Unterschied besteht darin, daß das eigentliche Maschinenprogramm (der Objektcode) bei der Assemblerprogammierung nicht direkt geschrieben wird, sondern zuvor ein sogenanntes Source File (Quelldatei) erstellt wird. Dieses Source File enthält zum einen die 65xx Befehle des erstellten Programmes in der Form, wie Sie das von den bisherigen Beispielprogrammen her kennen, in Mnemonics. Das Programm wird also nicht als Hexcodes, sondern in symbolischer Darstellung eingegeben. Desweiteren enthält das Source File Befehle für den Assembler.
Das Programm Assembler ist nämlich für die Übersetzung des Source Files in Objektcode (endgültiges Maschinenprogramm) zuständig. Die Assemblerbefehle innerhalb des Source Files nehmen Einfluß auf diese Übersetzung. Diese Einflußnahme ist sehr wichtig und nützlich. Mit den Möglichkeiten, die sich daraus ergeben, werden wir uns in diesem Buchteil beschäftigen.

Editor
Zur Erstellung des Source Files ist ein Editor erforderlich. Mit Hilfe des Editors läßt sich das Source File eingeben und ändern. Einen solchen Editor enthält das C 64 Betriebssystem bereits zur Eingabe und Änderung von BASIC Programmen.
Viele Assembler machen sich diese Tatsache zunutze und akzeptieren ein Source File, welches mit dem BASIC Editor wie ein BASIC Programm mit Zeilennummern eingegeben wurde. Der Assembler benötigt dann keinen eigenen Editor und es können

Befehlserweiterungen (Simons BASIC, EXBASIC usw.) für den BASIC Editor mitbenutzt werden. Funktionen wie Autonumber, Renumber, Find, Erase usw. sind auch für die Erstellung eines Source Files sehr nützlich.
Von den vorgestellten drei Assemblern erwarten PROFIASS und T.EX.AS. das Source File als mit dem C 64 BASIC Editor erstellt. Befehlserweiterungen sind hier also gut einzusetzen.
Der Assembler MAE 64 beinhaltet einen eigenen Editor. Dieser Editor ähnelt in seiner Anwendung dem BASIC Editor. Es werden ebenfalls Zeilennummern eingegeben und die Editierung ist bildschirmorientiert. Das heißt, daß Sie wie mit dem BASIC Editor jede Zeile auf dem Bildschirm als Eingabezeile verwenden können. Der MAE 64 Editor hat einige Funktionen wie Autonumber, Erase oder Find bereits eingebaut. Wir werden den MAE 64 Editor im folgenden nur insoweit erklären wie das zur Ausführung der verwendeten Beispiele erforderlich ist. Die Benutzung der besonderen Funktionen dieses Editors entnehmen Sie dann bitte Ihrem Handbuch.

Labels
Ein Label ist einfach eine Speicheradresse, die einem bestimmten Namen zugeordnet ist. Dieser Name darf eine beliebige Buchstabenfolge haben. Die Länge des Namens ist in der Regel auf eine bestimmte Anzahl Buchstaben begrenzt. Meist 6 oder 8 Buchstaben.
Nehmen wir einmal an, dem Namen "LOESCH" sei die Speicheradresse $5000 zugeordnet. Bei einem Sprungbefehl in einem Programm darf dann anstelle der Adresse $5000 das Label LOESCH angegeben werden. Anstelle von:

```
JSR $5000
```

dürfen Sie im Source File schreiben:

```
JSR LOESCH
```

Wenn der Assembler das Source File in Objektcode übersetzt, so setzt er anstelle von JSR das Byte $20 und anstelle von LOESCH die Bytes $00 und $50.
Die Zuordnung eines Labels zu einer Speicheradresse erfolgt,

indem der Labelname im Source File vor den Befehl geschrieben wird, dessen Adresse das Label erhalten soll:

```
LOESCH  LDA #$00
...
...
RTS
JSR LOESCH
```

Sie sehen, daß das Label eigentlich nicht einer bestimmten Adresse zugeordnet ist, sondern einer bestimmten Stelle im Programm. Die Adresse dieser bestimmten Stelle ergibt sich aus der Länge der vorhergehenden Befehle und der Startadresse des Programmes.
Die Startadresse eines Programmes wird zu Beginn des Source Files mit einem Assemblerbefehl angegeben. Wenn das folgende Maschinenprogramm im Source File nur Labels zur Adressenangabe benutzt, so kann das Programm einfach durch Angabe der Startadresse in jeden beliebigen Speicherbereich assembliert werden. Werden Befehle gelöscht oder hinzugefügt, so müssen keine Sprungadressen umgerechnet werden, da den Labels ja automatisch neue Adressen zugeordnet werden.
Sie sehen, daß allein durch die Einführung der Labels eine ganz erhebliche Erleichterung der Programmierung gegeben ist. Um das zu verdeutlichen, sehen Sie unten das Listing des Beispielprogrammes ZWEI (Seite 48) als Source File abgebildet:

```
Zeilennummer  Labels  Mnemonics

          10  Startadresse angeben
          20          LDY #$00
          30  VERZOE  JSR XDECR
          40          INY
          50          BMI END
          60          JMP VERZOE
          70  XDECR   LDX #$FF
          80  XDECR2  DEX
          90          BNE XDECR2
         100          RTS
         110  END     BRK
```

Die Ziffern zu Beginn jeder Zeile sind die Zeilennummern. Diese Zeilennummern dienen nur dem Editor und haben mit dem Assembler oder dem zu erzeugenden Objektcode nichts zu tun. Genau wie in BASIC können Zeilen eingefügt, gelöscht oder der Zeilenabstand anders gewählt werden.
In Zeile 10 wird die Startadresse angegeben. Das erfolgt bei den verschiedenen Assemblern mit verschiedenen Befehlen. Ich werde für jeden der drei Assembler die speziellen Befehle immer, so wie im ersten Teil des Buches auch, gesondert angeben. Die Festlegung der Labels und die Mnemonics ist bei den meisten Assemblern gleich.
Gehen Sie anhand des obigen Listings einen Programmlauf im Kopf durch. Überprüfen Sie durch Überlegung, ob das Programm tatsächlich durch Festlegung der Startadresse in jeden beliebigen Speicherbereich gebracht werden kann.

Variable

Die Variablen sind grundsätzlich mit den Labels vergleichbar. Auch die Variablen sind Buchstabenfolgen, denen ein bestimmter Wert zugeordnet ist. Die Zuordung erfolgt allerdings direkt durch einen Assemblerbefehl. Damit enthält eine Variable immer einen bestimmten Wert, der von der Startadresse unabhängig ist. Variablen können Werte von $00 bis $FFFF zugeordnet werden. Die entsprechenden Assemblerbefehle für die Zuordnung werden im einzelnen nachfolgend besprochen.
Wenn einem Label oder einer Variablen einmal ein Wert zugeordnet wurde, so unterscheiden sich Labels und Variable in der Handhabung nicht weiter. Ein Label kann also auch als Wert und eine Variable als Sprungadresse eingesetzt werden.
Variablen werden verwendet, um bestimmte Werte zu Beginn des Source Files festzulegen. Wenn diese Werte sich dann ändern sollen, so müssen nur die Variablen zu Beginn des Source Files geändert werden. Es ist auch sinnvoll, verwendete Betriebssystemroutinen mit einem Namen aufzurufen. Das wird möglich, indem die Adresse der Betriebssystemroutine einer Variablen zugeordnet wird. Das gilt auch für feststehende Vektoren usw.

Ein in dieser Weise aufgebautes Source File ist sehr flexibel und läßt sich auch meist leicht an andere 65xx Systeme anpassen.

Bevor Sie in diesem Teil des Buches den gelernten Stoff in die Praxis umsetzen können, müssen wir noch einige Theorie bearbeiten. Ab Absatz 1.3. werden Sie dann aber wieder ständig die Praxis mit Ihrem Assembler einüben. Ich bitte bis dahin um Geduld und unverminderte Aufmerksamkeit.

PROFIASS
Startadresse festlegen: *= Startadresse
Variable festlegen: Name = Wert
Labels oder Variable dürfen maximal 8 Zeichen haben und müssen mit einem Buchstaben beginnen.
Die Zeilen des Source Files werden mit dem BASIC Editor wie BASIC Zeilen eingegeben. Die erste Zeile eines Source Files muß den Assembler mit dem SYS Befehl aufrufen. Dadurch wird die Assemblierung mit dem RUN Befehl gestartet. Die ersten Zeilen müssen wie folgt lauten, wenn das Programm ab Adresse $5000 starten soll:

```
10 SYS 8 * 4096
20 *= $5000
30 ...
```

T.EX.AS.
Startadresse festlegen: .BA Startadresse
Variable festlegen: Name = Wert
oder: Name .DL Wert
Die Zeilen des Source Files werden mit dem BASIC Editor wie BASIC Zeilen eingegeben. Zu Beginn jeder Zeile muß ein REM Befehl oder ein Anführungszeichen stehen. Soll ein Programm ab Adresse $5000 starten, so müssen die ersten Zeilen wie folgt aussehen:

```
10 ".BA $5000
20 "...
```

Die Variablen- und Labelnamen müssen unmittelbar hinter dem Anführungszeichen beginnen. Andernfalls werden diese nicht richtig gelesen.

Die Assemblierung wird bei gestartetem Monitor mit dem Befehl .AS eingeleitet.

MAE 64
Startadresse festlegen: .BA
    Variable festlegen: Name .DE Wert
Die Namen von Labels und Variablen dürfen maximal 10 Zeichen haben und müssen mit einem Buchstaben beginnen.
Die Zeilen des Source Files werden mit einem eigenen Editor eingegeben. Der Editor erwartet wie der BASIC Editor die Angabe von Zeilennummern. Ein Autonumber läßt sich mit folgendem Befehl einstellen:

```
AUTO Zeilenschrittweite
```

Die erste Zeilennummer muß eingegeben werden. Ab dieser Zeile wird dann die jeweils im Abstand der vorgegebenen Schrittweite folgende Zeilennummer automatisch generiert. Durch Eingabe von // hinter einer generierten Zeilennummer wird die Autonumber Funktion abgeschaltet.
Soll ein Programm ab Adresse $5000 starten, so müssen die Zeilen wie folgt aussehen:

```
10 .BA $5000
20 ...
```

Fragen zu 1.1.

1.) Worin liegt der entscheidende Unterschied zwischen der Programmierung in Maschinensprache wie im ersten Teil dieses Buches beschrieben und der Programmierung mit Hilfe des Assemblers ?

2.) Wozu dient ein Editor ?

3.) Welche Bedeutung haben die Zeilennummern in einem Source File ?

4.) Welchen besonderen Vorteil bietet ein Source File, in welchem Sprungadressen nur als Labels oder Variable erscheinen ?

5.) Worin unterscheiden sich Labels und Variable ?

Antworten zu 1.1.

zu 1.) Dadurch, daß in Assembler ein Source File in den Objektcode übersetzt wird, gestaltet sich die Programmierung wesentlich komfortabler und flexibler, ohne daß ein Vorteil aufgegeben werden muß.

zu 2.) In diesem Fall dient der Editor zur Erstellung und Korrektur des Soure Files. Allgemein dient ein Editor zur Erstellung eines Textes, wobei auch ein Programm in einer beliebigen Programmiersprache als Text betrachtet werden kann.

zu 3.) Die Zeilennummern in einem Source File dienen nur dem Editor zur Festlegung der Reihenfolge der Befehlszeilen. Durch die Zeilennummern können wie von BASIC her gewohnt leicht Änderungen vorgenommen werden.

zu 4.) Wenn alle Sprungadressen in einem Source File durch Labels oder Variable festgelegt sind, so ist nur die Angabe der Startadresse mit einem Assemblerbefehl erforderlich, um den Objektkode für einen beliebigen Speicherbereich zu assemblieren.

zu 5.) Labels und Variable unterscheiden sich nur in der Zuordnung einer Adresse oder eines Wertes.
Dem Label wird eine Adresse zugeordnet, die einer bestimmten Stelle im Programm entspricht. Dadurch ist die zuzuordnende Adresse von der tatsächlichen Adresslage dieser Programmstelle abhängig.
Der Variablen wird meist zu Beginn des Source Files ein bestimmter Wert direkt zugeordnet. Dieser Wert ist von der Adresslage des Programmes unabhängig.

1.2. Adressierung / Operatoren / Kommentare

Die im Source File zu verwendenden Mnemonics entsprechen den bisher verwendeten. Glücklicherweise halten sich alle mir bekannten Assembler an diese vom 65xx Prozessorhersteller festgelegten Befehlssymbole.

Adressierung

Das gilt bis auf eine Ausnahme auch für die Festlegung der Art der Adressierung. Die Ausnahme bildet die Zeropageadressierung. Von den drei vorgestellten Assemblern erkennen PROFIASS und T.EX.AS. die Zeropageadressierung an dem Wert des / der Operanden. Liegt der Wert über $FF, so wird absolute Adressierung festgelegt, andernfalls Zeropage Adressierung. Es genügt also, im Souce File einfach:

```
LDA $60
```

für Zeropage Adressierung oder:

```
LDA $5000
```

für absolute Adressierung zu schreiben. Der Assembler MAE 64 erwartet die Zeropageadressierung wie folgt:

```
LDA *$60
```

Der Stern kennzeichnet also gesondert die Zeropage Adressierung. Das gilt auch für den Fall, daß der Operand durch eine Variable oder ein Label festgelegt wird. Dazu ein Beispiel:

```
10 ADRESSE1 .DE $60
20 ADRESSE2 .DE $5000
30 LDA *ADRESSE1
40 LDA ADRESSE2
```

Die Symbole zur Kennzeichnung der anderen Adressierungen finden Sie im Anhang unter 65xx Adressierungen. Sie haben alle Symbole für die Adressierung auch schon im ersten Teil

dieses Buches an den Assemblerlistings der Beispielprogramme kennengelernt.

Es ist noch darauf hinzuweisen, daß die meisten Assembler Zahlenangaben in dezimaler, hexadezimaler oder binärer Form erlauben. Zur Unterscheidung binärer und hexadezimaler Angaben von dezimalen Angaben dienen wie bereits bekannt das % und das $ Zeichen. Sie können also einen Operanden in einem der Zahlensysteme angeben:

```
LDA 255
LDA $FF
LDA %11111111
```

Diese drei Befehle werden alle zu den Bytes $A5 $FF assembliert, da der Wert des Operanden identisch ist.

Operatoren

Die Operatoren sind nicht mit der Operanden zu verwechseln. Genauer bezeichnet handelt es sich um arithmetische Operatoren. Diese Operatoren dienen dazu, den Wert eines Operanden oder einer Variablen erst ausrechnen zu lassen. Die meisten Assembler kennen die Operatoren + und -. Dazu einige Beispiele:

```
10 TEST = $AA + 3
20 TEST2 = TEST + %111010
30 LDA TEST
40 LDY TEST + 1
50 STA TEST2 - 1
60 STY TEST2
```

Die verschiedenen Assembler kennen zusätzlich zu den Operatoren + und - noch weitere Operatoren. Für die drei besprochenen Assembler werde ich diese Operatoren gesondert beschreiben.

Kommentare

Auch die Möglichkeit, die Maschinen- und Assemblerbefehle durch Kommentare innerhalb des Source Files zu erklären, kennen Sie bereits aus den Beispielprogrammen des ersten Teiles dieses Buches. Sobald in einer Source File Zeile ein

Semikolon steht, betrachtet der Assembler bei der Übersetzung in Objektcode alles nach dem Semikolon in dieser Zeile als Kommentar. Der Kommentar wird dann vom Assembler nicht weiter beachtet.

Sie kennen diese Möglichkeit auch bereits von BASIC her. Dort erlaubt der REM Befehl die Eingabe von Kommentaren. In BASIC verbrauchen solche Kommentare allerdings Programmspeicherplatz, während die Kommentare keinen Einfluß auf die Länge des durch die Assemblierung erzeugten Objektcode haben. Die Kommentare werden bei der Ausgabe eines Assemblerlistings meist auch sehr übersichtlich ausgegeben.

Um so ausführlicher und vor allem einsichtiger Sie Ihre Source Files kommentieren, um so wertvoller wird die Routine für Sie und andere. Sie werden schon nach einigen Tagen Programmierpause mit Ihren eigenen unkommentierten Routinen nur sehr schwer wieder zurechtkommen. Gerade bei der Maschinensprache- Programmierung ist eine gute Programmdokumentation elementar wichtig. Wir werden das in einem eigenen Kapitel noch genauer besprechen.

Nachfolgend finden Sie die drei Assembler im einzelnen zu den vorstehenden Funktionen besprochen. Der nächste Absatz wird dann nach ein wenig Theorie endlich alles bisher Besprochene in der Praxis vorstellen.

PROFIASS

Dieser Assembler kennt besonders viele Operatoren. Sie sehen alle erlaubten Operatoren nachfolgend zusammengefaßt und erklärt:

+ Werte addieren
- Werte subtrahieren
* Werte multiplizieren
/ Werte dividieren
! Werte ODER verknüpfen
& Werte UND verknüpfen
↑ Werte exclusiv ODER verknüpfen

> Verschiebung des Wertes links von dem > Zeichen um die Anzahl Stellen nach rechts, die rechts von dem > Zeichen festgelegt sind.

```
Beispiel:   LDA #%00010000 > 4 ;ergibt
            LDA #%00000001
```

< Verschiebung des Wertes links von dem < Zeichen um die Anzahl Stellen nach links, die rechts von dem < Zeichen festgelegt sind.

```
Beispiel:   LDA #%00000001 < 7 ;ergibt
            LDA #%10000000
```

Es können mehrere Operatoren zur Berechnung eines Ausdruckes verwendet werden. Die Berechnung erfolgt dann von links nach rechts. Diese Reihenfolge der Berechnung kann durch Setzen von Klammern noch verändert werden. Die Argumente und Ergebnisse der Berechnungen dürfen bis zu 16 Bit haben. Andernfalls wird eine Fehlermeldung ausgegeben.

Um Ihnen diese Möglichkeiten nicht als zu theoretisch erscheinen zu lassen, will ich Ihnen eine praktische Anwendung vorstellen:

Sie wollen die hochauflösende Grafik des C 64 benutzen. In Ihrem Programm muß die Startadresse der Grafikseite festgelegt werden. Zu diesem Zweck benötigen Sie einige

Informationen: Die Bits der Grafikstartadresse sind auf zwei Speicherstellen (Byte 1 / Byte 2) in Ihrem C 64 verteilt, wobei diese Bits nur die drei höchstwertigen Bits der Grafikstartadresse darstellen. Diese Bits stehen nicht stellenrichtig in den Speicheradressen. Die Grafikstartadresse setzt sich wie folgt zusammen:

```
Adressbit: F E D C B A 9 8 7 6 5 4 3 2 1 0

           B  B
           Y  Y   niederwertige Adressbits
           T  T   für Grafikstartadresse = 0
           E  E
           1  2
```

Von Byte 1 aus Adresse $DD00 stellen das 7. und 8. Bit die Bits F und E der Grafikstartadresse. Die Bits sind zusätzlich noch invertiert gesetzt. Ein 1 Bit ist also als 0 und ein 0 Bit als 1 zu verstehen. Von Byte 2 aus Adresse $D018 stellt das 3. Bit das Bit D der Grafikstartadresse.

Zusätzlich zur Grafikstartadresse muß noch die Videostartadresse festgelegt werden. Die Adressbits F und E der Videostartadresse entsprechen denen der Grafikstartadresse. Die Adressbits D, C, B und A der Videostartadresse entsprechen dem 5. 6. 7. und 8. Bit des Bytes 2.

Sie sehen, daß einige Verschiebungen und Umrechnungen erforderlich sind, um die gewünschten Startadressen in die entsprechenden Register zu bringen. In dem Source File zu dem Grafikprogramm soll es aber möglich sein, diese Startadressen direkt ohne Umrechnung einer Assemblervariablen zuzuordnen. Das wird bei dem Assembler PROFIASS durch richtige Ausnutzung der Operatoren wie folgt möglich:

```
 10 GRAPH = $A000  ;Grafikstartadresse
 20 VIDEO = $8C00  ;Videostartadresse
 30 LDA $DD00      ;Byte 1 holen
 40 AND #%11111100 ;Bit F und E isolieren
 50 ORA #(GRAPH > 8 & %11000000 ^ %11000000) > 6
                   ;Bit F und E setzen
 60 STA $DD00      ;Byte 1 einsetzen
 70 LDA $D018      ;Byte 2 holen
 80 AND #%00000111 ;Bit D, C, B und A isolieren
 90 ORA #(GRAPH > 10 & %00001111)!(VIDEO > 10 &  %00001111  <
4)
                   ;Bits D, C, B und A setzen
100 STA $D018      ;Byte 2 einsetzen
```

In diesem Beispiel soll die Grafikstartadresse auf $A000 und die Videostartadresse auf $8C00 festgelegt werden. Die Festlegung erfolgt in den Zeilen 10 und 20. Die Variable GRAPH entspricht der Grafikstartadresse und die Variable VIDEO der Videostartadresse.

Die Bits F und E für beide Startadressen werden in Zeile 50 gesetzt. Dort werden zuerst die Adressbits der in der Variablen GRAPH festgelegten Adresse in das Lowbyte an die 7. und 8. Stelle geschoben (GRAPH > 8) und dann isoliert (& %11000000). Daraufhin werden die Adressbits invertiert (^ %11000000) und an die richtige Stelle für Byte 1 geschoben (> 6). Die Klammern sind erforderlich, damit der gesamte Ausdruck um 6 Stellen verschoben wird. Mit dem Beispielwert für GRAPH erfolgen also diese Operationen:

```
GRAPH = %10100000 00000000   > 8
      = %         10100000   & %11000000
      = %         10000000   ^ %11000000
      = %         01000000   > 6
      = %         00000001
```

Die Zeile 50 entspricht demnach bei einem Wert der Variablen GRAPH von $A000 der Zeile:

```
ORA #$01
```

In Zeile 90 werden die Adressbits D, C, B, A der

Videostartadresse und das Adressbit D der Grafikstartadresse entsprechend den Variablenwerten festgelegt. Sehen wir uns die Operationen anhand der Beispielwerte an:

```
GRAPH = %10100000 00000000   > 10
      = %         00101000   & %00001111
      = %         00001000   ! mit Klammer
VIDEO = %10001100 00000000   > 10
      = %         00100011   & %00001111
      = %         00000011   < 4
      = %         00110000   ! mit Klammer
      = %         00111000
```

Die Zeile 90 entspricht demnach bei einem Wert der Variablen GRAPH von $A000 und der Variablen VIDEO von $8C00 der Zeile:

```
ORA #$38
```

Sie sehen, daß Sie durch die richtige Verwendung der Operatoren nahezu jeden Zusammenhang rein symbolisch darstellen können. Wenn Sie das auch in Ihren Source Files verwirklichen, werden Sie optimal flexible Routinen erstellen.

Wie die meisten Assembler, kennt auch PROFIASS die Möglichkeit, von einem 16 Bit Ausdruck nur das High- oder Lowbyte auszuwählen. Dazu ein Beispiel:

```
10 ADRESSE = $5000
20 LDA #<ADRESSE ;Lowbyte von ADRESSE in ACCU
30 LDY #>ADRESSE ;Highbyte von ADRESSE in Y- Reg.
```

Die Zeilen 20 und 30 entsprechen folgenden Zeilen:

```
20 LDA #$00 ;Lowbyte
30 LDY #$50 ;Highbyte
```

Sollten Sie für eine spezielle Anwendung die Assemblierung eines Befehls mit absoluter Adressierung benötigen, obwohl der Wert des Operanden kleiner 256 ist, so können Sie mit dem ! Zeichen die absolute Adressierung erzwingen. Dazu ein

Beispiel:

```
LDA $60  ergibt die Bytes $A5 $60
LDA !$60 ergibt die Bytes $AD $60 $00
```

T.EX.AS.
Dieser Assembler erlaubt die Operatoren + und - in der bereits beschriebenen Weise.
Wie die meisten Assembler kennt auch T.EX.AS. die Möglichkeit, von einem 16 Bit Ausdruck nur das High- oder Lowbyte auszuwählen. Dazu ein Beispiel:

```
10 ADRESSE = $5000
20 LDA #<ADRESSE ;Lowbyte von ADRESSE in ACCU
30 LDY #>ADRESSE ;Highbyte von ADRESSE in Y- Reg.
```

Die Zeilen 20 und 30 entsprechen folgenden Zeilen:

```
20 LDA #$00 ;Lowbyte
30 LDY #$50 ;Highbyte
```

Sollten Sie für eine spezielle Anwendung die Assemblierung eines Befehls mit absoluter Adressierung benötigen, obwohl der Wert des Operanden kleiner 256 ist, so können Sie mit dem Klammeraffen die absolute Adressierung erzwingen. Dazu ein Beispiel:

```
LDA  $60  ergibt die Bytes $A5 $60
LDA @$60 ergibt die Bytes $AD $60 $00
```

T.EX.AS. verlangt die Angabe der Accumulator Adressierung ohne besondere Kennzeichnung. Um den ACCU beispielsweise nach links zu verschieben ist also im Source File folgende Zeile einzugeben:

```
ASL
```

MAE 64
Dieser Assembler erlaubt die Operatoren + und - in der

bereits beschriebenen Weise.
Wie die meisten Assembler kennt auch MAE 64 die Möglichkeit, von einem 16 Bit Ausdruck nur das High- oder Lowbyte auszuwählen. Dazu ein Beispiel:

```
10 ADRESSE .DE $5000
20 LDA #L,ADRESSE ;Lowbyte von ADRESSE in ACCU
30 LDY #H,ADRESSE ;Highbyte von ADRESSE in Y- Reg.
```

Die Zeilen 20 und 30 entsprechen folgenden Zeilen:

```
20 LDA #$00 ;Lowbyte
30 LDY #$50 ;Highbyte
```

Fragen zu 1.2.

1.) Wie schreiben Sie in Assembler den Befehl: Lade den ACCU indirekt Y- indiziert ?

2.) Welche Adressierungen werden in Assembler nicht besonders gekennzeichnet ?

3.) Was verstehen Sie unter (arithmetischen) Operatoren ?

4.) In einem Source File ist der Variablen BSHAUS der Wert $AB1E zugeordnet worden. Der Wert der Variablen soll an den Adressen $5000 und $5001 abgelegt werden. Welche Zeilen sind erforderlich, wenn die Operanden nur durch Variable angegeben werden sollen ? Außer BSHAUS ist nur eine weitere Variable erforderlich.

5.) Wie läßt sich ein Source File kommentieren ?

6.) Welchen Nachteil haben Kommentare in einem BASIC Programm gegenüber Kommentaren in einem Source File ?

Antworten zu 1.2.

zu 1.) LDA ($xx),Y

zu 2.) Folgende Adressierungen benötigen in Assembler keine besondere Kennzeichnung: Implizierte / Absolute / Relative / bei den meisten Assemblern auch die Zeropage Adressierung nicht (MAE 64 - durch * kennzeichnen).

zu 3.) Die drei vorgestellten Assembler kennen alle drei die Operatoren + und -. Mit Hilfe dieser Operatoren lassen sich Variable berechnen. Dadurch kann die Übersichtlichkeit und Flexibilität eines Source Files erheblich gesteigert werden.

zu 4.) Nachfolgend die Lösung:

```
10 BSHAUS  = $AB1E
20 ADRESSE = $5000
30 LDA #<BSHAUS
40 LDY #>BSHAUS
50 STA ADRESSE
60 STA ADRESSE + 1
```

Für den Assembler MAE 64 sind die entsprechenden Assemblerbefehle zu verwenden.

zu 5.) Bei der Übersetzung eines Source Files in Objektcode ignoriert der Assembler alle Zeichen nach einem Semikolon in einer Zeile. Kommentare können also in jeder Zeile nach einem Semikolon eingegeben werden.

zu 6.) Kommentare in einem BASIC Programm belegen Programmspeicherplatz. Kommentare in einem Source File haben keinerlei Einfluß auf die Länge des erzeugten Objektcodes.

1.3. Woher das Source File / Wohin der Objektcode

Bisher haben Sie schon einiges über die Übersetzung des Source Files in Objektcode durch den Assembler erfahren. Es fehlt allerdings noch die Information, wohin der erzeugte Objektcode gebracht und wie der Assembler im einzelnen zur Übersetzung des Source Files veranlaßt wird.
Zur Erzeugung des Objektcodes gibt es grundsätzlich drei Möglichkeiten:

1. Kein Objektcode

Die erste dieser Möglichkeiten besteht darin, keinen Objektcode zu erzeugen. Das dient zu dem Zweck, ein Source File vor der eigentlichen Assemblierung auf etwaige Eingabefehler zu überprüfen.
Wenn Sie in einem Source File beispielsweise LDZ statt LDA eingeben, so wird diese Eingabe zunächst so in das Source File übernommen. Bei der Assemblierung kann der Assembler diese Zeile dann allerdings nicht übersetzen, da der Befehl LDZ nicht existiert. Der Assembler wird darauf mit einer entsprechenden Fehlermeldung reagieren.
Das gilt für alle unmöglichen Anweisungen im Source File. Also auch für Assemblerbefehle. Die meisten Assembler bieten die Möglichkeit, durch einen Assemblerbefehl festzulegen, ob die Assemblierung nach einem Fehler abgebrochen oder nur eine Fehlermeldung ausgegeben, die Assemblierung aber fortgesetzt werden soll.
Die Möglichkeit, keinen Objektcode zu erzeugen, kann bei längeren Source Files Zeit sparen, da der Assembliervorgang ohne Objektcodeerzeugung weniger Zeit benötigt. Man spricht dann auch von einem Syntaxcheck.

2. Objektcode ins RAM

Diese Funktion bringt den erzeugten Objektcode direkt an die festgelegte Adresse im RAM. Das Maschinenprogramm (der Objektcode) kann dann sofort durch den Monitor oder den SYS Befehl gestartet werden.
Diese Möglichkeit ist für kürzere Routinen die einfachste.

3. Objektcode auf Diskette oder Cassette

Wie sich schon aus der Überschrift ergibt, wird der Objektcode mit dieser Möglichkeit direkt auf Diskette oder Cassette als Maschinenprogramm abgespeichert. Das entspricht der SAVE Funktion des Monitors.
Diese Möglichkeit empfiehlt sich, wenn die Routine in jedem Fall abgespeichert werden soll oder bei längeren Programmen.

Das Source File kann in der Regel von der Diskette durch den Assembler geladen werden, oder es wird aus dem Arbeitsspeicher assembliert. Bei längeren Programmen kann es leicht vorkommen, daß das Source File mehr Speicherplatz als vorhanden benötigt. In einem solchen Fall lassen sich mehrere Source Files miteinander verketten. Dadurch kann der Objektcode bis zu 64 K Byte lang sein.
Die meisten Assembler können aus dem Source File ein formatiertes Asemblerlisting erzeugen. Das erfolgt in der Regel parallel zur Assemblierung. Ein Assemblerlisting enthält alle Informationen des Source Files einschließlich Zeilennummern und Kommentare. Zusätzlich werden meist noch die Adressen und die Befehle in Hexdarstellung ausgegeben. Sie kennen diese Darstellungsform bereits aus dem ersten Teil dieses Buches. Das Assemblerlisting wird über den Bildschirm oder den Drucker ausgegeben.

Nachfolgend finden Sie die entsprechenden Befehle zu den besprochenen drei Assemblern. Anschließend sollen Sie dann ein Source File eingeben und assemblieren.

Durch den Assemblerbefehl .OPT wird die Ausgabe des Objektcodes und des Assemblerlistings gesteuert. Nach dem .OPT Befehl können mehrere Anweisungen zur Ausgabesteuerung folgen. In diesem Absatz sind folgende mögliche Anweisungen besprochen worden:

P Ausgabe des formatierten Assemblerlisting auf dem Bildschirm.
P# Ausgabe des formatierten Assemblerlisting auf das zuvor in BASIC eröffnete File. # stellt die Filenummer des eröffneten Files dar.
O0 Objektcode direkt in das RAM assemblieren.
O# Objektcode Ausgabe auf das zuvor in BASIC eröffnete File. # stellt die Filenummer des eröffneten Files dar.

Sehen wir uns dazu einige mögliche Befehlszeilen in einem Source File an:

```
10 OPEN 1,8,1,"NAME" :REM Objektcode File
20 OPEN 2,4 :REM Assemblerlisting File
30 SYS 8 * 4096 :REM Assemblerstart
40 .OPT P2,O0,O1  ;Ausgabe festlegen
50 *= $5000       ;Programmstart
```

Sie sehen, daß die im .OPT Befehl angesprochenen Files vor dem SYS Befehl zum Assemblerstart eröffnet werden müssen. Durch den .OPT Befehl wird hier folgendes veranlaßt:
Das formatierte Assemblerlisting wird auf den Bildschirm und den Drucker ausgegeben (die P# Anweisung enthält die P Anweisung) / Der Objektcode wird direkt in das RAM ab Adresse $5000 assembliert / Der Objektcode wird auch auf Diskette unter der Bezeichnung "NAME" abgespeichert.
Wird weder die O0 noch die O# Anweisung gegeben, so wird kein Objektcode erzeugt.
Das Source File muß immer im BASIC Arbeitsspeicher wie ein BASIC Programm stehen. Soll ein Source File verkettet werden, so muß der letzte Assemblerbefehl des ersten Files wie folgt lauten:

```
XXXX .FILE 8,"Name des nächsten Files"
```

Das nächste File kann auf dieselbe Weise ein weiteres Source File aufrufen usw. Das letzte File der Kette muß wieder das erste File auf folgende Weise aufrufen:

```
XXXX .END 8,"Name des ersten Files"
```

PROFIASS erkennt selbständig "fatale" Fehler, die eine Fortsetzung der Assemblierung nicht erlauben wie Floppyfehler oder den Sprung zu einer nicht existierenden Zeilennummer (darauf kommen wir noch) und Fehler, die eine Fortsetzung der Assemblierung erlauben. Für ein fehlerhaftes Mnemonic wird beispielsweise eine Fehlermeldung ausgegeben und das Mnemonic durch BRK ersetzt, die Assemblierung aber fortgesetzt.

T.EX.AS.

Durch den .AS Befehl kann bei diesem Assembler bestimmt werden, ob das Sourcefile aus dem BASIC Arbeitsspeicher oder von Diskette her assembliert wird:

```
.AS          ;Source File aus Arbeitsspeicher
.AS "Name"   ;Source File von Diskette
```

Wie bereits erwähnt, ist der .AS Befehl ein Monitorbefehl und muß somit bei gestartetem Monitor gegeben werden.
Folgende Assemblerbefehle steuern die Ausgabe des Objektcodes und des Assemblerlistings:

.OC Keine Objektcode Ausgabe in das RAM.
.OD "Name" Objektcode auf Diskette unter "Name" ablegen.
.LI Assemblerlisting auf Bildschirm ausgeben.
.PR Assemblerlisting auch auf Drucker ab dem .LI Befehl ausgeben.
.NL Ausgabe des Assemblerlistings beenden.
.CE Assemblierung bei Fehler fortsetzen.

Der .OC, .OD, .PR und der .CE Befehl müssen vor dem .BA Befehl zur Festlegung der Startadresse im Source File stehen. Wird der .OC Befehl nicht gegeben, so wird immer direkt in das RAM assembliert. Die Festlegung von Variablen muß nach dem .BA Befehl erfolgen.

Sehen wir uns dazu einmal den Beginn eines Source Files an:

```
10 " .CE            ;Assemblierung trotz Fehler
20 " .PR            ;Assemblerlisting auf Drucker
30 " .OD "NAME"     ;Objektcode auf Diskette
40 " .OC            ;kein Objektcode ins RAM
50 " .BA $5000      ;Startadresse
60 " .LI            ;Assemblerlisting auf Bildsch.
70 "SPEICHER = $AA ;Variable definieren
80 " .NL            ;Assemblerlisting ausschalten
```

Wird dieses Source File assembliert, so wird der Objektcode auf Diskette unter der Bezeichnung "NAME" abgespeichert / Im RAM wird kein Objektcode abgelegt / Die Startadresse ist $5000 / Die Zeile 70 wird als Assemblerlisting auf den Bildschirm und den Drucker ausgegeben / Die Assemblierung wird bei einem Fehler nicht abgebrochen.
Zu dem Assemblerlisting bleibt noch zu bemerken, daß das Listing nicht formatiert wird und keine Adressen ausgegeben werden. Das Assemblerlisting entspricht bei diesem Assembler dem Source File. Es ist daher meist besser, den Objektcode aus dem RAM mit dem Monitor auszulisten, da dann die Adressen mit ausgegeben werden.
Wenn ein Source File länger als der zur Verfügung stehende Speicherplatz ist, so lassen sich mehrere Sourcefiles von einem Steuerungsfile aus aufrufen. Das kann wie folgt aussehen:

```
10 " .BA $5000     ;Startadresse
20 "VARIAB1 = $AA ;Variablen definieren
20 " *FILE1        ;erstes Source File assembl.
30 " *FILE2        ;zweites Source File
usw.
```

Die Namen der aufgerufenen Files dürfen dabei keine Leerzeichen enthalten.
Es ist noch zu erwähnen, daß der Monitor, wenn er bereits eingeladen und mit dem .EN Befehl verlassen wurde, mit dem Befehl SYS 2 wieder warm gestartet werden kann. Wenn ein Source File aus dem Speicher mit dem .AS Befehl assembliert werden soll, muß der Monitor mit SYS 2 warm gestartet werden.

Er darf also nicht unmittelbar von Diskette geladen oder mit SYS 6 * 4096 kalt gestartet worden sein.

MAE 64

Wie bereits erwähnt, wird das Source File mit diesem Assembler mit einem eigenen Editor erstellt. Der Befehl zur Assemblierung wird vom Editor aus gegeben. Folgende Befehle zur Assemblierung sind möglich:

```
AS            ;Assemblierung ohne Listing
AS L          ;Assemblierung mit Listing
AS "Name"     ;Assemblierung von Disk ohne Listing
AS "Name" L   ;Assemblierung von Disk mit Listing
TO I          ;Druckerausgabe einschalten
TO P          ;Druckerausgabe ausschalten
```

Um das Assemblerlisting auf den Drucker auszugeben, ist vor dem AS L Befehl der Befehl TO I zu geben. Der Befehl TO P schaltet die Druckerausgabe dann wieder aus.
Für die zuvor allgemein besprochenen Funktionen sind folgende Assemblerbefehle im Source File zu benutzen:

.OS Objektcode im RAM ablegen.
.OC Befehl .OS aufheben.
.RS Relativladerverschiebung einschalten
.RC Relativladerverschiebung ausschalten
.LS Assemblerlisting ausgeben
.LC Befehl .LS aufheben
.CE Assemblierung bei leichten Fehlern nicht abbrechen.

Das Programmpaket des MAE 64 enthält einen sogenannten Relativlader. Mit Hilfe dieses Relativladers kann ein bereits für einen bestimmten Adressbereich assembliertes Maschinenprogramm in einen anderen Adressbereich im RAM gebracht werden.
Der Relativlader rechnet zu diesem Zweck alle absoluten Adressen, die als Labels oder mit dem .DI Befehl (interne Symbole) definiert wurden, um. Das verschobene Maschinenprogramm wird direkt in das RAM an die neue Adresse gebracht. Der MAE 64 besitzt keine Möglichkeit, ein Maschinenprogramm direkt auf Diskette oder Cassette zu

erzeugen.
Der Relativlader benötigt ein Relativlademodul, welches vom Assembler nach dem OUT "Name" Befehl auf Diskette erzeugt wird. Um den Objektcode in Form eines als Maschinenprogramm ladbaren Programmes auf Diskette zu bringen, gibt es zwei Möglichkeiten:
1. Die Assemblierung erfolgt direkt in das RAM (.OS) und der dort abgelegte Objektcode wird dann mit Hilfe eines Monitors oder einiger BASIC Zeilen als Maschinenprogramm abgespeichert. Das hat den Nachteil, daß der Objektcode nicht im Programm- Speicherbereich des MAE 64 oder des Monitor liegen kann.
2. Aus einem mit dem Assembler erzeugten Lademodul wird mit einem eigenen Programm der Objektcode auf Diskette erzeugt. Der Objektcode kann dann an jeder beliebigen Speicherstelle stehen. Sie finden dieses Hilfsprogramm nachfolgend abgedruckt.
Das Programm erzeugt aus einem Relativlademodul mit beliebigem Namen ein Maschinenprogramm mit beliebigem Namen. Die Anfangsadresse des erzeugten Maschinenprogrammes entspricht der im Source File mit dem .BA Befehl festgelegten Adresse. Im Source File muß der .RC Befehl gegeben werden. Der .DI Befehl darf nicht verwendet werden.
Das Programm ist in BASIC geschrieben, um Ihnen das Verständnis und eventuelle Änderungen zu erleichtern. Auf die Funktion des Programmes will ich hier nicht näher eingehen, da dies zu speziell würde. Bei Interesse können Sie sich den Inhalt eines relativen Lademodules einmal genauer ansehen. Die Funktion dürfte dann leicht nachzuvollziehen sein.

Listing zur Erzeugung eines Maschinenprogrammes auf Diskette aus einem relativen Lademodul.

```
10 INPUT"RELATIVMODULNAME: ";RN$
20 PRINT:INPUT"PROGRAMMNAME:      ";PN$
30 OPEN 1,8,3,RN$+",S,R"
40 OPEN 2,8,1,PN$
300 I=0
310 GOSUB 2000
320 IF BF = 198 OR BF = 120 THEN I=I+1:IF I=4 THEN GOSUB 2000:GOTO 340
330 GOTO 310
340 PRINT:PRINT "STARTADRESSE (L/H): ";
350 GOSUB 4000:GOSUB 4000
360 PRINT:PRINT:PRINT "PROGRAMM:":PRINT
500 GOSUB 2000
510 IF BF = 15 THEN GOSUB 2000:PRINT "LABEL";:GOTO 510
515 IF BF = 31 OR BF = 47 THEN GOSUB 2000:GOSUB 2000:PRINT "MACRO";:GOTO510
520 IF BF = 63 THEN PRINT:PRINT".BY";:GOSUB 4000:GOSUB 2000:GOTO 510
530 IF BF = 79 THEN PRINT:PRINT".SE";:GOSUB 4000:GOSUB4000:GOSUB2000:GOTO510
535 IF BF = 127 THEN PRINT:PRINT".DS";:GOSUB 4000:GOSUB4000:GOSUB2000:GOTO510
540 PRINT:GOSUB 1000
550 FOR I = 0 TO A0
560 GOSUB 3000
570 GOSUB 2000
580 NEXT:GOTO 510
590 GOTO 500
1000 IF BF = 0 OR BF = 64 OR BF = 96 THEN A0 = 0:RETURN
1010 IF BF = 32 THEN A0 = 2:RETURN
1020 IF (BF AND 15) < 8 THEN A0 = 1:RETURN
1030 IF ((BF AND 15) = 8) OR ((BF AND 15) = 10) THEN A0 = 0:RETURN
1040 IF ((BF AND 15) = 9) AND ((BF AND 16) = 0) THEN A0 = 1:RETURN
1050 A0 = 2:RETURN
2000 GET#1,A$:IF ST=64 THEN CLOSE 1:CLOSE 2:CLR:END
2010 IF A$="" THEN BF=0:RETURN
2020 BF = ASC(A$):RETURN
3000 PRINT BF;:PRINT#2,CHR$(BF);:RETURN
4000 GOSUB 2000:GOSUB 3000:RETURN
READY.
```

Unter Berücksichtigung der vorhergehenden Informationen ist der Beginn eines Source Files mit MAE 64 wie folgt möglich:

```
10 .OC          ;kein Objektcode ins RAM
20 .RC          ;keine Adressverschiebung
30 .BA $5000    ;Startadresse
40 .LS          ;Listing einschalten
50VARI .DE $60 ;Variable definieren
60 .LC          ;Listing ausschalten
```

Der Objektcode wird nicht in das RAM assembliert / Es wird keine Adressverschiebung für den Relativlader vorgesehen / Die Startadresse ist $5000 / Die Zeile 50 wird als Listing ausgegeben / Die Variable VARI erhält den Wert $60.
Um ein Maschinenprogramm auf Diskette zu erzeugen, ist wie folgt vorzugehen:

* Nach der Assemblierung ist der Befehl OUT "NAME" zu geben.
* Der Assembler ist mit BA zu verlassen.
* Das zuvor vorgestellte Hilfsprogramm ist zu laden und zu starten.
* Der Relativmodulname ist mit NAME und der Programmname mit PNAME anzugeben.

Daraufhin wird das Maschinenprogramm entsprechend dem Source File erzeugt und ist mit LOAD "PNAME",8,1 wieder ladbar. Selbstverständlich können auch andere Namen gewählt werden. Der Warmstart kann dann mit dem Befehl SYS 20483 erfolgen. Wenn ein Source File mehr Speicherplatz als den zur Verfügung stehenden benötigt, so kann ein Steuermodul Abhilfe schaffen. Ein Steuermodul kann wie folgt aussehen:

```
10 .CT          ;als Steuermodul kennzeichnen
20 .OC          ;Objektcode nicht ins RAM
30 .RC          ;keine Adressverschiebung vorgesehen
40 .FI "FILE1" ;Source File 1 assemblieren
50 .FI "FILE2" ;Source File 2 assemblieren
60 .EN          ;Ende der Assemblierung
```

So können bis zu 64 K Byte lange Maschinenprogramme assembliert werden.
Der .EN Befehl darf nicht in einem der vom Steuermodul

aufgerufenen Source Files gegeben werden. Im Steuermodul oder einem einzelnen Source File muß der .EN Befehl allerdings immer die Assemblierung beenden.

****

Damit kommen wir wieder zum allgemeinen Teil der Assembler-Programmierung zurück. Diese lange Erläuterung der Besonderheiten der einzelnen Assembler war erforderlich, um im folgenden davon ausgehen zu können, daß Sie die grundsätzlichen Operationen und Befehle zur Assemblierung mit Ihrem Assembler kennen. Ich werde diese Vorgaben in den folgenden Beispielen also nur noch allgemein geben wie: Startadresse $XXXX, Assemblierung ins RAM, Listing ein, usw. Bevor Sie nun das erste Source File mit Ihrem Assembler assemblieren, muß ich noch kurz auf die grundsätzliche Arbeitsweise der besprochenen Assembler eingehen, da sonst eine Unklarheit auftauchen wird. Die Assembler bearbeiten das Source File in zwei Durchgängen (Pass 1 und Pass 2). In Pass 1 werden allen Labels und Variablen die entsprechenden Werte zugewiesen und einige Assemblerbefehle ausgeführt. In Pass zwei können dann die Werte und Adressen der Labels und Variablen eingesetzt werden. Die Übersetzung der Mnemonics erfolgt meist in Pass 1. Für einige Assemblerbefehle kann es wichtig sein zu wissen, in welchem Pass diese berücksichtigt werden. Wo dies für unsere Beispiele wichtig ist, werde ich darauf hinweisen. Ansonsten finden Sie die Informationen dazu in Ihrem Handbuch.
Um einmal ein vollständiges, lauffähiges Beispiel für jeden der drei Assembler zu geben, finden Sie nachfolgend das Beispielprogramm ZWEI mit allen zur Assemblierung notwendigen Befehlen für jeden Assembler einzeln aufgeführt:

PROFIMAT

Der Assembler wurde von der Programmdiskette eingeladen. Der C 64 befindet sich im BASIC Direktmodus. Das Source File ist wie ein BASIC Programm einzugeben:

```
 10 SYS 8 * 4096
 20 .OPT P,00 ;O.code ins RAM/Listing a.Bildsch.
 30 ZAEHL = 0 ;Variable definieren
 40 *= $5000  ;Startadresse $5000
100 LDY #ZAEHL
110 VERZOE  JSR XDECR
120 INY
130 BMI END
140 JMP VERZOE
150 XDECR LDX #$FF
160 XDECR2 DEX
170 BNE XDECR2
180 RTS
190 END BRK
```

Speichern Sie das Source File ab, bevor Sie mit RUN die Assemblierung starten. Sollten Fehler auftreten, so schlagen Sie in Ihrem Handbuch die entsprechenden Fehlermeldungen nach.

T.EX.AS.
Der Assembler wurde von der Diskette geladen und mit dem .EN Befehl wurde in den BASIC Direktmodus gesprungen. Das Source File ist wie ein BASIC Programm einzugeben:

```
 10 " .LI ;Listing ausgeben
 20 " .BA $5000 ;Startadresse
 30 "ZAEHL = 0 ;Variable definieren
100 " LDY #ZAEHL
110 "VERZOE JSR XDECR
120 " INY
130 " BMI END
140 " JMP VERZOE
150 "XDECR LDX #$FF
160 "XDECR2 DEX
170 " BNE XDECR2
180 " RTS
190 "END BRK
```

Achten Sie darauf, daß die Label- und Variablennamen wie abgebildet unmittelbar hinter dem Anführungszeichen beginnen.

Speichern Sie das Source File dann ab und starten den Assembler mit dem Befehl SYS 2 warm. Die Assemblierung wird mit dem Befehl .AS gestartet. Sollten Fehler auftreten, so schlagen Sie in Ihrem Handbuch bei den Fehlermeldungen nach.

MAE 64

Der Assembler wurde eingeladen und ist gestartet. Der Source File Speicher ist mit dem Befehl CLEAR gelöscht. Geben Sie den Befehl AUTO 10 (RETURN Taste - Autonumber ist eingeschaltet) und dann die erste Zeilennummer 10. Es folgt das Listing:

```
10 .LS ;Listing ausgeben
0020 .OS ;Objektcode ins RAM
0030 .RC ;keine Verschiebung
0040ZAEHL .DE 0 ;Variable definieren
0050 .BA $4000 ;Startadresse $4000
0060START LDY #ZAEHL
0070VERZOE JSR XDECR
0080 INY
0090 BMI END
0100 JMP VERZOE
0110XDECR LDX #$FF
0120XDECR2 DEX
0130 BNE XDECR2
0140 RTS
0150END RTS
0160 .EN
```

Mit // (RETURN Taste) beenden Sie den Autonumber Modus. Achten Sie darauf, daß zwischen Labels oder Variablen und der Zeilennummer kein Leerzeichen steht. Zwischen Assemblerbefehlen oder Mnemonics und den Zeilennummern muß dagegen ein Leerzeichen stehen. Speichern Sie das Source File mit dem Befehl PUT "SOU ZWEI" auf Diskette ab. Mit dem Befehl GET "SOU ZWEI" können Sie es wieder einladen. Der Befehl AS startet dann die Assemblierung. Sollten dabei Fehler auftreten, so sehen Sie in Ihrem Handbuch bei den Fehlermeldungen nach.
Der MAE 64 kann ein im RAM befindliches Maschinenprogramm mit dem Befehl RUN (Label) starten. Sie können also zum Start der

assemblierten Routine den Befehl RUN START geben.

****

Überprüfen Sie mit dem Monitor den im RAM erzeugten Objektcode.
Wenn alles funktioniert hat, so haben Sie jetzt Ihr erstes Source File assembliert. Eventuelle Unklarheiten müßten mit dieser praktischen Übung beseitigt sein. Andernfalls sollten Sie die entsprechenden Stellen in diesem Buch noch einmal nacharbeiten und eventuell eigene praktische Versuche durchführen. Auch wenn Ihnen der Stoff bis jetzt geläufig ist, kann ein wenig experimentieren mit dem Beispielprogramm Ihre Fertigkeiten nur verbessern.
Sie beherrschen jetzt die grundlegenden Kentnisse zur Programmierung in Assembler. Die folgenden Absätze dieses Kapitels werden die Feinheiten der Assembler- Programmierung trainieren, bis Sie in der Lage sind, ein völlig flexibles Source File zu einem Problem zu erstellen.

Fragen zu 1.3.

1.) Welche drei grundsätzlichen Möglichkeiten zur Ablage des Objektcodes kennen Sie ?

2.) Wo kann sich das Source File zur Assemblierung befinden ?

3.) Warum kann das Source File zu lang für den Speicher werden ? - Was ist in einem solchen Fall zu tun ?

4.) Wie verhält sich der Assembler während der Assemblierung bei einem Fehler im Source File ?

5.) Welche Vorteile bietet die Möglichkeit, den Objektcode als Maschinenprogramm auf Diskette abzulegen ?

6.) Warum muß die Assemblierung in zwei Passes erfolgen ?

Antworten zu 1.3.

zu 1.) 1. Kein Objektcode erzeugen (Syntaxcheck) -
2. Objektcode direkt in das RAM assemblieren -
3. Objektcode als Maschinenprogramm auf Diskette ablegen.

zu 2.) Das Source File kann entweder aus dem Arbeitsspeicher oder von Diskette assembliert werden.

zu 3.) Das Mnemonic LDA #$AA belegt im Arbeitsspeicher einschließlich Zeilennummer mindestens (je nach Assembler) 13 Bytes. Der aus diesem Mnemonic erzeugte Objektcode belegt nur 2 Bytes ($A9, $AA). Daher ist das Source File immer wesentlich länger als der daraus erzeugte Objektcode.
Aus dem obigen Beispiel läßt sich ein Faktor von etwa 5 bis 6 ableiten. Ist das fertige Maschinenprogramm also etwa 8 K Byte lang, so ist das zugehörige Source File etwa 40 bis 50 K Byte lang.
Ein so langes Source File paßt nicht in den Arbeitsspeicher. Die Assembler bieten daher die Möglichkeit, das Source File in mehrere Teile aufzuteilen, die bei der Assemblierung dann von Diskette nachgeladen werden.

zu 4.) Tritt bei der Assemblierung ein Fehler auf, so wird eine entsprechende Fehlermeldung ausgegeben. Bei leichten Fehlern kann die Assemblierung trotz des Fehlers fortgesetzt werden.

zu 5.) Wenn der Objektcode in einem Bereich stehen soll, in welchem der Assembler bereits steht, so darf der Objektcode nicht direkt in das RAM assembliert werden da der Assembler dadurch zerstört und abstürzen würde. In diesem häufig auftretenden Fall gibt es nur die Möglichkeit, den Objektcode auf Diskette abzulegen.
Auch zum Austesten eines längeren Programmes ist es zeitsparend, wenn man dieses nach einem Absturz beim Testen nicht neu assemblieren muß, sondern von Diskette nachladen kann.

zu 6.) Wenn der Assembler folgende Zeilen vorfindet:

```
10          JSR UNTERPGM
20 ....
30 .....
40 UNTERPGM LDA #$AA
50 ....
60 RTS
```

kann er zwar im ersten Pass das Mnemonic JSR in Zeile 10 in $20 übersetzen, die Operanden des Befehles muß er jedoch erst ermitteln. Daher wird im ersten Pass erst allen Variablen und Labels der entsprechende Wert zugewiesen. Im zweiten Pass können dann die Operanden der Befehle eingesetzt werden.

1.4. Tabellen- Variablenzuweisung

Innerhalb eines Programmes wird es häufig vorkommen, daß Text ausgegeben, Konstanten verarbeitet werden oder daß Speicherbereiche als Variablenspeicher dienen sollen. Da es Ziel der Assemblerprogrammierung ist, ein möglichst flexibles Programm zu erstellen, sollten auch diese Aufgaben im Source File symbolisch ohne feste Adressangabe lösbar sein.

Adresstabellen

In einer Adresstabelle stehen mehrere 16 Bit Adressen in der üblichen Reihenfolge Low- Highbyte unmittelbar hintereinander. Sie haben eine solche Adresstabelle bereits am Beispielprogramm ACHT (Seite 91 ff) kennengelernt.
In der Assemblerprogrammierung ist der Aufbau einer Adresstabelle sehr einfach. Nehmen wir einmal an, die Adressen einer Adresstabelle seien folgenden Labels zugeordnet:

```
LAB1 = $5000
LAB2 = $5100
LAB3 = $5200
```

Die drei besprochenen Assembler bauen die Adresstabelle dann wie folgt auf:

```
PROFIASS: .WORD LAB1,LAB2,LAB3

T.EX.AS.: .BY LAB1 LAB2 LAB3

MAE 64: .SE LAB1
        .SE LAB2
        .SE LAB3
```

Das Ergebnis der Assemblierung dieser Befehle ist die Bytefolge:

```
$00 $50 $00 $51 $00 $52
```

Die Bytes werden ab der aktuellen assemblier Adresse im Objektcode abgelegt.

Text und Einzelbytes

Wenn das Programm Texte ausgeben soll (Fragen, Menue usw.), so muß dieser Text innerhalb des Objektcodes im ASCII Code abgelegt sein. Auch das haben Sie am Beispielprogramm ACHT bereits kennengelernt. Die drei besprochenen Assembler erlauben die direkte Eingabe des Textes, ohne diesen zuvor in den ASCII Code umzuwandeln. Die Umwandlung erfolgt durch den Assembler.

Es kann auch erforderlich sein, den absoluten Wert eines Bytes im Objektcode ablegen zu müssen. Beispielsweise erfordert die Benutzung der Ausgaberoutine $AB1E den Abschluß der Texte mit einem $00 Byte. Die Assembler erlauben somit auch die Angabe des absoluten Wertes eines Bytes.

Nehmen wir an, das Wort "TEXT" sollte im Objektcode abgelegt und mit einem $00 Byte abgeschlossen werden. Die drei besprochenen Assembler lösen diese Aufgabe wie folgt:

```
PROFIASS: .ASC "TEXT":.BYTE 0

T.EX.AS.: .BY 'TEXT' 0

MAE 64: .BY 'TEXT' 0
```

Das Ergebnis der Assemblierung ist die Bytefolge:

```
$54 $45 $58 $54 $00
```

Die Bytes werden wieder ab der aktuellen Assemblier - Adresse abgelegt.

Speicherbereiche reservieren

Wenn das Maschinenprogramm RAM Speicherplätze zur Ablage von Adressen, Variablen oder Tabellen benötigt, so kann es nützlich sein, diese Speicherbereiche im Source File festzulegen.

Die drei besprochenen Assembler stellen zu diesem Zweck folgende Befehle zur Verfügung:

```
PROFIASS: *= *+X

T.EX.AS.: .DS X

MAE 64: .DS X
```

Für X kann entweder direkt die Anzahl Bytes des freizuhaltenden Bereiches oder ein Label / Variable stehen. Der Inhalt der freigehaltenen Bytes im Objektcode ist nicht festgelegt. Bei der Assemblierung wird die aktuelle Assemblier - Adresse einfach um den Wert von X erhöht und entsprechend viele Bytes von unbestimmtem Wert im Objektcode abgelegt. In diesem Bereich legt ja das Maschinenprogramm seine Daten ab.

In der bisher beschriebenen Form wird die Nützlichkeit dieser Assemblerbefehle eventuell noch nicht ganz deutlich. Es fehlt auch noch eine wichtige Information. Vor jeden der oben aufgeführten Befehle dürfen Sie ein Label setzen. Diesem Label wird dann die Adresse der Adresstabelle, des Textes, des absolut festgelegten Bytes oder des reservierten Speicherbereiches zugeordnet. Damit können die Operanden aller Befehle des Maschinenprogrammes, welche diese Bereiche bearbeiten, im Source File durch Labels angegeben werden. Es müssen keine absoluten Adressen im Source File angegeben werden. Das trägt wiederum erheblich zur Flexibilität des Source Files bei.
Dazu ein kleines Beispiel:

```
     PROFIASS           T.EX.AS.            MAE 64

 100 LDA #<TEXT         LDA #<TEXT          LDA #L,TEXT
 110 LDY #>TEXT         LDY #>TEXT          LDY #H,TEXT
 120 JSR $AB1E          JSR $AB1E           JSR $AB1E
 130 TEXT .ASC "TEST"   TEXT .BY 'TEST' 0   TEXT .BY 'TEST' 0
 140 .BYTE 0
```

Sie sehen, daß ein Source File auf diese Weise Objektcode für

jeden Speicherbereich zu assemblieren erlaubt. Es genügt, nur die Startadresse anzugeben. Sie können auch beliebig viele Befehle löschen oder einfügen, ohne daß Sie sich um die Umrechnung der Adressen kümmern müßten.
Zur Vertiefung der Kenntnis dieser Möglichkeiten wollen wir einmal das Beispielprogramm ACHT in Assembler darstellen. Die rechten Spalten werden nur aufgeführt, wenn sie sich von der linken Spalte unterscheiden - Die Zeichen -- bedeuten, daß diese Zeile für diesen Assembler nicht eingegeben wird:

```
    PROFIASS            T.EX.AS.            MAE 64

 10 SYS 8 * 4096        STARTADR = $5000      --
 20   --                START = STARTADR-1    --
 30   --                .BA STARTADR          --
100 .OPT P,00           .LI                 .LS
110   --                  --                .OS
120   --                  --                .RC
200 ;
210 ; *** Variable ***
220 ;
230 BSHAUS = $AB1E      BSHAUS = $AB1E      BSHAUS .DE $AB1E
240 GET = $FFE4         GET = $FFE4         GET .DE $FFE4
250 ZEIL = 3            ZEIL = 3            ZEIL .DE 3
400 ;
410 ; *** Startadresse ***
420 ;
430 *= $5000            (s.Zeile 10)        .BA $4000
500 ;
510 ; *** Menue Programm ***
520 ;
530 START JSR EING      JSR EING            START JSR EING
540 AND #ZEIL
550 ASL A               ASL                 ASL A
560 TAX
570 LDA SPTAB,X         LDA @SPTAB,X        LDA SPTAB,X
580 STA ZEIG            STA @ZEIG           STA ZEIG
590 INX
600 LDA SPTAB,X         LDA @SPTAB,X        LDA SPTAB,X
610 STA ZEIG+1          STA @ZEIG+1         STA ZEIG+1
620 LDA #>START-1       LDA #>START         LDA #H,START-1
```

```
630 PHA
640 LDA #<START-1      LDA #<START         LDA #L,START-1
650 PHA
660 JMP (ZEIG)
670 ;
680 ; *** Eingabe ***
690 ;
700 EING JSR GET
710 TAX
720 BEQ EING
730 RTS
740 ;
750 ; *** Aufrufbare Routinen ***
760 ;
770 PGM1 LDA #<NULL    PGM1 LDA #<NULL     PGM1 LDA #L,NULL
780 LDY #>NULL         LDY #>NULL          LDY #H,NULL
790 JSR BSHAUS
800 RTS
900 ;
910 PGM2 LDA #<EINS    PGM2 LDA #<EINS     PGM2 LDA #L,EINS
920 LDY #>EINS         LDY #>EINS          LDY #H,EINS
930 JSR BSHAUS
940 RTS
950 ;
960 PGM3 LDA #<ZWEI    PGM3 LDA #<ZWEI     PGM3 LDA #L,ZWEI
970 LDY #>ZWEI         LDY #>ZWEI          LDY #H,ZWEI
980 JSR BSHAUS
990 RTS
1000 ;
1010 PGM4 PLA
1020 PLA
1030 BRK               BRK                 RTS
1040 ;
1050 ; *** Adresstabelle ***
1060 ;
1070 SPTAB .WORD PGM1  SPTAB .BY PGM1      SPTAB .SE PGM1
1080 .WORD PGM2        .BY PGM2            .SE PGM2
1090 .WORD PGM3        .BY PGM3            .SE PGM3
1100 .WORD PGM4        .BY PGM4            .SE PGM4
1110 ;
1120 ; *** Texte ***
```

```
1130 ;
1140 NULL .ASC "NULL":.BYTE 0 ;nur für PROFIASS
1140                   NULL .BY 'NULL' 0 ;für T.EX.AS u. MAE
1150 EINS .ASC "EINS":.BYTE 0 ;nur für PROFIASS
1150                   EINS .BY 'EINS' 0 ;für T.EX.AS u. MAE
1160 ZWEI .ASC "ZWEI":.BYTE 0 ;nur für PROFIASS
1160                   ZWEI .BY 'ZWEI' 0 ;für T.EX.AS u. MAE
1170 ;
1180 ; *** Bereich für Sprungadresse ***
1190 ;
2000 ZEIG *= *+2       ZEIG .DS 2          ZEIG .DS 2
2010 ;
2020 .END              .EN                 .EN
```

Geben Sie das entsprechende Source File zu Ihrem Assembler ein und speichern es unter dem Namen SOU ACHT auf Diskette ab. Starten Sie dann die Assemblierung. Sollten Fehler auftreten, so korrigieren Sie diese mit Hilfe der ausgegebenen Fehlermeldungen.
Wenn Sie Ihr Source File ohne Fehler vorliegen haben, sollten Sie sich besonders die verwendeten Assemblerbefehle genau ansehen. Es ist wichtig, daß Ihnen der Einsatz dieser Befehle völlig klar ist. Vergleichen Sie auch mit Hilfe des Monitors den erzeugten Objektcode mit dem Listing des Beispielprogramm ACHT von Seite 91.

Bemerkungen zu den besprochenen Assemblern:

PROFIASS
Das vorstehende Source File enthält keine nicht bereits besprochene Befehle. Lassen Sie sich auch einmal das Assemblerlisting auf Ihren Drucker ausgeben. Dazu sind folgende Zeilen zu ändern bzw. zu ergänzen:

```
  2 OPEN 1,4
100 .OPT P1,00
```

T.EX.AS.
Im vorstehenden Source File sind zwei Besonderheiten für diesen Assembler zu finden: Zum ersten wird das Label START als Variable, die um 1 kleiner als die Startadresse ist,

definiert (Zeilen 10 und 20). Das ist erforderlich weil in den Zeilen 620 und 640 das Low- und Highbyte dieser Adresse benötigt werden. T.EX.AS. betrachtet den Operator > oder < als Operator höchster Priorität. Dadurch würde die Befehlszeile:

```
640 LDA #<START-1
```

bewirken, daß zuerst das Lowbyte von START isoliert und dann um 1 vermindert wird. Das führt in diesem Fall aber nicht zum gewünschten Ergebnis.
Zum zweiten wurde in den Zeilen 570, 580, 600 und 610 die absolute Adressierung durch den Klammeraffen erzwungen. Das ist bei diesem Assembler immer dann erforderlich, wenn das die Operanden stellende Label bis zu der betreffenden Zeile noch dicht definiert wurde. Würden Sie den Klammeraffen weglassen, so würde das zu einer fehlerhaften Assemblierung führen. Der Fehler wird allerdings nicht durch eine Fehlermeldung angezeigt. Also Vorsicht !
Wie Sie beispielsweise an Zeile 770 sehen, ist der Klammeraffe nicht erforderlich wenn, das Byte durch > oder < spezifiziert wurde.

MAE 64

Das zuvor abgebildete Source File enthält keine anderen Befehle als die bereits besprochenen. Der Objektcode wurde ab Adresse $4000 asssembliert, um den Assembler nicht zu zerstören. Lassen Sie sich auch einmal ein Assemblerlisting auf Ihren Drucker ausgeben. Dazu müssen Sie die Befehle:

```
TO I
AS
```

geben. Nachdem das Listing ausgegeben wurde, geben Sie dann den Befehl:

```
TO P
```

Fragen zu 1.4.

1.) Welchen Vorteil hat die rein symbolische Anlage einer Adresstabelle im Source File ?

2.) Welche Vorteile bringt die Eingabe von Texten im Source File ?

3.) Wie ist es möglich, eine Tabelle mathematischer Konstanten im Source File anzulegen ?

4.) Was müssen Sie beachten, wenn Sie im Source File Speicherplatz für Variable oder Tabellen des Maschinenprogrammes reservieren ?

5.) Können Sie zu Beginn jeder beliebigen Zeile des Source Files ein Label definieren ? Würde das entscheidende Vorteile bringen ?

Antworten zu 1.4.

zu 1.) Wenn Sie eine Adresstabelle im Source File nur mit Variablen und Labels, also rein symbolisch, anlegen, so bleibt die Assemblierung des Source Files völlig unabhängig von der festgelegten Startadresse oder Programmänderungen. Auch wenn die Adresstabelle Variable enthält, ist deren Änderung zu Beginn des Source Files wesentlich einfacher, als den Wert im gesammten Source File zu ändern.

zu 2.) Durch die Möglichkeit, Text im Source File ohne Codewandlung einzugeben, werden Eingabe und Änderung wesentlich erleichtert. Durch Setzen eines Labels zu Beginn des Textes läßt sich auch die Adresse des Textes symbolisch verarbeiten.

zu 3.) Mit Hilfe des .BYTE Befehl läßt sich jedes beliebige Byte im Objektcode erzeugen. Diese Bytes können dann natürlich auch mathematische Konstanten enthalten.
PROFIASS bietet hier eine besondere Funktion: Mit dem Befehl .FLP X wird der Wert von X im Fließkommaformat des C 64 (5 Bytes) im Objektcode abgelegt. Die Konstanten lassen sich so direkt mit den Betriebssystemroutinen verarbeiten.

zu 4.) Wenn Sie Speicherplatz für Maschinenprogramm - Variable reservieren, so ist das nur dann sinnvoll, wenn das Maschinenprogramm mit Sicherheit nicht im ROM (EPROM) arbeiten soll.

zu 5.) Sie dürfen zu Beginn jeder beliebigen Zeile des Source Files, nachdem die Startadresse festgelegt wurde, ein Label definieren. Dadurch können auch Tabellen, Texte und Konstanten rein symbolisch verarbeitet werden.

1.5. Labels und Variable ausgeben

Wie Sie bereits wissen, können Sie Labels oder Variablen beliebige Namen geben. Die einzigen Einschränkungen sind durch die maximale Zeichenzahl und die Vorschrift, bestimmte Zeichen nicht zu verwenden, gegeben.
Ein Label oder eine Variable kann allerdings nur einmal definiert werden. Es ist nicht möglich, derselben Variablen innerhalb desselben Source Files einen anderen Wert zuzuordnen. Daher müssen Sie für jede Sprungadresse und jede Variable eine neue Bezeichnung festlegen. Wenn Sie versuchen, denselben Labelnamen zweimal zu definieren, so quittiert der Assembler das bei der Assemblierung mit einer entsprechenden Fehlermeldung.
Die Bezeichnungen werden natürlich möglichst so gewählt, daß sie leicht einprägsam sind. Labels für eine bestimmte Routine innerhalb eines größeren Files sollten am besten mit denselben Buchstaben beginnen. Wenn Sie die Bezeichnungen möglichst kurz halten (4 - 6 Zeichen), sparen Sie sich Tipparbeit und Speicherplatz, der für die Bezeichnungen benötigt wird.
Für ein längeres Source File ist es oft erforderlich, sich eine Liste aller verwendeten Labels und Variablen ausgeben zu lassen, um die Übersicht zu behalten. Die meisten Assembler bieten deshalb die Möglichkeit, eine solche Liste auf den Bildschirm oder den Drucker auszugeben. Mit dem entsprechenden Assemblerbefehl wird der Assembler meist veranlaßt, sofort nach der Assemblierung die Liste aller verwendeten Labels und Variablen auszugeben.
Wir werden nachfolgend diese Möglichkeiten der einzelnen Assembler besprechen:

PROFIASS
Bei diesem Assembler beziehen sich die folgenden Befehle auf die Ausgabe von Labels und Variablen:

.SYM X Variable und Labels nach der Assemblierung auf das mit P festgelegte Gerät ausgeben. X legt die Anzahl Spalten der Ausgabe fest. Wird X nicht angegeben, so werden 4 Spalten ausgegeben.

.SST X,2,"Name" Alle Labels und Variable werden mit ihren Bezeichnungen und Werten auf das mit der Gerätenummer X angegebene (Floppy = 8) Gerät ausgegeben.

.LST X,2,"Name" Ein mit dem .SST Befehl abgespeichertes Label File wird eingeladen. Alle in diesem File abgespeicherten Labels und Variable können im Source File nach dem Einlesen verwendet werden, ohne diese eigens zu definieren.

Um also die zweispaltige Ausgabe der Labels auf den Bildschirm zu veranlassen, muß das Source File folgende Zeile enthalten:

```
100 .OPT P:.SYM 2
```

Sollen die Labels auf den Drucker vierspaltig ausgelistet werden, so sind folgende Zeilen erforderlich:

```
 10 OPEN 1,4
 20 SYS 8 * 4096
100 .OPT P1:.SYM
```

Mit Hilfe der Befehle .SST und .LST können Sie sich eine Bibliothek mit immer wieder verwendeten Variablen beispielsweise der Betriebssystemadressen anlegen. Sie brauchen diese Variablen dann nur einmal zu definieren und mit den Zeilen:

```
100 .SST 8,2,"BETR LABELS"
```

ein Label File auf Diskette anzulegen. Wenn Sie dann in einem Source File den Befehl:

```
100 .LST 8,2,"BETR LABELS"
```

geben, so können Sie diese Labels wieder verwenden ohne sie neu definieren zu müssen. Sie werden im Laufe Ihrer Programmierpraxis sicher noch weitere sinnvolle Anwendungen dieser beiden Befehle kennenlernen.

Die mit dem .SYM Befehl ausgegebene Liste ist nach der Reihenfolge der Definition geordnet. Und zwar in umgekehrter Reihenfolge. Es wird also zuerst das zuletzt definierte Label ausgelistet. Mit dem Hilfsprogramm SYMPRINT auf Ihrer Programmdiskette können Sie ein Label File alphabetisch sortiert auslisten. Das Label File muß dazu natürlich mit dem Befehl .SST zuvor erzeugt werden.
Probieren Sie diese Befehle einmal am Source File SOU ACHT aus.

T.EX.AS.
Bei diesem Assembler wird durch den .LA Befehl die Ausgabe einer Liste aller definierten Labels und Variablen veranlaßt. Ersetzen Sie im Source File SOU ACHT einmal die Zeile 100 mit folgender Zeile:

```
100 ".LA
```

Wenn Sie folgende Zeile noch hinzufügen, wird die Liste auf den Drucker ausgegeben:

```
110 ".PR
```

Die Befehle werden natürlich erst bei der Assemblierung nach dem .AS Befehl ausgeführt.
Die Liste wird in der Reihenfolge ausgegeben, in der die Labels oder Variablen definiert wurden.

MAE 64
Ihnen ist sicher aufgefallen, daß dieser Assembler immer eine Liste aller definierten Labels und Variablen ausgibt, wenn das Assemblerlisting ausgegeben wurde. Das gilt für die Ausgabe auf den Bildschirm und auf den Drucker.
Wenn Sie das unterbinden wollen, so ist das möglich, indem Sie in der letzten Zeile vor dem .EN Befehl den .LC Befehl geben. Probieren Sie das einmal am Source File SOU ACHT aus, indem Sie folgende Zeile hinzufügen:

```
2012 .LC
```

Nach dem AS Befehl wird jetzt nur das Assemblerlisting ohne

das Label Listing ausgegeben.
Nach einer Assemblierung können Sie jederzeit die definierten Labels mit dem Befehl LA ausgeben lassen. Wenn Sie zuvor mit dem TO I Befehl die Druckerausgabe eingeschaltet haben, so wird die Liste auf den Drucker ausgegeben.
Die Liste ist alphabetisch sortiert.

Fragen zu 1.5.

1.) Welche Einschränkungen für Label- und Variablennamen bestehen in der Regel ?

2.) Wozu dient die Auslistung der definierten Labels und Variablen ?

3.) Kann zwischen Labels und Variablen in der Auslistung unterschieden werden ? Begründen Sie Ihre Antwort.

Antworten zu 1.5.

zu 1.) Labels und Variablen müssen mit einem Buchstaben beginnen und haben eine von Assembler zu Assembler verschiedene Maximallänge. Bei manchen Assemblern dürfen die Namen auch nicht mit einem Mnemonic wie LDA o.ä. beginnen.

zu 2.) Bei einem langen, eventuell sogar verketteten Source File wird es sehr schwer, die Übersicht über die bereits verwendeten Namen zu behalten. Das wird mit einer Liste aller Labels und Variablen wesentlich einfacher. Besonders, wenn diese Liste alphabetisch sortiert ist.

zu 3.) Der Assembler unterscheidet nicht zwischen Labels und Variablen. Daher kann eine solche Unterscheidung auch nicht in der Auslistung der Labels und Variablen vorgenommen werden.

1.6. Druckerausgabe / Source File Gestaltung

Ab einer gewissen Länge des Source Files ist es nicht mehr möglich, das Programm ohne ausgedrucktes Assemblerlisting zu überschauen. Aber auch bei einem Ausdruck ist es sehr wichtig, diesen so übersichtlich wie möglich zu gestalten. Sie haben dafür mit dem Source File SOU ACHT bereits ein praktisches Beispiel kennengelernt.
Wir wollen einige Grundregeln zur Gestaltung des Source Files aufstellen:

** Jede eigenständige Routine wird optisch gut sichtbar durch mehrere Zeilen von den vorhergehenden und nachfolgenden Zeilen getrennt. Ein einfaches Mittel hierzu ist, in der Zeile nur ein Semikolon zu schreiben.
** Jede eigenständige Routine erhält eine gut erkennbare Überschrift. Die Überschrift kann mit dem * Zeichen gut erkennbar gemacht werden. Es ist auch zu empfehlen, daß die Überschrift das Einsprunglabel enthält oder nur aus diesem besteht.
** Auch Zeilen, die Assemblerbefehle, Variablendefinitionen, Tabellen, Texte usw. enthalten, werden deutlich durch Leerzeilen und Überschrift abgehoben.
** Alle Variablen werden immer zu Beginn des Source Files definiert.
** Texte, Tabellen und Bereiche für Maschinensprache-Variablen stehen immer zum Schluß des Source Files.

Im Laufe Ihrer Programmierpraxis werden Sie obige Regeln sicher noch abwandeln und ergänzen. Dabei sollte Übersichtlichkeit und Einheitlichkeit aber immer oberstes Gebot sein. Sie werden es sich selbst danken, wenn Sie ein eigenes älteres Source File ändern oder erweitern wollen.
Auch die meisten Assembler bieten einige Möglichkeiten zur übersichtlichen Gestaltung des Ausdruckes. Wir werden die speziellen Möglichkeiten der einzelnen Assembler im folgenden besprechen.

PROFIASS

Der Ausdruck mit diesem Assembler erfolgt mit einem Seitenvorschub nach der 60. Zeile. Der Seitenvorschub wird mit dem CHR$ (12) Befehl erzeugt. Sollte Ihr Drucker diesen Code nicht kennen, so erfolgt kein Seitenvorschub.

Sie können einen Seitenvorschub an jeder beliebigen Stelle in einem Source File mit dem .PAG Befehl erzwingen. Wenn Sie die Anzahl Zeilen pro Blatt von 60 ändern wollen, so können Sie das ebenfalls mit dem .PAG X Befehl erreichen. X steht hier für die Anzahl Zeilen vor einem Seitenvorschub.

Der Ausdruck beginnt normalerweise am linken Papierrand. Wenn Sie einen Heftrand benötigen, so können Sie mit dem .PAG ,X Befehl den Ausdruckbeginn um X Zeichen nach rechts verschieben. Soll nur der Rand festgelegt werden, so muß das Komma vor dem Wert für X stehen. Es können die Anzahl Zeilen pro Blatt und der Heftrand wie folgt gleichzeitig festgelegt werden:

```
.PAG Anz.Zeilen,Heftrand
```

Ergänzen oder ändern Sie das Source File SOU ACHT einmal um folgende Zeilen und starten dann die Assemblierung:

```
  2 OPEN 1,4
100 .OPT P1,OO:.PAG ,8
440 .PAG
```

Sie sehen, daß mit Hilfe des .PAG Befehl die Übersichtlichkeit noch verbessert werden kann.

T.EX.AS.

Dieser Assembler kennt nur die bisher besprochene einfache Ausgabe des unformatierten Source Files. Sie können sich eventuell mit einem Tool zur formatierten Auslistung von BASIC Programmen helfen.

MAE 64

Folgende Befehle beeinflussen die Ausgabe des Source Files oder des Assemblerlistings:

MA C Source File nach dem PR Befehl ohne Zeilennummern ausgeben.

MA S Source File nach dem PR Befehl mit Zeilennummern ausgeben.

FO C Das eingegebene Source File wird nicht formatiert

FO S x Das eingegebene Source File wird formatiert. x gibt die maximale Zeichenzahl für Labels und Variable an (max. 31).

HA S x Ausdruck des Assemblerlisting auf 66 Zeilen pro Seite festlegen (nach 66 Zeilen ein Seitenvorschub). Seitenzahl in letzter Zeile ausdrucken. x ist die Seitenzahl der ersten Seite.

HA C Befehl HA S aufheben.

Diese Befehle sind direkt einzugeben. Nach dem Start des Assemblers sind die Befehle MA S, FO S 10 und HA C gegeben. Im Source File können Sie mit dem Befehl .EJ einen Seitenvorschub erzwingen, wenn vorher der Befehl HA S direkt gegeben wurde.
Ergänzen Sie einmal das Source File SOU ACHT um nachstehende Zeile:

```
440 .EJ
```

Geben Sie dann direkt folgende Befehle:

```
HA S O
TO I
AS
```

Sie sehen die Wirkung der besprochenen Befehle. Experimentieren Sie noch solange mit diesen Befehlen, bis Ihnen deren Funktion geläufig ist.

Fragen zu 1.6.

1.) Wie grenzen Sie eigenständige Routinen im Source File optisch von anderen Zeilen ab ?

2.) Welche anderen Teile des Source Files werden ebenfalls optisch abgegrenzt ?

3.) Warum ist bei der Programmierung in Maschinensprache eine übersichtliche Gestaltung des Programmtextes noch wichtiger als in BASIC ?

Antworten zu 1.6.

zu 1.) Alle eigenständigen Routinen sollten im Source File durch Leerzeilen (nur Semikolon) und Überschrift von vorstehenden und nachfolgenden Zeilen abgegrenzt werden. Die Überschrift sollte das Einsprunglabel enthalten oder nur aus diesem bestehen.

zu 2.) Alle zusammengehörigen Teile des Source Files wie Variable definieren, Tabellen, Startadresse usw. sollten ebenfalls mit Leerzeilen und Überschrift abgegrenzt werden. Es sollten auch Variablen untereinander abgegrenzt werden, wenn sie zu verschiedenen Themen gehören (z.B. Betriebssystemadressen und Programmparameter o.ä.).

zu 3.) Trotz der starken Vereinfachung der Maschinensprache-Programmierung durch die Assemblersprache ist die Programmierung in Assembler immer noch sehr maschinennahe. Viele Aufgaben wie Wahl des Speicherbereiches, Ein- Ausgabe usw. müssen vom Programmierer gelöst werden. In einer höheren Programmiersprache wie BASIC übernimmt das Betriebssystem diese Aufgaben.
Es ist daher in Assembler besonders bei umfangreicheren Programmen fast unumgänglich die Aufgaben auf Programm- Module zu verteilen. Diese Module werden dann von verschiedenen Programmteilen genutzt. Wenn diese aber unübersichtlich irgendwo im Source File "vergraben" sind wird es sehr langwierig, ein Modul herauszufinden, um es zu benutzen. Wir werden darauf im 2. Kapitel noch genauer zurückkommen.

1.7. Bedingte und interaktive Assemblierung

Unsere bisherigen Bestrebungen waren darauf gerichtet, ein möglichst flexibles und übersichtliches Source File zu erstellen. Flexibel dahingehend, daß eine Assemblierung des Source Files in jeden beliebigen Speicherbereich nur durch Angabe der Startadresse möglich ist. Durch konsequente Anwendung der symbolischen Darstellung aller Operanden mußten hier auch keinerlei Einschränkungen gemacht werden.
Die Möglichkeit der bedingten Assemblierung geht hier noch einen Schritt weiter. Sie werden nachfolgend sehen, wie Sie nicht nur ein Source File erstellen, welches in jeden Speicherbereich assembliert werden kann. Durch ein und dasselbe Source File kann auch Objektcode generiert werden, der für verschiedene Rechner oder sogar für unterschiedliche Aufgaben verwendet werden kann.
Das hört sich komplizierter an als es tatsächlich ist. Das Grundprinzip der bedingten Assemblierung ist sehr einfach: Die Assemblierung eines bestimmten Teiles des Source Files wird davon abhängig gemacht ob eine bestimmte Bedingung erfüllt ist oder nicht. Das ist sehr ähnlich den BASIC Zeilen:

```
100 IF A = 1 THEN GOTO 200
110 PRINT "Bedingung erfüllt"
120 END
200 PRINT "Bedingung nicht erfüllt"
```

Dieses Beispiel auf Assembler übertragen hieße, daß die Source File Anweisungen in den Zeilen 110 und 120 nur dann in Objektcode übersetzt werden, wenn die Bedingung A <> 1 erfüllt ist. Andernfalls werden die Zeilen 110 und 120 bei der Assemblierung ignoriert.

Interaktive Assemblierung

Dieser Begriff steht für die Möglichkeit, während der Assemblierung Eingaben zu machen, die die Assemblierung dann in der gewünschten Weise beeinflußen. In Verbindung mit der bedingten Assemblierung lassen sich so Source Files aufbauen, die extrem flexibel sind. Sie werden hierzu nachfolgend ein

Beispiel kennenlernen.

Beispiel

Nehmen wir einmal an, Sie wollen ein Maschinenprogramm schreiben, das sowohl auf dem VC 20 als auch auf dem C 64 lauffähig ist. Das Programm benutzt die Betriebssystemroutine $AB1E (C 64).

Die Betriebssysteme des C 64 und des VC 20 sind fast völlig identisch (Das gilt übrigens auch für die Commodore Rechner der 4000'er und 8000'er Serie). Der Unterschied ist in der Adresslage der Betriebssysteme zu finden. Die für BASIC zuständigen Routinen befinden sich beim C 64 von Adresse $A000 - $BFFF und beim VC 20 von Adresse $C000 - $DFFF. Die Ausgaberoutine muß beim VC 20 also an Adresse $CB1E aufgerufen werden.

Der Objektcode soll für den C 64 ab Adresse $C000 und für den VC 20 ab Adresse $A000 beginnen. Es wäre jetzt möglich, das Source File für den C 64 zu erstellen und abzuspeichern. Dann könnten die Startadresse und alle Aufrufe der Routine $AB1E entsprechend für den VC 20 geändert und so zwei verschiedene Versionen des Source Files abgespeichert werden.

In diesem einfachen Fall wäre das sicher auch die einfachste Lösung. Wenn allerdings mehrere Betriebssystem- Routinen aufgerufen werden sollen und noch weitere Rechner- bedingte Unterschiede bestünden, wird die Lösung des Problems mit Hilfe der bedingten Assemblierung wesentlich einfacher und übersichtlicher.

Wir wollen die Lösung des zuvor gestellten Problemes erst nur mit Hilfe der bedingten Assemblierung und dann zusätzlich mit Hilfe der interaktiven Assemblierung für die besprochenen Assembler betrachten.

PROFIASS

Dieser Assembler stellt zwei Befehle für die bedingte Assemblierung zur Verfügung:

.IF Variable:Befehl

Wenn die Variable den Wert Null hat, so wird der Befehl oder die Befehle hinter dem Doppelpunkt ignoriert und der Befehl in der folgenden Zeile ausgeführt. Ist die Variable nicht = Null, so werden alle Befehle hinter dem Doppelpunkt ausgeführt. Anstelle der Variablen darf natürlich auch ein Label stehen. Die Bedingung, ob der Wert des Ausdruckes hinter dem .IF Befehl Null ist, wird auch geprüft, wenn zusätzliche Operatoren, auch mit anderen Variablen verknüpft, den Ausdruck hinter dem .IF Befehl bilden.

.GOTO Zeilennummer

Es wird unbedingt zu der Zeilennummer des Source Files gesprungen, die nach dem Befehl angegeben wurde. In der Regel steht dieser Befehl hinter dem .IF Befehl.

Das Source File für das gestellte Problem sieht wie folgt aus:

```
 10 SYS 8 * 4096
 50 .OPT P
100 ZIELR = 1 ;C 64 = 0 / VC 20 = 1
110 .IF ZIELR:BSHAUS = $CB1E:*= $A000:.GOTO 130
120 BSHAUS = $AB1E:*= $C000
130 JSR BSHAUS
```

Sie sehen, daß Sie die Befehle .IF und .GOTO fast genauso wie in BASIC handhaben können.
PROFIASS kennt keine eigenen Befehle für die interaktive Assemblierung. Durch einen besonderen Befehl dieses Assemblers .GTB ist es aber möglich, während der Assemblierung in den BASIC Modus zu springen.
Der Befehl in der ersten Zeile im Source File nach dem .GTB Befehl wird dann als BASIC Befehl abgearbeitet. In BASIC dürfen beliebige Operationen ausgeführt werden (PRINT, GET, POKE, PEEK, Berechnungen usw.). Nicht verwendet werden darf

der INPUT Befehl und es darf kein String definiert werden. Ansonsten würden, nachdem mit SYS 40954 die Assemblierung fortgesetzt wird, dort Fehler auftreten da wichtige Daten zerstört wurden.
Der .GTB Befehl wird in Pass 1 und in Pass 2 ausgeführt. Sie können mit PEEK (567) in BASIC feststellen, ob aus dem ersten oder zweiten Pass in BASIC gesprungen wurde:

```
Pass 1  PEEK (567) = 255
Pass 2  PEEK (567) = 0
```

Ein Zeiger in den Speicherplätzen 579 (Lowbyte) und 580 (Highbyte) zeigt in Pass 1 auf das Lowbyte des Wertes der im Source File zuletzt definierten Variablen oder Label. Diese Tatsache machen wir uns zunutze, um eine interaktive Assemblierung zu realisieren.
Im Source File wird unmittelbar vor dem .GTB Befehl eine Variable definiert. In den folgenden BASIC Zeilen wird zunächst der laufende Pass geprüft. Bei Pass 2 werden die BASIC Zeilen übersprungen und die Assemblierung fortgesetzt. Bei Pass 1 wird ein kleines Menue ausgegeben, auf eine Eingabe gewartet und entsprechend der Eingabe der Wert der Assemblervariablen verändert. In der darauf fortgesetzten Assemblierung kann die Eingabe mit Hilfe des .IF Befehls berücksichtigt werden. Dazu das folgende Source File:

```
 10 SYS 8 * 4096
 50 .OPT P
 60 .SYM
100 ZIELR = 1
110 .GTB
120 IF PEEK (567) = 0 THEN 160
130 PRINT "1 = C 64":PRINT "2 = VC 20"
140 POKE 198,0:WAIT 198,1:GET A
150 POKE PEEK (579) + PEEK (580) * 256,A - 1
160 SYS 40954
210 .IF ZIELR:BSHAUS = $CB1E:*= $A000:.GOTO 230
220 BSHAUS = $AB1E:*= $C000
230 JSR BSHAUS
```

Sie sehen an diesem einfachen Beispiel, wie Sie den .GTB

Befehl einsetzen können. Es gibt außer der interaktiven Assemblierung natürlich noch weitere Einsatzmöglichkeiten.
Durch eine weitere Möglichkeit dieses Assemblers erweitern sich die Anwendungen der bedingten Assemblierung noch: Variablen die mit dem = Zeichen festgelegt wurden, dürfen in ihrem Wert nicht mehr verändert werden. Variablen können aber auch mit dem ← Zeichen definiert werden. In diesem Fall darf die Variable beliebig oft mit dem ← Zeichen neu definiert werden. Sie können so Assemblerschleifen im Source File programmieren. Dazu ein Beispiel:
Sie haben mit den Beispielprogrammen DREI und VIER bereits zwei Möglichkeiten zur Übertragung eines Speicherbereiches innerhalb des RAM kennengelernt. Beide Programme verwenden einen Branchbefehl zum Aufbau einer Schleife, in welcher der zu übertragende Bereich Byte für Byte an die Zieladresse gebracht wird. Der Branchbefehl wird also mit jedem übertragenen Byte einmal abgearbeitet. Das kostet relativ viel Rechenzeit.
In besonders zeitkritischen Anwedungsfällen muß eine solche "Zeitverschwendung" vermieden werden, um den Anforderungen gerecht zu werden. Die schnellste Übertragung erfolgt ohne Schleife durch Aneinanderreihung von LDA und STA Befehlen. Um also einen Bereich von 256 Byte zu übertragen, sind 256 LDA und 256 STA Befehle erforderlich. Das bedeutet natürlich wiederum eine "Speicherplatzverschwendung". In bestimmten Fällen ist das aber nicht zu vermeiden.
Mit Hilfe einer Assemblerschleife können Sie aber wenigstens sehr viel Tipparbeit in einem solchen Fall sparen. Sie sehen nachfolgend das Source File zu diesem Problem:

```
 10 SYS 8 * 4096
 50 .OPT P,OO
 60 .SYM
 70 *= $5000
100 ZAEHL ← 10
110 LDA $4000+ZAEHL
120 STA $4100+ZAEHL
130 ZAEHL ← ZAEHL-1
140 .IF ZAEHL+1:.GOTO 110
150 RTS
```

Die Assemblierung dieses Source Files ergibt ein Maschinenprogramm, welches den Inhalt des Speicherbereich $4000 - $400A in kürzestmöglicher Zeit in den Bereich $4100 - $410A bringt.
Sie werden im Laufe Ihrer Programmierpraxis sicher noch für andere Zwecke eine Assemblerschleife einsetzen.

T.EX.AS.
Dieser Assembler besitzt keine Möglichkeit zur interaktiven oder bedingten Assemblierung. Eventuell lassen sich diese Möglichkeiten über sogenannte Softmodule einrichten. Das sind eigene Unterroutinen, die mit Hilfe des Befehl .PE aufgerufen werden.

MAE 64
Dieser Assembler stellt folgende Befehle zur bedingten Assemblierung zur Verfügung:

IFE Variable
    Assembliere, wenn die Variable den Wert 0 hat.
IFN Variable
    Assembliere, wenn die Variable einen Wert ungleich 0 hat.
IFP Variable
    Assembliere, wenn die Variable einen positiven Wert (einschließlich 0) hat.
IFM Variable
    Assembliere, wenn die Variable einen negativen Wert hat.

Diese Befehle beziehen sich immer auf einen Block innerhalb des Source Files, dessen Beginn durch den IFx Befehl und dessen Ende durch *** Zeichen festgelegt wird. Anstelle der Variablen darf auch ein Label oder ein mit den Operatoren berechneter Ausdruck stehen.
Wenn der Assembler während der Assemblierung auf einen dieser Befehle stößt, so wird überprüft, ob die dem Befehl entsprechende Bedingung für die Variable erfüllt ist. Ist das nicht der Fall, so wird die Assemblierung mit der ersten Zeile nach der nächsten Zeile mit *** Zeichen fortgesetzt. Ist die Bedingung erfüllt, so wird der Block assembliert.

Sie finden die Anwendung in dem nachfolgenden Source File zu dem zuvor gestellten Problem:

```
  10 .OC
  20 .RC
 100ZIELR .DE 0 ;C 64 = 0 / VC 20 = 1
 110 IFE ZIELR
 120 .BA $C000
 130BSHAUS .DE $AB1E
 140 ***
 150 IFE ZIELR-1
 160 .BA $A000
 170BSHAUS .DE $CB1E
 180 ***
 190 JSR BSHAUS
 200 .EN
```

Tippen Sie dieses Source File ab, speichern es unter dem Namen "BASS 1" ab und starten dann die Assemblierung mit dem Befehl AS L.
Wenn Sie keinen Fehler gemacht haben, können Sie durch Ändern der Variablenzuweisung in Zeile 100 die Assemblierung für den C 64 oder den VC 20 einstellen. Das Programm hat nur einen Befehl, an dessen Operanden Sie die Wirkung der bedingten Assemblierung sehen können.
Wir wollen dieses Source File mit Hilfe der interaktiven Assemblierung noch komfortabler machen.
Der MAE 64 stellt für die interaktive Assemblierung folgende Befehle zur Verfügung:

.PR "Text"

Der Text zwischen den Anführungszeichen wird während der Assemblierung in Pass 1 bei Erreichen des .PR Befehles ausgegeben. Der .PR Befehl entspricht somit dem PRINT Befehl in BASIC.

.IN Variable

Wenn der Assembler während der Assemblierung in Pass 1 auf diesen Befehl stößt, so wird die Assemblierung angehalten und ein Fragezeichen ausgegeben. Es kann dann ein Wert für die auf den .IN Befehl folgende Variable eingegeben werden. Der .IN Befehl entspricht somit dem INPUT Befehl in BASIC.
Die Variable muß zusätzlich vor dem .IN Befehl wie ein Label definiert werden (siehe nachfolgendes Source File). Die so definierte Variable kann dann wie jede andere Variable im Source File eingesetzt werden.

Bei der Assemblierung des nachfolgenden Source Files wird ein kleines Menue ausgegeben. Dieses Menue bietet Ihnen die Zeilen 0 = C 64 und 1 = VC 20 an. Sie können dann die Ziffer für den gewünschten Zielrechner angeben:

```
 10 .OC
 20 .RC
 30 .PR "0 = C 64"
 40 .PR "1 = VC 20"
 50ZIELR .IN ZIELR
110 IFE ZIELR
120 .BA $C000
130BSHAUS .DE $AB1E
140 ***
150 IFE ZIELR-1
160 .BA $A000
170BSHAUS .DE $CB1E
180 ***
190 JSR BSHAUS
200 .EN
```

Sie können das vorherige Source File leicht um die Zeilen 30, 40 und 50 ergänzen und die Zeile 100 löschen. Speichern Sie das Source File dann unter dem Namen "BASS 2" ab und starten die Assemblierung mit dem Befehl AS.
Ein weiterer Befehl dieses Assemblers kann für die bedingte Assemblierung nützlich sein. Der

SET Variable/Label = neuer Wert

Befehl. Dieser Befehl erlaubt es, einer bereits definierten Variablen oder Label einen anderen Wert zuzuordnen. Für den neuen Wert darf ein beliebiger Ausdruck stehen, der sich aus Zahlen, Variablen, Labels und Operatoren zusammensetzen darf. Der SET Befehl darf nicht für Variable, die mit dem .IN Befehl festgelegt wurden, angewendet werden.

Fragen zu 1.7.

1.) Auf welchem einfachen Prinzip beruht die bedingte Assemblierung ?

2.) Nennen Sie einige Beispiele für die Anwendung der bedingten Assemblierung.

3.) Welche Vorteile hat ein Source File, welches Sie mit Hilfe der bedingten Assemblierung auf eine bestimmte Verwendung einstellen können, gegenüber der Erstellung von einzelnen Source Files für jede einzelne Verwendung ? Denken Sie dabei auch an spätere Änderungen des Source Files und die Korrektur von Fehlern.

4.) Was bedeutet interaktive Assemblierung ?

5.) Welchen Vorteil bietet die Abfrage von Parametern während der Assemblierung mit Hilfe der Befehle für die interaktive Assemblierung ?

Antworten zu 1.7.

zu 1.) Die bedingte Assemblierung basiert auf dem einfachen Prinzip, daß ein bestimmter Teil des Source Files nur unter bestimmten Bedingungen assembliert wird.

zu 2.) Source Files können mit Hilfe der bedingten Assemblierung auf verschiedene Rechner eingestellt werden, für wählbare Teilaufgaben eingesetzt werden, von bestimmten Speicherkonfigurationen abhängig gemacht werden, die Assemblierung in nicht erlaubte Speicherbereiche erkennen usw. Ziel der bedingten Assemblierung ist es, ein sicher ausgetestetes Source File für ähnliche Verwendungen zu erstellen und so die Programmierarbeit zu rationalisieren.

zu 3.) Wenn Sie beispielsweise ein Maschinenprogramm schreiben wollen, welches auf verschiedenen Commodore Rechnern lauffähig ist, so können Sie mit Hilfe der bedingten Assemblierung ein Source File erstellen, welches die Besonderheiten der betreffenden Rechner berücksichtigt (Betriebssystem Adressen, Bildschirmformat usw.). Treten bei diesem Programm zu einem späteren Zeitpunkt Fehler auf oder sind Änderungen vorzunehmen, so müssen Sie die Korrekturen oder Ergänzungen nur in einem Source File vornehmen. Die Änderung wird dann in jedem Objektcode für die verschiedenen Rechner bei der Assemblierung ausgeführt. Das gilt für alle Anwendungen der bedingten Assemblierung.

zu 4.) Die interaktive Assemblierung erlaubt die Eingabe von Parametern während der Assemblierung. So kann der Verlauf der Assemblierung in der im Source File programmierten Weise beeinflußt werden.

zu 5.) Durch die Ausgabe von Informationen mit Hilfe eines Ausgabebefehles (Menue, Frage usw.) und die Eingabe von entprechenden Daten kann die bedingte Assemblierung für den Benutzer stark vereinfacht werden. Wenn ein Source File durch Befehle der bedingten Assemblierung für unterschiedliche Verwendungen geeignet ist, so ist es besonders nach längerer Zeit für den Benutzer einfacher, wenn die notwendigen Parameter während der Assemblierung im Dialog abgefragt werden.

Ebenso einfach, aber auch nützlich, für die Assembler-Programmierung, wie die bedingte und interaktive Assemblierung, ist der Einsatz von Macros.
Ein Macro ist mit einem Baustein vergleichbar der aus einigen Source File Zeilen besteht. Dieser Baustein kann beliebig oft kopiert und in ein Source File eingebaut werden. Dazu ein Beispiel: In vielen Programmen werden 16 Bit Zeiger benötigt. Ein solcher Zeiger besteht, wie Sie wissen, aus zwei aufeinanderfolgenden Bytes. Häufig ist es erforderlich diesen Zeiger zu de- oder incrementieren. Diese Aufgabe wird von folgender Routine ausgeführt:

```
$5000  INC $60
$5002  BNE $5006
$5004  INC $61
$5006  ...
```

Diese Routine erhöht den 16 Bit Zeiger $60 - $61 um 1. Ein erforderlicher Übertrag wird dabei berücksichtigt.
Wenn es innerhalb des Programmes öfter erforderlich ist den Zeiger zu incrementieren, so bieten sich zwei Lösungen an: Zum Ersten kann die Routine in ein Unterprogramm umgewandelt werden indem an Adresse $5006 der RTS Befehl gesetzt wird. Soll der Zeiger dann in Ihrem Programm incrementiert werden, so rufen Sie einfach mit dem JSR Befehl die Routine auf.
Diese Lösung ist sicher die einfachste. Liegt aber ein zeitkritisches Problem vor, so ist es oft erforderlich die Ausführungszeit für die Befehle JSR und RTS einzusparen. Die vorstehende Routine muß in einem sochen Fall also immer an der entsprechenden Stelle im Source File eingetragen werden.
An diesem Punkt bietet sich die zweite Lösung des Problemes mit Hilfe eines Macro an. Die vorstehende Routine läßt sich bei einem macrofähigen Assembler zu Beginn des Source Files als Baustein festlegen.

Das muß in der Regel in nachstehender Weise erfolgen:

```
 50 ZEIG = $60
100 Macroname = INCR60
110 INC ZEIG
120 BNE KUEBTR
130 INC ZEIG+1
140 KUEBTR
150 Macroende
```

Die Zeilen 100 und 150 sind dabei für die verschiedenen Assembler durch die entsprechenden Assemblerbefehle zu ersetzen. Grundsätzlich aber wird ein Macro in dieser Weise festgelegt. Innerhalb eines Macro dürfen alle außerhalb des Macro definierten Variablen und Labels in der üblichen Weise verwendet werden. Sollen Labels innerhalb eines Macros definiert werden (s.o. Zeile 140), so muß das meißt in etwas anderer Form als sonst geschehen. Wir werden das bei den einzelnen Assemblern besprechen.
Das Macro erhielt in der Zeile 100 den Namen INCR60. Mit diesem Namen kann die Befehlsfolge der Zeilen 110, 120 und 130 an jeder folgenden Stelle des Source Files aufgerufen werden.
Dazu wieder ein Beispiel:

```
1000 ....
1010 STA $4000
1020 Macro INCR60 aufrufen
1030 Macro INCR60 aufrufen
1040 JMP $5000
1050 ....
```

Wenn dieser Ausschnitt aus einem Source File assembliert wird, so entspricht der erzeugte Objektcode dem Objektcode den folgender Source File Ausschnitt erzeugen würde:

```
1000 ....
1010          STA $4000
1020          INC ZEIG
1030          BNE LAB1
1040          INC ZEIG+1
1050 LAB1     INC ZEIG
1060          BNE LAB2
1070          INC ZEIG+1
1080 LAB2     JMP $5000
1090 ....
```

Sehen Sie sich das Bespiel bitte genau an und machen sich die Zusammenhänge deutlich.
Sie sehen, daß hinter dem Begriff "Macro" ein einfacher Zusammenhang steht. Mit Hilfe von Macros ersparen Sie sich das mehrfache Eingeben von gleichen Routinen. Auf weitere Vorteile und Anwendungen von Macros werden wir noch zu sprechen kommen. Zunächst bleibt aber noch eine zusätzliche Möglichkeit im Zusammenhang mit Macros zu besprechen.
Das zuvor vorgestellte Macro ist bei jedem Aufruf identisch mit dem definierten Macro. Das sollte auch so sein. Um aber bei dem Bespiel der Incrementierung eines 16 Bit Zeigers zu bleiben, so werden Sie in einem Programm sicher häufig mehr als einen solchen Zeiger benötigen. Soll diese Aufgabe wie besprochen mit Hilfe von Macros gelöst werden, so ist es ohne weiteres möglich für jeden Zeiger ein Macro mit eigenenm Namen und den jeweiligen Zeigeradressen zu definieren.
Die meisten Assembler bieten aber die Möglichkeit Übergabeparameter festzulegen. Diese Übergabeparameter erlauben es dem Macro bei Aufruf Werte zu übergeben die bei der Assemblierung des Macros dann an die vorgesehenen Stellen (meist für Operanden) eingesetzt werden.

Sehen wir uns dazu wieder ein Beispiel an:

```
 50 ZEIG1 = $60
 60 ZEIG2 = $62
100 Macroname = INCR MZEIG
110 INC MZEIG
120 BNE KUEBTR
130 INC MZEIG+1
140 KUEBTR
150 Macroende

1000 ....
1010 STA $4000
1020 Macro INCR ZEIG1 aufrufen
1030 Macro INCR ZEIG2 aufrufen
1040 JMP $5000
1050 ....
```

Es wird in diesem Beispiel ein Übergabeparameter verwendet. Die Übergabe erfolgt indem der Wert des ersten Parameter nach dem Macroaufruf (ZEIG1 in Zeile 1020 und ZEIG2 in Zeile 1030) der ersten Macrovariablen (lokale Variable) nach dem Macronamen (MZEIG in Zeile 100) zugeordnet wird. Während der Assemblierung des Macros bei dem momentanen Aufruf enthält die Macrovariable diesen Wert. Bei einem späteren Aufruf enthält diese Macrovariable den dann übergebenen Wert.
Wenn das vorstehende Beispiel assembliert wird, so entspricht der erzeugte Objektcode dem Objektcode den folgendes Source File erzeugen würde:

```
1000 ....
1010          STA $4000
1020          INC ZEIG1
1030          BNE LAB1
1040          INC ZEIG1+1
1050 LAB1     INC ZEIG2
1060          BNE LAB2
1070          INC ZEIG2+1
1080 LAB2     JMP $5000
1090 ....
```

Sehen Sie sich das Bespiel bitte genau an und machen sich die Zusammenhänge deutlich.
Wenn das erforderlich ist, so können auch mehrere Parameter beim Aufruf des Macro übergeben werden. Dabei ist nur zu beachten, daß genausoviele Parameter hinter dem Macroaufruf wie Macrovariable hinter dem Macronamen stehen müssen. Das erste Parameter wird dann der ersten Macrovariablen- das zweite Parameter der zweiten Macrovariablen usw. übergeben.
Ein Macro darf fast beliebig (max. Speicherplatz) lang sein. Daher können auch längere Routinen oder Programmteile als Macro definiert werden. So ist es möglich eine Macrobibliothek anzulegen in der sich die bereits von Ihnen programmierten, häufig gebrauchten, Standardroutinen befinden. Dazu gehören Ein- Ausgaberoutinen, Grafikroutinen, Soundroutinen usw.
Bei der Erstellung eines Maschinenprogrammes laden Sie diese komplette Bibliothek an den Beginn des Source Files und Rufen dann in Ihrem Programm die benötigten Macros einfach auf. Das vereinfacht die Programmierung und besonders auch das Austesten eines Programmes enorm, da die Routinen aus der Macrobibliothek ja bereits ausgetestet sind. Selbstverständlich ist die Programmierung in dieser Form nur bei guter Dokumentation der Macrobibliothek möglich. Darauf werden wir im nächsten Kapitel genauer eingehen.
Auch der Einsatz der bedingten- und interaktiven Assemblierung in Verbindung mit Macros ist meist möglich und kann zur Flexibilität einer Macrobibliothek erheblich beitragen.

Nachfolgend besprechen wir den Einsatz von Macros am praktischen Beispiel zu den besprochenen Assemblern.
Wenn Sie diese Beispiele durchgearbeitet haben, dann kennen Sie die wichtigsten Funktionen Ihres Assemblers und können diese in der Praxis einsetzen. Das sollten Sie auch erst einmal ausführen. Erstellen Sie selbständig ein Source File zu einem Sie interessierenden Problem. Versuchen Sie dabei alle Funktionen Ihres Assemblers einzusetzen. Auch die bedingte-, interaktive- und Macroassemblierung.
Hier einige Vorschläge zu einem solchen Programm:

** Eingabe Routine

Eine Routine die eine Eingabe über die Tastatur erwartet. Die Eingabe soll mit der RETURN Taste abgeschlossen werden. Es soll möglich sein einen String bei Aufruf der Routine zu übergeben, der dann auf dem Bildschirm ausgegeben wird. Eventuell kann die Routine noch dahingehend ergänzt werden, daß bei Einsprung festgelegt werden kann, ob nur Buchstaben oder nur Zahlen angenommen werden. Die eingegebenen Zeichen sollen auf den Bildschirm und einen Puffer (RAM Bereich) ausgegeben werden.

** Menue Routine

Eine Routine ähnlich dem Bespielprogramm ACHT. Bei Aufruf soll der Routine die Adresse der ersten Menue Zeile übergeben werden. Die Menue Zeilen stehen dann wie im Beispiel ACHT jeweils mit einem $00 Byte abgeschlossen im Speicher. Nach dem $00 Byte der letzten Menue Zeile steht ein $FF Byte zur Ende- Erkennung. Es sollen nur die Zifferntasten angenommen werden für die eine entsprechende Menue Zeile vorhanden ist. Das soll die Routine selbst erkennen ($00 Bytes zählen - nur entsprechende ASCII Codes zulassen). Hinter dem $FF Byte soll unmittelbar die Sprungtabelle der aufrufbaren Routinen beginnen.
Es soll also möglich sein mit der Routine verschiedene Menue auszugeben, deren Zeilen und Sprungtabellen im Speicher an der übergebenen Adresse stehen. Die Routinen sollen in derselben Weise wie in ACHT aufgerufen werden. Ein besonderer Menue Punkt muß den Aussprung aus dem Menue ermöglichen.

Wenn Sie eines oder beide vorstehende Probleme nach den Vorgaben gelöst haben, so ist der Grundstock zu Ihre Macrobibliothek mit sicher oft benötigten Routinen bereits gelegt.
Im folgenden Kapitel lernen Sie ein System kennen, welches es Ihnen erlaubt, Programmodule zu erstellen.

PROFIASS

Mit diesem Assembler wird ein Macro wie folgt zu Beginn des Source Files festgelegt:

```
100 Macroname .MAC Macrovariablen
110 ....
120 .MEN ;Macroende
```

Innerhalb des folgenden Source Files kann ein so definiertes Macro dann wie folgt aufgerufen werden:

```
1000 'Macroname Parameter
```

Im Beispielprogramm ACHT wird für die aufrufbaren Routinen dreimal dieselbe Befehlsfolge verwendet. Wir wollen hier einmal ein Macro einsetzen. Nehmen Sie an dem Source File SOU ACHT zu diesem Zweck folgende Änderungen for:

Löschen der Zeilen: 780, 790, 800, 920, 930, 940, 970, 980, 990

Eingabe folgender Zeilen:

```
300;
302; *** MACRO ***
304;
310 UPG .MAC ADDR
320 LDA #<ADDR
330 LDY #>ADDR
340 JSR BSHAUS
350 RTS
360 .MEN

770 PGM1 'UPG NULL
910 PGM2 'UPG EINS
960 PGM3 'UPG ZWEI
```

Wenn Sie diese Änderungen vorgenommen haben, sehen Sie sich das ganze Source File noch einmal an und überzeugen sich davon, daß der erzeugte Objektcode durch dieses Source File sich nicht von dem des Source File SOU ACHT unterscheiden wird. Das ist der Fall wenn Sie genau die obigen Änderungen

vorgenommen haben.
Speichern Sie das Source File dann unter dem Namen "MACRO ACHT" auf Diskette ab und starten dann die Assemblierung. Überzeugen Sie sich dann mit dem Monitor das der erzeugte Objectcode dem des Source Files SOU ACHT genau entspricht.
Wir wollen jetzt das Source File noch etwas erweitern um den Einsatz von lokalen Labels und die Übergabe mehrerer Parameter zu demonstrieren.
Ein lokales Label (Label innerhalb eines Macro) wird wie in normales Label definiert. Bei der Verwendung des Label muß dem Labelnamen dann ein Punkt vorangestellt werden. Sie werden das jetzt am Beispiel sehen. Nehmen Sie dazu am soeben erstellten Source File MACRO ACHT folgende Änderungen vor:

```
260 ZAEHL = 5

310 UPG .MAC ADDR,ANAUSG
312 LDA #ANAUSG
314 STA ZAEHL
320 MSL LDA #<ADDR

342 DEC ZAEHL
344 BNE .MSL

770 PGM1 'UPG NULL,1
910 PGM2 'UPG EINS,2
960 PGM3 'UPG ZWEI,3
```

Versuchen Sie anhand des so veränderten Source Files einmal die Wirkung der Änderungen auf die Programmausführung zu erkennen.
Speichern Sie das Source File dann unter dem Namen "MACRO A 2" auf Diskette ab und starten dann die Assemblierung.
Es bleibt noch darauf hinzuweisen, daß bei verketteten Source Files durch den .FILE Befehl die Macros im ersten Source File nur einmal defininiert werden müssen. Labels die in Macros definiert wurden, werden mit dem .SYM Befehl ebenfalls ausgegeben. Zu dem Labelnamen wird dann immer die laufende Nummer des wievielten Aufrufes des Macro ausgegeben.
Bei der Ausgabe eines Assemblerlistings wird bei jedem Macroaufruf das komplette Macro ausgegeben. Sollten Sie das

nicht wünschen so können Sie den Assembler mit dem Befehl .OPT M dazu veranlassen nur den Macroaufruf auszulisten.
Sie sehen an diesen Beispielen, daß der praktische Einsatz von Macros keineswegs besonders kompliziert ist. Für die Anwendung von Macros in Ihrer Programmierpraxis habe ich bereits einige Beispiele gegeben. Es ist aber sehr wahrscheinlich, daß Sie die Macros noch für spezielle eigene Anwendungen einsetzen werden.

T.EX.AS.
Mit diesem Assembler wird ein Macro wie folgt zu Beginn des Source Files festgelegt:

```
100 Macroname .MD Macrovariablen
110 ....
120 .ME ;Macroende
```

Innerhalb des folgenden Source Files kann ein so definiertes Macro dann wie folgt aufgerufen werden:

```
1000 Macroname Parameter
```

Im Beispielprogramm ACHT wird für die aufrufbaren Routinen dreimal dieselbe Befehlsfolge verwendet. Wir wollen hier einmal ein Macro einsetzen. Nehmen Sie an dem Source File SOU ACHT zu diesem Zweck folgende Änderungen for:
Löschen der Zeilen: 780, 790, 800, 920, 930, 940, 970, 980, 990
Eingabe folgender Zeilen:

```
300 ";
302 "; *** MACRO ***
304 ";
310 "UPG .MD ADDR
320 " LDA #<ADDR
330 " LDY #>ADDR
340 " JSR BSHAUS
350 " RTS
360 " .ME
```

```
770 "PGM1 UPG NULL
910 "PGM2 UPG EINS
960 "PGM3 UPG ZWEI
```

Wenn Sie diese Änderungen vorgenommen haben, sehen Sie sich das ganze Source File noch einmal an und überzeugen sich davon, daß der erzeugte Objektcode durch dieses Source File sich nicht von dem des Source File SOU ACHT unterscheiden wird. Das ist der Fall, wenn Sie genau die obigen Änderungen vorgenommen haben.
Speichern Sie das Source File dann unter dem Namen "MACRO ACHT" auf Diskette ab und starten dann die Assemblierung. Überzeugen Sie sich dann mit dem Monitor, daß der erzeugte Objektcode dem des Source Files SOU ACHT genau entspricht.
T.EX.AS. bietet bei der Verwendung von Macros noch eine Funktion, die die Erstellung einer Macro Bibliothek sehr unterstützt. Es ist mit diesem Assembler möglich, Macros direkt von Diskette aufzurufen. Um den Assembler zum Aufruf des Macro von Diskette zu veranlassen, ist dem Macronamen einfach ein Doppelpunkt voranzustellen. Auf der Diskette muß sich dann das unter dem Macronamen abgespeicherte Macro Befinden.
Das wollen wir mit dem vorherigen Beispiel einmal ausführen. Laden Sie dazu das Source File MACRO ACHT ein und geben im BASIC Direktmodus die Befehle:

```
LIST 310 - 360
NEW
```

Löschen Sie den Macronamen UPG in Zeile 310 und geben Sie dann die Zeilen 310 bis 360 neu ein indem Sie einfach mit dem Cursor auf die Zeile 310 fahren und 6 mal die RETURN Taste betätigen. LIST muß dann folgendes Macro zeigen:

```
310 " .MD ADDR
320 " LDA #<ADDR
330 " LDY #>ADDR
340 " JSR BSHAUS
350 " RTS
360 " .ME
```

Speichern Sie dieses Macro unter dem Namen "UPG" auf Diskette ab. Laden Sie dann das Source File MACRO ACHT wieder ein und löschen die Zeilen 300 bis 360. Ändern Sie die Zeilen 770, 910 und 960 indem Sie jeweils einen Doppelpunkt vor den Macronamen setzen. Diese Zeilen stehen dann wie folgt in Ihrem Source File:

```
770 "PGM1 :UPG NULL
910 "PGM2 :UPG EINS
960 "PGM3 :UPG ZWEI
```

Speichern Sie das Source File dann unter dem Namen "MACRO A 2" auf Diskette ab und starten die Assemblierung. Wenn Sie alles richtig gemacht haben, so wird derselbe Objektcode wie mit den Source Files SOU ACHT und MACRO ACHT erzeugt. Überzeugen Sie sich mit dem Monitor davon.

Wir wollen jetzt das Source File noch etwas erweitern um den Einsatz von lokalen Labels und die Übergabe mehrerer Parameter zu demonstrieren.
Ein lokales Label (Label innerhalb eines Macro) wird wie ein normales Label definiert. Sie werden das jetzt am Beispiel sehen. Nehmen Sie dazu am soeben erstellten Source File MACRO ACHT folgende Änderungen vor:

```
260 "ZAEHL = 5

310 "UPG .MD ADDR,ANAUSG
312 " LDA #ANAUSG
314 " STA ZAEHL
320 "MSL LDA #<ADDR

342 " DEC ZAEHL
344 " BNE MSL

770 "PGM1 UPG NULL 1
910 "PGM2 UPG EINS 2
960 "PGM3 UPG ZWEI 3
```

Versuchen Sie anhand des so veränderten Source Files einmal die Wirkung der Änderungen auf die Programmausführung zu

erkennen.
Speichern Sie das Source File dann unter dem Namen "MACRO A 3" auf Diskette ab und starten die Assemblierung.
Bei verketteten Source Files müssen die Macros nur einmal zu Beginn des aufrufenden Source Files stehen. Bei der Ausgabe des Assemblerlistings werden normalerweise nur die Zeilen mit dem Macroaufruf ausgelistet. Mit dem Befehl .ML anstelle des .LI Befehl können Sie den Assembler dazu veranlassen bei jedem Macroaufruf das Macro im Listing auszugeben.
Lokale Labels werden mit dem .LA Befehl nicht ausgegeben.
Sie sehen an diesen Beispielen, daß der praktische Einsatz von Macros keineswegs besonders kompliziert ist. Für die Anwendung von Macros in Ihrer Programmierpraxis habe ich bereits einige Beispiele gegeben. Es ist aber sehr wahrscheinlich daß Sie die Macros noch für spezielle eigene Anwendungen einsetzen werden.

MAE 64

Mit diesem Assembler wird ein Macro wie folgt zu Beginn des Source Files festgelegt:

```
100!!!Macroname .MD (Macrovariablen)
110 ....
120 .ME ;Macroende
```

Werden Macros im Source File verwendet, so muß der .RS Befehl gegeben werden. Innerhalb des folgenden Source Files kann ein so definiertes Macro dann wie folgt aufgerufen werden:

```
1000 Macroname (Parameter)
```

Im Beispielprogramm ACHT wird für die aufrufbaren Routinen dreimal dieselbe Befehlsfolge verwendet. Wir wollen hier einmal ein Macro einsetzen. Nehmen Sie an dem Source File SOU ACHT zu diesem Zweck folgende Änderungen for:
Löschen der Zeilen: 780, 790, 800, 920, 930, 940, 970, 980, 990

Eingabe folgender Zeilen:

```
120 .RS

300;
302; *** MACRO ***
304;
310!!!UPG .MD (ADDR)
320 LDA #L,ADDR
330 LDY #H,ADDR
340 JSR BSHAUS
350 RTS
360 .ME

770PGM1 UPG (NULL)
910PGM2 UPG (EINS)
960PGM3 UPG (ZWEI)
```

Wenn Sie diese Änderungen vorgenommen haben, sehen Sie sich das ganze Source File noch einmal an und überzeugen sich davon, daß der erzeugte Objektcode durch dieses Source File sich nicht von dem des Source File SOU ACHT unterscheiden wird. Das ist der Fall, wenn Sie genau die obigen Änderungen vorgenommen haben.
Speichern Sie das Source File dann unter dem Namen "MACRO ACHT" auf Diskette ab und starten dann die Assemblierung. Überzeugen Sie sich dann mit dem Monitor das der erzeugte Objectcode dem des Source Files SOU ACHT genau entspricht.
Wir wollen jetzt das Source File noch etwas erweitern um den Einsatz von lokalen Labels und die Übergabe mehrerer Parameter zu demonstrieren.
Ein lokales Label (Label innerhalb eines Macro) wird wie in normales Label definiert. Dem Labelnamen müssen innerhalb der Macrodefinition allerdings drei Punkte vorangestellt werden. Sie werden das jetzt am Beispiel sehen. Nehmen Sie dazu am soeben erstellten Source File MACRO ACHT folgende Änderungen vor:

```
26OZAEHL .DE 5

31O!!!UPG .MD (ADDR ANAUSG)
312 LDA #ANAUSG
314 STA *ZAEHL
32O...MSL LDA #L,ADDR

342 DEC *ZAEHL
344 BNE ...MSL

77OPGM1 UPG (NULL 1)
91OPGM2 UPG (EINS 2)
96OPGM3 UPG (ZWEI 3)
```

Versuchen Sie anhand des so veränderten Source Files einmal die Wirkung der Änderungen auf die Programmausführung zu erkennen.
Speichern Sie das Source File dann unter dem Namen "MACRO A 2" auf Diskette ab und starten die Assemblierung.
Die in einem Source File verwendeten Macros müssen immer zu Beginn des Source Files definiert sein. Das gilt auch für verkettete Source File Module. Soll also ein Macro in mehreren Modulen eingesetzt werden, so müßte dieses Macro auch zu Beginn jedes Moduls neu definiert werden.
MAE 64 bietet mit dem .MG Befehl die Möglichkeit ein durch das Kontroll Modul eingeladenes Modul (mit .FI) immer im Speicher zu behalten. Wenn Sie die Macros die in mehreren Modulen aufgerufen werden in ein eigenes Modul schreiben und der erste Befehl dieses Moduls .MG ist, so können Sie dieses Modul als erstes im Kontroll Modul einlesen. Die Macros stehen dann für alle weiteren Module ohne neue Definition zur Verfügung. Es darf nur ein Modul mit dem .MG Befehl eingelesen werden.
Bei der Ausgabe des Assemblerlistings wird nur die Zeile des Macroaufrufes ausgegeben. Der Befehl .ES bewirkt die Ausgabe des ganzen Macros bei jedem Aufruf im Listing. Dieser Befehl kann mit dem Befehl .EC wieder rückgängig gemacht werden.
Lokale Labels werden nicht mit der Symboltabelle ausgegeben.
Sie sehen an diesen Beispielen daß der praktische Einsatz von Macros keineswegs besonders kompliziert ist. Für die Anwendung von Macros in Ihrer Programmierpraxis habe ich

bereits einige Beispiele gegeben. Es ist aber sehr wahrscheinlich daß Sie die Macros noch für spezielle eigene Anwendungen einsetzen werden.

Fragen zu 1.8.

1.) Was ist ein Macro ?

2.) Welche drei grundsätzlichen Regeln sind bei der Definition von Macros zu beachten ?

3.) Welcher Unterschied besteht zwischen Macrovariablen und normalen Variablen ?

4.) Was sind lokale Labels ?

5.) Nennen Sie einige Beispiele für den Einsatz von Macros.

Antworten zu 1.8.

zu 1.) Ein Macro ist eine festgelegte Source File Sequenz, die an beliebiger Stelle im Source File durch Aufruf, vom Assembler eingefügt wird.

zu 2.) 1. Macros müssen zu Beginn des ersten Source File definiert werden. 2. Die Definition muß den Anfang und das Ende des Macro durch einen entsprechenden Assemblerbefehl enthalten. 3. Das Macro muß einen Namen erhalten.

zu 3.) Macrovariable erhalten während der Assemblierung bei jedem Aufruf neue Werte und werden durch Parameterübergabe festgelegt.
Normale Variable werden in der Regel einmal definiert und können nur bei einigen Assemblern durch besondere Befehle im Wert verändert werden.

zu 4.) Lokale Labels werden in einem Macro, während das Macro aufgerufen wird, definiert. Durch die unterschiedliche Adresslage erhalten die Labels bei jedem Aufruf unterschiedliche Werte.

zu 5.) Wenn innerhalb desselben Programmes öfter dieselbe Befehlsfolge benötigt wird, so kann der Einsatz eines Macro die mehrmalige Eingabe ersparen.
Durch Anlegen einer Macro Bibliothek kann die Erstellung von Programmen erheblich vereinfacht werden.

2.1. Planung

Der Titel dieses Kapitels "Programmierung in Assembler" könnte Ihnen hier an dieser Stelle etwas verspätet erscheinen. Immerhin haben Sie das erste Maschinenprogramm bereits mit dem Beispielprogramm EINS eingegeben.
Warum ich dennoch diese Überschrift gewählt habe, hat folgende Gründe: Bisher haben Sie alle 65xx Befehle, alle Monitor- und Assemblerbefehle in Theorie und Praxis kennengelernt. Dieses Wissen ist selbstverständlich die Grundlage für die Maschinensprache- Programmierung. Wie ich bereits feststellte, entbehrt die reine Maschinensprache allerdings völlig der "Intelligenz" der höheren Programmiersprachen.
Der Einsatz der Assemblersprache mit dem Assembler bringt, wie Sie inzwischen wissen, bereits einigen Komfort in die Maschinensprache- Programmierung. Dennoch bleibt auch ein umfangreich kommentiertes Source File schwer lesbar wenn nicht ein gut durchdachtes System konsequent angewandt wird. Dieses System muß den kompletten Programmiervorgang von der Aufgabenbeschreibung, über den Entwurf, bis zur Codeerstellung und dem Austesten erfassen.
Sie werden in diesem Kapitel ein solches System kennenlernen. Die folgenden Erläuterungen sind naturgemäß etwas trocken. Das gilt auch für die Anwendung eines Systemes zur Erstellung von Programmen. Das System verlangt von Ihnen, daß Sie bevor Sie den ersten Befehl des zu erstellenden Programmes eingeben, eine umfassende Programmbeschreibung schriftlich und grafisch erstellen.
Diese Vorarbeit auf dem "Trockenen" fällt besonders dem Praktiker sehr schwer. Die Lösung ist meist schon in Ihrem Kopf und muß nur noch in die Tat, sprich Programm, umgesetzt werden. Jetzt wird von Ihnen verlangt, zuerst Unterlagen, die einen größeren Umfang als das eigentliche Programm besitzen, anzufertigen. Auch die Zeit zur Erstellung dieser Unterlagen übersteigt zumindest im Anfang die eigentliche Programmierzeit.

Gegen diese und weitere Einwendungen gegen den Einsatz eines umfangreich planenden und dokumentierenden Programmier-Systemes, kann ich an dieser Stelle nur aufgrund eigener "leidvoller" Erfahrung die Behauptung stellen, daß es ohne nicht geht.
Selbstverständlich ist es möglich ein Programm direkt aus dem Kopf auf die Tasten zu bringen. Bei einem umfangreichen Programm bereits ab einigen Hundert Bytes wird dieses Vorgehen allerdings nach einem manchmal schnellen Anfangserfolg zur Qual. Wenn Fehler auftreten, und wann wäre das nicht der Fall, oder später Änderungen vorgenommen werden sollen ist es oft einfacher ein ganz neues Programm zu schreiben, wenn keinerlei Unterlagen vorliegen.
Das gilt natürlich für die Programierung allgemein. Die Assemblerprogrammierung erfordert aber unbedingt ein gutes Programmiersystem damit die erstellten Programme ihre Aufgaben sicher erfüllen und auch in Zukunft ihre Anwendbarkeit erhalten.
Daher empfehle ich Ihnen von Anfang an nur mit dem vorgestellten System zu arbeiten und der Programmierung von "Spaghetti Code" keinen Raum zu lassen.

Aufgabenbeschreibung
Bei der Aufgabenbeschreibung werden alle Aufgaben die das zu erstellende Programm erfüllen soll schriftlich festgehalten. Jede Aufgabe wird dabei mit einer Ordnungsnummer versehen und optisch von den anderen Aufgaben durch einen Absatz und Leerzeilen getrennt. So ist es leichter die einzelnen Aufgaben zu ändern oder zu ergänzen. Bei den weiteren Programmunterlagen können Sie sich so auch auf die Ordnungsnummern der einzelnen Aufgaben beziehen.
Wenn Sie für alle Programmunterlagen ein Textsystem einsetzen, können Sie sich viele Aufgaben mit fertigen Formularen erleichtern und die Unterlagen sind auch nach der 20. Änderung noch sauber und übersichtlich.
Wir wollen die Aufgabenbeschreibung für die Eingaberoutine von Seite 195 einmal in diese Form bringen:

Aufgabenbeschreibung: EINGABEROUTINE

1.) Eingabe mehrerer Zeichen über die Tastatur erwarten.

2.) Eingabe mit der RETURN Taste abschließen.

3.) Bei Einsprung übergebenen String auf Bildschirm ausgeben.

4.) Durch Übergabe eines Flags bei Einsprung gesteuert nur Buchstaben oder nur Zahlen oder beides annehmen. Wird eine nicht erlaubte Taste betätigt, so soll dieser Tastendruck ignoriert werden.

5.) Die eingegebenen und erlaubten Zeichen sollen sofort auf den Bildschirm und in einen Puffer im RAM ausgegeben werden.

Grafischer Top - Down Entwurf

Wie aus der Bezeichnung Top - Down Entwurf bereits hervorgeht, wird bei diesem Entwurf an der Spitze der Aufgabenstellung begonnen und abwärts die einzelnen Teilaufgaben grafisch dargestellt.

Die Teilaufgaben entsprechen dabei nicht unbedingt genau den Aufgaben aus der Aufgabenbeschreibung. Vielmehr werden beim Top - Down Entwurf schon erste Überlegungen zur Programmierung der einzelnen Routinen angestellt. Es wird also dargestellt welche Aufgabe von welchem Unterprogramm gelöst wird und welches Unterprogramm welche anderen Unterprogramme aufruft.

Die Aufgabenstellung wird dabei Zeilenweise zergliedert. In der ersten Zeile steht nur die Gesamtaufgabe. Die zweite Zeile beschreibt die Unterprogramme die direkt von dem Hauptprogramm aufgerufen werden. Die Dritte Zeile beschreibt dann die Unterprogramme die jeweils von den Programmen der zweiten Zeile aufgerufen werden usw. Die Zuordnung von den aufgerufenen zu den aufrufenden Kästen, wird durch Linien kenntlich gemacht.

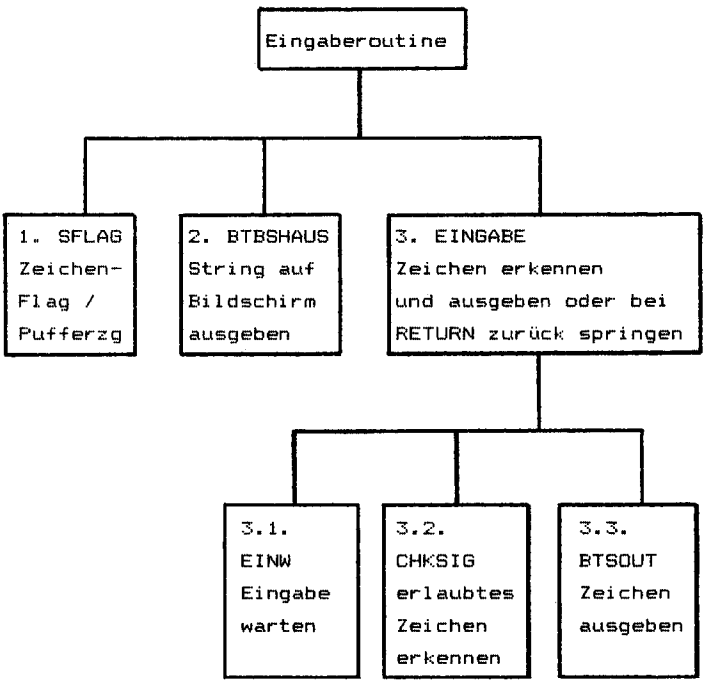
Die Aufgaben werden einzeln jeweils in umrahmten Kästen dargestellt. Die Kästen erhalten in der zweiten Zeile eine von links nach rechts fortlaufende Ordnungsnummer. Die Kästen der dritten Zeile erhalten Ordnungsnummern deren erste Stelle dieselbe Ziffer wie der sie aufrufende Kasten ist. Die Zweite Ziffer wird wieder von links nach rechts fortlaufend festgelegt. Dabei wird für bei jedem Wechsel der ersten Ziffer wieder bei 1 begonnen. Also: 1.1. / 1.2. / 1.3. / 2.1. / 2.2. / 2.3. usw.
Die weiteren Zeilen werden nach demselben System fortgesetzt. Für die vierte Zeile also: 1.1.1. / 1.1.2. / 1.2.1. / 2.1.1. usw.
Hinter oder unter der Ordnungsnummer jedes Kastens steht der Name der Routine in Großbuchstaben. Dieser Name wird im Programm auch als Label für die entsprechende Routine verwendet.
Es entsteht so ein übersichtliches Bild der zu erstellenden Routinen. Sie müssen sich dabei unbedingt an die beschriebene Hierarchie der Zeilen halten. Bei der Festlegung der Aufgaben eines Kasten muß die konkrete Umsetzung in ein Programm bereits berücksichtigt werden. Jeder Kasten steht für mindestens einige Maschinenbefehle. Die Verzweigung zu einem weiteren Kasten bedeutet einen JSR Befehl. Die Rückkehr aus einem Kasten erfolgt nur zur nächsthöheren Zeile mit dem RTS Befehl.
Sprünge zwischen den Kästen mit dem JMP oder einem Branch Befehl sind unbedingt zu unterlassen.
Ein Kasten kann auch von mehreren anderen Kästen aufgerufen werden. Das ist dann grafisch durch entsprechende Linien kenntlich zu machen. Die Ordnungsnummer eines solchen Kasten wird durch ein vorangestelltes U kenntlich gemacht. Hinter dem U stehen zeilenweise die Ordnungsnummern aller aufrufenden Kästen. Also:

```
U 1.3.
  2.1.6.
  2.
```

Dieser Entwurf kann bei kleinerem Umfang auch mit Hilfe eines Textsystems erstellt werden. Ein gutes Zeichenprogramm, welches die Anfertigung von Zeichnungen mit Text über mehrere Bildschirmseiten erlaubt, ist noch besser geeignet. Selbstverständlich kann der grafische Top - Down Entwurf auch

von Hand erstellt werden. Die Erstellung mit dem Computer ist aber besonders im Bezug auf Änderungen und Ergänzungen wesentlich komfortabler.
Bei der Erstellung sollen die Punkte der Aufgabenbeschreibung Punkt für Punkt abgestrichen werden, wenn alle für die Lösung der jeweiligen Aufgabe erforderlichen Kästen beschrieben sind. Der Top - Down Entwurf stellt dann eine umfassende Übersicht über die Lösung der gestellten Hauptaufgabe dar.

Ein solcher Entwurf sieht zu der zuvor erstellten Aufgabenbeschreibung wie folgt aus:



Modulbeschreibung
Ein Programmodul ist in diesem Fall ein Maschinenprogramm welches eine einzige Aufgabe erfüllt. Diese Aufgabe ist maximal mit einer Aufgabe aus der Aufgabenbeschreibung

identisch. Es kann allerdings auch eine Aufgabe aus der Aufgabenbeschreibung in mehrere Module zerlegt werden. Das ist dann sinnvoll, wenn eines dieser Module zusätzlich für eine andere Aufgabe aus der Aufgabenbeschreibung einsetzbar ist. Ein Modul kann andere Module aufrufen.
Ein Modul entspricht also jeweils genau einem Kasten aus dem grafischen Top - Down Entwurf.
Die Modulbeschreibung besteht aus zwei Teilen: Der Ablaufbeschreibung und der Übergabebeschreibung.

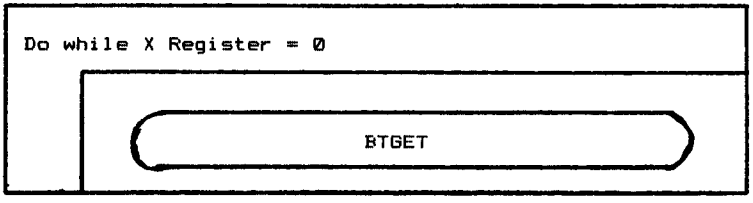
Ablaufbeschreibung
Die Ablaufbeschreibung beschreibt mit Hilfe von festgelegten grafischen Symbolen den Programmablauf eines Moduls. Dabei wird einmal das Programm mit Struktogrammen und zum anderen der Programmablauf mit einem Ablaufdiagramm beschrieben. Sie finden die verwendeten Struktogramme und die entsprechenden Ablaufdiagramme im Anhang abgebildet. Schlagen Sie dort einmal nach.

Struktogramm / Variablenliste
Mit Hilfe der im Anhang abgebildeten Struktogramme wird jedes Modul des grafischen Top - Down Entwurfes einzeln beschrieben. Wird ein Modul aufgerufen, so wird das aufgerufene Modul durch die im Top - Down Entwurf festgelegte Ordnungsnummer und den festgelegten Namen in Großbuchstaben bezeichnet.
Besteht ein Modul nur aus einer Betriebssystemroutine, so werden im Struktogramm alle Übergabeparameter nach Ein- und Ausgabe getrennt beschrieben. Zusätzlich wird die Funktion kurz angegeben. Der Aufruf der Betriebssystemroutine wird dann mit dem für Unterprogramme vorgesehenen Symbol dargestellt.
Während Sie die Struktogramme zu den Modulen erstellen gewinnt das endgültige Programm zunehmend an Gestalt. Dabei stellen Sie auch fest, welche Maschinensprache- Variablen Sie benötigen. Beispielsweise zur Übergabe von Parametern zwischen den Modulen, Flags, Tabellen, Puffer, Zeiger usw. Die Namen dieser Variablen werden im Struktogramm bereits so festgelegt wie diese auch im Source File benannt werden. Dabei sind einige Regeln zu beachten:

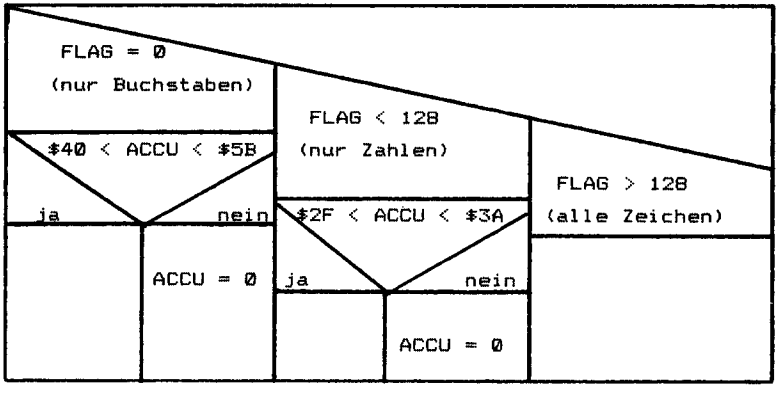
** Die Namen der Variablen die sich in der Zeropage befinden sollen, müssen mit ZP beginnen.
** Die Namen der Variablen deren Adresse im Source File durch freihalten von Bytes innerhalb des Objektcodes festgelegt werden soll, müssen mit SF beginnen.
** Die Namen der Variablen die das Lowbyte eines 16 Bit Zeiger darstellen, müssen mit ZG beginnen. Steht der Zeiger in der Zeropage, so stehen die Buchstaben ZG an der 3. und 4. Stelle des Namens.
** Die Namen der Variablen die eine Einsprungadresse für eine Betriebssystemroutine enthalten, müssen mit BT beginnen.

Auf diese Weise wird eine spätere Änderung der Variablenadressen erheblich vereinfacht.
Alle so festgelegten Variablennamen werden gleichzeitig in eine Variablenliste eingetragen. Diese Liste soll die Variablennamen und dahinter in Klammern die Ordnungsnummern der Module enthalten die diese Variable verwenden. Bei der Erstellung der Struktogramme wird die Liste fortlaufend geführt. Für die späteren Programmunterlagen ist die Liste dann alphabetisch zu sortieren. So sind dann auch leicht die verwendeten Zeropage Adressen zu überschauen.
Die Reihenfolge der Erstellung der Modulbeschreibungen ist entgegengesetzt der des grafischen Top - Down Entwurfs. Es wird also mit der untersten Zeile eines Zweiges des grafischen Top - Down Entwurfs begonnen. Sind alle Module eines Zweiges einer Zeile beschrieben, so werden die Module desselben Zweiges der nächsthöheren Zeile beschrieben. Ist das Modul der zweiten Zeile eines Zweiges beschrieben, so wird wieder mit dem nächsten Zweig auf der untersten Zeile begonnen. Analog zu dem Entwurf sprechen wir hier von einer Down - Top Beschreibung.
Sie finden nachfolgend die Struktogramme für alle Module des zuvor abgebildeten Top - Down Entwurfes.

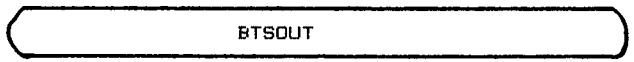
3.1. EINW

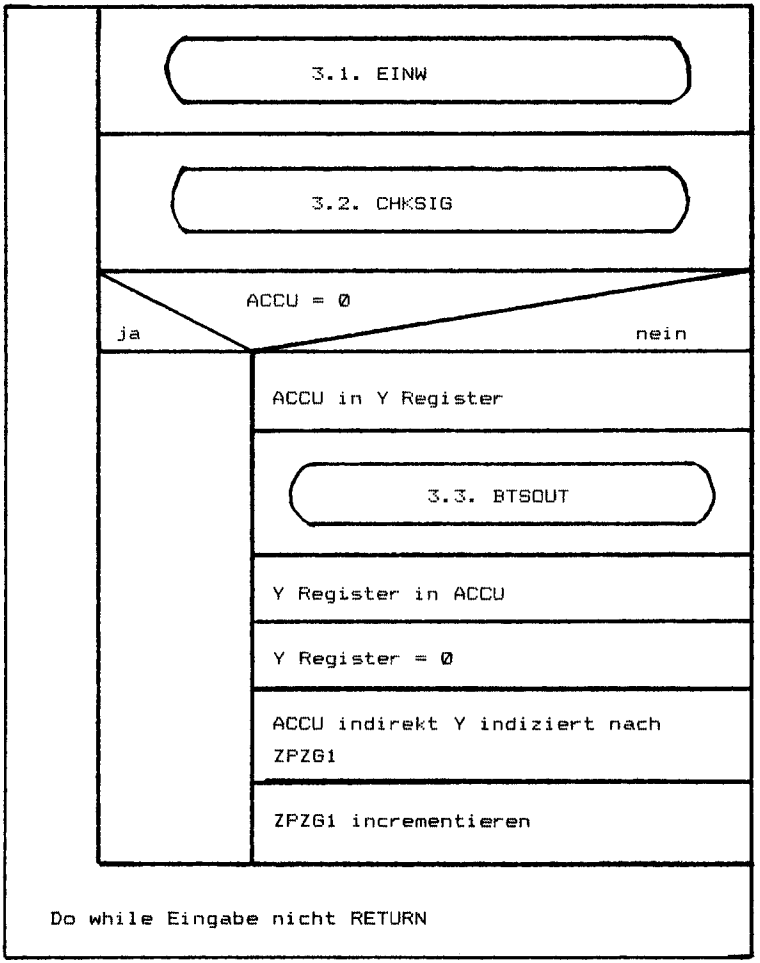


3.2. CHKSIG



Im ACCU befindliches ASCII codiertes Zeichen auf Bildschirm an aktueller Cursorposition ausgeben.
Übergabe Ein: ASCII Code im ACCU




3.1. EINW
3.2. CHKSIG
ACCU = Ø
ja
nein
ACCU in Y Register
3.3. BTSOUT
Y Register in ACCU
Y Register = Ø
ACCU indirekt Y indiziert nach ZPZG1
ZPZG1 incrementieren
Do while Eingabe nicht RETURN

1. SFLAG

X Register nach SFFLAG
ZPZG1 = ASPUFFER

2. BTBSHAUS

String auf Bildschirm ausgeben dessen Adresse im ACCU und Y Register übergeben wird
BTBSHAUS

Die Variablenliste ist während der Erstellung der vorstehenden Struktogramme wie folgt entstanden:

VARIABLENLISTE

BTGET (3.1.)
SFFLAG (3.2.)
BTSOUT (3.3.)
ZPZG1 (3.)
ASPUFFER (1.)
BTBSHAUS (2.)

Wenn diese Liste alphabetisch sortiert wird, so wird sie übersichtlicher:

VARIABLENLISTE (alphabetisch sortiert)

ASPUFFER (1.)
BTBSHAUS (2.)
BTGET (3.1.)
BTSOUT (3.3.)
SFFLAG (3.2.)
ZPZG1 (3.)

Ablaufdiagramm

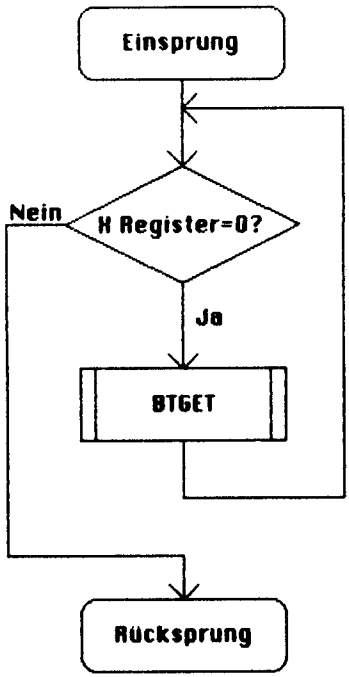
Mit Hilfe der Struktogramme läßt sich die Struktur einer Routine sehr gut entwerfen und beschreiben. Um den Ablauf einer Routine zu beschreiben und nachvollziehen zu können, ist es besser den Ablauf mit Hilfe eines Ablaufdiagrammes grafisch zu beschreiben. Das Ablaufdiagramm dient also weniger zur Entwicklung einer Routine sondern zur Beschreibung.

Im Anhang finden Sie die Struktogramme mit den zugehörigen Ablaufdiagrammen abgebildet, soweit die Übertragung sinnvoll ist. Es wird so sehr leicht zu einem Struktogramm ein Ablaufdiagramm zu erstellen.

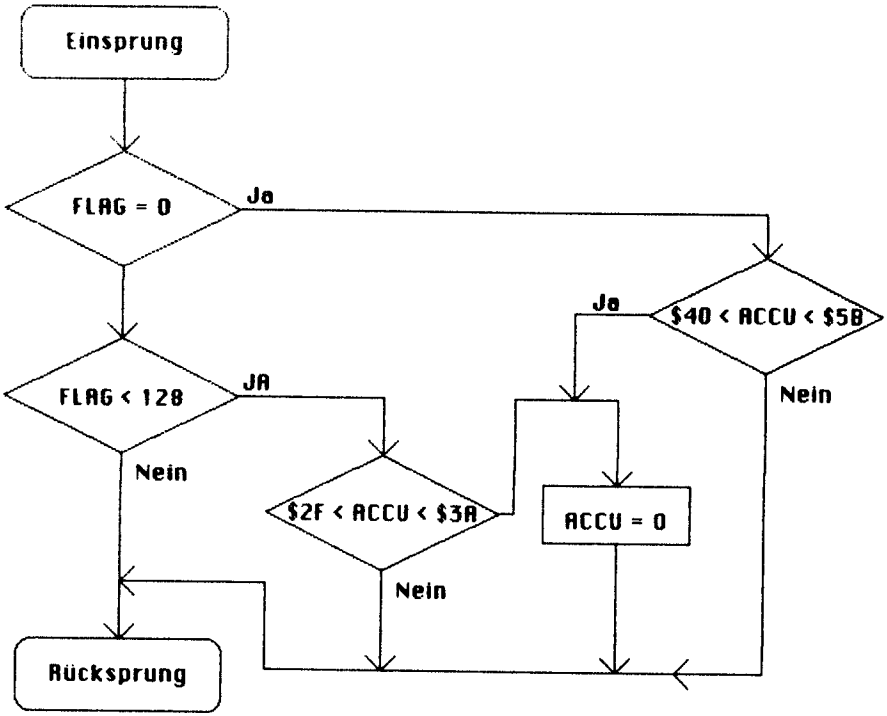
Eine besondere Vereinbarung ist noch zu treffen: Routinen die nur von einer Betriebssystemroutine repräsentiert werden, müssen nur durch das Ablaufdiagramm für ein Unterprogamm dargestellt werden. Alle anderen Routinen müssen mit einem Startsymbol beginnen und mit einem Endesymbol enden.

Sie finden nachfolgend die Struktogramme zu dem Beispielprogramm.

3.1. EINW

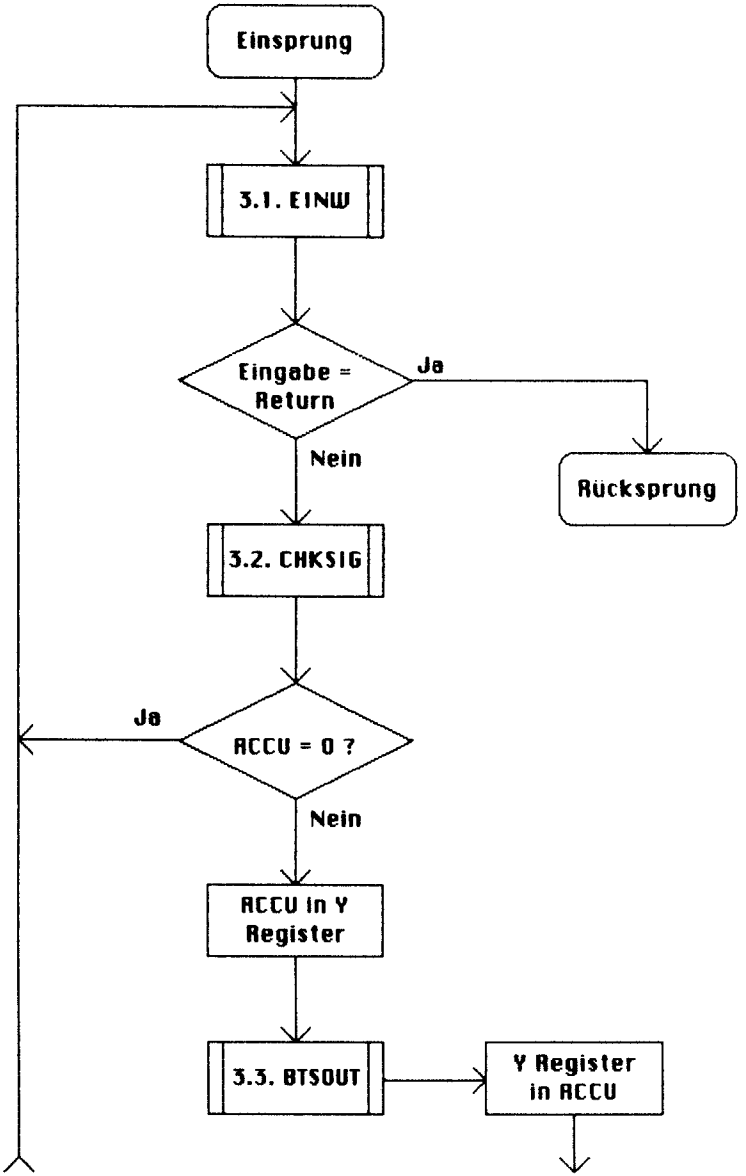


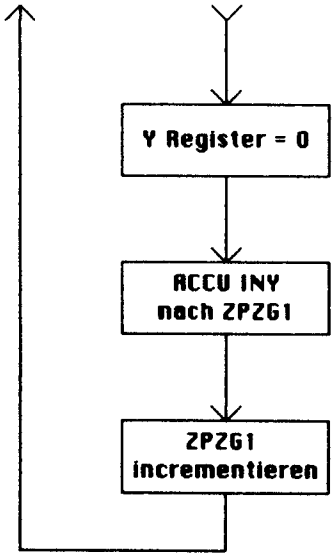
3.2. CHKSIG



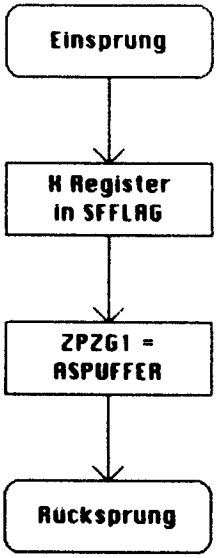
3.3. BTSOUT

3. EINGABE


Einsprung
3.1. EINW
Eingabe = Return
Ja
Nein
Rücksprung
3.2. CHKSIG
Ja
ACCU = 0 ?
Nein
ACCU in Y Register
3.3. BTSOUT
Y Register in ACCU



1. SFLAG



2. BTBSHAUS

Aufgrund dieser Unterlagen ist es jetzt sehr einfach, das fertige Programm zu erstellen.
Sie werden dazu im folgenden Absatz noch einige Hinweise erhalten und sollen dann selbständig das Source File zu dem gestellten Problem erstellen. Als letzte Programmunterlage werden dann noch die bereits erwähnten Übergabe- oder Kurzbeschreibungen vorgestellt.

Fragen zu 2.1.

1.) Nennen Sie die einzelnen Abschnitte der Programmerstellung nach dem vorgestellten System.

2.) Woher kommt die Bezeichnung "grafischer Top - Down Entwurf" ?

3.) Wie benutzen Sie die Aufgabenbeschreibung bei der Erstellung des grafischen Top - Down Entwurfes ?

4.) Dürfen Variable bei der Verwendung des vorgestellten Systemes beliebige Namen erhalten ?

5.) Welche Unstimmigkeit erscheint auf den ersten Blick bei dem Vergleich des Struktogramm und des Ablaufdiagramm zum Modul 3. EINGABE ?

6.) Warum werden zusätzlich zu den Struktogrammen noch Ablaufdiagramme erstellt ?

Antworten zu 2.1.

zu 1.) Nummerierte Aufgabenbeschreibung
Grafischer Top - Down Entwurf
Struktogramme
Ablaufdiagramme
Source File
Kurzbeschreibungen

zu 2.) Die gestellte Aufgabe wird grafisch aufgegliedert dargestellt. Dabei wird oben in der Grafik das Problem in weiter unten beschriebene Teilaufgaben aufgeteilt bis jede Teilaufgabe gelöst ist.

zu 3.) Bei der Erstellung des Top - Down Entwurfes werden die Aufgaben den Aufgabenbeschreibung einzeln abgestrichen wenn die entsprechende Aufgabe im Entwurf gelöst ist.

zu 4.) Die ersten beiden Buchstaben sind bei einigen Verwendungszwecken für Variable festgelegt.

zu 5.) Die Bedingung für den Rücksprung ist gegeben, wenn die RETURN Taste betätigt wurde. Im Struktogramm wird das durch die Schleifenbedingung beschrieben. Im Ablaufdiagramm wird der Rücksprung so beschrieben wie er auch im Source File programmiert werden wird.

zu 6.) Wie auch die zuvor gestellte Frage zeigt, beschreibt das Struktogramm nur die Struktur eines Moduls. Das erleichtert die Entwicklung einer Routine.
Ein Ablaufdiagramm dient der Beschreibung des Ablaufes einer Routine wenn die Struktur bereits festgelegt ist. Damit läßt sich die Routine leichter in ein Source File umsetzen und später leichter nachvollziehen.

2.2. Source File / Kurzbeschreibung

Bei der Erstellung des Source Files aufgrund der zuvor erstellten Unterlagen wird wieder in umgekehrter Reihenfolge vorgegangen. Sie beginnen also bei der Codierung mit dem ersten Modul der zweiten Zeile des Top - Down Entwurfes.
Die Module der zweiten Zeile werden dabei nicht als Unterprogramme aufgebaut sondern stehen aufeinanderfolgend im Source File und rufen ihrerseits die Module der unteren Zeilen als Unterprogramme auf.
Noch ein Hinweis zur Codierung der Module: Da jedes Modul nur eine einzige Aufgabe erfüllt ist es sehr einfach die Funktion eines einzelnen Moduls zu testen. Ein solcher Test erfolgt am einfachsten durch kleine Hilfsroutinen die direkt mit dem Monitor erstellt werden.
Wenn Ihr Monitor die Prozessorregister direkt setzen kann und diese nach der Rückkehr aus dem zu testenden Modul anzeigt können die Module meist ohne Hilfsprogramm getestet werden.
Bei der Fehlersuche in einem Modul kann zusätzlich noch der Einzelschritt- und der Unterbrechungsmodus nützlich sein.
Erstellen Sie also erst das gesammte Source File. Dazu finden Sie nachfolgend die zu verwendenden Betriebsystemroutinen:

```
BTBSHAUS = $AB1E ;String ausgeben
BTGET = $FFE4 ;Tastatur lesen
BTSOUT = $FFD2 ;Zeichen im ACCU ausgeben
```

Der Zeropage Zeiger ZPZG1 soll die Adressen $05 und $06 belegen. Also:

```
ZPZG1 = $05
```

Die Startadresse legen Sie zum Testen auf Adresse $5000 sofern Ihr Monitor diesen Bereich nicht belegt. Starten Sie dann die Assemblierung ohne Objektcode zu erzeugen und korrigieren eventuell auftretende Fehler im Source File. Assemblieren Sie dann das korrekte Source File mit Objektcode Erzeugung in das RAM.
Starten Sie dann Ihren Monitor und testen kurz jedes Modul mit einer Hilfsroutine oder durch setzen der Register mit dem

Monitor. Eventuell müssen Sie den RTS Befehl eines Modul zum Testen durch den BRK Befehl ersetzen. Um das Modul 2. BTBSHAUS zu testen müssen Sie im RAM mit dem Monitor einen String eingeben dessen Adresse Sie dem Modul dann übergeben.
Sind alle Tests zufriedenstellend verlaufen und eventuell das Source File entsprechend der gefundenen Fehler korrigiert, so können Sie das gesamte Programm mit einer entsprechenden Hilfsroutine testen. Hier dürften dann keine Fehler mehr auftreten.
Bei der Erstellung kommerzieller Software sollten Sie die Testroutinen mit einem Source File erstellen und diese Source Files den Programmunterlagen anfügen. Ein auftretender Fehler läßt sich so auch von einem anderen Programmier leichter finden da die Testbedingungen nachvollziehbar sind. Das gilt auch für komplexere eigene Programme.
Sie finden nachfolgend das Assemblerlisting zu dem gestellten Problem:

```
100:   5000                    .OPT P130
110:   5000                    .PAG 41,10
1000:                  ;
1002:                  ; *** VARIABLE ***
1004:                  ;
1010:  033C            ASPUFFER =    $033C
1020:  AB1E            BTBSHAUS =    $AB1E
1030:  FFE4            BTGET    =    $FFE4
1040:  FFD2            BTSOUT   =    $FFD2
1050:  0005            ZPZG1    =    $05
1100:                  ;
1102:                  ; *** STARTADRESSE ***
1104:                  ;
1110:  5000                    *=    $5000
1200:                  ;
1202:                  ; *** 1. SFLAG ***
1204:                  ;
1210:  5000 8E 50 50 SFLAG     STX   SFFLAG
1220:  5003 A2 3C              LDX   #<ASPUFFER
1230:  5005 86 05              STX   ZPZG1
1240:  5007 A2 03              LDX   #>ASPUFFER
1250:  5009 86 06              STX   ZPZG1+1
1300:                  ;
1302:                  ; *** 2. BTBSHAUS ***
1304:                  ;
1310:  500B 20 1E AB           JSR   BTBSHAUS
1400:                  ;
1402:                  ; *** 3. EINGABE ***
1404:                  ;
1410:  500E 20 2C 50 EINGABE   JSR   EINW     ;EINGABE HOLEN
1420:  5011 C9 0D              CMP   #$0D     ;RETURN TASTE TESTEN
1430:  5013 F0 16              BEQ   END3     ; JA- DANN RUECKSPRUNG
1440:  5015 20 33 50           JSR   CHKSIG   ;AUF ERLAUBT TESTEN
1450:  5018 A8                 TAY           ;FLAGS SETZEN / ACCU ZWSP.
1460:  5019 F0 F3              BEQ   EINGABE  ;IGNORIEREN
1470:  501B 20 D2 FF           JSR   BTSOUT   ;ZEICHEN AUSGEBEN
1480:  501E 98                 TYA           ;ACCU AUS ZWISCHENSPEICHER
1490:  501F A0 00              LDY   #0
1500:  5021 91 05              STA   (ZPZG1),Y  ;ZEICHEN IN PUFFER
```

```
1510:  5023 E6 05          INC  ZPZG1   ;ZEIGER INCREMENTIEREN
1520:  5025 D0 E7          BNE  EINGABE
1530:  5027 E6 06          INC  ZPZG1+1
1540:  5029 D0 E3          BNE  EINGABE  ;UNBEDINGTER SPRUNG
1550:  502B 60      END3   RTS           ;RUECKSPRUNG
2000:                ;
2002:                ; *** 3.1. EINW ***
2004:                ;
2010:  502C 20 E4 FF EINW  JSR  BTGET
2020:  502F AA             TAX
2030:  5030 F0 FA          BEQ  EINW
2040:  5032 60             RTS
2100:                ;
2102:                ; *** 3.2. CHKSIG ***
2104:                ;
2110:  5033 AE 50 50 CHKSIG LDX  SFFLAG  ;FLAG LADEN
2120:  5036 D0 09          BNE  TZAHL    ;FLAG TESTEN
2130:  5038 C9 41          CMP  #$41
2140:  503A 90 11          BCC  NICHT    ;NICHT ERLAUBT
2150:  503C C9 5B          CMP  #$5B
2160:  503E B0 0D          BCS  NICHT    ;NICHT ERLAUBT
2170:  5040 60      RUECK  RTS           ;RUECKSPRUNG MIT ZEICHEN
2180:  5041 30 FD   TZAHL  BMI  RUECK    ;ALLE ZEICHEN ERLAUBT
2190:  5043 C9 2F          CMP  #$2F
2200:  5045 90 06          BCC  NICHT    ;NICHT ERLAUBT
2210:  5047 C9 3A          CMP  #$3A
2220:  5049 B0 02          BCS  NICHT    ;NICHT ERLAUBT
2230:  504B 90 F3          BCC  RUECK    ;UNBEDINGTER SPRUNG
2240:  504D A9 00   NICHT  LDA  #0
2250:  504F 60             RTS           ;RUECKSPRUNG OHNE ZEICHEN
2300:                ;
2302:                ; *** BEREICH FUER VARIABLE ***
2304:                ;
2310:  5050        SFFLAG  *=   *+1
Ü5000-5051
NO ERRORS
```

Modul- Kurzbeschreibung

Die einzelnen Module sollten zusätzlich zu den grafischen Beschreibungen noch in einer Kurzbeschreibung umrissen werden. Diese Kurzbeschreibung dient zur späteren Benutzung des Moduls durch andere Programme oder Programmteile.

Die Kurzbeschreibung wird mit dem Namen des Moduls überschrieben und enthält neben einer kurzen Funktionsbeschreibung alle Ein- und Ausgabeparameter der Routine. Zusätzlich werden noch die benutzten Register und Machinensprache- Variablen aufgeführt.

Ruft das Modul andere Module auf, so sind diese ebenfalls aufzuführen. Zudem muß noch die benötigte Anzahl Bytes des Stapelspeichers (auch für JSR Befehle) angegeben werden.

Aufgrund der Kurzbeschreibung soll es möglich sein das Modul zu benutzen ohne das Source File studieren zu müssen.

Hier ist es sehr hilfreich mit Hilfe einer Textverarbeitung ein Formular zu erstellen das dann für jedes Modul einfach ausgefüllt wird. Das Formular muß dann wie folgt aussehen:

Name:

Funktion:

Parameter Ein:

Aus:

Benutzte Register:

Unterroutinen:

Stackbedarf: Bytes

Benutzte Variable:

Bemerkungen:

Sie finden nachfolgend die Module der Beispielroutine in Kurzbeschreibungen erklärt:

Name: 1. SFLAG

Funktion: Flag für Zeichenauswahl und Adresse des Eingabepuffer setzen.

Parameter Ein: X Register = Flag für Zeichenauswahl
Assemblervariable Aspuffer = Adresse des Puffer

Aus: --

Benutzte Register: X

Unterroutinen: --

Stackbedarf: -- Bytes

Benutzte Variable: ASPUFFER

Bemerkungen: --

Name: 2. BTBSHAUS

Funktion: String auf Bildschirm ausgeben

Parameter Ein: ACCU / Y Register = Stringadresse

Aus: --

Benutzte Register: alle

Unterroutinen: verschiedene Betriebssystemroutinen

Stackbedarf: unbekannt

Benutzte Variable: unbekannt

Bemerkungen: String muß mit O Byte enden

Name: 3. EINGABE

Funktion: Auf Tastendruck warten, Zeichen erkennen und falls erlaubt ausgeben, bei RETURN Rücksprung

Parameter Ein: Flag an Adresse SFFLAG
Flag = O = nur Buchstaben
Flag < 128 = nur Zahlen
Flag >= 128 = alle Zeichen
ZPZG1 = Zeiger auf Zeichenpuffer

Aus: Eingegebene Zeichen im Puffer

Benutzte Register: alle

Unterroutinen: 3.1. EINW
3.2. CHKSIG
3.3. BTSOUT

Stackbedarf: mindestens 2 Bytes

Benutzte Variable: SFFLAG
ZPZG1

Bemerkungen: Durch die benutzten Betriebssystemroutinen kann der Stackbedarf nicht exakt angegeben werden.

Name: 3.1. EINW

Funktion: Auf die Betätigung einer Taste warten und Tastencode in ASCII übergeben.

Parameter Ein: --

```
          Aus: ASCII Code der betätigten Taste in ACCU und  X
Register

Benutzte Register: ACCU
                   X Register

Unterroutinen: unbekannt

Stackbedarf: unbekannt

Benutzte Variable: unbekannt

Bemerkungen:



Name: 3.2. CHKSIG

Funktion: Erlaubtes  Zeichen  erkennen.  Zeichen  durch  Flag
wählbar.

Parameter Ein: Zeichen im ACCU
               Flag an Adresse SFFLAG
               Flag = 0 = nur Buchstaben
               Flag < 128 = nur Zahlen
               Flag >= 128 = alle Zeichen

          Aus: erlaubtes Zeichen im ACCU
               bei nicht erlaubtem Zeicen ACCU = 0

Benutzte Register: ACCU
                   X Register

Unterroutinen: --

Stackbedarf: -- Bytes

Benutzte Variable: --

Bemerkungen:
```

Name: 3.3. BTSOUT

Funktion: Einzelnes Zeichen an aktueller Cursorposition ausgeben.

Parameter Ein: auszugebendes Zeichen ASCII codiert im ACCU

Aus: --

Benutzte Register: ACCU

Unterroutinen: unbekannt

Stackbedarf: unbekannt

Benutzte Variable: unbekannt

Bemerkungen: --

Sie sollten diese Kurzbeschreibungen unbedingt für jedes Modul erstellen. Wenn Sie ein Modul zu einem späteren Zeitpunkt wieder verwenden wollen wird die Kurzbeschreibung Ihnen viel Zeit sparen.

Damit haben Sie alles kennengelernt was für die ersten völlig eigenständig zu erstellenden Maschinenprogramme notwendig ist. Um alle Möglichkeiten Ihres Computers mit der Maschinensprache zu nutzen steht Ihnen damit ein profundes Grundwissen zur Verfügung.
Sie sollten sich jetzt ein nicht allzu umfangreiches Problem aus Ihrem Interessensbereich stellen und dieses auf den erarbeiteten Grundlagen lösen. Eventuell erweitern Sie die zuvor erstellte Eingaberoutine zu einer Routine die Sie in allen Ihren Maschinenprogrammen dann einsetzen können. Denkbar wäre eine DELETE oder INSERT Funktion, erlaubte Zeichen weiter unterteilen usw.
Wenn Sie Routinen des Betriebssystem verwenden wollen, und darum kommen Sie früher oder später nicht herum, so benötigen Sie in jedem Fall ein dokumentiertes ROM Listing Ihres

Computers. Sie finden ein solches in 2.
Weitere Bücher zur Vertiefung und Ergänzung der in diesem Buch erarbeiteten Grundlagen sind in den Literaturhinweisen aufgeführt.
Im folgenden Absatz habe ich die Vorgehensweise zur Erstellung eines Assemblerprogrammes noch einmal zusammengefaßt.

Fragen zu 2.2.

1.) In welcher Reihenfolge werden die Module im Source File erstellt ?

2.) Welche Schritte unternehmen Sie wenn das Source File erstellt ist ?

3.) Wozu dient die Kurzbeschreibung eines Moduls ?

Antworten zu 2.2.

zu 1.) Es wird mit dem ersten Modul der zweiten Zeile begonnen.

zu 2.) 1. Assemblierung ohne Objektcode
2. Assemblierung mit Objektcode
3. Testen aller Module mit Hilfe des Monitors
4. Testen des gesamten Programmes

zu 3.) Mit Hilfe der Kurzbeschreibung eines Moduls kann das Modul von einem anderen Programm oder Programmteil benutzt werden ohne daß Sie sich die notwendigen Informationen aus dem Source File heraussuchen müssen.

2.3. Zusammenfassung

Diese Zusammenfassung soll Ihnen als Leitfaden für die Erstellung der ersten eigenen Assemblerprogramme dienen. Sie müssen dann nicht im vorhergehenden Text die einzelnen Schritte heraussuchen. Wenn Ihnen die genaueren Zusammenhänge eines Schrittes nicht mehr ausreichend geläufig sind, so finden Sie die entsprechende Textstelle leicht mit Hilfe des Index im Anhang.

Aufgabenbeschreibung
Die Aufgabenbeschreibung wird erstellt indem die Gesammtaufgabe in Teilaufgaben unterteilt und diese einzeln beschrieben werden. Jede Teilaufgabe erhält dabei eine fortlaufende Ordnungsnummer.

Grafischer Top - Down Entwurf
Das Hauptproblem wird von oben nach unten in immer kleinere Teilaufgaben aufgegliedert bis nur noch einzelne Aufgaben zu lösen sind. Die Aufgaben der Aufgabenbeschreibung werden dabei einzeln abgestrichen.
Jede Teilaufgabe wird dabei in einem mit einer Ordnungsnummer versehenen Kasten kurz beschrieben und erhält einen Namen.
Ein Kasten kann auch von mehreren anderen Kästen benutzt werden.
Direkte Sprünge von einem Kasten in den anderen sind in keinem Fall erlaubt. Andere Kästen werden immer mit JSR aufgerufen und mit RTS verlassen. Das ist schon hier zu berücksichtigen.

Struktogramme

Mit Hilfe von Struktogrammen wird die Lösung der Aufgaben der einzelnen Module entworfen. Dabei werden die Variablennamen bereits festgelegt und in der Variablenliste notiert. Für die ersten beiden Buchstaben der Variablennamen gelten in einigen Fällen folgende Richtlinien:

ZP Zeropage Variable
ZG 16 Bit Zeiger
SF Variablenplatz im Objektcode
BT Adresse einer Betriebssystemroutine

Ablaufdiagramm

Aufgrund der Struktogramme werden die Ablaufdiagramme erstellt. Dabei können logische Fehler auftauchen die eine Änderung der entworfenen Module erfordern.

Source File

Aufgrund der Struktogramme und der Ablaufdiagramme wird das Source File erstellt. Dabei wird mit dem ersten Modul der zweiten Zeile des grafischen Top - Down Entwurfes begonnen.
Die Module der zweiten Zeile sind keine Unterprogramme.
Innerhalb eines Moduls sind alle Programmier- Techniken erlaubt. Module ab der dritten Zeile dürfen nur mit RTS verlassen werden und müssen den Stack so verlassen wie bei Einsprung in das Modul vorgefunden.

Syntax Check

Das fertige Source File wird zunächst assembliert ohne Objektcode zu erzeugen. Eventuelle Assemblerfehler werden so entdeckt.

Module einzeln Testen

Das im Bezug auf den Assembler korrekte Source File wird dann erneut assembliert mit Objektcode Erzeugung in einen vom Monitor oder Assembler nicht belegten RAM Bereich.
Dann wird mit Hilfe des Monitor jedes Modul einzeln getestet. Dazu erforderliche Routinen können als Source File erstellt oder direkt mit dem Monitor eingegeben werden. Bei wichtigen oder Umfangreichen Programmen sollten die Testroutinen in Source Files erstellt und den Programmunterlagen hinzugefügt

werden.

Komplettes Programm testen
Wenn alle vorhergehenden Schritte wie beschrieben durchgeführt werden, darf hier kein Fehler mehr auftauchen. Andernfalls haben Sie einen der Schritte nicht korrekt durchgeführt.

Wenn Sie von Anfang an in dieser Weise vorgehen werden Sie Programme erstellen die wesentlich Komfortabler als BASIC Programme sind und die Vorteile der Maschinensprache zusätzlich besitzen.
Nach relativ kurzer Zeit steht Ihnen dann eine Modul Bibliothek zur Verfügung mit der in Verbindung mit Macros, interaktiver- und bedingter Assemblierung die Programmierzeit auf ein Minimum reduziert wird ohne einen Vorteil aufzugeben.

Dazu wünsche ich Ihnen viel Freude und Erfolg.

ASCII / Commodore Bildschirmcode Tabelle

% = ASCII gleich Bildschirmcode
- = Kein Bildschirmcode mit dieser Funktion

Zeichen	ASCII	Bildschirmcode Satz 1	Satz 2
* Farben			
Weiss	5 $05	-	-
Rot	28 $1C	-	-
Grün	30 $1E	-	-
Blau	31 $1F	-	-
Schwarz	144 $90	-	-
Violett	156 $9C	-	-
Gelb	158 $9E	-	-
Türkis	159 $9F	-	-
* Zeichensatz Steuercodes			
groß/klein blockieren	8 $08	-	-
groß/klein freigeben	9 $09	-	-
RVS on	18 $12	-	-
RVS off	146 $92	-	-
Kleinschr.	14 $0E	-	-
Großschr.	142 $8E	-	-
* Cursor Steuercodes			
Return	13 $0D	-	-
Shift Return	141 $8D	-	-
CRSR down	17 $11	-	-
CRSR right	29 $1D	-	-
CRSR up	145 $91	-	-
CRSR left	157 $9D	-	-
HOME	19 $13	-	-
CLR HOME	147 $93	-	-

Zeichen	ASCII	Satz 1	Satz 2
* Edit Steuercodes			
DELETE	20 $14	-	-
INSTED	148 $94	-	-
* Zeichen / Zahlen			
Space	32 $20	%	%
Shift Space	160 $A0	96 $60	96 $60
!	33 $21	%	%
"	34 $22	%	%
#	35 $23	%	%
$	36 $24	%	%
%	37 $25	%	%
&	38 $26	%	%
'	39 $27	%	%
(	40 $28	%	%
)	41 $29	%	%
*	42 $2A	%	%
+	43 $2B	%	%
,	44 $2C	%	%
-	45 $2D	%	%
.	46 $2E	%	%
/	47 $2F	%	%
0	48 $30	%	%
1	49 $31	%	%
2	50 $32	%	%
3	51 $33	%	%
4	52 $34	%	%
5	53 $35	%	%
6	54 $36	%	%
7	55 $37	%	%
8	56 $38	%	%
9	57 $39	%	%
:	58 $3A	%	%
;	59 $3B	%	%
<	60 $3C	%	%
=	61 $3D	%	%
>	62 $3E	%	%
?	63 $3F	%	%
@	64 $40	0 $00	0 $00

Zeichen	ASCII	Satz 1	Satz 2
π	126 $7E	94 $5E	-
* Kleinbuchstaben			
a	65 $41	-	1 $01
b	66 $42	-	2 $02
c	67 $43	-	3 $03
d	68 $44	-	4 $04
e	69 $45	-	5 $05
f	70 $46	-	6 $06
g	71 $47	-	7 $07
h	72 $48	-	8 $08
i	73 $49	-	9 $09
j	74 $4A	-	10 $0A
k	75 $4B	-	11 $0B
l	76 $4C	-	12 $0C
m	77 $4D	-	13 $0D
n	78 $4E	-	14 $0E
o	79 $4F	-	15 $0F
p	80 $50	-	16 $10
q	81 $51	-	17 $11
r	82 $52	-	18 $12
s	83 $53	-	19 $13
t	84 $54	-	20 $14
u	85 $55	-	21 $15
v	86 $56	-	22 $16
w	87 $57	-	23 $17
x	88 $58	-	24 $18
y	89 $59	-	25 $19
z	90 $5A	-	26 $1A
* Großbuchstaben			
A	193 $C1	1 $01	65 $41
B	194 $C2	2 $02	66 $42
C	195 $C3	3 $03	67 $43
D	196 $C4	4 $04	68 $44
E	197 $C5	5 $05	69 $45
F	198 $C6	6 $06	70 $46
G	199 $C7	7 $07	71 $47
H	200 $C8	8 $08	72 $48

Zeichen	ASCII	Satz 1	Satz 2
I	201 $C9	9 $09	73 $49
J	202 $CA	10 $0A	74 $4A
K	203 $CB	11 $0B	75 $4B
L	204 $CC	12 $0C	76 $4C
M	205 $CD	13 $0D	77 $4D
N	206 $CE	14 $0E	78 $4E
O	207 $CF	15 $0F	79 $4F
P	208 $D0	16 $10	80 $50
Q	209 $D1	17 $11	81 $51
R	210 $D2	18 $12	82 $52
S	211 $D3	19 $13	83 $53
T	212 $D4	20 $14	84 $54
U	213 $D5	21 $15	85 $55
V	214 $D6	22 $16	86 $56
W	215 $D7	23 $17	87 $57
X	216 $D8	24 $18	88 $58
Y	217 $D9	25 $19	89 $59
Z	218 $DA	26 $1A	90 $5A

Die Bildschirmcodes plus 128 oder Oder- Verknüpft mit $80 ergeben die Zeichen revers dargestellt.
Die Codes der Grafikzeichen entnehmen Sie bitte dem c 64 Handbuch Seiten 133 - 137.

65xx Flags

N- Negativflag
Nach einer Lade-, Rechen-, Logik-, Decrementier- oder Incrementier- Operation entspricht das N- Flag dem 8. also höchsten Datenbit.

V- Überlaufflag
Nach einer Addition oder einer Subtraktion entspricht das V- Flag dem 7. Ergebniss- Bit. Das V- Flag kann mit dem CLV Befehl gelöscht werden.

B- Breakflag
Nach einer durch den BRK Befehl erfolgten Programmunterbrechung ist das Breakflag gesetzt. Der RTI Befehl setzt es zurück.

D- Dezimalflag
Das Dezimalflag zeigt an ob der Prozessor im Dezimal- oder im Binär- Betrieb arbeitet. Flag = 1 - Dezimalbetrieb / Flag = 0 - Binärbetrieb. Das Flag läßt sich durch die Befehle: CLD und SED löschen und setzen.

I- Interruptflag
Das Interruptflag wird gesetzt wenn durch einen Low- Pegel am Prozessorpin IRQ ein Hardware- Interrupt erzeugt wurde. Bei gesetztem I- Flag wird kein Hardware- IRQ angenommen. Das I- Flag läßt sich durch die Befehle: CLI und SEI löschen und setzen. Der RTI Befehl löscht das I- Flag.

Z- Zeroflag
Ist das Ergebnis einer Lade-, Arithmetik-, Logik-, Decrementier- oder Incrementir- Operation = 0, dann ist das Z- Flag gesetzt.

C- Carrybit

Ist das Ergebnis einer Addition größer 255 so ist das Carrybit gesetzt. Vor einer Addition mit dem Befehl ADC muß das Carrybit gelöscht sein. Das kann mit dem Befehl: CLC erfolgen. Ist das Ergebnis einer Subtraktion kleiner 0, so wird das Carrybit gelöscht. Vor einer Subtraktion mit dem Befehl: SBC muß das Carrybit gesetzt sein. Das kann mit dem Befehl: SEC erfolgen.

65XX ADRESSIERUNG

**** Die implizierte Adressierung (implied) ****

Kürzel: IMP Assembler: Anz. Operanden: 0

Alle Einbyte - Befehle die zu ihrer Ausführung keine Operanden benötigen.

**** Die unmittelbare Adressierung (immediate) ****

Kürzel: IM Assembler: # Anz. Operanden: 1

Das unmittelbar auf das Befehlsbyte folgende Byte stellt das Datenbyte dar.

**** Die direkte, absolute Adressierung (absolute)

Kürzel: AB Assembler: Anz. Operanden: 2

Die beiden unmittelbar auf das Befehlsbyte folgenden Bytes zeigen auf die Adresse des Datenbytes oder bilden eine Startadresse. Die Reihenfolge der Bytes ist: Low Byte - High Byte der Adresse.

**** Die nullseitige Adressierung (Zeropage) ****

Kürzel: ZP Assembler: Anz. Operanden: 1

Das unmittelbar auf das Befehlsbyte folgende Byte stellt das Low Byte der Adresse des Datenbytes. Das High Byte der Adresse ist automatisch null.

**** Die Accumulator Adressierung ****

Kürzel: AC Assembler: A Anz. Operanden: 0

Der Befehl bezieht sich nur auf den Accumulator und benötigt keine Daten.

**** Die relative Adressierung ****

Kürzel: R Assembler: Anz. Operanden: 1

Das unmittelbar auf das Befehlsbyte folgende Byte beinhaltet eine Sprungweite vom momentanen Programmzähler- Stand. Ist das 8. Bit dieses Bytes gesetzt, so wird der aktuelle Programmzähler- Stand um den negierten Wert der restlichen 7 Bit vermindert (= Sprung zurück). Ist das 8. Bit nicht gesetzt, so wird der aktuelle Programmzähler- Stand um den Wert der restlichen 7 Bit erhöht (= Sprung vorwärts).

**** Die indirekte Adressierung ****

Kürzel: IND Assembler: ( ) Anz. Operanden: 2

Die beiden unmittelbar auf das Befehlsbyte folgenden Bytes zeigen auf das erste Byte der Adresse, an welcher das Low- und Highbyte der Sprungadresse stehen.

**** Die nullseitig / absolut X- oder Y- indizierte Adressierung

Kürzel: ZPX/ABX Assembler: ,x Anz. Operanden: 1/2
ZPY/ABY ,y

Zu dem / den beiden unmittelbar auf das Befehlsbyte folgenden Bytes wird der Wert des X oder Y Registers addiert. Das Ergebnis zeigt auf die Adresse des Datenbytes.

**** Die X- indiziert indirekte Adressierung

Kürzel: INX Assembler: (,x) Anz. Operanden: 1

Das unmittelbar auf das Befehlsbyte folgende Byte wird zu dem Wert des X- Register addiert. Das Ergebnis der Addition zeigt auf eine Zeropage Adresse. An diese Zeropage Adresse steht das Low- und darauf folgend das Highbyte der Adresse des Datenbyte.

**** Die indirekt Y- indizierte Adressierung ****

Kürzel: INY Assembler: ( ),y Anz. Operanden: 1

Das unmittelbar auf das Befehlsbyte folgende Byte zeigt auf eine Zeropage Adresse. An dieser Adresse steht das Low- und darauf folgend das Highbyte zu dem der Wert des Y- Register addiert wird. Das Ergebnis zeigt auf die Adresse des Datenbytes.
Bei der indirekten X indizierten Adressierung muß das X Register geradzahlige Werte haben.

**** Prozessor - Speicher ***************

```
LDA Lade den Inhalt einer Speicherzelle in den ACCU
    IM  - #      - $ A9 - 2
    AB  -        - $ AD - 4       Flags: N Z
    ZP  -        - $ A5 - 3
    INX - ( ,x)  - $ A1 - 6
    INY - ( ),y  - $ B1 - 5
    ZPX - ,x     - $ B5 - 4
    ABX - ,x     - $ BD - 4
    ABY - ,y     - $ B9 - 4

LDX Lade den Inhalt einer Speicherzelle in das X Register
    IM  - #      - $ A2 - 2
    AB  -        - $ AE - 4       Flags: N Z
    ZP  -        - $ A6 - 3
    ABY - ,y     - $ BE - 4
    ZPY - ,y     - $ B6 - 4

LDY Lade den Inhalt einer Speicherzelle in das Y Register
    IM  - #      - $ AO - 2
    AB  -        - $ AC - 4       Flags: N Z
    ZP  -        - $ A4 - 3
    ABX - ,x     - $ BC - 4
    ZPX - ,x     - $ B4 - 4

STA Speichere den ACCU nach angegebener Adresse
    AB  -        - $ 8D - 4       Flags: keine
    ZP  -        - $ 85 - 3
    INX - ( ,x)  - $ 81 - 6
    INY - ( ),y  - $ 91 - 6
    ZPX - ,x     - $ 95 - 4
    ABX - ,x     - $ 9D - 5
    ABY - ,y     - $ 99 - 5

STX Speichere das X Register nach angegebener Adresse
    AB  -        - $ 8E - 4       Flags: keine
    ZP  -        - $ 86 - 3
    ZPY - ,y     - $ 96 - 4
```

```
STY Speichere das Y Register nach angegebener Adresse
    AB  -       - $ 8C - 4       Flags: keine
    ZP  -       - $ 84 - 3
    ZPX - ,x    - $ 94 - 4

**** Transfere innerhalb des Prozessors ****

TAX ACCU nach X Register
    IMP - $ AA - 2       Flags: N Z

TAY ACCU nach Y Register
    IMP - $ A8 - 2       Flags: N Z

TSX Stapelzeiger nach X Register
    IMP - $ BA - 2       Flags: N Z

TXS X Register in Stapelzeiger
    IMP - $ 9A - 2       Flags: keine

TXA X Register nach ACCU
    IMP - $ 8A - 2       Flags: N Z

TYA Y Register nach ACCU
    IMP - $ 98 - 2       Flags: N Z

**** Arithmetische Befehle **************

ADC C Flag und adressiertes Byte zum ACCU addieren.  Ergebnis  in
    ACCU
    IM  - #     - $ 69 - 2
    AB  -       - $ 6D - 4       Flags: N Z C V
    ZP  -       - $ 65 - 3
    INX - ( ,x) - $ 61 - 6
    INY - ( ),y - $ 71 - 5
    ZPX - ,x    - $ 75 - 4
    ABX - ,x    - $ 7D - 4
    ABY - ,y    - $ 79 - 4
```

```
SBC C Flag und adressiertes Byte vom ACCU subtrahieren.  Ergebnis
    in ACCU
    IM  - #      - $ E9 - 2
    AB  -        - $ ED - 4       Flags: N Z C V
    ZP  -        - $ E5 - 3
    INX - ( ,x)  - $ E1 - 6
    INY - ( ),y  - $ F1 - 5
    ZPX - ,x     - $ F5 - 4
    ABX - ,x     - $ FD - 4
    ABY - ,y     - $ F9 - 4

DEC  Die  adressierte  Speicherzelle  wird  um  1  erniedrigt
    (decrementiert)
    AB  -        - $ CE - 6       Flags: N Z
    ZP  -        - $ C6 - 5
    ZPX - ,x     - $ D6 - 6
    ABX - ,x     - $ DE - 7

DEX X Register decrementieren
    IMP - $ CA - 2       Flags: N Z

DEY Y Register decrementieren
    IMP - $ 88 - 2       Flags: N Z

INC  Die  adressierte  Speicherzelle  wird  um  1  erhöht
    (incrementiert)
    AB  -        - $ EE - 6       Flags: N Z
    ZP  -        - $ E6 - 5
    ZPX - ,x     - $ F6 - 6
    ABX - ,x     - $ FE - 7

INX X Register incrementieren
    IMP - $ E8 - 2       Flags: N Z

INY Y Register incrementieren
    IMP - $ C8 - 2       Flags: N Z
```

**** Vergleichs - Befehle ***************

CMP Adressiertes Byte ohne C Flag vom ACCU subtrahieren. ACCU wird nicht verändert. Nur Flags werden gesetzt

```
IM  - #      - $ C9 - 2
AB  -        - $ CD - 4       Flags: N Z C
ZP  -        - $ C5 - 3
INX - ( ,x)  - $ C1 - 6
INY - ( ),y  - $ D1 - 5
ZPX - ,x     - $ D5 - 4
ABX - ,x     - $ DD - 4
ABY - ,y     - $ D9 - 4
```

CPX Adressiertes Byte ohne C Flag vom X Register subtrahieren. X Register wird nicht verändert. Nur Flags werden gesetzt

```
IM  - #      - $ EO - 2
AB  -        - $ EC - 4       Flags: N Z C
ZP  -        - $ E4 - 3
```

CPY Adressiertes Byte ohne C Flag vom Y Register subtrahieren. Y Register wird nicht verändert. Nur Flags werden gesetzt

```
IM  - #      - $ CO - 2
AB  -        - $ CC - 4       Flags: N Z C
ZP  -        - $ C4 - 3
```

BIT Der ACCU und das adressierte Byte werden bitweise UND verknüpft. Der ACCU wird nicht verändert. Nur die Flags werden gesetzt.

```
AB  -        - $ 2C  - 4      Flags: N Z V
ZP  -        - $ 24  - 3
```

**** Logische Operationen ***************

AND Der ACCU und das adressierte Byte werden bitweise UND verknüpft. Ergebnis in ACCU

```
IM  - #      - $ 29 - 2
AB  -        - $ 2D - 4       Flags: N Z
ZP  -        - $ 25 - 3
INX - ( ,x)  - $ 21 - 6
INY - ( ),y  - $ 31 - 5
ZPX - ,x     - $ 35 - 4
ABX - ,x     - $ 3D - 4
ABY - ,y     - $ 39 - 4
```

ORA Der ACCU und das adressierte Byte werden bitweise ODER verknüpft. Ergebnis in ACCU

```
IM  - #      - $ 09 - 2
AB  -        - $ 0D - 4       Flags: N Z
ZP  -        - $ 05 - 3
INX - ( ,x)  - $ 01 - 6
INY - ( ),y  - $ 11 - 5
ZPX - ,x     - $ 15 - 4
ABX - ,x     - $ 1D - 4
ABY - ,y     - $ 19 - 4
```

EOR Der ACCU und das adressierte Byte werden bitweise Exclusiv-Oder verknüpft. Ergebnis in ACCU

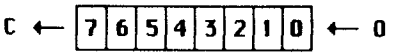
```
IM  - #      - $ 49 - 2
AB  -        - $ 4D - 4       Flags: N Z
ZP  -        - $ 45 - 3
INX - ( ,x)  - $ 41 - 6
INY - ( ),y  - $ 51 - 5
ZPX - ,x     - $ 55 - 4
ABX - ,x     - $ 5D - 4
ABY - ,y     - $ 59 - 4
```

**** Verschiebebefehle ******************

ASL Das adressierte Byte wird um 1 Bit nach links geschoben. Das linke Bit kommt in das C Flag. Ein Nullbit wird nachgeschoben

AB	-	$ OE	- 6
ZP	-	$ O6	- 5
AC	- A	$ OA	- 2
ZPX	- ,x	$ 16	- 6
ABX	- ,x	$ 1E	- 7

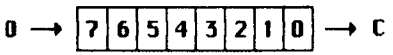
Flags: N Z C



LSR Das adressierte Byte wird um 1 Bit nach rechts geschoben. Das rechte Bit kommt in das C Flag. Ein Nullbit wird nachgeschoben

AB	-	$ 4E	- 6
ZP	-	$ 46	- 5
AC	- A	$ 4A	- 2
ZPX	- ,x	$ 56	- 6
ABX	- ,x	$ 5E	- 7

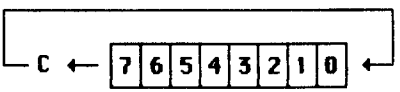
Flags: N Z C



ROL Das adressierte Byte wird um 1 Bit nach links geschoben. Das linke Bit kommt in das C Flag. Das C Flag wird nachgeschoben

AB	-	$ 2E	- 6
ZP	-	$ 26	- 5
AC	- A	$ 2A	- 2
ZPX	- ,x	$ 36	- 6
ABX	- ,x	$ 3E	- 7

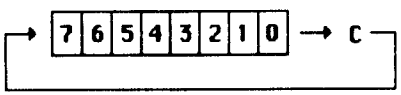
Flags: N Z C



ROR Das adressierte Byte wird um 1 Bit nach rechts geschoben. Das rechte Bit kommt in das C Flag. Das C Flag wird nachgeschoben

AB	-	$ 6E	- 6
ZP	-	$ 66	- 5
AC	- A	$ 6A	- 2
ZPX	- ,x	$ 76	- 6
ABX	- ,x	$ 7E	- 7

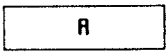
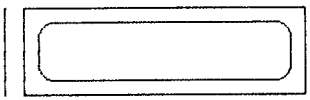
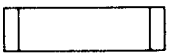
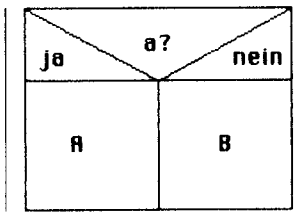
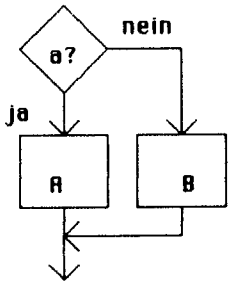
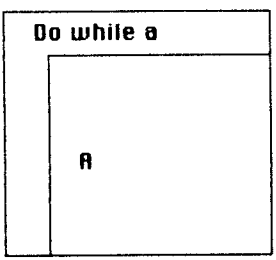
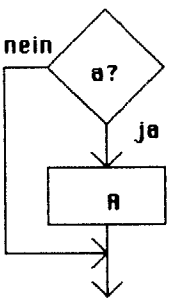
Flags: N Z C

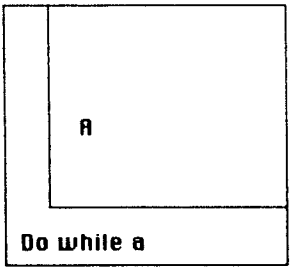


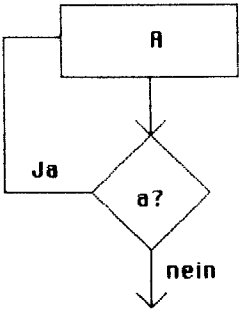
```
**** Flags setzen ***********************

CLC C Flag löschen
    IMP - $ 18 - 2

CLD D Flag löschen. Binärbtrieb ein.
    IMP - $ D8 - 2

CLI Interrupt Flag löschen. IRQ möglich.
    IMP - $ 58 - 2

CLV V Flag löschen
    IMP - $ B8 - 2

SEC C Flag setzen
    IMP - $ 38 - 2

SED D Flag setzen. Dezimalbetrieb ein.
    IMP - $ F8 - 2

SEI Interrupt Flag setzen. Kein IRQ möglich.
    IMP - $ 78 - 2

**** Unbedingte Sprünge ****************

JMP Springe an angegebene Adresse
    AB  -       - $ 4C - 3       Flags: keine
    IND - ( )   - $ 6C - 5

**** Bedingte Sprünge ******************

BCC Springe wenn C=0 ......... R - $ 90
BCS Springe wenn C=1 ......... R - $ B0
BNE Springe wenn Z=0 ......... R - $ D0
BEQ Springe wenn Z=1 ......... R - $ F0
BPL Springe wenn N=0 ......... R - $ 10
BMI Springe wenn N=1 ......... R - $ 30
BVC Springe wenn V=0 ......... R - $ 50
BVS Springe wenn V=1 ......... R - $ 70

    Alle Branchbefehle benötigen 2 Zyklen.
```

```
**** Stapelspeicher Operationen *********

PHA ACCU in Stapelspeicher
    IMP - $ 48 - 3       Flags: keine

PHP Statusregister in Stapelspeicher
    IMP - $ 08 - 3       Flags: keine

PLA Stapelspeicher in ACCU
    IMP - $ 68 - 4       Flags: keine

PLP Stapelspeicher in Statusregister
    IMP - $ 28 - 4       Flags: keine

**** Unterprogramm Befehle *************

JSR Springe in Unterprogramm an angegebener Adresse
    AB  -       - $ 20 - 6       Flags: keine

RTS Rücksprung von Unterprogramm
    IMP - $ 60 - 6       Flags: keine

**** Interrupt Befehle ******************

BRK IRQ Interrupt Softwaremäßig erzeugen.  IRQ  Vector  ($  FFFE)
    zeigt auf Adresse
    IMP - $ 00 - 7       Flags: B

RTI Rücksprung von Interrupt Routine
    IMP - $ 40 - 6
```

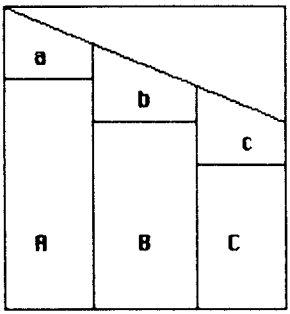
STRUKTOGRAMME / ABLAUFDIAGRAMME

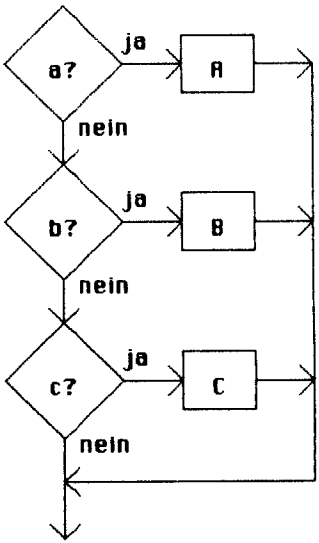
Struktogramm	Ablaufdiagramm	Code Beispiel
		LDA $00
		JSR HY
		BNE LAB LDA #$500 LAB LDA #$FF
		LAB BNE END INX JMP LAB END ...






```
LAB INX
BNE LAB
END ...
```






```
BNE N1
LDA $00
JMP END
N1 BMI N2
LDA $01
JMP END
N2 LDA $02
END ...
```

I N D E X

BRK	112
BVC	45
BVS	45
CLC	75
CLD	75
CLI	75
CLV	75
CMP	59
CPX	59
CPY	59
DEC	47
DEX	47
DEY	47
EOR	66
INC	47
INX	47
INY	47
JMP	44
JSR	44
LDA	23
LDX	23
LDY	23
LSR	67
NOP	120
ORA	66
PHA	89
PHP	89
PLA	89
PLP	89
ROL	67
ROR	67
RTI	111
RTS	44
SBC	78
SEC	75
SED	75
SEI	75
STA	24
STX	24
STY	24
TAX	66

LITERATURVERZEICHNIS

1 Englisch - Das Maschinensprachebuch für Fortgeschrittene
Data Becker Verlag

2 Angerhausen / Brückmann / Englisch / Gerits
64 intern
Data Becker Verlag

3 Plenge - Das Grafikbuch zum C 64
Data Becker Verlag

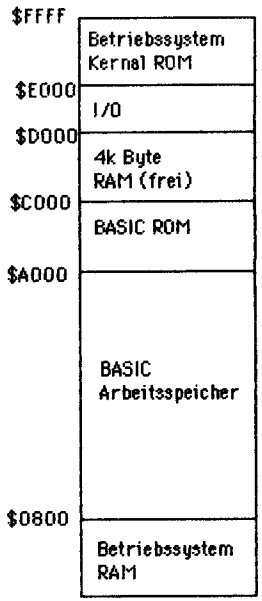
4 Dachsel - Das Musikbuch zum C 64
Data Becker Verlag

5 Zaks / Lesea
Mikroprozessor Interface Techniken
Sybex Verlag

6 Haugg
Software Engineering und ihre Qualitätssicherung
Franzis Verlag

C 64 Speichersteuerung

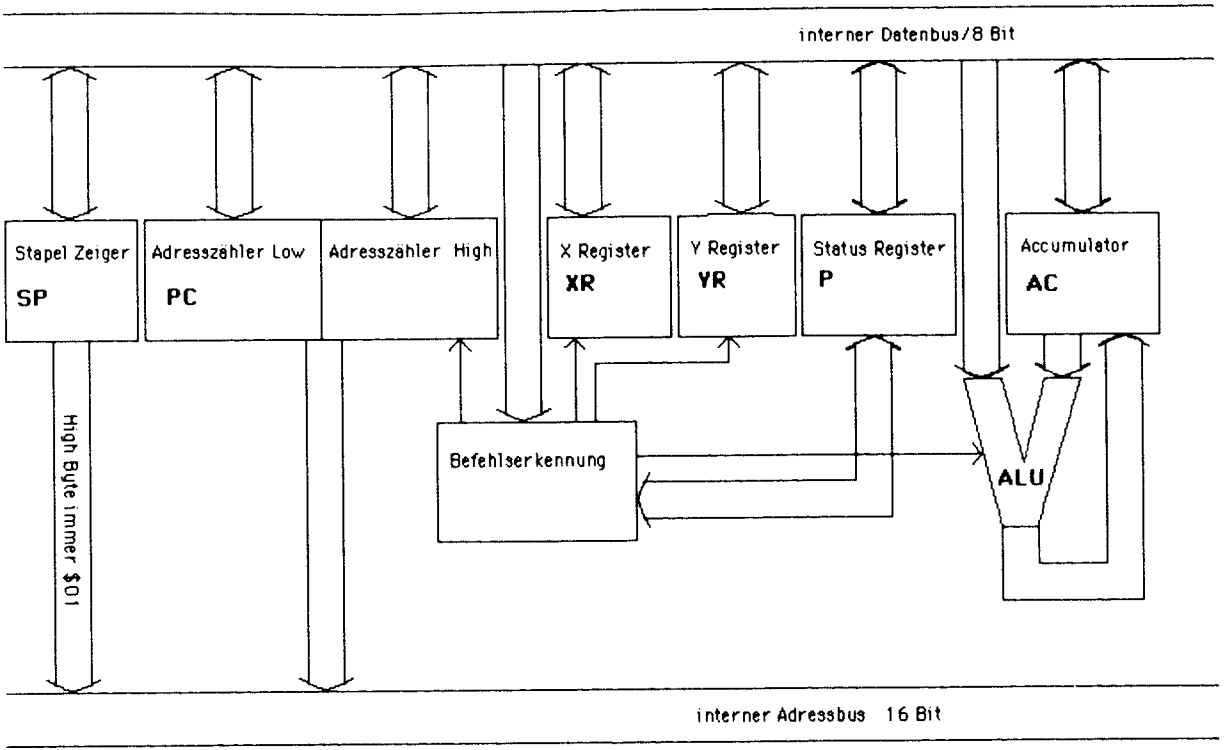
Prozessor Port		Exopansion Slot		
LORAM	HIRAM	EXROM	GAME	Wirkung
1	1	1	1	Einschaltzustand
0	1	X	1	BASIC ROM ($A000 - $BFFF) durch RAM ersetzen
1	0	X	1	Alles ROM ($A000 - $BFFF) und $E000 - $FFFF durch RAM ersetzen. Nur I/O Bereich ei
0	0	1	1	Nur RAM eingeschaltet
1	1	0	0	$8000 - $C000 = ext. ROM
0	1	0	0	$A000 - $C000 = ext. ROM
1	1	0	1	$8000 - $A000 = ext. ROM
CHAREN immer = 1				



C 64 Speicherbelegung

direkt nach dem Einschalten

Programmiermodell 65xx



DAS STEHT DRIN:

Dieses Buch bietet dem interessierten Programmierer eine leichtverständliche Einführung in die weitverbreiteten Assembler PROFI-ASS, SM MAE und T.EX.AS. mit hilfreichen Tips & Tricks, Hinweisen und Zusatzprogrammen. Gleichzeitig dient es als praxisorientiertes Handbuch mit Erläuterungen wichtiger Begriffe und Befehle.

Aus dem Inhalt:

- Arbeiten mit dem Monitor
- Disassemblieren
- Mnemonics, Labels, Variable
- Adressierung
- Tabellen- und Variablenzuweisung
- Wohin mit dem Objektcode?
- Bedingte Assemblierung
- Druckerausgabe
- Macros

  und vieles mehr

UND GESCHRIEBEN HAT DIESES BUCH:

Heribert Schmidt ist als Buch und Softwareautor (Finanzgenie, Trainingsbuch zu Datamat) spezialisiert auf Problemlösungen mit Mikrocomputern für Selbständige und Privathaushalte - wobei er besonderen Wert auf den praktischen Einsatz der Anwendungen legt.

ISBN 3-89011-071-1