Vüllers

C 16
C 116
Plus/4

Maschinensprache

EIN DATA BECKER BUCH

Vüllers

C 16
C 116
Plus/4

Maschinensprache

EIN DATA BECKER BUCH

ISBN 3-89011-206-4

2. überarbeitete Auflage

Copyright © 1987 DATA BECKER GmbH
Merowingerstraße 30
4000 Düsseldorf

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

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

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

Vorwort

Die heutigen Homecomputer wie z.B. der C16, C116 oder Plus/4 sind alle mit der Programmiersprache BASIC ausgerüstet. Diese Programmiersprache ist leicht erlernbar und in den meisten Fällen auch ausreichend. BASIC gehört zu den sogenannten höheren Programmiersprachen wie auch C oder Pascal, die auch als problemorientierte Programmiersprachen bezeichnet werden.

Im Gegensatz zu diesen höheren Programmiersprachen steht die Maschinensprache, also die Programmiersprache, die der Prozessor eigentlich versteht. Der Prozessor ist das eigentliche Arbeitstier in Ihrem Computer. Er sorgt dafür, daß sich überhaupt etwas "tut".

Maschinensprache ist gegenüber BASIC bei weitem nicht so leicht erlernbar und komfortabel, doch steht diesem die extrem hohe Geschwindigkeit der Maschinensprache gegenüber. Dazu ein Beispiel:

Programmieren Sie schon eine zeitlang in BASIC, so sind Sie sicherlich auch schon einmal an die Stelle gekommen, an der Ihr BASIC-Programm einfach zu langsam war. Oder Sie standen vor dem Problem, etwas programmieren zu wollen, was von BASIC aus nicht zu programmieren war, weil Sie dazu in die innere Struktur Ihres Rechners hätten eingreifen müssen.

An dieser Stelle setzt dann das Anwendungsgebiet der Maschinensprache ein. Diese ist im Gegensatz zum eingebauten BASIC bis zu 1000 mal schneller. Ein Nachteil der Maschinensprache ist aber ohne Zweifel, wie oben schon erwähnt, der hohe Schwierigkeitsgrad dieser Programmiersprache. Um diese Hürde zu überwinden und jedem Anwender eines C16, C116 oder Plus4 die Maschinensprache zugänglich zu machen, habe ich dieses

Buch geschrieben. Ich hoffe, hierdurch einer großen Zahl von Programmierern die Technik der Maschinenspracheprogrammierung zur Verfügung zu stellen und Ihnen zu helfen, Ihre BASIC-Programme noch effektiver zu gestalten.

Dieter Vüllers, im Mai 1986

Achtung: **Aus drucktechnischen Gründen ist der "Hochpfeil" in diesem Buch als ^ dargestellt.**

Inhaltsverzeichnis

1. Einführung

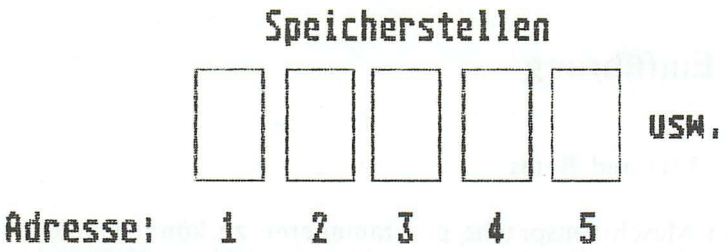
1.1 Bits und Bytes

Um in Maschinensprache programmieren zu können, müssen wir zuerst einmal die Grundlagen hierzu erklären. Das Bindeglied zwischen BASIC und der Maschinensprache stellen unter anderem die beiden BASIC-Befehle PEEK und POKE dar. Liest man sich jedoch im Handbuch die Erklärung des POKE-Befehls durch, so liest man dort etwas von RAM, Adressen, Bytes und so weiter. Wie ist diese Erklärung nun zu verstehen?

Bestimmt haben Sie auch schon gehört, daß Ihr Computer 16K RAM (C16) besitzt. Was bedeutet nun RAM? RAM ist eine englische Abkürzung und bedeutet "Random Access Memory" oder auf Deutsch "Speicher mit wahlfreiem Zugriff". Es handelt sich also um die Bezeichnung für einen Speicher.

In der normalen Umgangssprache verstehen wir unter einem Speicher einen Raum oder etwas ähnliches, in dem wir etwas ablegen oder speichern können. Wir können also etwas in unseren Speicher legen und es bei Bedarf wieder herausnehmen. Genauso können Sie sich auch einen Computerspeicher vorstellen. Unser Computer besitzt 16384 (C16) Schubladen, in denen wir etwas ablegen können. Wenn wir wollen, können wir auch in diesen Schubladen nachschauen, was sie enthalten oder etwas anderes in sie hineinlegen.

In der Computerfachsprache bezeichnet man eine solche Schublade als Speicherstelle oder Speicherplatz. Stellt man sich nun vor, diese Schubladen wären alle in einer Reihe nebeneinander angeordnet, so können wir jeder Schublade eine Nummer oder auch Adresse zuordnen. Die Speicherstelle 3 wäre dann die dritte Schublade von links. Symbolisch kann man sich einen Computerspeicher daher so vorstellen:



Da wir einen Computer besitzen und Computer auch als "Rechenmaschinen" bezeichnet werden, ist es logisch, daß man in einen Computerspeicher nicht irgendetwas hineinpacken kann, sondern nur Zahlen.

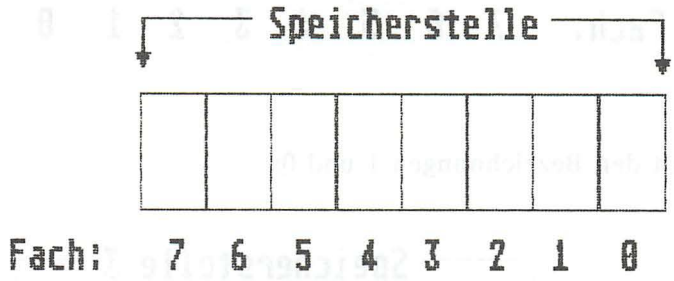
In jede Schublade oder Speicherstelle paßt eine Zahl zwischen 0 und 255, also beispielsweise 25 oder 129. Eine solche Zahl im Bereich von 0 bis 255 bezeichnet man als ein Byte.
Noch einmal zusammengefaßt:

Ein Computerspeicher besteht aus nebeneinander liegenden Speicherzellen. Die Nummer einer Speicherzelle in dieser Reihe bezeichnet man als Adresse. In jede Speicherzelle passt ein Wert zwischen 0 und 255, ein sogenanntes Byte.

Kommen wir nun zum Aufbau eines Bytes. Bleiben wir hierzu bei dem Vergleich mit den Schubladen:

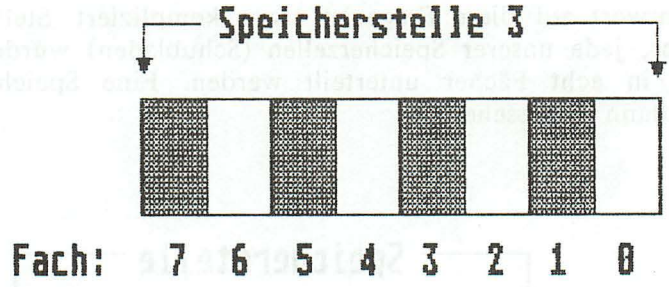
Wie oben schon gesagt wurde, stellt ein Byte eine Zahl im Bereich von 0 bis 255 dar, warum aber nicht eine Zahl im Bereich von 0 bis 100000?

Die Antwort auf diese Frage ist etwas kompliziert. Stellen Sie sich vor, jede unserer Speicherzellen (Schubladen) würde noch einmal in acht Fächer unterteilt werden. Eine Speicherzelle würde dann so aussehen:

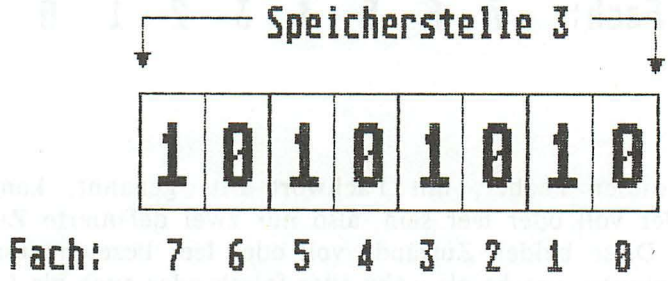


Jedes dieser Fächer, mit Fachwort Bits genannt, kann nun entweder voll oder leer sein, also nur zwei definierte Zustände haben. Diese beiden Zustände voll oder leer bezeichnet man in der Computersprache als wahr oder falsch oder auch als 1 (wahr) oder 0 (falsch), anstelle von voll oder leer.
Dazu ein Beispiel:

Nehmen wir an, wir wollten etwas in die Speicherstelle 3 legen. Wir füllen z.B. das erste Fach, das dritte, das fünfte und das siebte Fach. Unsere Speicherzelle sähe dann jetzt so aus:



oder mit den Bezeichnungen 1 und 0:



Um nun zu erklären, wie sich aus dieser Folge von Nullen und Einsen eine Zahl im Bereich von 0 bis 255 ergibt, wie sie der Computer in einem seiner Speicherplätze ablegen kann, schauen wir uns zuerst einmal an, wie das uns bekannte Dezimalsystem funktioniert:

Im Dezimalsystem existieren zehn Ziffern, daher auch der Name 'Dezimal'-System. Schauen wir uns einmal an, wie sich der Wert einer Zahl anhand der einzelnen Ziffern berechnen läßt. Nehmen wir als Beispiel einmal die Zahl 1986:

```
Stelle: 3 2 1 0
Ziffer: 1 9 8 6
```

Wie wir sehen, gibt die nullte Stelle die Einerwerte an, die erste die Zehnerwerte, die zweite die Hunderterwerte und die dritte Stelle die Tausenderwerte. Der Unterschied zweier aufeinanderfolgender Stellen liegt also immer im Faktor 10 (1, 10, 100, 1000, 10000).

Diese Werte (1, 10, 100) entsprechen den Zehnerpotenzen 10^0, 10^1, 10^2.

Der Wert der Zahl 1986 errechnet sich also folgendermaßen:

```
  Ziffer der 0. Stelle mal 1    (10^0)
+ Ziffer der 1. Stelle mal 10   (10^1)
+ Ziffer der 2. Stelle mal 100  (10^2)
+ Ziffer der 3. Stelle mal 1000 (10^3)
= 6*1 + 8*10 + 9*100 + 1*1000 = 1986
```

Analog funktioniert dieses System jetzt auch, wenn wir nur zwei Ziffern haben. Dieses System nennt man dann Binärsystem. Entsprechend der Anzahl der Ziffern im Binärsystem hat jede nächsthöhere Stelle in einem Byte auch nicht den zehnfachen Wert, sondern nur den zweifachen. Es handelt sich also nicht immer um Zehnerpotenzen, sondern nur um Zweierpotenzen.

Die nullte Stelle gibt daher die Einerwerte (2^0=1), die erste Stelle die Zweierwerte (2^1=2), die zweite Stelle die Viererwerte (2^2=4) usw. an. Der dezimale Wert einer Binärzahl errechnet sich daher folgendermaßen, nehmen wir hierzu wieder unser Byte von vorhin:

```
Stelle:    7 6 5 4 3 2 1 0
Binärzahl: 1 0 1 0 1 0 1 0
```

```
  Ziffer der 0. Stelle mal 1   (2^0)
+ Ziffer der 1. Stelle mal 2   (2^1)
+ Ziffer der 2. Stelle mal 4   (2^2)
+ Ziffer der 3. Stelle mal 8   (2^3)
+ Ziffer der 4. Stelle mal 16  (2^4)
+ Ziffer der 5. Stelle mal 32  (2^5)
+ Ziffer der 6. Stelle mal 64  (2^6)
+ Ziffer der 7. Stelle mal 128 (2^7)
=   0*  1 + 1*  2 + 0*  4 + 1*  8
  + 0* 16 + 1* 32 + 0* 64 + 1*128
  = 2+8+32+128
  = 170
```

Sicherlich verstehen Sie jetzt auch, wieso man mit einem Byte nur Zahlen im Bereich von 0 bis 255 darstellen kann. Sind alle Bits gleich 0, so ergibt sich der Wert 0, sind dagegen alle Bits gleich 1 so ergibt sich der Wert 1+2+4+8+16+32+64+128 = 255. Mit einem Byte lassen sich also 256 verschiedene Zahlen darstellen (0 bis 255).

Noch einmal zusammengefaßt:

Ein Computerspeicher besteht aus nebeneinander liegenden Speicherstellen. Die Nummer einer Speicherstelle in dieser Reihe bezeichnet man als Adresse. In jede Speicherstelle paßt ein Byte. Ein Byte besteht aus acht Bits, die jeweils nur den Wert 1 oder 0 annehmen können. Jede Stelle entspricht der ihr zugehörigen Zweierpotenz. Die dritte*) Stelle entspräche dann z.B. 2^3=8, sofern das entsprechende Bit gesetzt ist. Addiert man die Werte aller gesetzten Bits, so erhält man den Wert des gesamten Bytes.

*) Null mitgezählt!

Sicherlich ist Ihnen auch schon aufgefallen, daß Binärzahlen recht schwierig zu handhaben sind. Deshalb hat man noch ein zweites Zahlensystem eingeführt, das sogenannte Hexadezimalsystem. Dieses Hexadezimalsystem ergibt sich aus der Aufteilung eines Bytes in zwei Teile, den sogenannten Nibbles. Ein Byte sieht aufgeteilt dann so aus:

```
      Halb-     Halb-
      byte      byte
Bit: 7 6 5 4   3 2 1 0
```

Betrachtet man jetzt jedes Halbbyte getrennt, also

```
Bit:  3  2  1  0 - 3  2  1  0
Wert: 8  4  2  1 - 8  4  2  1

=    2^3 2^2 2^1 2^0 2^3 2^2 2^1 2^0
```

so erkennt man, daß sich durch jedes Halbbyte eine Zahl im Bereich von 0 bis 15 (1+2+4+8=15) darstellen läßt. Um jetzt jedes Halbbyte durch eine Ziffer darstellen zu können, benötigte man ein Zahlensystem, das sechzehn Ziffern besitzt. Da aber nur die Ziffern 0 bis 9 existieren, hat man noch die ersten Buchstaben des Alphabets mit zur Hilfe genommen.

Die Ziffern des Hexadezimalsystems lauten daher:

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

Um jetzt z.B. die Dezimalzahl 113 im Hexadezimalsystem auszudrücken, geht man folgendermaßen vor:

1. Man stellt die Zahl im Binärsystem dar. Die Zahl 113 sieht dann folgendermaßen aus:

```
Bit:  7 6 5 4 3 2 1 0
Wert: 0 1 1 1 0 0 0 1
```

2. Man teilt das Byte in zwei Halbbytes, die für die Zahl 113 dann so aussehen:

```
Bit:  3 2 1 0   3 2 1 0
Wert: 0 1 1 1   0 0 0 1
```

3. Man berechnet für jedes Halbbyte den dezimalen Wert im Bereich von 0 bis 15:

```
Wert: 0 1 1 1   0 0 0 1
      0+4+2+1   0+0+0+1
        7         1
```

4. Dann zieht man die beiden erhaltenen Ziffern zu einer Zahl zusammen. Für die dezimale Zahl 113 lautet die hexadezimale Darstellung dann $71.

Will man umgekehrt vorgehen, verfährt man wie beim Binär- oder Dezimalsystem. Wir erinnern uns:

Beim Dezimalsystem hatte jede Stelle den Wert ihrer Zehnerpotenz, beim Binärsystem jede Stelle den Wert ihrer Zweierpotenz. Folglich hat im Hexadezimalsystem jede Stelle den Wert ihrer Sechzehnerpotenz. Der dezimale Wert der hexadezimalen Zahl $D3 berechnet sich daher folgendermaßen:

3 * 16^0 + D * 16^1 = 3 * 1 + 13 * 16 = 211

Zur Unterscheidung der drei bekannten Zahlensysteme stellt man jeder Zahl ein Erkennungssymbol voran. Dezimalzahlen erkennt man daran, daß sie kein vorangestelltes Symbol besitzen.

Hexadezimalzahlen ist ein Dollarzeichen "$" vorangestellt, während Binärzahlen an einem vorangestellten Prozentzeichen "%" zu erkennen sind.

1.2 Von BASIC zu Maschinensprache

Nachdem wir nun über die Grundbegriffe der Maschinensprache informiert sind, wollen wir einmal versuchen, die Funktion des POKE-Befehls zu analysieren. Die Parameterübergabe des POKE-Befehls ist folgende:

POKE Adresse,Wert

Die Adresse muß im Bereich von 0-65535 und der Wert im Bereich von 0 bis 255 liegen. Sicherlich können Sie sich jetzt vorstellen, was mit dieser "Adresse" gemeint ist, denn wir haben diesen Begriff ja schon einmal besprochen. Es ist nämlich wieder die Nummer der Speicherstelle, oder -wie in unserem Beispiel- die Nummer der Schublade gemeint. Der zweite Parameter, der Wert darf irgendein gültiger mathematischer Ausdruck sein, dessen Erbegnis im Bereich von 0 bis 255 liegt, also durch ein Byte dargestellt werden kann. Mit dem POKE-Befehl kann man also bestimmte Werte in einer Speicherstelle ablegen. Geben Sie z.B. folgenden POKE-Befehl ein:

POKE 216,15 (RETURN)

Sie legen also den Wert 15 in der Speicherstelle 216 ab. Das Byte sieht nun in der Binärdarstellung so aus:

```
Bits: 7 6 5 4 3 2 1 0
Byte: 0 0 0 0 1 1 1 1
Wert: 1 6 3 1 8 4 2 1
      2 4 2 6
      8
=             8+4+2+1
=               15
=                   $0F (Hexadezimal)
```

Wir kennen jetzt also schon einen Befehl, mit dem die Verbindung zwischen BASIC und der Maschinensprache hergestellt werden kann.

Analog zum POKE-Befehl gibt es natürlich auch noch eine BASIC-Funktion, mit der wir nachschauen können, welcher Wert in einer Speicherstelle enthalten ist. Geben Sie hierzu bitte folgendes ein:

PRINT PEEK (216) (RETURN-Taste)

(Die Zahl in der Klammer hinter der PEEK-Funktion ist wieder die Adresse der Speicherstelle, die angesprochen werden soll!)

Als Erbegnis steht jetzt die Zahl 15 auf dem Bildschirm. Geben Sie jetzt bitte noch einmal das gleiche ein und drücken Sie wieder die RETURN-Taste. Sie sehen, das Ergebnis ist 15; der Inhalt einer Speicherstelle wird beim Auslesen also nicht verändert.

Noch einmal zusammengefaßt:

Mit dem POKE-Befehl ist es möglich, einen bestimmten Wert in eine bestimmte Speicherstelle zu schreiben. Analog hierzu ist es mit der PEEK-Funktion möglich, diesen Wert wieder auszulesen. Der Inhalt der Speicherstelle wird durch das Auslesen nicht verändert.

1.3 Der Maschinensprachemonitor

Um Maschinenprogramme einzugeben müßte man eigentlich direkt die einzelnen Bits der Speicherstellen ändern. Maschinensprache wäre dann extrem schwierig zu programmieren. Man müßte schließlich für jeden Befehl den Befehlskode aus einer Tabelle ermitteln, relative Sprünge selbst berechnen usw.

Aus diesem Grund gibt es die sogenannten Assembler. Assembler erlauben die Eingabe der Maschinensprachebefehle als sogenannte Mnemonics. Mnemonics sind praktisch Symbole für

die einzelnen Maschinensprachebefehle. Der Mnemonic LDA #$10 entspräche in der Maschinensprache der Bytefolge $A9 $10.

In diesem Buch besprechen wir die Programmierung hauptsächlich anhand der Mnemonics, da diese die Programmierung in Maschinensprache wesentlich erleichtern. Der Vollständigkeit halber sind natürlich auch die Befehlskodes in hexadezimaler Form angegeben.

Um auf Ihrem C16 (C116, Plus/4) in Maschinensprache zu programmieren, empfielt sich der eingebaute Monitor. Mit diesem (Maschinensprache)monitor ist natürlich kein Datensichtgerät gemeint, sondern ein Programm mit dem Sie Maschinenspracheprogramme in Form der eben beschriebenen Mnemonics eingeben können. Außerdem können Sie sich noch Speicherinhalte anschauen, diese verändern, vergleichen, laden, speichern usw. Maschinenprogramme können auch in Form eines Disassemblerlistings angezeigt werden. Unter einem Disassembler versteht man ein Programm, daß den reinen Maschinenkode in Form von Mnemonics, also Schlüsselwörtern anzeigt.

Um noch einmal auf die Mnemonics zurückzukommen: Ein Mnemonic besteht immer aus drei Buchstaben, die eine Abkürzung der Befehlsfunktion darstellen. Als Beispiel sei an dieser Stelle der Befehl TAX angeführt. Diese Abkürzung steht für Transfer Akku to X-Register. Sie sehen unter einem Mnemonic kann man sich als Programmierer eher etwas vorstellen als unter der Zahl $AA. Auf die einzelnen Assemblerbefehle werden wir in den anschließenden Kapiteln natürlich noch genauer eingehen. Doch nun erst einmal zu den Funktionen, die uns der Maschinensprachemonitor bietet. Um diese Funktionen anzusprechen, müssen wir den Maschinensprachemonitor erst aufrufen. Hierzu dient der BASIC-Befehl MONITOR, gefolgt von der RETURN-Taste. Der Maschinensprachemonitor meldet sich nun folgendermaßen:

```
MONITOR

  PC  SR AC XR YR SP
;FF00 00 00 FF 00 F8
```

Der Cursor steht unter dieser Meldung, die man auch als Monitorprompt bezeichnet. Diese Meldung zeigt uns an, das sich ab jetzt alle Befehle auf die Monitorfunktionen beziehen:

A:	Assemble	Assemblieren
C:	Compare	Vergleichen
D:	Disassemble	Disassemblieren
F:	Fill	Füllen
G:	Go	Starten
H:	Hunt	Suchen
L:	Load	Laden(Floppy, Recorder)
S:	Save	Speichern(Floppy, Recorder)
T:	Transfer	Kopieren
V:	Verify	Vergleichen(Floppy, Recorder)
X:	Exit	Ausgang zum BASIC

Außer diesen Befehlen existieren noch der Punkt, das Größer-Zeichen und das Semikolon. Zu diesen Befehlen kommen wir später noch. Doch jetzt erst einmal die Funktionsbeschreibung und Anwendung der einzelnen Monitorbefehle. Wenn möglich wird hierbei noch ein Beispiel gegeben.

A bzw. Assemble

Mittels des 'A'-Befehls können Sie Maschinenprogramme in der Mnemonicschreibweise eingeben. Hinter dem Befehl muß eine Adresse im 16-Bit-Format folgen. Wie alle Zahlenangaben im Monitor, muß auch diese Zahl im Hexadezimalsystem stehen. Die Angabe eines '$' vor der Adresse kann aber entfallen. Nach dieser Adresse folgt nun der Mnemonic und darauf folgend der Parameter. Dazu ein Beispiel:

```
A 3000 LDA #$50 (Returntaste)
^   ^   ^   ^
a   b   c   d
```

'a' ist in diesem Fall der Monitorbefehl für den Direktassembler. Hierauf folgt dann die Adresse (b), der kein '$' vorangestellt sein muß. Auf diese Adresse folgt nun der Befehlskode (c) mit seinem Parameter (d).

Nachdem eine Zeile eingegeben worden ist, muß diese Zeile mit dem Druck der RETURN-Taste abgeschlossen werden. Der Monitor gibt dann in der nächsten Zeile ein A und die neue Adresse an. Dies hat den Vorteil, daß Sie nicht jedesmal ausrechnen müssen, wie lang der Befehl mit seinen Parametern war und was demzufolge dann die neue Adresse wäre. In unserem Fall würde vom Monitor nach der RETURN-Taste folgendes ausgegeben werden:

```
A 3002
```

Der Cursor würde zwei Stellen hinter der Adresse stehen.
Die Befehlswörter (Mnemonics) müssen dem üblichen Standart entsprechen, der natürlich auch in diesem Buch benutzt wird. Zahlen und Adresse können leider nur in hexadezimaler Form eingegeben werden. Als Entschädigung hierfür berechnet der Monitor allerdings die Sprungziele aller relativen, bedingten Sprungbefehle. Sie brauchen also nur die Adresse wie beim JMP-Befehl eingeben. Tritt hierbei eine Bereichsüberschreitung, ein syntaktischer Fehler oder eine falsche Adressierungsart auf, so wird dies durch ein Fragezeichen (?) hinter der Zeile angezeigt.

Zuletzt muß nur noch gesagt werden, daß der Punkt (.) dem 'A'-Befehl völlig gleichgestellt ist, Sie können also auch in Disassemblerlistings (Siehe bei 'D') nach Herzenslust manipulieren.

C oder Compare

Mit dem Comparebefehl können mühelos zwei Speicherbereiche verglichen werden. Man gibt hierzu lediglich die Anfangs- und

Endadresse des ersten Bereichs und die Anfangsadresse des zweiten Bereichs an. Nach Druck der RETURN-Taste werden dann alle Adressen im zweiten Bereich angezeigt, die nicht den gleichen Inhalt haben, wie die korrespondierende Speicherstelle im ersten Bereich. Hierzu ein Beispiel:

Wir wollen die Bereichen $E000 bis $E003 mit dem Bereich ab $F000 vergleichen. Die beiden Bereiche sehen wie folgt aus:

```
$E000: $20 $A4 $CA $91
$F000: $05 $8C $37 $05
```

Geben Sie nun bitte die folgende Befehlsfolge ein:

```
C E000 E003 F000 (RETURN-Taste)
```

Es werden die folgenden Zahlen ausgegeben:

```
E000 E001 E002 E003
```

Wie Sie sehen können, stimmt keine der Zahlen überein, weder $20 mit $05 noch $A4 mit $8C. Daher werden alle Adressen ausgegeben, die überprüft wurden.

D oder Disassemble

Mittels des 'D'-Befehls kann man genau die dem 'A'-Befehl entgegengesetzte Funktion des Maschinensprachemonitors ansprechen. Mittels dieses Befehls ist es nämlich möglich Speicherinhalte als Mnemoniclisting anzuzeigen. Es werden also die Bytes, wenn möglich, in Befehlsworte wie LDA, STA usw. übersetzt. Stellt ein Byte im Speicher keinen gültigen Befehlskode dar, so werden drei Fragezeichen ausgegeben. Da jeder Zeile ein Punkt (.) vorangestellt ist (Siehe 'A'-Befehl), ist es möglich die Befehle durch einfaches Ändern der Befehlsworte zu ändern. Dazu ein Beispiel:

Geben Sie einmal die folgende Zeile ein und betätigen Sie anschließend die RETURN-Taste:

```
D A7B5 A7DD (RETURN-Taste)
```

Wie Sie sehen, wird vom Monitor das folgende Disassemblerlisting ausgegeben. Es handelt sich hierbei übrigens um den SYS-Befehl des BASIC-Interpreters:

```
. A7B5  20 E1 9D JSR $9DE1
. A7B8  A9 A7    LDA #$A7
. A7BA  48       PHA
. A7BB  A9 CE    LDA #$CE
. A7BD  48       PHA
. A7BE  AD F5 07 LDA $07F5
. A7C1  48       PHA
. A7C2  AD F2 07 LDA $07F2
. A7C5  AE F3 07 LDX $07F3
. A7C8  AC F4 07 LDY $07F4
. A7CB  28       PLP
. A7CC  6C 14 00 JMP ($0014)
. A7CF  08       PHP
. A7D0  8D F2 07 STA $07F2
. A7D3  8E F3 07 STX $07F3
. A7D6  8C F4 07 STY $07F4
. A7D9  68       PLA
. A7DA  8D F5 07 STA $07F5
. A7DD  60       RTS
```

Die Endadresse des Bereichs, der disassembliert werden soll, muß nicht unbedingt abgegeben werden. Wird z.B. nur die Anfangsadresse angegeben, so wird das Listing immer Stückweise ausgegeben. Will man sich dann das Listing noch weiter anschauen, so gibt man ein 'D' gefolgt von der RETURN-Taste ein.

F oder Fill

Mit dem 'F'-Befehl kann ein bestimmter Speicherbereich mit einem definierten Wert gefüllt werden. Die Parameterangabe ist daher wie folgt:

Zuerst die Anfangs- und die Endadresse des Bereichs, der gefüllt werden soll. Auf diese beiden Werte folgt dann ein 8-Bit-Wert, mit dem dann alle Speicherstellen dieses Speicherbereichs befüllt werden.

Beispiel:

```
F 3000 4000 00
```

Der Bereich von $3000 bis incl. $4000 wird mit dem Wert $00 gefüllt. Dieser Befehl wird meist dann angewendet, wenn für ein Programm bestimmte Ausgangsbedingungen geschaffen werden müssen.

G oder Go

Mittels des 'G'-Befehls können Maschinenprogramme im Speicher gestartet werden. Abgeschlossen sollten diese Programme durch einen BRK-Mnemonic. Dieser Befehl bewirkt einen Rücksprung in den Monitor. Der Programmzeiger (PC) zeigt nach einem solchen Break immer auf die folgende Adresse nach dem BRK-Befehl. Hierdurch ist es relativ einfach Maschinenprogramme zu prüfen, indem man an allen wichtigen Stellen BRK-Befehle einbaut und das Programm dann schrittweise austestet. Die Parameterübergabe beim 'G'-Befehl ist folgende:

Es wird nur die Startadresse des Programms hinter dem 'G'-Befehl angegeben. Bei einem Programmstart durch diesen Befehl werden die Registerinhalte entsprechend der Anzeige gesetzt (Siehe 'R'-Befehl).

Zum 'G'-Befehl jetzt noch ein Beispiel. Geben Sie bitte die folgende Zeile ein:

```
G 867C (RETURN-Taste)
```

Der 'G'-Befehl springt in diesem Fall direkt auf den Befehlskode $00, der der Kode für den BRK-Mnemonic ist. Wegen des BRK-Befehls wird dann wieder zum Monitor gesprungen, dessen Meldung dann so aussieht:

```
BREAK
   PC  SR AC XR YR SP
; 867E 30 00 FF 00 F8
```

Wie Sie sehen, steht der Programmzähler PC (siehe auch Kap. 1.4) auf der Adresse $867E, er zeigt also auf die dem BRK-Befehl folgende Speicherstelle. Dies ist so eingerichtet worden, um nach einem BRK-Befehl das Programm an der Stelle fortsetzen zu können, an der es auch abgebrochen wurde. Wird nur 'G' eingegeben und dann die RETURN-Taste betätigt, so wird das Maschinenprogramm gestartet, das an der Position des PC beginnt. In diesem Fall wäre das also die Adresse nach dem BRK-Befehl.

H bzw. Hunt

Um bestimmte Bytefolgen oder auch Texte im Speicher zu finden, dient dieser Befehl. Als Parameter erwartet er die Anfangs- und die Endadresse des zu durchsuchenden Bereichs. An diese beiden Adressen schließt sich dann noch unmittelbar die Bytefolge oder der Text an, der gesucht werden soll. Um z.B. die Bytefolge $0A $0B $0C im Bereich von $A000 bis $AFFF zu suchen, müßte man folgendes eingeben:

```
H A000 AFFF 0A 0B 0C
```

Als Ergebnis würden dann alle Speicherstellen ausgegeben werden, an denen eine solche Bytefolge beginnt.

Ein weiterer Leckerbissen ist es, auch nach Texten zu suchen. Will man z.B. nach dem Text 'READY.' im ROM suchen, so kann man hierzu folgende Variation des 'H'-Befehls anwenden:

```
H 8000 FFFF 'READY.
```

Vor dem Text muß also ein Hochkomma (kein Anführungszeichen!) stehen. Am Ende des Textes darf allerdings kein Hochkomma stehen, da dieses sonst als Teil des Suchtextes interpretiert werden würde.

L oder Load

Um Maschinenprogramme von einem externen Speichermedium, wie z.B. einer Floppy oder dem Recorder, in den Speicher zu bekommen, existiert der Monitorbefehl 'L'. Er enspricht in seiner Parameterübergabe in etwa dem normalen LOAD-Befehl des BASIC's. Der Name muß also in Anführungszeichen eingeschlossen sein und die Geräteadresse durch ein Komma vom Namen getrennt werden. Die Geräteadresse muß als zweistellige Hexzahl eingegeben werden. Dazu zwei Beispiele:

```
L "TEST",08 (RETURN-Taste)
```

Das Programm 'TEST' wird von der Floppy in den Speicher geladen.

```
L "TEST",01 (RETURN-Taste)
```

Das Programm 'TEST' wird in diesem Fall von der Datasette (recorder) in den Speicher geladen.

Durch den 'L'-Befehl werden Maschinenprogramme immer in den Speicherbereich geladen, aus dem Sie auch gespeichert wurden. Liegt das gespeicherte Programm also im Bereich zwischen $3000 und $3FFF, so wird es auch wieder in diesen Bereich geladen.

M oder Modify

Um sich Speicherbereiche in Hexadezimaldarstellung bzw. ASCII-Zeichen anzuschauen bzw. zu ändern, existiert der 'M'-befehl. Die Parameterübergabe ist identisch mit der des 'D'-Befehls, also entweder keine, eine oder zwei Adressen, die den anzuzeigenden Bereich kennzeichnen. Schauen wir uns z.B. einmal den Bereich von $A000 bis $A020 an. Geben Sie dazu die folgende Zeile ein:

```
M A000 A020 (RETURN-Taste)
```

Der Ausdruck sieht dann so aus:

```
>A000 80 76 38 93 16 82 38 AA :.68...8*
>A008 3B 20 80 35 04 F3 34 81 :; .5.34.
>A010 35 04 F3 34 80 80 00 00 :5.34....
>A018 00 80 31 72 17 F8 20 B0 :..12.8 0
>A020 A2 F0 02 10 03 4C 1C 99 :"0...l..
```

Alle Zeichen, die in diesem Fall unterstrichen sind, werden auf dem Bildschirm oder dem Drucker revers dargestellt. Um nun einen Speicherinhalt zu ändern fahren Sie am besten mit dem Cursor auf das zu ändernde Byte und geben das neue Byte an. Nachdem Sie dann die RETURN-Taste betätigt haben, wird das Byte im Speicher geändert und auch die ASCII-Darstellung korrigiert. Ein Ändern der Speicherinhalte durch das Ändern der ASCII-Darstellung ist nicht möglich.

Nun zu einem Befehl, der eigentlich schon im 'M'-Befehl integriert ist. Gemeint ist das Größer-Zeichen ('>'). Hinter diesem Zeichen folgt nun die Adresse und eine Bytefolge. Geben Sie z.B. einmal folgende Zeile ein:

```
>3000 00 01 02 03 04 05 06 07 (RETURN-Taste)
```

Wenn wir uns jetzt mittels des 'M'-Befehls den Speicherinhalt anschauen, erhalten wird das folgende Ergebnis:

```
M 3000 (RETURN-Taste)
```

Ausdruck:

```
>3000 00 01 02 03 04 05 06 07 :..
```

Es steht also genau das im Speicher, was wir eingegeben haben. Das Größer-Zeichen dient also ebenfalls zum Ändern von Speicherstellen.

R oder Register

Der 'R'-Befehl zeigt den Inhalt der Prozessorregister an. Geben Sie 'R' gefolgt von der RETURN-Taste ein, so sieht das Ergebnis in etwa so aus:

```
    PC  SR  AC  XR  YR  SP
; FF00 00  00  FF  00  F9
```

Wollen Sie nun die Registerinhalte ändern, so überschreiben Sie die alten Werte. Bei einem Programmstart mit dem 'G'-Befehl werden die Prozessorregister dann so initialisiert, wie Sie dies vorgesehen haben.

Auch hierzu ein Beispiel:

```
A 3000 JSR $FFD2
A 3003 BRK
```

Dann geben Sie bitte 'R' ein und drücken die RETURN-Taste - Sie erhalten die aktuellen Registerinhalte. Fahren Sie nun mit dem Cursor auf die Hexzahl unter den Buchstaben 'ac'und schreiben dort '2a'. Dann starten Sie unser Maschinenprogramm mit dem Befehl:

```
G 3000 (RETURN-Taste)
```

Wie Sie sehen, wird ein Sternchen ('*') ausgegeben und dann wieder in den Monitor gesprungen. Ändern Sie nun den Wert 2A in den Wert 41 und drücken Sie dei RETURN-Taste. Wenn Sie nun das Maschinenprogramm wieder mit 'G 3000'starten, wird anstelle des Sternchens wieder ein 'A'ausgegeben.

Erklärung: Die Routine $FFD2 gibt ein Zeichen auf den Bildschirm aus. Der ASCII-Kode (siehe Tabelle im Handbuch) muß hierzu im Akkumulator stehen (Kap. 1.4). Da wir die Routine zweimal mit verschiedenen Akkuinhalten aufrufen, wird auch jedesmal ein anderes Zeichen ausgegeben.

Um die Registerinhalte zu ändern, reicht es auch schon, nur ein Semikolon (';'), gefolgt von den neuen Registerinhalten einzugeben. Dieser Vorgang könnte z.B. so aussehen:

```
; FF00 00 00 FF 00 F9
```

S oder Save

Das 'S'-Kommando dient zum Abspeichern von Maschinenprogrammen auf einem externen Speichermedium, wie dies z.B. eine Floppy oder ein Recorder ist. Nach dem Befehlsbuchstaben folgt in Anführungszeichen der Name, unter dem das Programm abgelegt werden soll. Dieser Name sollte eine Länge von 16 Zeichen nicht überschreiten. Dann folgen durch Kommata getrennt die Geräteadresse, die Anfangsadresse und die Endadresse des Speicherbereichs, der gespeichert werden soll. Die Geräteadresse muß als zweistellige Hexadezimalzahl eingegeben werden. Für den Recorder wäre das 01 und für die Floppy z.B. 08.

Um z.B. den Speicherinhalt von $3000 bis incl. $30FF auf dem Recorder zu sichern, muß man folgendes eingeben:

```
S "Programmname",01,3000,3100
```

Die zweite Adresse muß also immer um eins größer sein als die Endadresse des Programms im Speicher. Wie Sie sehen, muß bei der Endadresse $30FF im Speicher die Adresse $3100 ($30FF+1) angegeben werden.

T oder Transfer

Mit Hilfe dieses Befehls lassen sich Speicherbereiche kopieren. Man kann z.B. den Bereich von $3000 bis $3FFF in den Bereich $1000 bis $1FFF verschieben. Aufgrund dieses Verfahrens müssen drei Adressen hinter dem Befehl angegeben werden:

Als erstes die Anfangs- sowie die Endadresse des zu kopierenden Speicherbereichs und danach die Anfangsadresse des Be-

reichs, in den der Speicherbereich verschoben werden soll. Um den Bereich von $3000 bis $3FFF in den Bereich von $1000 bis $1FFF zu kopieren, muß man also folgendes eingeben:

```
T 3000 3FFF 1000 (RETURN-Taste)
```

Überlappen sich der Ausgangs- und Zielbereich, so ist nur eine Verschiebung von einem höher liegenden in einen tieferliegenden Bereich möglich.

V oder Verify

Mit diesem Befehl kann ein Programm auf Diskette oder Kassette mit einem Programm im Speicher verglichen werden. Differiert das Programm im Speicher von dem Programm auf dem Datenträger, so wird 'ERROR'ausgegeben, andernfalls erscheint nur der Cursor. Die Parameterübergabe des 'V'-Befehls ist denkbar einfach, sie entspricht der Übergabe beim 'L'-Befehl.

X oder Exit

Und nicht zuletzt der Befehl zum Verlassen des Monitors. Nach 'X' und der RETURN-Taste befindet man sich wieder im ganz normalen BASIC-Editor und kann BASIC-Programme eingeben. Die Monitorbefehle können jetzt nicht mehr angewendet werden. Um wieder in den Monitor zu gelangen, müssen Sie wieder MONITOR, gefolgt von der RETURN-Taste eingeben.

1.4 Der 7501-Mikroprozessor

Einleitend sei erst einmal gesagt, daß der 7501-Mikroprozessor zur Gruppe der 65XX-Prozessoren gehört und softwarekompatibel zu allen Prozessoren dieses Typs ist. Konkret heißt dies für Sie, daß Sie auch Literatur für den 6502 oder 6510 zu Rate ziehen können, oder bereits in 6502-Assembler geschriebene Programme nur noch an Ihr System, nicht aber an Ihren Prozessor anpassen müssen.

Bevor wir uns nun mit dem internen Aufbau des 7501 beschäftigen, sollte erst einmal geklärt werden, was man sich überhaupt unter einem Mikroprozessor vorzustellen hat:

Ein Prozessor ist ein Mikrochip, der vom Anwender programmiert werden kann. Er holt sich seine Befehle aus dem Speicher des Computers. Einige Bytes stellen Befehlswörter für den Prozessor dar; andere wiederum werden vom Prozessor nicht verstanden und führen zum sogenannten "Absturz" des Systems, d.h., nichts geht mehr. Abhilfe schafft in einem solchen Fall nur noch das Auslösen eines Resets (Resettaster) oder das Ausschalten des Geräts.

Ein Prozessor kann bei weitem nicht so komfortabel programmiert werden, wie man dies z.B. in BASIC oder Pascal kann. Er kennt praktisch nur Befehle zum Laden, Speichern und Bearbeiten von Bytes. Darüber hinaus kann er auch noch 8-Bit-Werte addieren oder subtrahieren und Vergleiche anstellen. Mehr ist ohne weitere Bausteine nicht möglich. Ist dagegen noch ein Bildschirmcontroller an den Prozessor angeschlossen, kann der Prozessor die Funktion dieses Chips kontrollieren und z.B. Buchstaben auf dem Datensichtgerät erscheinen lassen.

Sie sehen also, in Maschinensprache müssen Sie fast jede Problemlösung selber programmieren.

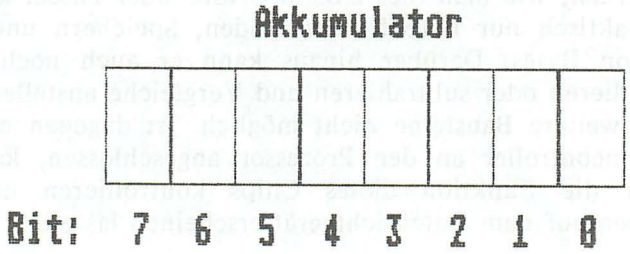
Doch nun zum internen Aufbau eines Prozessors:

Für die oben genannten Aufgaben besitzt der 7501 sechs Register. Diese Register sind kein Teil des normalen RAM-Speichers, sondern liegen innerhalb des Prozessors. Auf sie kann nur mittels der Maschinensprachebefehle zugegriffen werden.
Die einzelnen Register sind der Akkumulator, das X- und Y-Register, der Programmzähler, das Statusregister und der Stapelzeiger. Nun zum Aufbau und zur Funktion der einzelnen Register:

1. Der Akkumulator (Akku):

   Der Akku ist das Hauptarbeitsregister des Prozessors. In ihm laufen alle wesentlichen arithmetischen und logischen Befehle ab. Außerdem existieren für den Akku auch noch die meisten Adressierungsarten; doch dazu später noch. Weiterhin ist noch zu sagen, daß der Akkumulator ein 8-Bit-Register ist, er also nur mit Werten im Bereich von 0 bis 255 hantieren kann.

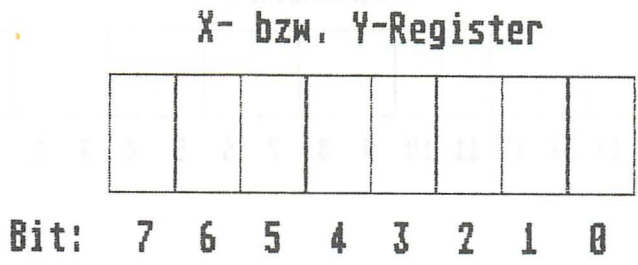
Aufbau des Akkumulators:



2. Das X-Register (XR):

   Wegen seiner Aufgabe als Zählregister wird dieses Register auch noch Indexregister genannt. Seine Hauptfunktion besteht in der Bearbeitung von Tabellen, die nur mittels des X oder Y-Registers möglich ist. Auch das X- und Y-Register sind 8-Bit-Register und umfassen daher nur den Wertebereich eines Bytes.

Aufbau des X- und Y-Registers:



3. Das Y-Register (YR):

   Prinzipiell entspricht dieses Register in Aufbau und Funktion dem X-Register. Der einzige Unterschied liegt in einem etwas anderen Reservoir an Adressierungsarten.

4. Der Programmzähler (PC):

   Der Programmzähler ist im Gegensatz zu allen anderen Registern ein 16-Bit-Register, daß heißt, er umfasst 16 Bits. Mit 16 Bit, einem sogenannten Adressbyte, kann man 2^16=65535 verschiedene Adressen ansprechen. Der Inhalt des Programmzählers zeigt immer auf die Adresse des nächsten zu bearbeitenden Befehls. Der Programmzeiger kann vom Benutzer nicht gesetzt oder gelöscht werden, sondern wird vom Prozessor selbst verwaltet.

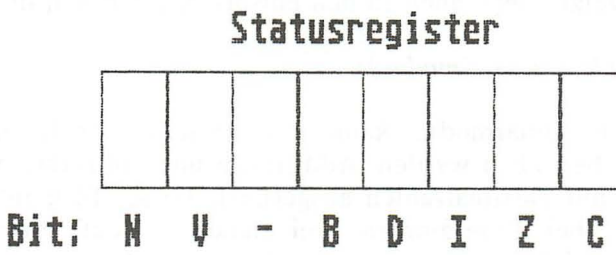
Aufbau des Programmzählers:

5. Das Statusregister: SR

Das Statusregister ist wiederum ein 8-Bit-Register von denen jedoch nur 7 eine Aufgabe besitzen. Es dient dazu, das Ergebnis eines Befehls oder eines Vergleiches festzuhalten; z.B., ob die letzte Zahl, mit der operiert wurde, gleich 0 oder größer als 127 gewesen ist usw.. Die einzelnen Bits des Statusregister bezeichnet man als sogenannte Flags. Sie sind im Zusammenhang mit den Vergleichs- und Sprungbefehlen von großer Bedeutung.

Ist ein Bit gleich 0, so ist das dementsprechende Flag falsch. Ist das Bit dagegen gleich 1, so ist das dementsprechende Flag wahr.

Aufbau des Statusregisters:



Die Flags haben nun folgende Bedeutung:

'C' Carryflag (Über- bzw. Unterlaufsflag):

Das Carryflag zeigt an, ob bei einer Addition ein Überlauf oder bei einer Subtraktion ein Unterlauf aufgetreten ist. Desgleichen gibt es auch darüber Auskunft, ob bei einem Vergleich der Vergleichswert kleiner oder größer als der Wert war, mit dem verglichen worden ist.

'Z' Zeroflag (Nullflag):

War das Ergebnis einer Operation gleich 0, so wird dieses Flag gesetzt. Weiterhin wird es auch gesetzt, wenn bei einem Vergleich beide Werte identisch, die Differenz also gleich 0 war.

'I' Interrupt Disable (Interruptflag):

Mit diesem Flag kann entschieden werden, ob ein Maschinenprogramm durch einen Interrupt (Unterbrechung) unterbrochen werden darf oder nicht. Ein Interrupt wird durch Periphe-

riebausteine ausgelöst, wie z.B. dem im TED eingebauten Timer. Ist dieses Flag gesetzt, so wird in keinem Fall zur Interruptroutine verzweigt (siehe auch in den entsprechenden Kapiteln).

'D' Decimal (Dezimalmodus):

Außer dem Binärmodus kann der Prozessor noch im Dezimalmodus betrieben werden. Additionen und Subtraktionen werden dann mit Dezimalzahlen ausgeführt. Dieses Flag interessiert meist nur bei Programmen, bei denen es auf eine höhere Rechengenauigkeit ankommt (z.B. Mathemat 64).

'B' Break (Abbruch):

Das Breakflag wird gesetzt, sofern ein Interrupt nicht durch ein Peripheriechip, sondern durch den Maschinensprachebefehl BRK ausgelöst worden ist. Dieser Befehl wird eigentlich nur zum Austesten von Assemblerprogrammen angewendet.

'V' Overflow (Überlauf):

Rechnet man mit positiven und negativen Zahlen, so wird das Vorzeichen durch das 7. Bit eines Bytes dargestellt. Das Overflowflag zeigt nun an, ob ein Überlauf in dieses Vorzeichenbit aufgetreten ist, da der Prozessor nicht erkennen kann, ob mit acht oder mit sieben Bit gerechnet wird. Auch dieses Bit ist im Rahmen dieses Buches ohne Interesse.

'N' Negativ (Negativflag):

Das Negativflag wird gesetzt, wenn bei einer Operation das Ergebnis größer als 127 war, das 7. Bit also gesetzt ist. Wie schon beim Overflowflag angeführt, kann man das 7. Bit als Vorzeichenbit benutzen.

6. Der Stapelzeiger (Stackpointer):

Der Stapelzeiger oder Stackpointer zeigt in den sogenannten Stapel oder Stack. Der Stack dient für die Ablage der Rücksprungadresse bei Unterprogrammen

und zu Sicherung der Registerinhalte. Die Funktion des Stacks und des Stackpointers werden wir in dem Kapitel über die Stackbefehle noch eingehend besprechen.

2. Der Befehlssatz des 7501

2.1 Die Adressierungsarten

Bevor ich Ihnen den Befehlssatz des 7501 vorstelle, erkläre ich zuerst einmal die verschiedenen Adressierungsarten. Unter den Adressierungsarten versteht man grob gesagt die Art und Weise, unter der der Operant eines Befehls ermittelt wird. Also z.B., ob er direkt angegeben, aus einer Speicherstelle gelesen oder mittels eines Zeigers bestimmt wird. Es gibt folgende Möglichkeiten zu bestimmen, von wo ein Operant stammen soll:

- Unmittelbare Adressierung
- Absolute Adressierung
- Zeropageadressierung
- Indizierte Adressierung
- Indizierte Zeropageadressierung
- Indirekt indizierte Adressierung
- Indiziert indirekte Adressierung

Bei der Erklärung der Adressierungsarten wird folgendermaßen vorgegangen:

Als erstes wird festgestellt, wie die Adressierungsart funktioniert. Dann wird ein Beispiel gegeben und gezeigt, was passiert und wie der Befehl mit seinen Parametern im Speicher abgelegt wird. Wenn möglich, wird dann auch noch gezeigt, wie die jeweilige Adressierungsart in BASIC zu realisieren gewesen wäre. Und zu guter letzt wird noch erklärt, wie die Syntax dieses Mnemonics (AssemblerBefehlswortes) ist.

1. Unmittelbare Adressierung

Eine unmittelbare Adressierung liegt vor, wenn der Wert, mit dem operiert werden soll, direkt hinter dem Befehl steht. Man operiert also unmittelbar mit einem konstanten Wert, der noch Teil des Befehls ist. Ein Beispiel für einen unmittelbaren Befehl wäre z.B. der folgende LDA-Befehl:

```
LDA #$0F
```

Nach der Ausführung dieses Befehls hätte der Akku den Inhalt $0F und der Programmzeiger wäre um zwei Positionen weitergerückt. Es handelt sich also um einen 2-Byte Befehl, der im Speicher folgendermaßen abgelegt wird:

1. Byte	2. Byte
Befehlskode	Operand mit dem der Akku geladen wird
A9	0F

Die Syntax dieses Befehls ist folgende:

Als erstes der Befehlsmnemonic, dann ein Gitterkreuz (#) und dann sofort der Wert mit dem der Akku geladen wird.

```
Befehlswort #Wert
```

2. Die absolute Adressierung

Will man nicht immer mit einem bestimmten Wert, sondern ständig mit dem Inhalt derselben Speicherstelle operieren, so bietet sich hierfür die absolute Adressierung an. Sie belegt drei Bytes, den Befehlskode und dann die Adresse der Speicherstelle, deren Inhalt in den Akkumulator geladen werden soll. Beispiel:

```
LDA $FF00
```

Nach der Ausführung dieses Befehls hat der Akku den gleichen Inhalt wie die angegebene Speicherstelle. In unserem Beispiel hätte der Akku also den Inhalt der Speicherstelle $FF00. Angenommen diese Speicherstelle hätte den Inhalt $05, so hätte auch der Akku nach der Ausführung des obigen Befehls den Inhalt $05.

Die Anordnung im Speicher ist nun schon etwas komplizierter. Als erstes steht dort natürlich das Befehlswort, dann folgt die Adresse. Da die Adresse im Bereich von 0 bis 65535 liegen kann, belegt sie auch zwei Bytes im Speicher. Die Anordnung

dieser beiden Bytes ist nun aber entgegen der Erwartung genau umgekehrt. Es wird nämlich zuerst das niederwertige Byte (LOW-Byte) und dann das höherwertige Byte (HIGH-Byte) im Speicher abgelegt. Unter dem niederwertigen Byte versteht man die untersten acht Bits des Adressbytes (Bits 0-7), während man unter einem höherwertigen Byte die obersten acht Bits (8-15) eines Adressbytes versteht. Diese Anordnung einer 16-Bitzahl im Speicher sollten Sie sich unbedingt merken, sie tritt noch sehr häufig auf.

Bei der absoluten Adressierung wird der Befehl also folgendermaßen im Speicher abgelegt:

1. Byte	2. Byte	3. Byte
Befehlskode	Adressbyte LOW	Adressbyte HIGH
AD	00	FF

Syntaktisch ist die absolute Adressierung wie folgt aufgebaut:

```
Befehlswort $Adresse
```

Es entfällt also nur das Gitterkreuz und der Parameter muß eine 16-Bitzahl sein (Ausnahme siehe 3.)

3. Zeropageadressierung

Diese Adressierungsart ist von der Funktion her völlig identisch mit der absoluten Adressierung. Man könnte sie daher sozusagen als absolute Zeropageadressierung bezeichnen. Jetzt erst einmal zur Klärung des Begriffes Zeropage:

Man kann den Speicher Ihres Computer in 256 Pages aufteilen. Eine Page besteht jeweils aus 256 Bytes (256*256=65536). Unter diesen Pages gibt es eine unterste Page mit der Nummer 0. Auf Englisch ist dies dann die Zeropage. Des öfteren findet man auch den Begriff Nullseite.

Der wesentliche Unterschied zwischen absoluter- und Zeropageadressierung liegt in der anzugebenden Adresse. Während die Adresse bei der absoluten Adressierung eine 16-Bit-Zahl war, ist sie bei der Zeropageadressierung nur eine 8-Bit-zahl. Man

kann mit der Zeropageadressierung also nur den Adressraum von 0 bis 255 bearbeiten, eben die sogenannte Zeropage.

Ein Beispiel für die Zeropageadressierung wäre z.B.:

```
LDA $FF
```

Es wird der Inhalt der Adresse $FF in den Akkumulator geladen. Die Anordnung im Speicher dürfte klar sein, da es sich - wie gesagt - nur um einen 2-Byte-Befehl handelt:

1. Byte	2. Byte
Befehlskode	$Adresse (0-255)

Syntax:

```
Befehlswort $Zeropageadresse (0-255)
```

4. Indizierte Adressierung

Die indizierte Adressierung ermöglicht es, auf einfache Art und Weise Tabellen zu bearbeiten. Das Prinzip:

Wie bei der absoluten Adressierung wird nicht der Wert, mit dem operiert werden soll, direkt angegeben, sondern aus einer Speicherstelle entnommen. Entgegen der absoluten Adressierung kann diese Speicherstelle aber bei der indizierten Adressierung noch um bis zu 255 Bytes von der angegeben Adresse variieren. Dies wird folgendermaßen ermöglicht:

Zur angegeben Adresse wird einfach noch der Inhalt des X- bzw. des Y-Registers hinzuaddiert und aus dieser Speicherstelle dann der Operant entnommen. Beispiel:

```
LDA $FF00,Y
```

Nehmen wir an, das Y-Register hätte den Inhalt $10. Die Adresse, aus der der Operant entnommen würde, berechnete sich dann folgendermaßen:

```
Adresse + Inhalt von YR = $FF00 + $10 = $FF10
```

In unserem Beispiel würde also der Inhalt der Speicherstelle $FF10 in den Akkumulator geladen. Der Speicheraufbau sieht bei der indizierten Adressierung genauso aus wie bei der absoluten Adressierung:

```
1. Byte        2. Byte        3. Byte
Befehlskode    LOW-Adresse    HIGH-Adresse
```

Nun noch der syntaktische Aufbau bei der indizierten Adressierung:

```
Befehlswort $Adresse,X oder Befehlswort $Adresse,Y
```

Die indizierte Adressierung funktioniert also sowohl mit dem X- als auch mit dem Y-Register.

5. Indizierte Zeropageadressierung

Die indizierte Zeropageadressierung entspricht genau der normalen indizierten Adressierung, nur das es sich um eine 8-Bit-Adresse handelt. Ansonsten gilt alles für die normale indizierte Adressierung Gesagte. Der Speicheraufbau entspricht der normalen Zeropageadressierung:

```
1. Byte        2. Byte
Befehlskode    Adresse
```

Auch die Syntax ist wieder identisch mit der normalen indizierten Adressierung:

```
Befehlswort $Adresse,X oder Befehlswort $Adresse,Y
```

Die Adresse muß hierbei allerdings im Bereich zwischen 0 und 255 liegen, wie dies charakteristisch für die Zeropageadressierung ist.

6. Indirekt indizierte Adressierung

Obwohl diese Adressierungsart relativ schwer zu verstehen ist, ist sie doch extrem leistungsfähig. Im Gegensatz zur unmittelbaren oder absoluten Adressierung, bei denen entweder der Operant oder die Adresse, aus der der Operant entnommen werden sollte, bekannt war, ist bei der indirekt indizierten Adressierung direkt keines von beiden bekannt. Die Adresse wird indirekt ermittelt und dann noch indiziert, d.h., zu der indirekt ermittelten Adresse wird noch der Inhalt des Y-Register hinzugezählt.

Die Ermittlung der Adresse ist nun schon etwas komplizierter. Zum besseren Verständnis zuerst einmal ein Beispiel:

```
LDA ($D8),Y
```

Das Y-Register soll in diesem Fall den Wert $10, die Speicherstelle $D8 den Wert $00 und die Speicherstelle $D9 den Wert $FF enthalten.

Der Inhalt der angegebenen und der darauf folgenden Speicherstelle wird praktisch als Zeiger auf die Adresse benutzt, zu der dann noch der Inhalt des Y-Registers addiert werden würde. Im Klartext heißt das:

Es wird der Inhalt der angegebenen Speicherstelle $D8 geholt. Der Inhalt dieser Speicherstelle stellt dann das LOW-Byte der Adresse dar, auf die eigentlich zugegriffen werden soll. Das HIGH-Byte dieser Adresse steht nun in der folgenden Speicherstelle; In unserem Fall in der Speicherstelle $D9. Zuletzt wird dann noch zu dieser Adresse der Inhalt des Y-Registers addiert. In unserem Beispiel lautete die Adresse, auf die dann zugegriffen werden würde, wie folgt:

```
Inhalt der Speicherstelle $D8 = $00
Inhalt der Speicherstelle $D9 = $FF
Die Adresse lautet daher:     $FF00
```

Zu dieser Adresse wird nun noch der Inhalt des Y-Registers addiert, das in unserem Beispiel den Wert $10 enthält. Die endgültige Adresse lautet dann:

```
$FF00 + Inhalt des Y-Registers = $FF00 + $10 = $FF10
```

In unserem Beispiel würde der Akkumulator dann mit dem Inhalt der Speicherstelle $FF10 geladen.

Der Parameter hinter dem Befehlswort darf nur eine 8-Bit-Adresse sein. Die Speicherstellen, die die Adresse der eigentlichen Speicherstelle enthalten, können also nur innerhalb der Zeropage liegen. Außerdem arbeitet diese Adressierung nur mit dem Y-Register.

7. Indiziert indirekte Adressierung

Diese Adressierungsart funktioniert in etwa genauso wie die indirekt indizierte Adressierung. Die einzigen beiden Unterschiede sind, daß diese Art der Adressierung nur mit dem X-Register funktioniert und zur angegeben Zeropageadresse zuerst der Inhalt des X-Registers addiert wird. Aus der so errechneten Speicherstelle wird dann die Adresse geholt (erst LOW, dann HIGH), aus der der eigentliche Operant entnommen werden soll. Beispiel:

```
LDA ($D8,X)
```

Das X-Register enthalte wieder den Wert $10, die Speicherstelle $E8 den Wert $00 und die Speicherstelle $E9 den Wert $FF. Die eigentliche Adresse errechnet sich nun folgendermaßen:

Zuerst wird zur angegeben Adresse der Inhalt des X-Register addiert:

```
$D8 + $10 = $E8
```

Aus der sich ergebenden und der darauf folgenden Speicherstelle wird nun die Adresse der Speicherstelle ermittelt, in der sich der Operand befindet. Bei unserem Beispiel sähe das dann so aus:

```
Inhalt der Speicherstelle $E8 = $00
Inhalt der Speicherstelle $E9 = $FF
Sich ergebende Adresse:       $FF00
```

Bei unserem oben stehenden Befehl würde der Akkumulator also den Inhalt der Speicherstelle $FF00 bekommen.

Alle oben angegebenen Adressierungsarten funktionieren natürlich nicht nur mit dem LDA-Befehls sondern auch mit einem Großteil der anderen Befehle. Welche Addressierungsarten bei den einzelnen Befehlen möglich sind wird jeweils angegeben.

2.2.1 Die Ladebefehle LDA, LDX, LDY

Den LDA-Befehl kennen Sie ja schon aus dem vorhergehenden Kapitel. Analog hierzu existieren auch noch Ladebefehle für die beiden Indexregister. Es sind dies die Befehle LDX und LDY. Mit dem LDX-Befehl kann man das X-Register mit einem 8-Bit-Wert laden, während man das Y-Register mittels des LDY-Befehls laden kann. Im Gegensatz zum LDA-Befehl, bei dem alle Adressierungsarten erlaubt sind, sind bei den beiden Befehle LDX und LDY nicht ganz so viele Adressierungsarten möglich. Bei den Ladebefehlen werden folgende Flags beeinflußt:

1. Das Negativflag, sofern der Wert, mit dem das Register geladen wird, größer als 128 ($7F) ist. In diesem Fall kann die Zahl auch als negative Zahl interpretiert werden.
2. Das Zeroflag, sofern der Wert, der in das Register geladen wird, gleich 0 ist.

Welche Adressierungsarten für die einzelnen Befehle möglich sind, entnehmen Sie bitte der untenstehenden Tabelle, die wie folgt angelegt ist:

In der ersten Spalte steht der Name der Adressierungsart, dann folgt ein allgemeines Beispiel zum besseren Verständnis. Dann folgen mehrere Spalten für die einzelnen Befehle. Steht an einer

Position keine Zahl, so heißt das, daß der Befehl in dieser Adressierungsart nicht existiert. Ansonstem gibt die Zahl den Befehlskode an.

Hier nun die Tabelle:

Adressierungsart	Beispiel	LDA	LDX	LDY
unmittelbar	Befehl #$Byte	$A9	$A2	$A0
absolut	Befehl $Adresse	$AD	$AE	$AC
absolut x-indiziert	Befehl $Adresse,X	$BD	---	$BC
absolut y-indiziert	Befehl $Adresse,Y	$B9	$BE	---
zeropage	Befehl $Z.adresse	$A5	$A6	$A4
zeropage x-indiziert	Befehl $Z.adresse,X	$B5	---	$B4
zeropage y-indiziert	Befehl $Z.adresse,Y	---	$B6	---
indirekt indiziert	Befehl ($Z.adresse),Y	$B1	---	---
indiziert indirekt	Befehl ($Z.adresse,X)	$A1	---	---

2.2.2 Die Speicherbefehle STA, STX, STY

Das Gegenstück zu den Ladebefehlen LDA, LDX und LDY stellen die Befehle STA, STX und STY dar. Mit diesen Befehlen ist es möglich den Inhalt eines Registers in den Speicher zu schreiben. Bei den Adressierungsarten stehen Ihnen fast alle Adressierungsarten wie bei den Ladebefehlen zur Verfügung. Eine Ausnahme bildet natürlich die unmittelbare Adressierung, da es ja sinnlos wäre, den Akku in den Akku zu schreiben. Bei den Speicherbefehlen wird kein Flag beeinflusst, da sich die Befehle ja auf den Speicher und nicht auf die Register beziehen. Die möglichen Adressierungsarten sind folgende:

Adressierungsart	Beispiel	STA	STX	STY
unmittelbar	Befehl #$Byte	---	---	---
absolut	Befehl $Adresse	$8D	$8E	$8C
absolut x-indiziert	Befehl $Adresse,X	$9D	---	---
absolut y-indiziert	Befehl $Adresse,Y	$99	---	---
zeropage	Befehl $Z.adresse	$85	$86	$84
zeropage x-indiziert	Befehl $Z.adresse,X	$95	---	$94

zeropage y-indiziert	Befehl $Z.adresse,Y	---	$96	---
indirekt indiziert	Befehl ($Z.adresse),Y	$91	---	---
indiziert indirekt	Befehl ($Z.adresse,X)	$81	---	---

2.2.3 Die Transferbefehle TAX, TXA, TAY, TYA

Um die Registerinhalte des Akkus und des X- und Y-Registers intern zu übertragen, existieren die sogenannten Transferbefehle. Mit ihnen ist es möglich, z.B. den Inhalt des Akkus in das Y-Register und umgekehrt zu übertragen.

TAX: Mit dem TAX Befehl besteht die Möglichkeit, den Inhalt des Akkus in das X-Register zu übertragen. Der Inhalt des X-Registers wird hierbei überschrieben und der Inhalt des Akkus bleibt erhalten. Steht z.B. im Akku der Wert $01 und im X-Register der Wert $02, so enthält sowohl der Akkumulator als auch das X-Register nach der Ausführung des TAX-Befehls den Wert $01.

TAY: Der TAY hat die gleiche Funktion wie der TAX-Befehl, nur daß er für den Akkumulator und das Y-Register gilt.

TXA: Dieser Befehl stellt die Umkehrfunktion des TAX-Befehls dar. Es wird also der Inhalt des X-Registers in den Akkumulator kopiert. Der Wert im Akku geht hierbei verloren.

TYA: Dieser Befehl stellt das Gegenteil zum TAY-Befehl dar und funktioniert entsprechend dem TXA-Befehl.

Bei den Transferbefehlen werden eventuell folgende zwei Flags des Statusregisters beeinflusst:

1. Das Negativflag, falls der Wert, der kopiert wurde, größer als 127 ($7F) war.

2. Das Zeroflag, falls der Wert, der kopiert wurde, gleich 0 war.

Um die Inhalte zweier Register untereinander zu vertauschen, ohne das der Inhalt eines Registers verloren geht, verwendet man am besten die folgende Methode:

```
Akku - XR:    Akku - YR:    XR - YR:
STA $03       STA $03       STX $03
TXA           TYA           STY $04
LDX $03       LDY $03       LDX $04
                            LDY $03
```

Da die Transferbefehle nur registerintern arbeiten, ist es logisch, daß es für sie keine verschiedenen Adressierungsarten gibt. Daher an dieser Stelle nur die Befehlskodes der Befehle:

```
TAX: $AA
TAY: $A8
TXA: $8A
TYA: $98
```

Da die Befehle TSX und TXS keine direkten Befehle für die drei wichtigsten Register sind, finden Sie diese beiden Befehle im Kapitel 2.2.13.

2.2.4 Arithmetische Befehle ADC, SBC

Wie Sie vielleicht schon an anderer Stelle in diesem Buch gelesen haben, ist der 7501-Prozessor nur in der Lage Werte zu addieren oder zu subtrahieren. Hierzu dienen die beiden Befehle ADC und SBC. Um zwei Werte zu addieren werden logischerweise zwei Werte benötigt. Der eine Wert steht prinzipiell im Akkumulator während der zweite Operant durch verschiedene Adressierungsarten ermittelt werden kann. Das Ergebnis einer Addition bzw. Subtraktion steht immer im Akkumulator.

Bei der Addition zweier 8-Bit-Werte kann es vorkommen, daß sich das Ergebnis der Addition nicht mit einem Byte darstellen lassen kann. In diesem Fall tritt ein sogenannter Übertrag auf. Einen Übertrag erkennt man daran, daß das Carryflag gesetzt ist. Das Carryflag dient also als 8. Bit. Somit lassen sich dann auch ohne weiteres Werte im Bereich eines Bytes addieren, ohne daß das Ergebnis durch einen nicht registrierten Überlauf verfälscht werden würde. Schauen wir uns einmal eine Addition mit einem solchen Überlauf an:

Gehen wir davon aus, daß der Akku den Wert $54 enthält und wir den Wert $F3 addieren wollen. Binär ausgedrückt sieht die Addition dann so aus:

```
$54 = %01010100
$F3 = %11110011
```

Daraus folgt dann:

```
  %01010100
+ %11110011
= 101000111
```

Man erkennt also, daß die binäre Addition ähnlich der dezimalen Addition durchgeführt wird.

0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 0 plus Überlauf

Ein Übertrag wird wie bei der dezimalen Addition auf die nächste Stelle verrechnet. Zum besseren Verständnis hier noch einmal die Erklärung unseres Beispiels von oben. Zum Addieren zweier Binärwerte gehen wir von rechts nach links, also vom 0. zum 7. Bit vor. Unsere Addition lautet demnach:

```
Bit 0: 0+1+0 (Rest)=1 Rest 0
Bit 1: 0+1+0 (Rest)=1 Rest 0
Bit 2: 1+0+0 (Rest)=1 Rest 0
Bit 3: 0+0+0 (Rest)=0 Rest 0
Bit 4: 1+1+0 (Rest)=0 Rest 1
Bit 5: 0+1+1 (Rest)=0 Rest 1
Bit 6: 1+1+1 (Rest)=1 Rest 1
Bit 7: 0+1+1 (Rest)=0 Rest 1
Über-:   0+1 (Rest)=1 Rest 0
lauf
```

Wie oben schon gesagt, wird der Übertrag dieser Addition in das Carry-Flag übernommen:

```
LDA #$54
ADC #$F3
```

Ist kein Übertrag entstanden so ist das Carryflag 0, ist aber ein Übertrag entstanden, so ist das Carryflag 1. Will man aber Integerzahlen eines größeren Formats, z.B. Integerzahlen mit einer Länge von zwei Bytes erhalten, so addiert man einfach diese Werte hintereinander. Zuerst die beiden unteren und dann die beiden oberen. Tritt hierbei ein Übertrag auf, so wird es etwas problematisch. Aus diesem Grund wird bei einer Addition das Carryflag mit betrachtet. Ist es gleich 1, so ist der Wert mit dem operiert wird gleich dem Wert im Akku+1. Daher muß vor jeder Operation das Carryflag gelöscht werden. Eine Zweibyteaddition sähe dann z.B. so aus, sofern der erste Wert in den Speicherstellen $D8/$D9 und der zweite Wert in den Speicherstellen $DA/$DB enthalten ist:

```
CLC      ;Carryflag löschen (Kap. 2.2.10)
LDA $D8  ;LOW-Byte 1. Wert
ADC $DA  ;+ LOW-Byte 2. Wert
STA $DA  ;LOW-Ergebnis in $DA
LDA $D9  ;HIGH-Byte 1. Wert, Carry enthält Übertrag
ADC $DB  ;+ HIGH-Byte 2. Wert
STA $DB  ;HIGH-Ergebnis in $DB
```

Das Ergebnis dieser Zweibyteaddition steht jetzt in den Speicherstellen $DA (LOW-Byte) und $DB (HIGH-Byte).

Subtraktion: Diese geschieht in etwa wieder analog zur Addition, denn auch bei der Subtraktion wird ein Wert vom Akkuinhalt abgezogen und wieder im Akku abgelegt. Auch bei der Subtraktion kann es vorkommen, daß das Ergebnis nicht mehr im Bereich zwischen 0 und 255 liegt, sondern kleiner als 0 ist. Ein solcher Unterlauf wird dadurch gekennzeichnet, daß das Carryflag nach der Subtraktion gelöscht ist. Es muß also vor einer Subtraktion mittels des Befehls SEC (Kap. 2.1.10) gesetzt werden. Wie kommt es aber nun zur Beeinflussung des Carryflags? Schauen wir uns dazu wieder ein Beispiel an:

Der Wert im Akkumulator sei dazu $54 und der Wert der abgezogen werden soll sei $F4. Binär ausgedrückt sehen diese beiden Werte so aus:

```
$54 = %01010100
$F3 = %11110011
```

Daraus folgt dann:

```
  %01010100
- %11110011
= 001000001
```

Da das 8. Bit gelöscht war, gilt also der folgende Zusammenhang:

```
0 - 0 = 0
0 - 1 = 1 plus Unterlauf
1 - 0 = 1
1 - 1 = 0
```

Zum besseren Verständnis hier noch einmal alle einzelnen Schritte der Subtraktion:

```
0. Bit: 0-1-0 (Rest) = 1 Rest 1
1. Bit: 0-1-1 (Rest) = 0 Rest 1
2. Bit: 1-0-1 (Rest) = 0 Rest 0
3. Bit: 0-0-0 (Rest) = 0 Rest 0
4. Bit: 1-1-0 (Rest) = 0 Rest 0
5. Bit: 0-1-0 (Rest) = 1 Rest 1
6. Bit: 1-1-1 (Rest) = 1 Rest 1
7. Bit: 0-1-1 (Rest) = 0 Rest 1
8. Bit: 1-0-1 (Rest) = 0 Rest 0
```

Bem.: Das achte Bit ist das gesetzte Carryflag.

Wie Sie sehen, ist das Carryflag wieder ein wichtiger Bestandteil der Subtraktion. Wird es bei einer Subtraktion gelöscht, so bedeutet dies einen Unterlauf.

Der dezimale Wert unserer erhaltenen Binärzahl ist nun 97, obwohl bei der normalen Subtraktion von 84-243 der Wert -159 das Ergebnis ist. In welchem Bezug stehen diese beiden Ergebnisse zueinander? Subtrahiert man 97 von 256, so erhält man ebenfalls 159. Das gelöschte Carryflag nach der Subtraktion sagt uns also, daß wir das Ergebnis als negative Zahl interpretieren müssen. Eine solche Zahl bezeichnet man dann als Zweierkomplement. Den richtigen Wert erhalten wir, wenn wir alle Bits des Ergebnisses umdrehen und dann 1 addieren. In der Praxis sähe das so aus:

```
%01100001

Alle Bits umdrehen: %10011110
Eins addieren:      %10011111
Dezimaler Wert:      159
```

Wir müssen allerdings daran denken, daß wir diesen Wert als negativen Wert betrachten müssen.

Nachdem wir nun die Funktionsweisen vom ADC- und SBC-Befehl besprochen haben, hier die Tabelle mit allen möglichen Adressierungsarten:

Adressierungsart	Beispiel	ADC	SBC
unmittelbar	Befehl #$Byte	$69	$E9
absolut	Befehl $Adresse	$6D	$ED
absolut x-indiziert	Befehl $Adresse,X	$7D	$FD
absolut y-indiziert	Befehl $Adresse,Y	$79	$F9
zeropage	Befehl $Z.adresse	$65	$E5
zeropage x-indiziert	Befehl $Z.adresse,X	$75	$F5
zeropage y-indiziert	Befehl $Z.adresse,Y	---	---
indirekt indiziert	Befehl ($Z.adresse),Y	$71	$F1
indiziert indirekt	Befehl ($Z.adresse,X)	$61	$E1

2.2.5 Die Logischen Befehle

Außer den Verknüpfungsmöglichkeiten durch Addieren oder Subtrahieren, besitzt der 7501 noch die sogenannten logischen Befehle. Darunter versteht man Befehle zur UND, ODER und EXKLUSIV ODER Verknüpfung. Bei den logischen Befehlen werden, wie bei den arithmetischen Befehlen, wieder zwei Werte benötigt. Der erste dieser beiden Werte steht immer im Akkumulator, während der zweite Wert durch die verschiedene Adressierungsarten ermittelt werden kann. Bei der Verknüpfung der beiden Werte miteinander wird immer bitweise vorgegangen, also immer zwei Bits nach den entsprechenden Verknüpfungsvorschriften verknüpft.

Das Ergebnis einer logischen Verknüpfung steht immer im Akkumulator. Nun zu den drei Arten der logischen Verknüpfung:

1. Die UND-Verknüpfung

Bei der UND-Verknüpfung wird immer dann das entsprechende Bit im Ergebnis gesetzt, wenn im ersten UND im zweiten Wert das jeweilige Bit gesetzt ist. Hieraus folgen die Verknüpfungsvorschriften für die UND-Verknüpfung:

```
0 UND 0 = 0
0 UND 1 = 0
1 UND 0 = 0
1 UND 1 = 0
```

Wir wollen einmal als Beispiel die beiden Werte $59 und $F4 miteinander verknüpfen:

```
$59 = %01011001
$F4 = %11110100

     %01011001
UND  %11110100
  =  %01010000
```

Das Ergebnis der UND-Verknüpfung ist also $50. Analog funktioniert die UND-Verknüpfung nun auch beim Prozessor. Der entsprechende Befehl hierzu lautet AND. In Assembler sähe unsere UND-Verknüpfung daher so aus:

```
LDA #$59 ;ersten Wert in Akkumulator laden
AND #$F4 ;mit zweitem Wert verknüpfen
```

Das Ergebnis $50 steht nach Ausführung dieser beiden Befehle im Akku. Der AND-Befehl wird z.B. benötigt, wenn man bestimmte Bits in einem Byte testen will. Dazu ein Beispiel:

Wir wollen testen, ob das 6. Bit in der Speicherstelle $D8 gesetzt ist. Dazu gehen wir folgendermaßen vor:

```
LDA $D8 AND #%01000000 ;6. Bit testen
```

Ist nach der Ausführung dieser beiden Befehle der Akkuinhalt gleich 0 (Zeroflag gesetzt), so war das getestete Bit nicht gesetzt. Ist der Akkuinhalt jedoch ungleich 0 (Zeroflag gelöscht), dann war das getestete Bit gesetzt.

Außer dem Zeroflag wird durch den AND-Befehl gegebenenfalls noch das Negativflag gesetzt, falls das Ergebnis größer als

$7F war, das 7. Bit im Ergebnis also gesetzt war. Welche Adressierungsarten für den AND-Befehl möglich sind, können Sie der Tabelle am Ende dieses Kapitels entnehmen.

2. Die ODER-Verknüpfung

Auch bei der ODER-Verknüpfung werden die beiden Werte bitweise verknüpft. Ist ein Bit im ersten Wert ODER ein Bit im zweiten Wert gesetzt, so wird auch das entsprechende Bit im Ergebnis gesetzt. Es gelten die folgenden Verknüpfungsvorschriften:

```
0 ODER 0 = 0
0 ODER 1 = 1
1 ODER 0 = 1
1 ODER 1 = 1
```

Dazu ein Beipiel, wir verknüpfen unsere beiden Binärzahlen diesmal mittels einer ODER-Verknüpfung:

```
$59 = %01011001
$F4 = %11110100

     %01011001
ODER %11110100
  =  %11111101
```

Wie Sie sehen, ist das Ergebnis auch 1, wenn beide Bits gleich 1 sind. Dieser Art der ODER-Verknüpfung bezeichnet man daher als inklusive ODER-Verknüpfung.

In der Maschinensprache des 7501 stellt der ORA-Befehl die ODER-Funktion dar. In Maschinensprache sähe unsere Verknüpfung dann so aus:

```
LDA #$59 ;ersten Wert in Akku laden
ORA #$F4 ;mit zweitem Wert ODER-verknüpfen
```

Die ODER-Verknüpfung wird meist dazu benutzt, Bits in einem Byte zu setzen, ohne die anderen Bits zu beeinflussen, wie dies z.B. beim normalen STA-Befehl geschehen würde. Um z.B. Bit 6 in der Speicherstelle $D8 zu setzen, würde man so vorgehen:

```
LDA $D8         ;Inhalt aus $D8 in Akku
ORA #%01000000 ;mit Bit 6 ODER-verknüpfen
STA $D8         ;und wieder in $D8 speichern
```

3. Die EXKLUSIV-ODER-Verknüpfung

Die EXKLUSIV-ODER-Verknüpfung (kurz: EXOR) funktioniert fast genauso wie die normale ODER-Verknüpfung. Der einzige Unterschied zwischen beiden ist, daß das Ergebnis gleich 0 ist, falls beide Bits gleich 1 sind. Diese Art der ODER-Verknüpfung bezeichnet man daher auch als exklusive ODER-Verknüpfung. Die Verknüpfungsvorschriften lauten daher fast analog zu denen der ODER-Verknüpfung:

```
0 EXOR 0 = 0
0 EXOR 1 = 1
1 EXOR 0 = 1
1 EXOR 1 = 0
```

Dazu wieder das Beispiel mit $59 und $F4:

```
$59 = %01011001
$F4 = %11110100

     %01011001
EXOR %11110100
   = %10101101
```

Anwendung findet die EXOR-Verknüpfung z.B. dann, wenn Werte oder Bits invertiert werden sollen. Wir werden daher noch an anderer Stelle wieder auf die EXOR-Verknüpfung eingehen.

In der Maschinensprache heißt der Befehl zur EXKLUSIV-ODER-Verknüpfung EOR. Daher würde unsere EXOR-Verknüpfung in Maschinensprache so aussehen:

```
LDA #$59 ;ersten Wert in Akku laden
EOR #$F4 ;mit zweitem Wert EXOR-verknüpfen
```

Außer dem schon oben genannten AND-Befehl gibt es noch den BIT-Befehl. Auch seine Funktion beruht auf einer UND-Verknüpfung, werden hierbei keine Registerinhalte verändert. Trifft der Prozessor bei der Abarbeitung eines Maschinenprogramms auf einen BIT-Befehl, so führt er eine UND-Verknüpfung zwischen dem Akkumulatorinhalt und der adressierten Speicherstelle durch. Ist das Ergebnis der UND-Verknüpfung gleich 0, so wird das Zeroflag gesetzt. Ist das Ergebnis ungleich 0, so wird das Zeroflag gelöscht. Weiterhin wird noch das 6. Bit der adressierten Speicherstelle in das Overflowflag und das 7. Bit der adressierten Speicherstelle in das Negativflag übernommen. Dazu ein Beispiel:

```
BIT $1000
```

Der Akku enthalte $E3 und die Speicherstelle $1000 den Wert $BC. Die UND-Verknüpfung sieht dann so aus:

```
$E3 = %11100011
$BC = %10111100

      %11100011
UND   %10111100
  =   %10100000
```

Das Zeroflag wird gelöscht, das Overflowflag und das Negativflag gesetzt.

Für den BIT-Befehl existieren nur die beiden Adressierungsarten Zeropage und Absolut:

```
Zeropage BIT $Z.adresse $24
Absolut  BIT $adresse   $2C
```

Hier nun die Tabelle mit allen möglichen Adressierungsarten für die Befehle AND, ORA, EOR:

Adressierungsart	Beispiel	AND	ORA	EOR
unmittelbar	Befehl #Byte	$29	$09	$49
absolut	Befehl $Adresse	$2D	$0D	$4D
absolut x-indiziert	Befehl $Adresse,X	$3D	$1D	$5D
absolut y-indiziert	Befehl $Adresse,Y	$39	$19	$59
zeropage	Befehl $Z.adresse	$25	$05	$45
zeropage x-indiziert	Befehl $Z.adresse,X	$35	$15	$55
zeropage y-indiziert	Befehl $Z.adresse,Y	---	---	---
indirekt indiziert	Befehl ($Z.adresse),Y	$31	$11	$51
indiziert indirekt	Befehl ($Z.adresse,X)	$21	$01	$41

2.2.6 Die Vergleichsbefehle CMP, CPX, CPY

Wir kommen nun zu einer sehr wichtigen Befehlsgruppe, den Vergleichsbefehlen. Mittels der Vergleichsbefehle können zwei Werte miteinander verglichen werden. Der erste Wert muß immer im Akku, im X- oder im Y-Register stehen, während der zweite Wert durch verschiedene Adressierungsarten ermittelt werden kann.

Das Prinzip eines Vergleiches zweier Werte ist nun folgendes. Der zweite Wert wird vom ersten Wert abgezogen, also das adressierte Byte von einem Registerinhalt (Akku, X, Y). Diese Subtraktion geschieht aber prozessorintern; es werden also keine Registerinhalte beeinflußt außer den Flags.

Durch die Vergleichsbefehle werden nämlich das Carry-, das Zero- und das Negativflag auf folgende Art und Weise beeinflußt:

C=1 Das Carryflag wird gesetzt, wenn der erste Wert größer gleich dem zweiten Wert ist, die Subtraktion also entweder einen positiven Wert oder 0 ergibt.

Carry = 1 bedeutet: W1 größer, gleich W2

C=0 Das Carryflag wird gelöscht, wenn der erste Wert kleiner als der zweite Wert ist, die Subtraktion also ein negatives Ergebnis liefert.

Carry = 0 bedeutet: W1 kleiner W2

Z=1 Das Zeroflag wird auf 1 gesetzt, wenn beide Werte identisch sind, die Subtraktion also 0 ergibt.

Zero = 1 bedeutet: W1 gleich W2

Z=0 Ist das Zeroflag nach einem Vergleich gelöscht, so bedeutet dies, daß die beiden Werte verschieden sind.

Zero = 0 bedeutet: W1 ungleich W2

N=1 Ist das Negativflag nach einem Vergleich gesetzt, so heißt dies, daß das Ergebnis der Subtraktion größer als $7F ist. Dies ist der Fall, wenn eine der folgenden Bedingungen erfüllt ist:

1. W1 + $80 kleiner W2 (Für W1 kleiner W2)
2. W1 - $80 größer W2 (Für W1 größer W2)

N=0 Ist das Negativflag nach einem Vergleich gelöscht, so heißt das, daß die Differenz zwischen beiden kleiner als $80 gewesen ist.

1. W1 + $80 größer W2 (Für W1 kleiner W2)
2. W1 - $80 kleiner W2 (Für W1 größer W2)

(Unter W1 ist in allen Fällen der Wert zu verstehen der im Akku, im X- oder im Y-Register steht. Unter dem zweiten Wert wird hier der Wert verstanden, der mittels einer der Adressierungsarten ermittelt worden ist.)

Nun zu den einzelnen Befehlen:

Es existieren drei Vergleichsbefehle CMP, CPX und CPY. Der CMP-Befehl vergleicht einem Wert mit dem Inhalt des Akkumulators, der CPX-Befehl einen Wert mit dem Inhalt des X-Registers und der CPY-Befehl einen Wert mit dem Inhalt des Y-Registers. Für den CMP-Befehl existieren die meisten Adressierungsarten (siehe Tabelle unten).

Nun einige Beispiele zu den Vergleichsbefehlen. Wir arbeiten hier mit dem CMP-Befehl, das Prizip gilt natürlich genauso für die anderen beiden Vergleichsbefehle:

Gehen wir im ersten Fall davon aus, daß der Akku den Inhalt $4F enthält und er mit dem Wert $3B verglichen werden soll. Die interne Subtraktion sieht dann so aus:

```
  $4F     LDA #$4F
- $3B     CMP #$3B
= $14     C=1; Z=0; N=0
```

Das Ergebnis der internen Subtraktion wäre in unserem Fall also $14. Da kein Unterlauf aufgetreten ist, wird das Carryflag gesetzt. Beide Werte waren verschieden, daher wird das Zeroflag gelöscht. Schließlich wird noch das Negativflag auf 0 gesetzt, da das Ergebnis der Subtraktion kleiner als $7F war.

Im zweiten Fall sollen nun beide Werte identisch sein. Nehmen wir dazu einmal $3B sowohl für den ersten als auch für den zweiten Wert an:

```
  $3B     LDA #$3B
- $3B     CMP #$3B
= $00     C=1; Z=1; N=0
```

Wie Sie sehen, ist das Ergebnis der internen Subtraktion 0 gewesen. Da bei dem Ergebnis 0 noch kein Unterlauf aufgetreten ist, wird das Carryflag wieder gesetzt. Auch das Zeroflag wird diesmal gesetzt, da beide Werte identisch waren, das Ergebnis der Subtraktion also 0 war. Auch in diesem Fall bleibt das Negativflag gelöscht, da das Ergebnis kleiner als $7F war.

Im dritten Fall wollen wir annehmen, daß der erste Wert kleiner als der zweite Wert ist. Für den zweiten Wert behalten wir dazu den Wert $eB bei, während wir für den ersten Wert $0B annehmen.

```
  $0B     LDA #$0B
- $3B     CMP #$3B
= $D0     C=0; Z=1; N=1
```

Da diesmal der zweite Wert größer als der erste war, bei der Subtraktion also ein Unterlauf aufgetreten ist, wird das Carryflag gelöscht. Beide Werte waren auch in diesem Beipiel nicht identisch, daher ist das Zeroflag gelöscht. Das Negativflag wird diesmal gesetzt, da das Ergebnis ($D0) größer als $7F ist.

Man kann die interne Subtraktion also durchaus mit einer normalen SBC-Subtraktion vergleichen. Die Bedingungen für die Flags sind identisch, nur das der Akku nach Abschluß der Subtraktion nicht das Ergebnis enthält, sondern nur die Flags verändert werden.

Um nun zu testen, wie sich zwei Werte zueinander verhalten, muß man folgendermaßen vorgehen:

W1 ist größer W2 wenn C=1 und Z=0

W1 ist gleich W2 wenn Z=1

W1 ist kleiner W2 wenn C=0

W1 ist größer, gleich W2 wenn C=1

Zur Überprüfung auf W1 größer W2 müssen also jeweils zwei Flags getestet werden.

Hier nun die Tabelle mit allen möglichen Adressierungsarten:

Adressierungsart	Beispiel	CMP	CPX	CPY
unmittelbar	Befehl #$Byte	$C9	$E0	$C0
absolut	Befehl $Adresse	$CD	$EC	$CC
absolut x-indiziert	Befehl $Adresse,X	$DD	---	---
absolut y-indiziert	Befehl $Adresse,Y	$D9	---	---
zeropage	Befehl $Z.adresse	$C5	$E4	C
zeropage x-indiziert	Befehl $Z.adresse,X	$D5	---	---
zeropage y-indiziert	Befehl $Z.adresse,Y	---	---	---
indirekt indiziert	Befehl ($Z.adresse),Y	$D1	---	---
indiziert indirekt	Befehl ($Z.adresse,X)	$C1	---	---

2.2.7 Die bedingten Sprünge -BEQ, BNE, BCS, BCC, BMI, BPL, BVS, BVC

Um auf bestimmte Ereignisse und Flags zu reagieren, existieren die sogenannten Sprungbefehle. Je nachdem, ob ein Flag gesetzt ist oder nicht, kann gesprungen werden. Es existieren für das Zero-, das Carry, das Negativ- und das Overflowflag jeweils zwei Sprungbefehle. Ein Sprungbefehl, bei dem gesprungen wird, wenn das Flag gelöscht ist und einer, bei dem gesprungen wird, wenn das Flag gesetzt ist. Außer dem Befehlskode benötigt der Prozessor bei einem bedingten Sprungbefehl natürlich noch eine Zieladresse. Unter der Zieladresse versteht man den neuen Wert des Programmzählers, also die Stelle von der sich der Prozessor den nächsten Befehlskode holt. Normalerweise gibt man Adressen immer im 16-Bitformat an, worauf wir auch schon eingegangen sind. Diese Art der Angabe der Adresse wird uns auch noch bei dem unbedingten Sprungbefehl begegnen. Bei den bedingten Befehlen wurde allerdings eine andere Art der Adressierung gewählt, die relative Adressierung.

Da man bei bedingten Spüngen selten über einen großen Raum springen muß, haben sich die Entwickler der 65XX-Prozessoren

(7501) eine neue Art der Adressierung ausgedacht. Eben die relative Adressierung. Was muß man sich nun unter der sogenannten relativen Adressierung vorstellen?

Der erste Unterschied zur normalen, absoluten Adressierung ist, daß die Adresse nicht durch zwei Bytes, sondern nur durch ein Byte definiert wird. Die bedingten Sprungbefehle bestehen also nicht aus drei, sondern aus zwei Bytes. Dies hat natürlich den Vorteil, daß sie erheblich schneller bearbeitet werden, als die drei Bytes langen Befehle.

Nun zum Prinzip der relativen Adressierung. Wie der Name schon sagt, wird das Sprungziel relativ angegeben. Um eine Adresse relativ anzugeben, benötigt man einen Bezugspunkt, in diesem Fall unseren Augangspunkt, den bedingten Sprungbefehl. Eine relative Angabe des Sprungziels könnte etwa so aussehen:

Springe von der momentanen Adresse zehn Speicherstellen nach vorn

oder:

Springe von der momentanen Adresse zehn Speicherstellen nach hinten

Besonders wichtig ist, daß von der momentanen Adresse gesprungen wird. Verschiebt man ein Maschinenprogramm, in dem relative Sprungbefehle vorkommen, in einen anderen Speicherbereich, so braucht man die Sprungziele nicht ändern, da diese sich ja nur auf die Entfernung zwischen Sprungquelle und Sprungziel beziehen. Die bedingten Sprünge beziehen sich also nicht absolut auf eine bestimmte Speicherstelle.

Wie wir schon wissen, kann man mit einem Byte 256 verschiedene Zahlen (0-255) darstellen. Dieser Wertebereich reichte also aus, um maximal 256 Bytes nach vorn oder nach hinten zu springen. Besser wäre es natürlich, wenn man sowohl nach hinten als auch nach vorn springen könnte. Man müßte dazu mit einem Byte sowohl negative als auch positive Zahlen darstellen können. Vielleicht erinnern Sie sich in diesem Zusammenhang

noch an den SBC-Befehl oder an die Ausführungen über das Negativflag. Dort haben wir nämlich schon die Möglichkeit kennengelernt, mit einem Byte sowohl positive als auch negative Zahlen darzustellen. Man betrachtete dabei nämlich das 7. Bit als Vorzeichenbit. War das 7. Bit in einer Zahl gesetzt, so war diese Zahl als negative Zahl im Zweierkomplement (siehe unten) zu betrachten. War das Bit 7 gelöscht, also gleich 0, so handelte es sich um eine ganz normale positive Zahl. Genauso verhält es sich nun auch bei den bedingten Sprungbefehlen. Auch hier wird das 7. Bit als Vorzeichenbit angesehen. Ist es gesetzt, so wird nach hinten gesprungen, ist es gelöscht, wird nach oben gesprungen (Mit 'oben' und 'unten' sind in diesem Zusammenhang die Adressen gemeint. Die Adresse $2000 liegt z.B. weiter oben als die Adresse $1000).

Bei der Berechnung einer negativen Zahl ging man folgendermaßen vor:

Man drehte alle Bits um (1 wird zu 0 und 0 zu 1) und addierte dann 1 auf. Verständlicher ausgedrückt könnte man die Berechnung einer negativen Zahl folgendermaßen vornehmen:

Ist das 7. Bit einer Zahl gesetzt, so wird diese als negative Zahl angesehen. Der absolute Wert (Wert ohne Vorzeichen, der absolute Wert von -127 und 127 ist in beiden Fällen 127) der Zahl errechnet sich dann folgendermaßen:

128 minus Wert der Bits 0 bis 6. Vor das Ergebnis dieser Berechnung brauchen Sie dann nur noch ein Minus zu setzen und schon haben Sie den Wert unser negativen Zahl.

Sollte Ihnen die Berechnung der Zieladresse reichlich kompliziert und undurchsichtig vorgekommen sein, so kann ich Sie beruhigen: Die Berechnung der Zieladresse bei einem bedingten Sprung nimmt uns der Maschinensprachemonitor oder ein guter Assembler ab. Sie brauchen die Zieladresse dann nur absolut angeben (z.B. BEQ $1000). Der Maschinensprachemonitor berechnet dann automatisch den Parameter des Sprungbefehls und weist Sie

auch auf eine Bereichsüberschreitung hin. Denn mit den bedingten Sprungbefehlen kann man maximal 128 Bytes zurück und 127 Bytes nach vorne springen.

Nun zu den einzelnen Sprungbefehlen:

BEQ (Branch on EQual):

Es wird gesprungen wenn das Zeroflag gesetzt ist, die letzte Operation also 0 ergeben hat. Beispiel:

```
LDA #$20      ;Akku mit $20 laden
CMP #$20      ;gleich, daher Zeroflag = 1
BEQ $Adresse ;Zeroflag = 1, also Sprung nach $Adresse
```

BNE (Branch on Not Equal):

Hier wird gesprungen, wenn das Zeroflag gelöscht (0) ist. Das Zeroflag wird z.B. gelöscht, wenn bei einem Vergleich beide Werte nicht identisch waren. Beispiel:

```
LDA #$20      ;Akku mit $20 laden
CMP #$20      ;Ungleich daher Zeroflag = 0
BNE $Adresse ;Zeroflag = 0, also Sprung nach $Adresse
```

BCS (Branch on Carry Set):

Ist das Carryflag gesetzt, so wird zur angegeben Adresse gesprungen. Beispiel:

```
LDA #$20      ;Akku mit $20 laden
CMP #$10      ;Kein Unterlauf, daher Carry = 1
BCS $Adresse ;Carryflag = 1, also Sprung nach $adresse
```

BCC (Branch on Carry Clear):

Ist das Carryflag gelöscht, dann wird der Sprung ausgeführt. Beispiel:

```
LDA #$20      ;Akku mit $20 laden
CMP #$30      ;Unterlauf, daher Carry = 0
BCC $Adresse ;Carryflag = 0, also Sprung nach $Adresse
```

BMI (Branch on MInus):

Ist das siebte Bit im Statusregister gesetzt, das Negativflag also gleich 1, so wird beim BMI-Befehls gesprungen. Beispiel:

```
LDA #$80      ;Akku mit $80 laden, $80 größer als $7F
BMI $Adresse ;Negativflag = 1, also Sprung zu $Adresse
```

BPL (Branch on PLus):

Ist das Negativflag gelöscht, wird bei diesem bedingten Sprungbefehl gesprungen. Beispiel:

```
LDA #$20      ;Akku mit $20 laden, $20 kleiner als $80
BPL $Adresse ;Negativflag = 0, also Sprung zu $Adresse
```

BVS/BVC (Branch on Overflow Set bzw. Clear):

Zum Abfragen ob das Overflowflag gesetzt ist oder nicht, dienen diese beiden Befehle. Ist das Overflowflag gesetzt, so wird bei BVS gesprungen. Andernfalls wird der Sprung bei BVC ausgeführt. Diese beiden Befehle spielen nur beim Rechnen mit vorzeichenbehafteten Zahlen eine Rolle. Ihre Verwendung ist dementsprechend selten.

Um die anderen Flags zu testen, gibt es keine Befehle. Man muß dies dann mittels der Stackbefehle lösen (Kap. 2.2.13). Um z.B. das Breakflag zu testen, müßte man etwa so vorgehen:

```
PHP           ;Statusregister auf Stack legen
PLA           ;Byte (Statusregister) vom Stack holen
AND #$10      ;Bit 4 ausfiltern
BNE $Adresse ;Sprung falls Breakflag gesetzt
```

Analog muß man auch vorgehen, wenn man testen will, ob das Interrupt oder Dezimalflag gesetzt ist. Da dies allerdings nicht oft der Fall ist, wird die oben stehende Lösung nur der Vollständigkeit halber erwähnt.

Um noch einmal darauf hinzuweisen:
In den meisten Assemblern oder Maschinensprachemonitoren (z.B. TEDMON) ist es möglich, die Adresse hinter einem bedingten Sprungbefehl, wie z.B. BEQ oder BCS, als 16-Bit-Adresse anzugeben. Dies dient nur zur Vereinfachung für den Programmierer. Im Speicher wird die Adresse als ein Byte abgelegt.

Hier nun die Befehlskodes der einzelnen Sprungbefehle, die natürlich nur in der relativen Adressierung existieren:

```
BEQ: $F0
BNE: $D0
BCS: $B0
BCC: $90
BMI: $30
BPL: $10
BVS: $70
BVC: $50
```

2.2.8 Die unbedingten Sprünge JMP

Außer den bedingten Sprüngen gibt es natürlich noch die unbedingten Sprünge, die ähnlich dem GOTO-Befehl in BASIC immer ausgeführt werden. Mit einem solchen JMP-Befehl kann man auch im ganzen Adressraum hin- und herspringen. Hieraus resultiert dann, wie schon im vorhergehenden Kapitel angesprochen, daß beim JMP-Befehl die Zieladresse mittels eines 16-Bitwertes dargestellt werden muß. Außer dieser absoluten Adressierung, die wie beim LDA-Befehl funktioniert, also im Speicher erst das Befehlswort und dann die Adresse im LOW-HIGH-Format (siehe Kap. 2.1), existiert noch eine andere Adressierungsart für den JMP-Befehl. Diese Adressierungsart haben wir in etwa schon bei der indirekt-indizierten Adressie-

rung besprochen. Es handelt sich nämlich um die indirekte Adressierung. Erinnern wir uns, wie die indirekt-indizierte Adressierung funktioniert:

Der Parameter hinter dem LDA-Befehl gab eine Zeropage-adresse an, in der das LOW-Bit (Bits 0-7) der anzusprechenden Adresse stand. In der darauffolgenden Speicherstelle in der Zeropage stand dann das HIGH-Byte (Bits 8-15). Diese beiden einzelnen Bytes wurden dann zu einer Adresse zusammengesetzt und zu dieser Adresse dann noch der Inhalt des Y-Registers addiert.

Fast genauso funktioniert jetzt auch der indirekte JMP-Befehl. Es entfällt lediglich die Indizierung durch das Y-Register und die Speicherstellen, in denen die Adresse des eigentlichen Sprungzieles steht, müssen nicht unbedingt im Bereich der Zero-page liegen.

Der indirekte JMP-Befehl funktioniert also so:

Aus der angegeben Speicherstelle und der auf diese Speicher-stelle folgenden Speicherstelle wird die Adresse ermittelt, zu der gesprungen werden soll. Hierbei steht das LOW-Byte der Zieladresse in der ersten und das High-Byte der Zieladresse in der zweiten Speicherstelle. Dazu wieder ein Beispiel:

In der Speicherstelle $1000 stehe der Wert $D8 und in der fol-genden Speicherstelle ($1001) der Wert $F4.

```
JMP ($1000)
```

Der Prozessor holt sich nun den Inhalt ($D8) der angegebenen Speicherstelle ($1000) und setzt diesen für die untersten 8 Bits der Zieladresse ein. Die Adresse sieht bis jetzt also so aus:

	HIGH-Byte								LOW-Byte							
Bit:	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
Wert:	?	?	?	?	?	?	?	?	1	1	0	1	1	0	0	0

Dann holt er sich den Inhalt ($F4) der folgenden Speicherstelle ($1001). Die Adresse sieht dann endgültig so aus:

	HIGH-Byte								LOW-Byte							
Bit:	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
Wert:	1	1	1	1	0	1	0	0	1	1	0	1	1	0	0	0

Zusammengesetzt lautet die Zieladresse also $F4D8. Und genau an dieser Adresse setzt der Prozessor nun seine Arbeit fort.

Die Befehlskodes für die beiden Adressierungsarten lauten:

```
Absolut:  $4C
Indirekt: $6C
```

2.2.9 Die Zählbefehle INC, INX, INY, DEC, DEX, DEY

Um in einer Programmiersprache effektiv programmieren zu können, benötigt diese Möglichkeiten zum wiederholten Ausführen einer Funktion mit geänderten Parametern. Von BASIC oder Pascal her kennen Sie sicherlich die FOR-Schleife. Diese Art der Schleifenkonstruktion erfordert eine sogenannte Laufvariable, die innerhalb der Schleife hoch- bzw. heruntergezählt wird. In BASIC würde das wahrscheinlich so aussehen:

```
FOR I=1 TO 100:PRINT I:NEXT I
```

oder in Pascal so:

```
FOR i:=1 TO 100 DO WRITELN (i);
```

Auf diese Art der Schleifenkonstruktion werden Sie sicherlich nicht gern verzichten, doch leider ist in Maschinensprache eine komfortabele Schleifenprogrammierung nicht vorgesehen. Wir müssen die Schleifen also selbst programmieren, indem wir innerhalb unserer Schleife eine Speicherstelle oder ein Register bis zu einem bestimmten Wert immer um 1 erniedrigen oder erhöhen.

Hierzu dienen die sogenannten Zählbefehle. Es existieren jeweils zwei Zählbefehle für die beiden Indexregister (X/Y) und allgemein für beliebige Speicherstellen. Bei den Zählbefehlen für Speicherstellen spricht man auch von sogenannten READ-MODIFY-WRITE-Befehlen, da der Prozessor zuerst den Inhalt der angegebenen Speicherstelle liest (READ), diesen dann modifiziert (MODIFY) und dann wieder in die Speicherstelle schreibt (WRITE), ohne das ein Registerinhalt verändert wird.

Beginnen wir nun mit der Erklärung der Zählbefehle für die beiden Indexregister:

Zur Erhöhung des Inhaltes der beiden Indexregister existieren der INX- und der INY-Befehl. Der Inhalt eines Indexregisters wird dabei immer um den Wert 1 erhöht. Tritt hierbei ein Überlauf auf, so wird dieser nicht registriert, es wird also wieder bei 0 zu zählen begonnen. In einem Schema sieht das dann so aus:

```
1,2,3,...253,254,255,0,1,2...
```

Bei einem solchen Überlauf wird auch nicht das Carryflag beeinflusst. Lediglich das Negativ- und das Zeroflag werden von den sogenannten Inkrementierbefehlen beeinflusst. Ist nach der Ausführung eines INX oder INY-Befehls der Inhalt des Indexregisters gleich 0, wird das Zeroflag gesetzt. Anderenfalls wird es gelöscht. Ähnlich verhält es sich auch mit dem Negativflag. Bei Registerinhalten, die kleiner als $7F sind, bleibt es gelöscht; bei größeren Inhalten wird es gesetzt. Um nun eine Schleife von $00 bis $FF zu programmieren muß man so vorgehen:

```
      LDX #$00 ;X-Register mit 0 laden
LOOP  INX      ;Den Inhalt des X-Registers um 1 erhöhen
      CPX #$FF ;Ist der Inhalt ungleich $FF, so ist Z=0
      BNE LOOP ;Bei Ungleichheit springen
```

Der Befehl INC, der zum Erhöhen von Speicherinhalten dient, arbeitet analog. Er beeinflusst jedoch nur die Flags des Statusregisters und die angegebene Speicherstelle, nicht aber den Inhalt des Akkus oder eines Indexregisters.

Das Gegenteil zu den Befehlen INX, INY und INC stellen die Befehle DEX, DEY und DEC dar. Auch sie beeinflussen nur das Zero- und das Negativflag. Ein Unterlauf wird ebenfalls nicht registriert. Das Herunterzählen sähe schematisch so aus:

```
255,254,253,...3,2,1,0,255,254...
```

Um nun eine Schleife von $FF bis $00 programmieren zu können, kann man so vorgehen:

```
      LDX #$FF ;X-Register mit $FF laden
LOOP  DEX      ;Inhalt des X-Registers vermindern
      CPX #$00 ;Mit 0 vergleichen, wenn gleich, dann Z=1
      BNE LOOP ;Ungleich, dann weiter zählen
```

Dieses Programm würde unsere Anforderungen voll erfüllen, doch auch das folgende Programm arbeitet zu unserer vollen Zufriedenheit:

```
      LDX #$FF ;X-Register mit $FF laden
LOOP  DEX      ;Inhalt des X-Registers vermindern
      BNE LOOP ;Ungleich 0, dann weiter zählen
```

Das zweite Programm funktioniert ebenfalls, da der DEX-Befehl ja schon das Zeroflag beeinflusst. Ist der Inhalt des X-Registers gleich 0, wird das Zeroflag sowieso gesetzt. Der Inhalt des X-Registers braucht daher nicht mehr umständlich überprüft zu werden. Überlegen Sie sich also gut wo und ob Sie einen Vergleichsbefehl einsetzen müssen, denn gerade Schleifen gehören zu den zeitkritischsten Programmteilen. Im ersten Programm wird der CPX-Befehl 255 mal durchlaufen!

Hier die möglichen Adressierungsarten für den INC- und DEC-Befehl und die Befehlskodes der anderen Zählbefehle:

Adressierungsart	Beispiel	INC	DEC
absolut	Befehl $Adresse	$EE	$CE
absolut x-indiziert	Befehl $Adresse,X	$FE	$DE
absolut y-indiziert	Befehl $Adresse,Y	---	---
zeropage	Befehl $Z.adresse	$E6	$C6
zeropage x-indiziert	Befehl $Z.adresse,X	$F6	$D6

```
INX: $E8
INY: $C8
DEX: $CA
DEY: $88
```

2.2.10 Die Flagbefehle SEC, CLC, SED, CLD, SEI, CLI, CLV

Um einzelne Flags im Statusregister zu setzen oder zu löschen, existieren die sogenannten Flagbefehle. Einige dieser Flagbefehle haben wir schon im Zusammenhang mit den arithmetischen Operationen kennengelernt. Es existieren jeweils für das Carry-, das Dezimal- und das Interruptflag zwei Befehle, die das entsprechende Flag beeinflussen. Jeweils ein Befehl, um das spezifische Flag zu löschen und noch ein Befehl, um es gegebenenfalls zu setzen. Für das Overflowflag existiert nur ein Befehl zum Löschen des Flags. Hier die Beschreibung der einzelnen Befehle:

SEC, CLC:

Mit dem Befehl SEC (SEt Carry) wird das Carryflag gesetzt. Die gegenteilige Funktion hat der Befehl CLC (CLear Carry). Beide Befehle haben eine große Bedeutung bei den arithmetischen und logischen Befehlen.

SED, CLD:

Diese beiden Befehle dienen zum Setzen (SED: SEt Decimal) und zum Löschen (CLD: CLear Decimal) des Dezimalflags. Ist das Dezimalflag gesetzt, so arbeitet der Prozessor im Dezimalmodus,

andernfalls im normalen Binärmodus. Auf den Dezimalmodus gehen wir aus Gründen der Vollständigkeit noch am Ende dieses Kapitels gesondert ein.

SEI, CLI:

Durch den Befehl SEI (SEt Interrupt disable) und CLI (CLear Interrupt disable), kann das Interruptflag gesetzt (SEI) und gelöscht (CLI) werden. Ist das Interruptflag gesetzt, so sind alle Interrupts gesperrt. Eine Ausnahme bildet hierbei der sogenannte NMI-Interrupt (Non Maskerable Interrupt). Auf die verschiedenen Arten des Interrupts werden wir noch in einem speziellen Kapitel eingehen (siehe Inhaltsverzeichnis).

CLV:

Durch den Befehl CLV (CLear oVerflow) wird das Overflowflag gelöscht. Ein Befehl zum Setzen des Overflowflags existiert wegen dessen untergeordneter Bedeutung nicht.

Hier nun die Tabelle mit den Befehlskodes des einzelnen Befehle:

```
CLC: $18
SEC: $38
CLD: $D8
SED: $F8
CLI: $58
SEI: $78
CLV: $B8
```

An dieser Stelle will ich Ihnen noch, wie oben schon angekündigt, den sogenannten Dezimalmodus erklären:

Außer im sogenannten Binärmodus, den wir ja schon kennen, kann der Prozessor noch in einer anderen Betriebsart betrieben werden. Diese Betriebsart ist der sogenannte Dezimalmodus. Eingesetzt wird dieser Modus allerdings nur sehr selten, so z.B. in Mathematikprogrammen usw., wo es auf eine erhöhte

Rechengenauigkeit ankommt. Aus diesem Grund ist der Dezimalmodus den meisten Maschinenspracheprogrammierern auch nur vom Namen her bekannt.

Im Dezimalmodus kann durch ein Byte eine Zahl im Bereich von 0 bis 99 dargestellt werden. Um dies zu ermöglichen, wird das Byte wieder in zwei Teile gespalten, die sogenannten Nibbles (siehe auch Kap. 1.2). Durch vier Bits lassen sich nun, wie schon aus Kap. 1.2 bekannt ist, Zahlen im Bereich von 0 bis 16 darstellen. Im Dezimalmodus werden allerdings nur die Zahlen 0 bis 9 verwendet. Die unteren vier Bits stellen im Dezimalmodus die Einerstelle und die oberen vier Bits die Zehnerstelle dar. Hat das untere Halbbyte z.B. den Wert 9 und das obere Halbbyte den Wert 8, so hat das Byte den Wert 89, vorausgesetzt natürlich, daß der Dezimalmodus aktiv ist. Der Dezimalmodus wird ebenfalls von den arithemtischen Befehlen ADC und SBC registriert. Es ist also auch möglich, Zahlen im Dezimalformat zu addieren oder zu subtrahieren.

Durch das Zusammensetzen mehrerer Bytes können Zahlen mit beliebig vielen Stellen aufgebaut werden. Will man mit positiven und negativen Zahlen arbeiten, so muß man noch ein Halbbyte für das Vorzeichen reservieren. Bei Kommazahlen muß nur die Stelle, an der das Komma steht, registriert werden. Noch einmal zusammengefasst:

Im Dezimalmodus kann durch ein Byte eine Zahl im Bereich von 0 bis 99 dargestellt werden. Jedes Halbbyte stellt hierbei eine Ziffer im Bereich von 0 bis 9 dar. Hieraus folgt, daß nicht alle Bitkombinationen für den Dezimalmodus definiert sind. Dazu ein Beispiel:

Die Binärzahl %01100111 hat im Dezimalmodus folgenden Wert:

```
Bit:  0 1 1 0  0 1 1 1
Wert: 0 4 2 0  0 4 2 1
Erg.:    6        7
```

Der dezimale Wert unserer Binärzahl im Dezimalmodus ist also 67.

Wie funktionieren aber Addition und Subtraktion im Dezimalmodus? Dazu vorweg drei Beispiele:

```
Beispiel 1:

   %01110000     7 0    Flags vorher
 + %00100000   + 2 0    C=0; N=0; V=0
 = %10010000   = 9 0    C=0; N=1; V=1

Beispiel 2:

   %10010000     9 0    Flags vorher:
 + %00100000   + 2 0    C=0; N=0; V=0
 = %00010000   = 1 0    C=1; N=1; V=0

Beispiel 3:

   %01000100     4 4    Flags vorher:
 - %01010101   - 5 5    C=1; N=0; V=0
 = %10001001   = 8 9    C=0; N=1; V=0
```

Wie kann man die Ergebnisse der einzelnen Beispiele nun interpretieren?

In den ersten beiden Beispielen wurde das Carryflag jedesmal vor einer Addition auf 0 gesetzt, also gelöscht. Bei der Subtraktion (Beisp. 3), wurde das Carryflag vor der Subtraktion gesetzt. Es wurde also genauso vorgegangen, wie bei der normalen, binären Addition bzw. Subtraktion.

Das Wesentlichste, das sich geändert hat, ist das Verknüpfungsschema. Es wird nämlich nicht mehr bitweise vorgegangen, sondern sozusagen ziffernweise. Intern werden zuerst die beiden dezimalen Werte der unteren Bithälften addiert. Das Ergebnis dieser Addition wird dann wieder durch die untersten vier Bits im Ergebnis dargestellt. Trat bei dieser Addition ein Überlauf auf, so wird dieser bei der Addition der oberen Bytehälften be-

rücksichtigt. Tritt bei der Addition des oberen Nibbles ein Überlauf auf, so wird dieser in das Carryflag übernommen. Das Carryflag stellt also praktisch die Hunderter dar.

Die Subtraktion verläuft in etwa analog, also ein Unterlauf wird auch immer auf die nächste höhere Stelle verrechnet. Tritt allerdings bei der Subtraktion der beiden oberen Bytehälften ein Unterlauf auf, wird das Carryflag in diesem Fall gelöscht. Das Ergebnis ist dann als negative Zahl zu betrachten (Beisp. 3).

Nun zur Funktion des Negativ- und des Overflowflags. Wie wir anhand des ersten Beispiels sehen können, wird das Overflowflag immer dann gesetzt, wenn sich die Zehnerziffer nicht mehr durch drei Bits darstellen läßt, also das 7. Bit gesetzt ist.

Wie berechnet man nun aber den absoluten Wert, wenn ein Unterlauf aufgetreten ist? Im Binärsystem drehten wir hierzu alle Bits um und addierten 1. Fast genauso funktioniert dies auch im Dezimalmodus:

Als erstes führen wir eine EXKLUSIV-ODER-Verknüpfung mit dem dezimalen Wert 99 durch. Im Dezimalmodus sieht dieser Wert so aus:

```
Bit:  76543210
Wert: 10011001
```

Oder Hexadezimal ausgedrückt ebenfalls $99. Zu dem Ergebnis dieser Verknüpfung addieren wir dann noch 1 und schon haben wird das absolute Ergebnis unserer Subtraktion. Führen wir den eben beschriebenen Prozess einmal anhand unseres dritten Beispiels durch:

```
Ergebnis: %10001001
EXOR:     %10011001
Ergebnis: %00010000
plus 1:   %00000001
Ergebnis: %00010001
```

Das Ergebnis ist also 11. Rechnen wir dies einmal nach:

```
  44
- 55
=-11
```

Wir müssen uns das Ergebnis 11 also nur noch als negative Zahl vorstellen. Das korrekte Ergebnis unserer Subtraktion lautet also -11.

Sollten Sie nicht vorhaben im Dezimalmodus zu programmieren, so macht es überhaupt nichts, wenn Sie sich jetzt noch nichts unter der dezimalen Betriebsart vorstellen können. Benötigen Sie den Dezimalmodus aber doch einmal, so kommt mit dem Interesse bestimmt auch das Verständnis für die oben beschriebenen Vorgänge.

2.2.11 Die Verschiebebefehle ASL, LSR, ROL, ROR

Außer den logischen und arithmetischen Befehlen existieren noch die sogenannten Verschiebebefehle zum Bearbeiten eines Bytes. Man kann mit ihrer Hilfe z.B. den Akku um eine Stelle nach links oder nach rechts verschieben. Es werden zwei Arten von Verschiebebefehlen unterschieden. Als erstes die Befehle, die ein Register nach links respektive rechts verschieben und in das freiwerdende Bit eine 0 schieben. Bei dieser Art an Verschiebebefehlen kommt das herausgeschobene Bit ins Carryflag. War es 1, so ist das Carryflag gesetzt (1), war es 0, so ist das Carryflag gelöscht.

Schauen wir uns hierzu das Schema der Linksverschiebung an:

Für die Linksverschiebung durch ASL:

```
Vor der Verschiebung:

Carry = 0; Bit:  7 6 5 4 3 2 1 0
           Wert: 1 0 1 0 1 0 1 0
```

```
Nach der Verschiebung:

Carry = 1; Bit:  7 6 5 4 3 2 1 0
           Wert: 0 1 0 1 0 1 0 0
```

Die Rechtsverschiebung durch LSR funktioniert analog. Es wird halt nur das 0. Bit ins Carry geschoben und in das 7. Bit eine 0 eingesetzt.

Für die Linksverschiebung existiert also der Befehl ASL (Arithmetik Shift Left), für die Rechtsverschiebung der Befehl LSR (Logical Shift Right).

Die zweite Art der Verschiebebefehle sind die Befehle, die ein Byte zyklisch rotieren. Bei ihnen wird in das freiwerdende Bit nicht eine 0, sondern der Inhalt des Carryflags geschoben. Das herausgeschobene Bit wird dann im Carryflag abgelegt.

Wichtig ist bei dieser Art der Verschiebung die Reihenfolge der einzelnen Prozesse. Erst wird das Byte verschoben und das herausgeschobene Bit intern gespeichert. Dann wird der Inhalt des Carryflags an die Stelle des freiwerdenden Bits gesetzt. Erst nachdem dies geschehen ist, wird der zwischengespeicherte Wert des herausgeschobenen Bits in das Carryflag geschoben. Es ist also nicht möglich ein Byte einfach nach links zu verschieben, so daß das 7. Bit an die Stelle des 0. Bits tritt. Wie man jedoch so etwas bewerkstelligen kann, erfahren Sie weiter unten.

Schauen wir uns diesmal das Schema der Rechtsverschiebung an:

Für die Rechtsverschiebung durch ROR:

```
Vor der Verschiebung:

Carry = 0; Bit:  7 6 5 4 3 2 1 0
           Wert: 1 0 1 0 1 0 1 0
```

```
Nach der Verschiebung:

Carry = 0; Bit:  7 6 5 4 3 2 1 0
           Wert: 0 1 0 1 0 1 0 1
```

Das Carryflag hat also jetzt den Inhalt des ehemaligen 0. Bit, während der ehemalige Inhalt des Carryflags jetzt an der Stelle des 7. Bits steht. Die Linksverschiebung funktioniert analog.

Für die Linksverschiebung existiert in diesem Fall der Befehl ROL (ROtate Left) und für die Rechtsverschiebung der Befehl ROR (ROtate Right).

Anwendungsbeispiele:

Die Verschiebebefehle werden häufig benutzt, um Werte zu verzweifachen, zu vervierfachen usw..

Verschiebt man nämlich ein Byte um eine Stelle nach links, so entspricht dies einer Multiplikation mit 2. Analog dazu entspricht die Verschiebung nach rechts einer Division durch 2. Um also z.B. dem Inhalt des Akkus zu vervierfachen müßte man zweimal (2*2=4) den Befehl ASL A ausführen lassen (das 'A' hinter dem Befehlswort zeigt dem Assembler oder Monitor nur an, daß es sich um die Akkumulatoradressierung handelt). Die möglichen Überträge müssen natürlich berücksichtigt werden und gegebenenfalls in ein weiteres Byte geschoben werden, so daß es sich nach Abschluß der Verschiebung um einen 16-Bit-Wert handelt. Dazu ein Beispiel:

Die Speicherstelle $D8 enthalte den Wert $FF, die folgende Speicherstelle $D9 den Wert 0. Der Inhalt der Speicherstelle $D8 soll nun verachtfacht werden (2*2*2=8). Dazu würde man folgendermaßen vorgehen:

```
ASL $D8
ROL $D9
ASL $D8
```

```
ROL $D9
ASL $D8
ROL $D9
```

Wie funktioniert unser Programm nun? Als erstes wird der Inhalt der Speicherstelle $D8 um eine Position nach links verschoben. An die Stelle des 0. Bits rückt eine 0, das 7. Bit wird ins Carry geschoben. Um nun den Übertrag zu berücksichtigen, der ja eigentlich das 8. Bit darstellt, müsste dieser in ein weiteres Byte geschoben werden. Diese Aufgabe übernimmt die Befehlssequenz ROL $D9. Die Speicherstelle $D9 wird hierbei um eine Bit nach links verschoben und der Inhalt des Carryflags ( ) an die Position des 0. Bits gesetzt wird.

Führen wir die oben genannte Befehlsfolge dreimal aus, so haben wird den Inhalt der Speicherstelle $D8 verachtfacht. Allerdings handelt es sich jetzt um einen 16-Bit-Wert, da er ja durch zwei Speicherstellen dargestellt wird.

Wie oben schon angeschnitten, hier eine Lösungsmöglichkeit um z.B. den Akku zyklisch zu verschieben:

```
        LDA #%10101010; Akku mit Wert laden
                        Negativflag enthält Wert des 7. Bits
        CLC           ; Carry vorsorglich löschen
        BPL LABEL     ; 7. Bit = 0, dann Sprung
        SEC           ; 7. Bit = 1, Carry auch = 1
LABEL   ROL A         ; Akku rotieren.
```

Bei der Rotation wird der Inhalt des Carryflags in das 0. Bit geschoben. Das Carryflag enthält aber den Inhalt des Negativflags, das wiederum den Inhalt des 7. Bits enthält. Operiert man mit Speicherstellen, so findet der BIT-Befehl seine Anwendung (Kap. 2.2.5).

Hier die Tabelle mit den Adressierungsarten und Befehlskodes:

Adressierungsart	Beispiel	ASL	LSR	ROL	ROR
akkumulator	Befehl A	$0A	$4A	$2A	$6A
unmittelbar	Befehl #$Byte	---	---	---	---
absolut	Befehl $Adresse	$0E	$4E	$2E	$6E
absolut x-indiziert	Befehl $Adresse,X	$1E	$5E	$3E	$7E
absolut y-indiziert	Befehl $Adresse,Y	---	---	---	---
zeropage	Befehl $Z.adresse	$06	$46	$26	$66
zeropage x-indiziert	Befehl $Z.adresse,X	$16	$56	$36	$76
zeropage y-indiziert	Befehl $Z.adresse,Y	---	---	---	---
indirekt indiziert	Befehl ($Z.adresse),Y	---	---	---	---
indiziert indirekt	Befehl ($Z.adresse,X)	---	---	---	---

Unter der Adressierungsart 'akkumulator' versteht man in diesem Fall, wenn sich der Verschiebebefehl direkt auf den Akkumulator bezieht. Diese Adressierungsart wird in der Regel durch ein 'A' hinter dem Befehlswort gekennzeichnet.

2.2.12 Die Stackbefehle PHA, PLA, PHP, PLP, TXS, TSX

Um die Funktion der Stackbefehle erklären zu können, muß an dieser Stelle zuerst einmal geklärt werden, was überhaupt ein Stack, oder mit anderem Namen ein Stapel, ist.

Unter einem Stack versteht man einen Speicherbereich im RAM-Speicher, der für den Prozessor reserviert ist. Dieser reservierte Speicherbereich beginnt an der Adresse $0100 und endet an der Adresse $01FF. Er belegt also die gesamte erste Page (Siehe Kap. 2.1).

Der Prozessor benutzt den Stapel hauptsächlich zum Abspeichern von Rücksprungadressen beim Aufruf von Unterprogrammen oder zur Zwischenspeicherung von Registerinhalten. Um nun zu wissen, an welcher Stelle der nächste Wert abgelegt werden soll, existiert der sogenannte Stapelzeiger oder Stackpointer. Der Stackpointer zeigt immer auf die nächste freie Adresse. Legt man jetzt einen Wert mittels eines speziellen Befehls auf den Stapel, so wird der Wert an der Adresse $0100 plus dem Inhalt des Stackpointers abgespeichert. Ist dies geschehen, so wird der

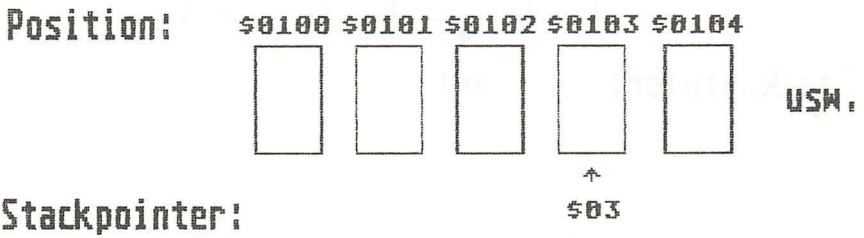
Stackpointer um 1 vermindert, so daß er immer auf die nächste freie Position im Stack zeigt. Soll nun ein Wert vom Stack geholt werden, so funktioniert das ganze Prinzip anders herum:

Der Stackpointer wird um 1 erhöht und dann der Wert geholt, der an der Adresse $0100 + Stackpointer (SP) steht. Legt man also mehrere Werte hintereinander auf den Stack, so wird als erstes immer nur der Wert geholt, der als letztes auf den Stack gelegt wurde. Dieses Prinzip bezeichnet man auch als LIFO-System. LIFO bedeutet LAST IN - FIRST OUT, was soviel bedeutet wie:

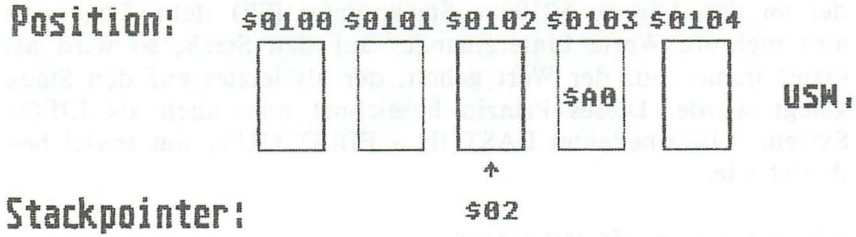
Als letztes rein, als erstes raus

Daher kommt auch die Bezeichnung Stapel. Auf einen Stapel legt man immer etwas an der obersten Position ab. Will man etwas vom Stapel herunternehmen, so muß man der Reihe nach alle Dinge von oben herunternehmen. Als Beispiel wollen wir uns einmal anschauen, wie es aussieht, wenn man drei Werte auf den Stapel legt und sie danach wieder herunternimmt. Der Stapelzeiger soll bei diesem Beispiel an der Position $03 stehen unsere drei Werte seien $A0, $CF und $01.

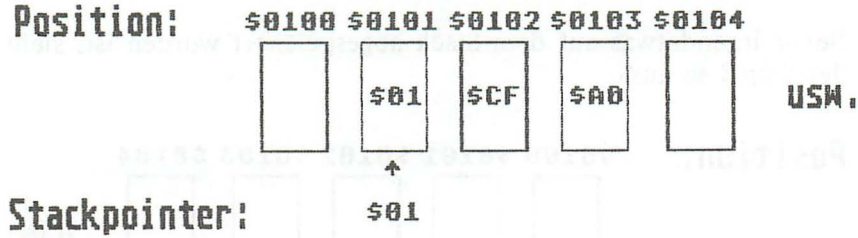
Bevor irgendetwas auf dem Stack abgespeichert worden ist, sieht der Stapel so aus:



Nun soll der erste Wert auf den Stack gelegt werden. Als erstes wird der Wert an der Adresse $0100 + Stackpointer abgespeichert. Danach wird der Stackpointer um eins vermindert. Der Stack sieht jetzt so aus:

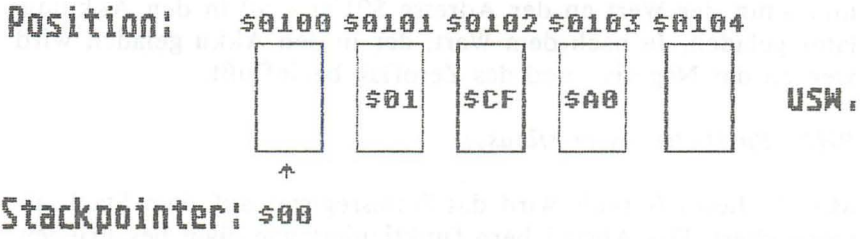


Das Ablegen der anderen beiden Werte funktioniert analog. Betrachten wir nun unser Beispiel, wenn alle drei Werte bereits auf dem Stack abgelegt worden sind. Der Stack sieht dann so aus:



Der Stackpointer zeigt also auf die nächste freie Position im Stack.

Soll nun ein Wert vom Stack geholt werden, wird der Stackpointer um eine Position erhöht. Er zeigt dann, wie folgt, auf die Position $01:



Dann wird der Wert geholt, der an dieser Position steht; es ist der Wert $01. Das Herunterholen der anderen beiden Werte funktioniert wieder analog. Nachdem der komplette Vorgang abgeschlossen ist, steht der Stackpointer wieder auf der Position $03. Wir haben also den Wert, den wir zuletzt auf dem Stack abgespeichert haben, zuerst wieder vom Stack geholt. Den Wert, den wir als erstes auf dem Stack abgelegt haben, haben wir als letztes wieder vom Stack geholt. Es handelt sich also tatsächlich um das LIFO-System.

Da wir nun wissen, was ein Stack bzw. ein Stackpointer ist, wollen wir nun auf die einzelnen Befehle zur direkten Stackmanipulation eingehen:

PHA (PusH Akkumulator):

Mittels des PHA-Befehls ist es möglich, den Inhalt des Akkus auf den Stack zu legen. Der Wert wird dabei an die Adresse $0100 + SP geschrieben. Anschließend wird der Stackpointer (SP) um 1 vermindert. Der Akkumulatorinhalt wird durch diesen Befehl nicht beeinflußt. Auch Flags beeinflußt dieser Befehl nicht.

PLA (PulL Akkumulator):

Dieser Befehl stellt den entgegengesetzten Befehl zum PHA-Befehl dar. Es wird nämlich der Stackpointer zuerst um 1 erhöht und dann der Wert an der Adresse $0100 + SP in den Akkumulator geladen. Je nach dem Wert, der in den Akku geladen wird, werden das Negativ- und das Zeroflag beeinflußt.

PHP (PusH Prozessor status):

Mittels dieses Befehls wird das Statusregister auf dem Stack abgespeichert. Das Abspeichern funktioniert wie oben beschrieben.

Dieser Befehl ist besonders dann sinnvoll, wenn Routinen aufgerufen werden sollen, ohne daß durch diese Routinen das Statusregister verändert werden soll. Man legt dann einfach das Statusregister auf den Stack, ruft die Routine auf, und lädt nach der Ausführung der Routine wieder den alten Inhalt in das Statusregister.

PLP (PulL Prozessor status):

Durch diesen Befehl wird ein Byte vom Stack geholt und im Statusregister abgelegt. Die Anwendung dieses Befehls entspricht der des PHP-Befehls, nur eben mit der umgekehrten Funktion.

Nun kommen wir zu Befehlen, die wir eigentlich auch schon im Kapitel über die Transferbefehle hätten besprechen können. Es handelt sich nämlich um zwei Befehle zur Manipulation des Stackpointers, wie dies z.B. bei Einschalten des Computers notwendig ist. Man kann nämlich mittels des Befehls

TXS

den Inhalt des X-Registers in den Stapelzeiger übertragen. Das Gegenteil kann durch den Befehl

TSX

erreicht werden. Durch diesen Befehl wird nämlich der Inhalt des Stapelzeigers in das X-Register übertragen.

Um noch einmal die Anwendung der Stackbefehle zu demonstrieren, hier ein Beispiel:

Aktivieren Sie hierzu bitte den Maschinensprachemonitor durch die Eingabe von 'MONITOR' und anschließendem Drücken der RETURN-Taste.

Der Monitor meldet sich nun wie folgt:

```
MONITOR

   PC  SR AC XR YR SP
; FF00 00 00 FF 00 F8
```

Geben Sie jetzt bitte das folgende Programm ein:

```
A 3000 PHA
A 3001 BRK
```

Starten Sie jetzt das Programm mit 'G 3000', gefolgt von der RETURN-Taste. Der Monitor meldet sich jetzt:

```
BREAK
   PC  SR AC XR YR SP
; 3003 30 00 FF 00 F7
```

Wenn Sie einmal den Inhalt des Stapelzeigers vor und nach unserem Programmaufruf betrachten, dann werden Sie erkennen, daß sich die beiden Werte um 1 unterscheiden. Der Inhalt des Stapelzeigers SP ist nach unserem Programmaufruf um 1 kleiner als vorher. Wie läßt sich dies erklären?

In unserem Programm haben wir den Inhalt des Akkumulators mittels des Befehls PHA auf dem Stack abgelegt. Nun wissen wir aber, daß nach einem solchen Prozess der Stackpointer um 1 vermindert wird, also von $F8 auf $F7.

Probieren wir doch einmal den umgekehrten Prozeß. Ändern Sie als erstes bitte den Inhalt des Akkumulators auf die folgende Art und Weise:

```
; 3003 30 FF FF 00 F7
```

Wir ändern also den Inhalt des Akkumulators, um hinterher erkennen zu können, daß wirklich wieder der alte Wert ($00) im Akku steht. Hier nun das Maschinenprogramm:

```
A 3000 PLA
A 3001 BRK
```

Starten Sie jetzt wie oben das Programm wieder mit 'G 3000' und dem Druck der RETURN-Taste. Der Monitor meldet sich jetzt so:

```
BREAK
   PC  SR AC XR YR SP
; 3003 32 00 FF 00 F8
```

Der Stackpointer hat also wieder den alten Wert $F8 und auch der Akku hat seinen alten Inhalt, nämlich $00.

Zur Erklärung des Befehls TXS hier ein kleiner Ausschnitt der NMI-Routine des Betriebssystems. Die NMI-Routine wird immer dann aufgerufen, wenn der RESET-Knopf gedrückt wurde. Hier nun der Ausschnitt:

```
. F2A4  A2 FF     LDX #$FF
. F2A6  78       SEI
. F2A7  9A       TXS
. F2A8  D8       CLD
```

Wie Sie sehen, wird zuerst das X-Register des Prozessors mit dem Wert $FF geladen. Dann wird mittels des SEI-Befehls der IRQ-Interrupt verhindert, was uns jetzt aber noch nicht interessieren soll. Nun folgt der Befehl, auf den es uns besonders ankommt. Wie oben schon beschrieben, wird durch den TXS-Befehl der Inhalt des X-Registers als neuer Stapelzeiger gesetzt. Da das X-Register hier den Wert $FF enthält, hat der Stapelzeiger nun auch den Wert $FF. Der Inhalt des X-Registers bleibt dabei erhalten.

Hier nun die Befehlskodes der einzelnen Befehle:

```
PHA: $48
PLA: $68
PHP: $08
PLP: $28
TXS: $9A
TSX: $BA
```

2.2.13 Die Unterprogrammbefehle JSR, RTS

An dieser Stelle wollen wir auf die Unterprogrammtechnik eingehen. Die Unterprogrammbefehle sind mit Sicherheit zwei der wichtigsten Maschinensprachebefehle.

Von BASIC her kennen Sie wohl schon die beiden Befehle GOSUB und RETURN. Der GOSUB-Befehl dient dazu, ein Unterprogramm aufzurufen, während der RETURN-Befehl dazu dient, ein Unterprogramm abzuschließen und mit dem Programmablauf hinter dem GOSUB-Befehl fortzufahren. Prinzipiell verhält es sich in der Maschinensprache genauso; es existiert ein Befehl zum Aufrufen eines Unterprogramms und einer zum Abschließen eines Unterprogramms. Der Befehl zum Aufrufen heißt JSR (Jump SubRoutine) und der Befehl zum Beenden eines Unterprogramms RTS (ReTurn from Subroutine). Damit nun nach der Ausführung eines Unterprogramms an die Stelle hinter dem JSR-Befehl gesprungen werden kann, muß die Adresse, von der der Sprungbefehl erfolgt ist, irgendwo abgespeichert werden. Wie eben schon angesprochen, wird hierfür

der Stack verwendet. Aufgrund des LIFO-Prinzips können fast beliebig viele Unterprogramme ineinander verschachtelt werden. Begrenzt wird die Anzahl der möglichen Unterprogramme nur durch die Größe des Stacks.

Kommen wir nun genauer zur Funktionsweise der Unterprogrammbefehle JSR und RTS:

Trifft der Prozessor bei der Abarbeitung eines Maschinenspracheprogramms auf den Befehlskode für den JSR-Befehl, so geschieht folgendes:

1. Der Prozessor nimmt die augenblickliche Position des Programmzeigers, die immer noch auf den Befehlskode für den RTS-Befehl steht. Auf diese Adresse addiert der Prozessor dann noch den Wert zwei. Der Programmzähler steht nun auf dem HIGH-Byte der Unterprogrammadresse. Diese Adresse wird nun seitens des Prozessors in ein LOW und ein HIGH-Byte zerlegt.

2. Der Prozessor legt das HIGH-Byte dieser Adresse auf den Stack und vermindert den Stackpointer um 1.

3. Dann legt er das LOW-Byte der Adresse auf den Stack und vermindert den Stackpointer wieder um 1. Insgesammt ist der Stackpointer jetzt um den Wert 2 kleiner, als vor dem Unterprogrammaufruf. Auf dem Stack liegt jetzt die Adresse an der der Befehlskode für den JSR-Befehl stand, plus 2.

4. Jetzt holt sich der Prozessor die Adresse, die hinter dem Befehlskode für den JSR-Befehl steht. Diese Adresse gibt die Adresse an, an der das Unterprogramm beginnt. Sie wird im LOW-High-Format im Speicher abgelegt. Diese Adresse wird nun als neuer Inhalt für den Programmzeiger angenommen. Der Prozessor arbeitet also jetzt das Unterprogramm ab.

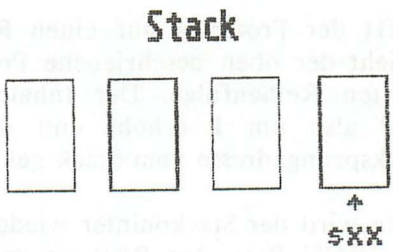
5. Trifft der Prozessor auf einen RTS-Befehl, so geschieht der oben beschriebene Prozeß in der umgekehrten Reihenfolge. Der Inhalt des Stackpointers wird also um 1 erhöht und das LOW-Byte der Rücksprungadresse vom Stack geholt.

6. Dann wird der Stackpointer wieder um 1 erhöht und das HIGH-Byte der Rücksprungadresse vom Stack geholt. Der Stackpointer hat jetzt den Wert, den er vor dem Aufruf des Unterprogramms hatte. Wie oben schon gesagt, zeigte die Adresse, die auf den Stack gelegt worden ist, auf das HIGH-Byte der Adresse hinter dem JSR-Befehl. An dieser Stelle soll das Maschinenprogramm natürlich nicht fortgesetzt werden. Daher wird die Adresse, die gerade vom Stack geholt worden ist, noch einmal um 1 erhöht; sie zeigt nun also auf den nächsten Maschinensprachebefehl hinter dem JSR-Befehl.

7. Zuletzt wird der Programmzeiger noch auf die Adresse gesetzt, die gerade vom Stack geholt worden ist (plus 1!). Das Programm wird jetzt wieder ganz normal hinter dem JSR-Befehl fortgesetzt.

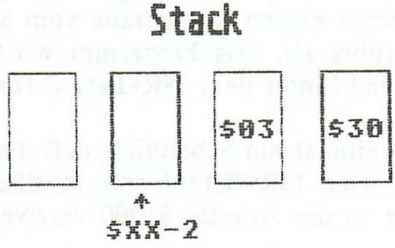
Dazu hier noch einmal ein Schaubild, daß dies noch einmal verdeutlichen soll. Der JSR-Befehl soll in diesem Fall zu einem Unterprogramm an der Adresse $1000 verzweigen:

Adresse	$3000	$3001	$3002	$3003	$3004
Inhalt:	Prg.-Text	JSR-Kode	Adr.-LOW	Adr.-HIGH	Prg.Text
Programm zeiger:	^ $3000				

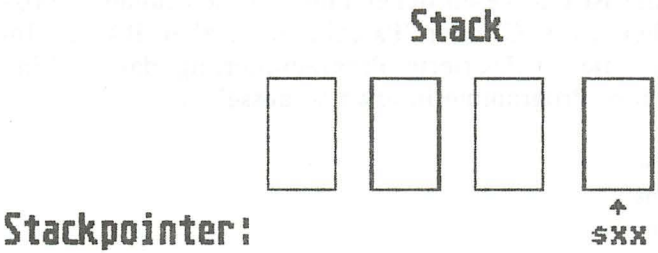


Nun trifft der Prozessor auf den Befehlskode für den JSR-Befehl. Er legt den aktuellen Inhalt des Programmzeigers plus 2 auf den Stack. Insgesamt sieht das so aus:

Adresse:	$3000	$3001	$3002	$3003	$3004
Inhalt:	Prg.-Text	JSR-Kode	Adr.-LOW	Adr.-HIGH	Prg.-Text
Programmzeiger:		^ $3001			



Jetzt arbeitet der Prozessor das Unterprogramm an der Adresse $1000 ab, bis er auf einen RTS-Befehl stößt. Dann holt er sich wieder das LOW und das HIGH-Byte der Rücksprungadresse vom Stack. Der Stack sieht dann wieder so aus:



Die Adresse, die vom Stack geholt worden ist, lautete $3003. Zu diese Adresse addiert der Prozessor jetzt noch den Wert eins. Die endgültige Adresse lautet dann $3004. Genau an dieser Adresse setzt der Prozessor dann seine Arbeit fort:

Adresse	$3000	$3001	$3002	$3003	$3004
Inhalt:	Prg.-	JSR-	Adr.-	Adr.-	Prg.
	Text	Kode	LOW	HIGH	Text
Programm:					^
zeiger:					$3004

Um ein Programm so flexibel wie möglich zu gestalten, sollten Sie versuchen, es in möglichst viele Teilprogramme zu zerlegen. Diese Teilprogramme sollten Sie als Unterprogramme programmieren und zwar so, daß man sie mit beliebigen Parametern aufrufen kann. Wollen Sie z.B. eine Routine zum Invertieren eines bestimmten Bildschirmbereichs schreiben, laden Sie die Speicherstellen, die Sie hierzu benötigen, nicht mit festen Werten in der Routine, sondern Sie bestimmen in der Routine nur, in

welchen Speicherstellen die Parameter stehen sollen. Hierdurch ist es dann später ohne große Probleme möglich, auch andere Bildschirmbereiche zu invertieren.

Diese Art der Programmierung nennt man strukturierte Programmierung. Sie ist ein wesentlicher Punkt in den höheren Programmiersprachen ,wie 'C' oder 'Pascal', nicht aber BASIC. Im Endeffekt sollte die strukturierte Programmierung darauf hinauslaufen, daß ihre Programme in etwa so aussehen:

```
Unterroutine 1
Unterroutine 2
Unterroutine 3
      .          Unterprogrammteil
      .
      .
Unterroutine X
JSR Unterprogramm A
JSR Unterprogramm B
      .                Hauptprogramm
      .
      .
```

Versuchen Sie möglichst so zu programmieren, sofern es von der Laufzeit her realisierbar ist.

Zum Schluß hier noch die Befehlskodes des JSR und des RTS-Befehls sowie die Parameterübergabe des JSR-Befehls:

```
JSR $Adresse

JSR: $20
RTS: $60
```

2.2.14 Die Interruptbefehle BRK, RTI

Obwohl die Interruptmöglichkeiten des 7501 noch an anderer Stelle weit ausführlicher behandelt werden, sollen hier der Vollständigkeit halber noch die beiden Interruptbefehle vorgestellt

werden. Es sind dies der BRK-Befehl mit dem Befehlskode $00 und der RTI-Befehl mit dem Befehlskode $40. BRK bedeutet soviel wie 'BReaK' und RTI soviel wie 'ReTurn from Interrupt'. Bei dem BRK-Befehl handelt es sich um einen Befehl zum Auslösen eines Softwareinterrupts. Der Prozessor verzweigt bei einem solchen Softwareinterrupt in der Regel zum eingebauten Maschinensprachemonitor. Der RTI-Befehl dagegen ist dem RTS-Befehl sehr ähnlich. Der einzige Unterscheid zum RTS-Befehl ist, daß beim RTI-Befehl noch der Inhalt des Statusregisters vom Stack geholt wird und der RTI-Befehl zum Verlassen einer Interruptroutine dient. Aus diesem Grund sollten Sie nicht etwa auf die Idee kommen, mit dem RTI-Befehl an Stelle des RTS-Befehls zu operieren.

2.2.15 Keine Operation NOP

Bei allen 65XX-kompatiblen Prozessoren existiert sogar noch ein Befehl, der überhaupt nichts vollbringt. Es ist dies der NOP-Befehl mit dem Befehlskode $EA. Obwohl er keine Funktion besitzt, dauert seine Abarbeitung natürlich eine gewisse Zeit, so daß er als Verzögerungsbefehl in Schleifen eingesetzt werden kann. Zweitens ist er recht nützlich, wenn Befehle aus einem Programm entfernt werden sollen, ohne daß sich das ganze Programm verschiebt. Man schreibt dann in die Speicherstellen, die vorher durch die zu löschenden Befehle belegt gewesen sind, den Befehlskode $EA. Hierfür eignet sich z.B. der 'F'-Befehl des Maschinensprachemonitors.

3. Anwendungen der Maschinensprache

3.1 Ausgeben eines Zeichens

Nachdem wir in den vorhergehenden Kapiteln die Grundlagen und den Befehlssatz des 7501 besprochen haben, kommen wir in den nun folgenden Kapiteln zu Anwendungen der Maschinensprache. Diese Anwendungen müssen nicht unbedingt äußerst komplex und optimal programmiert sein. Sie sollen Ihnen nur die verschiedenen Techniken und Adressierungsarten nahelegen. So ist es durchaus möglich, daß es für ein Problem bessere Lösungsalgorithmen gibt, als die in diesem Buch beschriebenen. Dies liegt dann aber daran, daß solche Lösungsbeispiele von Profis für Profis geschrieben wurden, während ich in diesen Kapiteln versuchen will, Ihnen die Schleifenprogrammierung usw. darzulegen.

Kommen wir jetzt zu der ersten Aufgabenbstellung, die wir mit Hilfe der Maschinensprache auf verschiedene Arten und Weisen lösen wollen. Es soll nämlich ein Zeichen auf dem Bildschirm ausgegeben werden. Da in der Maschinensprache kein PRINT-Befehl existiert, wollen wir uns zuerst einmal der zweiten Möglichkeit zuwenden, in BASIC ein Zeichen auf dem Bildschirm darzustellen. Gemeint ist das POKEN in den Bildschirmspeicher. Der Bildschirmspeicher ist ein Teil des normalen RAM-Speichers, aus dem der Bildschirmcontroller (TED) seine Informationen über das Aussehen des Bildschirminhaltes bezieht. Steht z.B. in der ersten Adresse des Videorams der Wert 65, so bedeutet dies für TED, daß er in der linken oberen Ecke des Bildschirms ein 'A' darstellen soll. Dazu muß noch gesagt werden, daß der Bildschirm zeilenweise aufgebaut ist. Das heißt, daß die erste Zeile die Speicherstellen 0-39, die zweite Zeile die Speicherstellen 40-79, die dritte Zeile die Speicherstellen 80-119 usw. des Bildschirmspeichers belegt. Der Bildschirmspeicher liegt übrigens von der Adresse 3072 bis 4071. Nähere Auskünfte hierzu erteilt entweder das Handbuch oder das Grafikbuch zum C16/116 und Plus/4, das ebenfalls im DATA-BECKER Verlag erschienen ist.

Setzen wir also erst einmal von BASIC aus ein 'A' in die linke obere Ecke des Bildschirms. Löschen Sie dazu zuerst den Bildschirm mittels der Tastenkombination SHIFT+CLR/HOME und betätigen Sie dann noch fünfmal die RETURN-Taste. Ist dies geschehen, geben Sie bitte die folgende Zeile gefolgt von der RETURN-Taste ein:

```
POKE 3072,65 (RETURN-Taste)
```

(Um zu ermitteln, welche Zahl im Bildschirmspeicher welchem Zeichen entspricht, verwenden Sie bitte die POKE-Kode-Tabelle im Handbuch)

Wie Sie sehen, erscheint in der linken oberen Ecke des Bildschirms ein 'A', wenn wir die Speicherstelle 3072 auf 65 setzen. Versuchen wir nun unsere BASIC-Lösung in Maschinensprache zu übertragen:

Wie wir schon besprochen haben, existieren nur drei Speicherbefehle - der STA, der STX und der STY-Befehl. Einen dieser drei Speicherbefehle müssen wir also schon auf jeden Fall benutzen. Nehmen wir also beispielsweise den STA-Befehl. Um mittels des STA-Befehls den Wert 65 in die Speicherstelle 3072 zu schreiben, müssen wir den Akkumulator zuerst mit dem Wert 65 laden. Zusammengefaßt sieht unsere Problemlösung in Maschinensprache dann so aus:

```
MONITOR

   PC  SR AC XR YR SP
; FF00 00 00 FF 00 F8

A 3000 LDA #$41  ;Akku mit 65 laden
A 3002 STA $0C00 ;in Speicherstelle 3072 schreiben
A 3005 BRK       ;zurück zum Monitor
```

(Die Texte hinter den Semikolons dürfen, inclusive der Semikolons, nicht eingegeben werden. Sie dienen nur der Dokumentierung!)

Löschen Sie nun den Bildschirm mittels SHIFT+CLR/HOME und drücken Sie fünfmal die RETURN-Taste. Geben Sie nun die folgende Zeile ein:

```
G 3000 (RETURN-Taste)
```

Wie Sie sehen, erscheint auch diesmal ein 'A' in der linken oberen Ecke des Bildschirms. Wir sind also schon in der Lage, einzelne Zeichen auf dem Bildschirm darzustellen. Doch was machen wir, wenn wir 100 mal ein 'A' auf dem Bildschirm ausgeben wollen? Wir könnten natürlich das folgende programmieren:

```
A 3000 LDA #$41  ;Akku mit 65 laden
A 3002 STA $0C00 ;in die Speicherstelle 3072 schreiben
A 3005 STA $0C01
A 3008 STA $0C02
       .
       .
       .
       STA $0C63
       BRK
```

Diese Möglichkeit ist denkbar unpraktisch und langsam. Daher ist es sinnvoller, eine Schleife zu programmieren. Dabei gibt es wieder mehrere Möglichkeiten. Fangen wir mit der Einfachsten an:

Da wir ja nur hundert Speicherstellen mit dem gleichen Wert füllen wollen, bietet sich zuerst einmal die indizierte Adressierung an. Erinnern wir uns noch einmal kurz, was unter der indizierten Adressierung zu verstehen ist:

Bei der indizierten Adressierung wurde eine Adresse angegeben, zu der dann immer der Inhalt des X oder des Y-Registers addiert wurde. Will man also ab einer bestimmten Speicherstelle noch 99 weitere Speicherstellen ansprechen, so erhöht man in einer Schleife immer das X- oder das Y-Register um den Wert 1. Schreiben wir also das entsprechende Programm:

Als erstes müssen wir den Inhalt des Indexregisters auf 0 setzen. Da wir mit LDA und STA arbeiten wollten, müssen wir das Y-Register nehmen. Der erste Befehl in unserem Maschinenprogramm wäre also:

```
LDY #$00
```

Dann müssen wir den Akkumulator mit dem Wert laden, der jeweils in die Speicherstellen geschrieben werden soll. Den Befehl hierzu hatten wir oben oben:

```
LDA #$41
```

Jetzt kommen wir zur Programmierung unserer Schleife. Das erste, was in unserer Schleife erledigt werden muß, ist den Wert in die Speicherstelle 3072 plus dem Inhalt des Y-Registers zu schreiben. Als Mnemonic sieht diese Anweisung so aus:

```
STA $0C00,Y ;Akku an Speicherstelle 3072+YR schreiben
```

Nun müssen wir den Inhalt des Y-Registers um eins erhöhen und testen, ob der Inhalt schon 100 ist, also schon die ersten hundert (0-99) Positionen im Bildschirmspeicher mit einem 'A' gefüllt worden sind. Zum Erhöhen existiert der Befehl INY und zum Vergleichen der Befehl CPY. Ergänzen wir unser Programm also um die folgenden beiden Befehle:

```
INY      ;Inhalt des YR um eins erhöhen
CPY #$64 ;mit hundert vergleichen
```

War der Inhalt des Y-Registers ungleich 100, so wird das Zeroflag gelöscht, andernfalls gesetzt. Wir müssen unsere Schleife also solange ausführen, wie das Zeroflag bei unserem Vergleich gelöscht ist. Für diesen Sprung bei gelöschtem Zeroflag existiert der Befehl BNE. Da der indizierte STA-Befehl der erste Befehl in unsere Schleife war, müssen wir also wieder zu ihm springen. Abgeschlossen werden sollte unser Programm erneut durch einen BRK-Befehl, um wieder in den Monitor zu springen. Komplett sieht unser Programm jetzt so aus:

```
MONITOR

   PC  SR AC XR YR SP
; 300E 00 41 FF 64 F8

A 3000 LDY #$00     ;YR mit 0 laden
A 3002 LDA #$41     ;Akku mit 65 laden
A 3004 STA $0C00,Y ;Akku an 3072+YR speichern
A 3007 INY          ;YR um 1 erhöhen
A 3008 CPY #$64     ;mit 100 vergleichen
A 300A BNE $3004    ;ungleich, dann Sprung
A 300C BRK          ;zum Monitor
```

Löschen Sie nun bitte wieder den Bildschirm mittels SHIFT+CLR/HOME und betätigen Sie circa fünfmal die RETURN-Taste. Jetzt können Sie die folgende Zeile eingeben und die RETURN-Taste drücken:

```
G 3000 (RETURN-Taste)
```

Fast gleichzeitig mit dem Druck der RETURN-Taste erscheinen die A's auf dem Bildschirm. Um unser Programm noch weiter zu optimieren, könnte man jetzt noch folgendes ändern:
Würde man durch den ersten Befehl das Y-Register nicht mit $00 sondern mit $63 laden und das Y-Register dann innerhalb der Schleife herunter- statt heraufzählen, so könnte man sich den Vergleichsbefehl CPY sparen. Man müßte dann allerdings anstelle des BNE-Befehl einen BMI-Befehl einsetzen. Optimiert sieht unser Programm also schließlich folgendermaßen aus:

```
MONITOR

   PC  SR AC XR YR SP
; 300E 00 41 FF 64 F8

A 3000 LDY #$63     ;YR mit 99 laden
A 3002 LDA #$41     ;Akku mit 65 laden
A 3004 STA $0C00,Y ;Akku an 3072+YR speichern
```

```
A 3007 DEY          ;YR um 1 vermindern
A 3008 BPL $3004    ;größer 0, dann Sprung
A 300A BRK          ;zum Monitor
```

Nun zur Funktionsweise des Programms. Wie Sie sehen, haben wir zuerst das Y-Register mit dem dezimalen Wert 99 geladen. Die dann folgenden Befehle sind dann wieder identisch mit denen des ersten Programms. Da das Programm aber jetzt von 99 bis 0 inclusive zählen soll, muß der Inhalt des Y-Registers um eins vermindert werden. Da der DEY-Befehl auch das Negativflag beeinflußt, können wir dieses Flag für unsere Verzweigung benutzen. Der Wertebereich, der beim DEY-Befehl auftritt, soll im Bereich von 99-0 liegen. Bei keinen dieser Zahlen ist also das 7. Bit gesetzt. Infolgedessen wird auch nicht das Negativflag gesetzt, solange die Zahlen im Bereich von 99 bis 0 liegen. Tritt allerdings ein Unterlauf auf, so nimmt das Y-Register den Wert 255 an. Da dieser Wert aber negativ ist, wird das Negativflag gesetzt und daher beim BPL-Befehl auch nicht mehr gesprungen. Sie sehen, zu tüfteln kann nie schaden; erst recht dann nicht, wenn dadurch das 99malige Durchlaufen eines recht zeitintensiven Vergleichbefehls verhindert wird.

Bis jetzt haben wir immer kleine Bereiche gefüllt, wie aber sieht es aus, wenn wir den ganzen Bildschirm füllen wollen? Natürlich existieren auch hier wieder mehrere Möglichkeiten:

1. Wir wenden die indirekt indizierte Adressierung an und durchlaufen dreimal eine Schleife von 0-255 und einmal eine Schleife von 0-232, da der Bildschirmspeicher nur 1000 Bytes umfaßt.

2. Wir arbeiten mit einer verschachtelten Schleife und teilen den Bildschirm in z.B. 25 Abschnitte mit jeweils 40 Bytes auf (25*40=1000).

Sowohl vom Programmieraufwand, als auch von der Übersichtlichkeit her, ist die zweite Lösung sicherlich besser. Aus diesem Grund hier noch einmal die genaue Aufschlüsselung des Prinzips:

Um einen Speicherraum von exakt 1000 Bytes zu füllen, teilt man diesen Bereich in 25 Bereiche zu jeweils 40 Bytes auf. Dann konstruiert man ein Programm, in dem 25 mal eine Schleife von 0-39 durchlaufen wird. Wir verschachteln also zwei Schleifen. Nach jedem Durchlauf der äußeren Schleife erhöhen wir den Startwert für die innere Schleife um 40. In BASIC würde das so aussehen:

```
100 AD=3072 : REM Startadresse Bildschirmspeicher
110 FOR Y=0 TO 24 : REM Äußere Schleife für Zeilen
120 :   FOR X=0 TO 39 : REM Innere Schleife für Spalten
130 :       POKE AD+X,65 : REM Kode für 'A' in Speicher
140 :   NEXT X : REM X um 1 erhöhen
150 :   AD=AD+40 : REM AD um 40 erhöhen
160 NEXT Y : REM Äußere Schleife um 1 erhöhen
```

Natürlich hätte man die äußere Schleife auch mit 'STEP 40' programmieren können, aber die Übertragung in Maschinensprache würde dann erheblich schwieriger aussehen.
Versuchen wir doch jetzt einmal, das BASIC-Programm in ein Maschinenprogramm umzuschreiben, denn das Prinzip ist ja das gleiche:

Dazu müssen wir als erstes, wie in Zeile 100, die Startadresse des Bildschirmspeichers festlegen. Da wir mit der indirekt indizierten Adressierung arbeiten werden, nehmen wir hierfür am besten zwei Zeropageadressen. Hier bieten sich dann die Adressen $D8/$D9 an, da sie vom Betriebssystem nicht benutzt werden. Die ersten vier Befehle unseres Maschinenprogramms müssen also so aussehen:

```
LDA #$00 ;LOW-Byte von 3072
LDY #$0C ;HIGH-Byte von 3072
STA $D8  ;LOW-Byte zuerst
STY $D9  ;dann das HIGH-Byte speichern
```

Nun müssen wir noch einen Indexzähler für die Zeilen einrichten. Wir nehmen hier am besten das X-Register, da wir das Y-Register noch bei der indirekt indizierten Adressierung benötigen werden, die ja nicht mit dem X-Register funktioniert. Der

Übersichtlichkeit halber wollen wir die beiden Schleifen wieder mit Vergleichsbefehlen programmieren, also nicht mit Programmiertricks, wie oben mit dem BPL-Befehl. Wir laden daher das X-Register erst einmal mit $00:

```
LDX #$00 ;X-Register mit 0 laden
```

Danach müssen wir auch noch das Y-Register auf Null setzen. Nun müssen wir noch den Akkumulator mit dem Bildschirmkode laden, der ausgegeben werden soll. Diesen Befehl legen wir innerhalb der ersten Schleife ab, da der Akkuinhalt später beim Addieren von 40 verändert wird. Wir ergänzen also:

```
LDA #$41 ;Bildschirmkode für 'A'
```

Jetzt können wir mit der inneren Schleife beginnen. Als erstes kommt wieder der Befehl zum Speichern des Bildschimkodes in der momentanen Adresse. Hierzu benutzen wir die indirekt indizierte Adressierung. In den Speicherstellen $D8/D9 steht immer die Adresse für den Zeilenanfang und mittels des Y-Registers laufen wir dann alle 40 Spalten einer Zeile durch.

```
LABEL STA ($D8),Y
```

Jetzt müssen wir natürlich das Y-Register um eins erhöhen und testen ob schon 40mal ein 'A' ausgegeben wurde. Wenn nicht, springen wir mittels eines BNE-Befehl wieder zu dem STA-Befehl:

```
INY       ;Y-Register um eins erhöhen
CPY #$28  ;Mit 40 vergleichen
BNE LABEL ;bei Ungleichheit zu STA-Befehl
```

Damit wäre die innere Schleife fertig. Wir müssen jetzt nur noch den Inhalt der Speicherstellen $D8/$D9 um 40 erhöhen, sowie auch das X-Register inkrementieren. Ist der Inhalt des X-Registers gleich 25, so ist das Programm beendet und wir springen mittels des BRK-Befehls wieder in den Maschinensprachemonitor. Hier nun das komplette Programm:

```
A 3000  LDA #$00    ;LOW-Byte von 3072
A 3002  LDY #$0C    ;HIGH-Byte von 3072
A 3004  STA $D8     ;Erst LOW-Byte
A 3006  STY $D9     ;dann HIGH-Byte speichern
A 3008  LDX #$00    ;Zeilenzähler auf 0
A 300A  LDY #$00    ;Spaltenzähler auf 0
A 300C  LDA #$41    ;Akku mit Kode für 'A' laden
A 300E  STA ($D8),Y ;'A' in Speicher schreiben
A 3010  INY         ;Spaltenzähler erhöhen
A 3011  CPY #$28    ;Schon 40 Spalten?
A 3013  BNE $300E   ;Nein, dann weiter ausgeben
A 3015  CLC         ;Carry für ADC vorbereiten
A 3016  LDA $D8     ;LOW-Byte in Akku laden
A 3018  ADC #$28    ;40 addieren
A 301A  STA $D8     ;und wieder speichern
A 301C  BCC $3020   ;Kein Übertrag, dann Sprung
A 301E  INC $D9     ;Übertrag! HIGH-Byte erhöhen
A 3020  INX         ;Zeilenzähler erhöhen
A 3021  CPX #$19    ;Schon 25 Zeilen?
A 3023  BNE $300A   ;Nein, dann wieder Schleife
A 3025  BRK         ;Ja! Dann zum Monitor
```

Interessant ist an diesem Programm auch die Erhöhung einer 16-Bit-Zahl um 40. Es wird nämlich nur das HIGH-Byte um eins ehöht, wenn das Carry gesetzt ist, also ein Übertrag aufgetreten ist.

Bis jetzt haben wir die Zeichen immer direkt in den Bildschirmspeicher geschrieben, doch es gibt auch noch andere Möglichkeiten, ein Zeichen auszugeben. In BASIC kann mittels des PRINT-Befehls an jeder beliebigen Stelle ein Zeichen ausgegeben werden. Innerhalb dieses Textes, der ausgegeben werden soll, können auch noch Steuerzeichen usw. vorkommen. Es muß also schon im Betriebssystem unseres Rechners ein Unterprogramm geben, das ein Zeichen ausgibt.

Diese Routine, die den Namen 'BSOUT' trägt, beginnt bei der Adresse $FFD2. Ruft man diese Adresse mittels eines JSR-Befehls auf, so wird an der momentanen Cursorposition das Zeichen ausgegeben, dessen ASCII-Kode im Akkumulator steht. In-

nerhalb dieser Unterroutine werden das X-, das Y- und das Statusregister zwischengespeichert, so daß die Routine keinen Registerinhalt verändert. Geben wir jetzt einmal ein Zeichen mittels der BSOUT-Routine des Betriebssystems aus. Aktivieren Sie hierzu den Maschinensprachemonitor, der sich wie folgt meldet:

```
MONITOR

   PC  SR AC XR YR SP
; 300C 00 2A FF 00 F8
```

Geben Sie jetzt das folgende Maschinensprogramm ein:

```
A 3000 LDA #$0D  ;Akku mit RETURN-Kode laden
A 3002 JSR $FFD2 ;RETURN-Kode ausgeben
A 3005 LDA #$2A  ;Akku mit Kode für '*' laden
A 3007 JSR $FFD2 ;'*' ausgeben
A 300A BRK       ;zum Monitor
```

Starten Sie nun unser eben eingegebenes Maschinenprogramm durch folgenden Befehl:

```
G 3000 (RETURN-Taste)
```

Die Ausgabe unseres Maschinenprogramms sieht jetzt so aus:

```
*
BREAK
   PC  SR AC XR YR SP
; 300C 70 2A FF 00 F8
```

Es wir also ein Sternchen ('*') unter dem 'G'-Befehl ausgegeben. In unsrem Maschinenprogramm wird zuerst der Akkumulator mit dem dezimalen Wert 13 geladen. Dieser Wert entspricht dem ASCII-Kode für die RETURN-Taste. Dann wird mit diesem Wert die BSOUT-Routine an $FFD2 aufgerufen. Da der RETURN-Kode ausgegeben werden sollte, wird der Cursor auf den Anfang der nächsten Zeile gesetzt und wieder zurückgesprungen. In unserem Programm wird dann der Akkumulator

mit dem dezimalen Wert 42 geladen. Dieser Wert ist der ASCII-Kode für das Sternchen ('*'). Da wir nun die Routine BSOUT aufrufen, wird das Sternchen an der aktuellen Cursorposition ausgegeben. Da die Cursorposition in diesem Fall in der ersten Spalte, der Zeile unter dem 'G'-Befehl war, wird das Sternchen auch dort ausgegeben.

Sie sehen, die BSOUT-Routine ist sehr viel komfortabler als das POKEN in den Bildschirmspeicher. Um jetzt mittels der BSOUT-Routine 256 Zeichen auszugeben, genügt es, eine Schleife von 0 bis 255 zu konstruieren, in der dann der Akku mit dem auszugebenden ASCII-Wert geladen wird und dann die BSOUT-Routine aufgerufen wird. Hier jetzt das Programm, das 256mal ein 'A' auf den Bildschirm ausgibt:

```
A 3000  LDY #$00  ;Y-Register mit Startwert laden
A 3002  LDA #$41  ;Akku mit ASCII-Wert von 'A' laden
A 3004  JSR $FFD2 ;BSOUT aufrufen - 'A' ausgeben
A 3007  INY       ;Y-Register erhöhen
A 3008  BNE $3004 ;größer als 255, nein dann Sprung
A 300A  BRK       ;zum Monitor
```

Wir laden also den Akku vor unserer Schleife mit dem ASCII-Wert des Buchstabens, der ausgegeben werden soll. Dann rufen wir 256mal die BSOUT-Routine auf. Beim BNE-Befehl wird solange wieder zum JSR-Befehl verzweigt, bis der Inhalt des Y-Registers überläuft, also wieder null wird. Diese Abfrage ist möglich, da ja der INY-Befehl vor dem bedingten Sprungbefehl steht und der INY-Befehl unter anderem auch das Zeroflag beeinflußt.

Will man mittels der BSOUT-Routine 1000 Zeichen ausgeben, so gestaltet sich auch dies einfacher als mit dem direkten Schreiben in den Speicher. Wir brauchen nämlich nur 1000mal hintereinander die BSOUT-Routine aufrufen, also eine verschachtelte Schleife konstruieren. Allerdings hat die BSOUT-Routine einen Haken. Wird nämlich in der rechten unteren Ecke ein Buchstabe ausgegeben, so wird der Bildschirm nach oben gerollt. Um dieses Manko zu umgehen, geben wir nur 99 Zeichen aus. Es bietet sich dann die folgende Verschachtelung an:

```
9 * 111 = 999
```

Die äußere Schleife läuft also von 0 bis 8 und die innere Schleife von 0 bis 110 (die 0 nicht vergessen!). Dadurch wird die BSOUT-Routine exakt 999mal aufgerufen. Da unser Akkuinhalt innerhalb der beiden Schleifen nicht verändert wird, brauchen wir den LDA-Befehl nicht innerhalb einer der beiden Schleifen schreiben, sondern beginnen das Programm gleich mit ihm. Nehmen wir der Einfachheit halber wieder das 'A':

```
LDA #$41 ;Akku mit ASCII-Kode für 'A' laden
```

Nun müssen die beiden Indexregister initialisiert werden. Nehmen wir das X-Register für die äußere Schleife und das Y-Register für die innere Schleife.

Nun gibt es zwei Möglichkeiten, die Schleifen zu konstruieren. Erstens: Wir lassen die äußere Schleife von 0 bis 8 laufen und überprüfen, ob der Wert des X-Registers schon 9 ist. Zweitens : Wir lassen die äußere Schleife von 9 bis 1 laufen und überprüfen, ob der Inhalt des X-Registers schon gleich 0 ist. Diese beiden Möglichkeiten existieren natürlich auch für die innere Schleife, hier allerdings mit den beiden Möglichkeiten, die Schleife von 0 bis inclusive 110 oder von 111 bis inclusive 1 laufen zu lassen. Da bei der zweiten Lösung zwei Vergleichsbefehle entfallen, entscheiden wir uns natürlich für diese Möglichkeit. Rechnen Sie doch einmal nach, insgesamt sparen wir dadurch über 999 Vergleichsbefehle!

Wir laden unsere beiden Indexregister also mit folgenden Werten:

```
LDX #$09       ;XR für äußere Schleife mit 9 laden
LOOP1 LDY #$6F ;YR für innere Schleife mit 111 laden
```

Nun müssen wir innerhalb der inneren Schleife die BSOUT-Routine aufrufen:

```
LOOP2 JSR $FFD2 ;Zeichen im Akku ('A') ausgeben
```

Nun vermindern wir den Zähler für die innere Schleife (YR) um eins und testen, ob der Inhalt des Y-Registers noch ungleich null ist. Ist dies der Fall, springen wir wieder zum JSR-Befehl:

```
DEY       ;Indexzähler für äußere Schleife dekrementieren
BNE LOOP2 ;Schon 0, Nein, dann zum JSR-Befehl
```

Dieselbe Prozedur müssen wir jetzt noch für die äußere Schleife durchführen und dann wieder zum Monitor springen. Vollständig sieht unsere Programm dann so aus:

```
A 3000  LDA #$93  ;ASCII-Kode für SHIFT+CLR/HOME
A 3002  JSR $FFD2 ;ausgeben
A 3005  LDA #$41  ;ASCII-Kode für 'A'
A 3007  LDX #$09  ;XR mit Wert für äußere Schleife
A 3009  LDY #$6F  ;YR mit Wert für innere Schleife
A 300B  JSR $FFD2 ;Akku ('A') ausgeben
A 300E  DEY       ;innere Schleife erniedrigen
A 300F  BNE $300B ;schon Null? Nein, dann zum JSR-Befehl
A 3011  DEX       ;äußere Schleife erniedrigen
A 3012  BNE $3009 ;schon Null? Nein, dann innere Schleife
A 3014  BRK       ;Fertig, zum Monitor
```

Wie Sie sehen, wird als erstes noch der ASCII-Wert 147 mittels der BSOUT-Routine ausgegeben. Dieser ASCII-Wert entspricht der Tastenkombination SHIFT+CLR/HOME. Es wird also der Bildschirm gelöscht.

Zum Starten geben Sie bitte 'G 3000' ein!

Wenn Sie nicht recht glauben wollen, daß tatsächlich der ganze Bildschirm bis auf die untere rechte Ecke gefüllt wird, geben Sie bitte die folgende, kleine Änderung ein:

```
A 3014  JMP $3014 ;Endlosschleife
```

Der Prozessor springt dann unendlich lange zur Speicherstelle $3014. Um diese Endlosschleife zu verlassen, drücken Sie bitte die RUN/STOP-Taste und gleichzeitig den RESET-Knopf. Sie landen dann wieder im Maschinensprachemonitor.

3.2 Ausgeben eines Textes

Im vorhergehenden Kapitel haben Sie kennengelernt, wie Sie einzelne Zeichen, oder mehrmals hintereinander ein Zeichen ausgeben konnten. Nun kommt es natürlich nicht sehr häufig vor, daß man nur ein Zeichen ausgeben will. Meist handelt es sich um Texte oder andere Zeichenfolgen, egal welcher Art. Aus diesem Grund wollen wir jetzt besprechen, wie man Texte auf den Bildschirm ausgibt. Erinnern wir uns zuerst, wie man ein Zeichen ausgeben konnte. Erstens bestand die Möglichkeit, es direkt in den Bildschirmspeicher zu schreiben, zweitens es über die BSOUT-Routine des Betriebssystems auszugeben. Logischerweise ist die Ausgabe mittels BSOUT das elegantere Verfahren. Da wir aber auch die verschiedenen Adressierungsarten und Programmiertechniken besprechen wollen, besprechen wir beide Möglichkeiten.

Um einen Text ausgeben zu können, müssen wir uns vorher einige Gedanken hierzu machen. Der Text soll vorzugsweise hintereinander im Speicher abgelegt werden, also etwa so, wie in einer Tabelle. Um nun den Text ausgeben zu können, müssen wir auf jeden Fall die Anfangsadresse der Tabelle kennen. Für die Endadresse gibt es nun mehrere Möglichkeiten:

Erstens müssen wir wissen, wie lang unser Text ist. Ist dies bekannt, so können wir unseren Indexzähler auf eine Überschreitung des Tabellenendes überprüfen. Diese Methode hat allerdings einen entscheidenden Nachteil:

Stellen Sie sich vor, Ihr Programm soll auf einen Tastendruck reagieren, beispielsweise bei den Ziffern von 0 bis 9 jeweils einen anderen Text ausgeben. Um dies zu realisieren, müssen Sie die folgenden Tabellen anlegen:

a) eine Tabelle mit den Texten
b) eine Tabelle mit den Anfangsadressen der Texte
c) eine Tabelle mit der Länge der jeweiligen Texte

Dies ist natürlich etwas umständlich. Einfacher wäre es, wenn wir unseren Text durch ein Endsymbol abschließen würden,

beispielsweise durch ein Byte mit dem Wert 0. Wir würden dann solange die Zeichen ausgeben, bis das Zeichen aus der Tabelle den Kode 0 hat. Wäre diese Marke erreicht, so wäre unser Text beendet.

Bevor wir nun auf diese beiden Möglichkeiten ganauer eingehen, ein kleines Hilfsprogramm. Leider unterstützt der eingebaute Maschinensprachemonitor die Verarbeitung von Texten in keiner Weise. Um nun Texte im Bildschirmkode im Speicher abzulegen, müßten wir für jeden Buchstaben in der Bildschirmkodetabelle nachschauen, welchen Kode dieses Zeichen hat. Diesen Kode müssten wir dann noch in das hexadezimale Format umrechnen und im Speicher ablegen. Diese Prozedur kann bei längeren Texten natürlich sehr ermüdend sein. Daher habe ich ein Programm geschrieben, das Ihnen diese Arbeit abnimmt. Nach dem Programmstart geben Sie einfach den gewünschten Text und die Anfangsadresse dieses Textes ein. Diese Adresse kann sowohl als dezimale als auch als hexadezimale Zahl eingegeben werden. Ist diese Eingabe geschehen, so wandelt das BASIC-Programm den Text um und legt ihn im Speicher ab. Danach gibt das Programm den nächsten freien Speicherplatz, sowie die Länge des Textes aus. Arbeiten Sie mit einer Endmarkierung, müssen Sie diese gegebenenfalls mittels des Monitors an Ihren Text anfügen. Hier nun das BASIC-Programm:

```
100 rem ***********************
110 rem * --- poke-insert --- *
120 rem ***********************
130 :
140 input "text";a$
150 :
160 input "startadresse";ad$
170 if left$(ad$,1)<>"$" then ad=val(ad$) : goto 200
180 ad$=right$(ad$,len(ad$)-1)
190 ad=dec(ad$)
200 :
210 scnclr : print a$
220 :
230 for i=0 to len(a$)-1
```

```
240 :  poke ad+i,peek(3072+i)
250 next i
260 :
270 scnclr
280 :
290 print "naechste freie adresse:"
300 print "dez. ";ad+i;
310 print "hex. ";hex$(ad+1)
320 :
330 end
```

Das Prinzip dieses kleinen Programms ist ebenso einfach, wie wirkungsvoll. Es wird der eingegebene Text mittels des PRINT-Befehls in die linke, obere Ecke geschrieben. Dann wird der Text wieder in Form des Bildschirmkodes aus dem Bildschirmspeicher geschrieben. Wir schreiben den Text also mit BSOUT in den Bildschirmspeicher, um ihn dann in unserem Maschinenprogramm wieder direkt in den Bildschirmspeicher zu schreiben.

Nun aber zur ersten möglichen Ausgabeart. Wir wollten hierbei die Länge des Textes angeben. Vorzugsweise gehen wir folgendermaßen vor:

Als Zeiger innerhalb unseres Textes nehmen wir ein Indexregister, hier das Y-Register. Mittels dieses Registers und der indizierten Adressierung laden wir den auszugebenden Wert aus der Tabelle in den Akkumulator:

```
Position: 0 1 2 3 4 5 6 7 8
-------------------------
Text:     P R O B E T E X T
```

Der LDA-Befehl hierfür sieht so aus:

```
LDA $Tabellenanfang,Y
```

Dann schreiben wir, wieder mittels der indizierten Adressierung, den Wert in den Bildschirmspeicher. Hierzu nehmen wir den STA-Befehl, da er ja des Gegenstück zum LDA-Befehl darstellt.

Die Adresse hinter dem STA-Befehl ist die Adresse im Bildschirmspeicher, an der das erste Zeichen ausgegeben werden soll:

```
STA $Adresse vom Textanfang,Y
```

Dann erhöhen wir das Indexregister (hier YR) und testen, ob schon alle Zeichen ausgegeben worden sind. Hierfür nehmen wir zweckmäßigerweise den CPY-Befehl, der das Y-Register mit einem anderen Wert vergleicht. Als Beispiel wollen wir jetzt den Text "PROBETEXT" ausgeben. Wie Sie sehen, ist dieser Text neun Buchstaben lang, wir müssen also prüfen, ob der Indexzähler schon den Wert neun erreicht hat (die 0 nicht vergessen!).

Bis jetzt sieht unser Programm also so aus:

```
A 3000 LDY #$00     ;Indexregister auf Null
A 3002 LDA $300E,Y  ;Kode aus Tabelle holen
A 3005 STA $0C00,Y  ;Kode in Bildschirmspeicher schreiben
A 3008 INY          ;Indexregister erhöhen
A 3009 CPY #$09     ;schon alle Zeichen ausgegeben?
A 300B BNE $3002    ;Nein, dann weiter ausgeben
A 300D BRK          ;Und wieder zum Monitor
```

Um nun den Text in Form des Bildschirmkodes im Speicher abzulegen, bieten sich Ihnen zwei Möglichkeiten an:
Erstens Sie benutzen das oben stehende Programm oder zweitens Sie geben im Monitor diese beiden Zeilen ein und drücken nach jeder Zeile die RETURN-Taste:

```
>300E 10 12 0F 02 05 14 05 18
>3016 14
```

Wollen Sie das obenstehende Hilfsprogramm nutzen, so starten Sie es bitte und geben auf die Frage 'TEXT?', den Text "PROBETEXT" ein. Danach drücken Sie bitte die RETURN-Taste. Nun fragt das Programm nach der Adresse, ab der die Tabelle abgelegt werden soll. Geben Sie hier bitte '$300E' ein und drücken die RETURN-Taste. Das Hilfsprogramm generiert nun die Tabelle. Danach können Sie den Monitor mit dem Befehl 'MONITOR' starten.

Starten Sie dann das Maschinenprogramm mit dem Befehl:

```
G 3000 (RETURN-Taste)
```

Wie Sie sehen, wird der Text ordnungsgemäß ausgegeben. Ändern wir unser Programm nun so ab, daß es mit einem 0-Byte als Endmarkierung arbeitet. Es sollen also solange Buchstaben ausgegeben werden, bis auf ein Byte getroffen wird, das den Wert null hat. Der Nachteil bei dieser Methode ist allerdings, daß ein Zeichen nicht mehr verwendet werden kann. In diesem Fall ist es das Zeichen mit der Bildschirmkodenummer 0, also der Klammeraffe.

Wie schon bei der ersten Methode, benötigen wir wieder einen Indexzähler sowie einen indizierten Ladebefehl. Bis hierhin ist das Programm also völlig identisch. Bevor wir jedoch das Zeichen mittels des Speicherbefehls STA ausgeben können, müssen wir überprüfen, ob es sich eventuell um ein 0-Byte handelt. Es empfiehlt sich hier, den BEQ-Befehl zu benutzen. War der Wert, der in den Akku geladen wurde nämlich gleich 0, so wird gesprungen; in diesem Fall an das Ende unserer Ausgaberoutine. Nach dem BEQ-Befehl folgt dann der STA-Befehl und der INY-Befehl sowie ein Sprungbefehl. Obwohl es sich eigentlich um einen unbedingten Sprungbefehl handeln müßte, wird hier trotzdem der BNE-Befehl benutzt und zwar aus folgendem Grund:

Eine einwandfreie Funktion der Routine ist sowieso nur dann gewährleistet, wenn der auszugebende Text höchstens 255 Zeichen, exklusive der Endmarkierung, lang ist. Wird diese Länge überschritten, so läuft das Y-Register über und es wird wieder bei 0 begonnen. Da schon aus diesem Fall das Y-Register nie 0 werden darf, kann an dieser Stelle der BNE-Befehl benutzt werden. Hier nun erst einmal das Programm, das Sie am besten mit dem Monitor eingeben sollten:

```
A 3000 LDY #$00    ;Indexregister mit 0 laden
A 3002 LDA $300E,Y ;Kode aus Tabelle holen
A 3005 BEQ $300D   ;Endmarkierung?, Ja dann Sprung
```

```
A 3007 STA $0C00,Y ;Nein!, Zeichen ausgeben
A 300A INY         ;Indexregister erhöhen
A 300B BNE $3002   ;Immer springen
A 300D BRK         ;Zum Monitor
```

Geben Sie nun bitte mittels des Monitors die folgenden zwei Zeilen ein, oder benutzen Sie wieder unser Hilfsprogramm von weiter oben:

```
>300E 10 12 0f 02 05 14 05 18
>3016 14 00
```

Sollten Sie den Text mittels des weiter oben vorgestellten Hilfsprogramms eingegeben haben, so müssen Sie noch die Endmarkierung hinzufügen. Dies kann entweder durch den folgenden POKE-Befehl:

```
POKE DEC("3017"),0 (RETURN-Taste)
```

oder im Maschinensprachemonitor durch die folgende Eingabe bewirkt werden:

```
>3017 00
```

Jetzt ist unser Programm wieder komplett. Sie können es wieder durch den Befehl 'G 3000', gefolgt von der RETURN-Taste innerhalb des Monitors starten. Die Tabelle mit dem Text liegt wieder an der gleichen Stelle und unser Programm ist auch gleich lang geblieben. Unser Programm ist jetzt universeller und etwas sparsamer, da wir nicht jedesmal eine neue Länge angeben brauchen.

Außer dadurch, daß man direkt auf den Bildschirmspeicher zugreift, kann man natürlich Texte auch noch mittels der Betriebssystemroutine BSOUT ausgeben. Auch hier kann man das Ende entweder durch die Angabe der Textlänge oder durch eine Endmarkierung definieren. Da der Maschinensprachemonitor auch für Buchstaben und Texte im ASCII-Format keine Eingabemöglichkeit bietet, wird an dieser Stelle ebenfalls ein Pro-

gramm vorgestellt, mit dem ASCII-Texte in Maschinenprogramme eingebunden werden können:

```
100 rem ************************
110 rem * --- ascii-insert --- *
120 rem ************************
130 :
140 a$=""
150 :
160 :
170 :
180 input "startadresse";ad$
190 if left$(ad$,1)<>"$" then ad=val(ad$)-1 : goto 220
200 ad$=right$(ad$,len(ad$)-1)
210 ad=dec(ad$)-1
220 :
230 for i=1 to len(a$)
240 :   poke ad+i,asc(mid$(a$,i,1))
250 next i
260 :
270 print "naechste freie adresse:"
280 print "dez. ";ad+i;
290 print "hex. ";hex$(ad+i)
300 print "laenge";i
310 :
320 end
```

Der Text der in das Maschinenprogramm eingebunden werden soll, wird diesmal nicht eingegeben, wenn das Programm läuft, sondern innerhalb des Programms der Stringvariablen 'a$' zugewiesen. Dies wurde so realisiert, da der Commodore-ASCII-Kode eine Vielzahl von nicht druckbaren Steuerzeichen enthält, die natürlich mit übernommen werden sollen. Sie können also z.B. den folgenden Ausdruck für 'a$' angeben:

```
140 a$="das maschinensprachebuch"
150 a$=chr$(147)+a$
```

Selbstverständlich wird vom Programm wieder die Länge des Textes und die nächste freie Speicherstelle ausgegeben. Arbeiten Sie mit einer Endmarkierung, so müssen Sie diese mittels des POKE-Befehls an die nächste freie Position schreiben, sofern Sie die Endmarkierung nicht schon innerhalb von 'a$' angegeben haben. Um einen Text durch eine solche Endmarkierung abzuschließen, müßten Sie die folgende Zeile eingeben:

```
POKE Ende+1,,ASC("Endmarkierung") (RETURN-Taste)
```

Hier nun das Programm, das ohne Endmarkierung arbeitet. Der einzige Unterschied zu der Variante mit dem STA-Befehl ist nur der Austausch eben dieses STA-Befehl gegen den Unterprogrammaufruf mit JSR $FFD2. Um den Text einzugeben, gehen Sie bitte wie oben beschrieben vor oder geben Sie einfach innerhalb des Monitors die folgenden zwei Zeilen ein·

```
>300E 10 12 0F 02 05 14 05 18
>3016 14

A 3000 LDY #$00     ;Indexregister auf Null
A 3002 LDA $300E,Y ;Kode aus Tabelle holen
A 3005 JSR $FFD2    ;Kode durch BSOUT ausgeben
A 3008 INY          ;Indexregister erhöhen
A 3009 CPY #$09     ;schon alle Zeichen ausgegeben?
A 300B BNE $3002    ;Nein, dann weiter ausgeben
A 300D BRK          ;Und wieder zum Monitor
```

Auch wenn man mit einem Nullbyte als Endmarkierung arbeitet, ändert sich nicht viel. Es wird nämlich ebenfalls nur der indizierte STA-Befehl durch den Unterprogrammaufruf der Routine BSOUT ersetzt. Vom Speicherplatz her liegt also kein Unterschied zwischen der Ausgabe durch direktes Schreiben in den Bildschirmspeicher und der Ausgabe durch BSOUT vor. Ein wesentlicher Vorteil der Textausgabe über die Routine BSOUT ist allerdings folgender:

Sie kennen sicherlich in BASIC die Befehle CMD und PRINT#. Mittels CMD kann die Bildschirmausgabe auf ein logisches File umgelenkt werden. Dies kann eine Datei auf einer Diskette oder einer Kassette sein.

Den gleichen Vorteil können Sie nun auch bei BSOUT nützen. Mittels spezieller Routinen im Betriebssystem ist es möglich, die Bildschirmausgabe über BSOUT auf ein File umzulenken. Dieser Vorteil existiert beim direkten Schreiben in den Bildschirmspeicher natürlich nicht. Daher sollten Sie Texte und Zeichen möglichst immer über logische Routinen, wie z.B. BSOUT eine darstellt, ausgeben.

Da Texte auch vom Betiebssystem und vom BASIC-Interpreter (siehe weiter unten) ausgegeben werden müssen, existieren natürlich schon mehrere Ausgaberoutinen in den ROM's Ihres Rechners. Von diesen Ausgaberoutinen wollen wir die zwei praktikabelsten besprechen. Beide Routinen geben Texte im ASCII-Format an. Also Ihre Texte können auch Steuerzeichen für die Farben oder den REVERS-Modus beinhalten. Die erste dieser Routinen ist die Routine mit dem Namen 'PRIMM'. Um einen Text mit der PRIMM-Routine auszugeben, müssen Sie folgendermaßen vorgehen:

Als erstes rufen Sie die Routine PRIMM mittels eines JSR-Befehls auf. Direkt hinter dem HIGH-Byte dieser Zieladresse folgt nun der Text im Speicher. Die Routine PRIMM gibt nun solange ein Zeichen nach dem anderen aus, bis sie auf ein Nullbyte trifft. Ist dies der Fall, wird direkt zur Speicherstelle hinter dem Nullbyte zurückgesprungen. Der auszugebende Text wird also einfach in das Programm mit eingebaut. Dazu ein Beispiel:

Wir wollen in unserem Maschinenprogramm die Zeichenfolge 'ABCD' ausgeben. Dazu gehen wir folgendermaßen vor:
Als erstes rufen wir die Routine PRIMM auf, die an der Adresse $FF47 oder 65359 beginnt. Hinter dem HIGH-Byte dieser Adresse folgt nun unmittelbar unsere Zeichenfolge 'ABCD'. Dann folgt ein Nullbyte um die Zeichenfolge abzuschließen.

Hinter diesem Nullbyte folgt nun wieder der ganz normale Programmtext, an dem die Programmausführung fortgesetzt wird, wenn der Text ausgegeben worden ist. Hier das Maschinenprogramm um die Zeichenfolge 'ABCD' auszugeben:

```
A 3000 JSR $FF4F    ;PRIMM-Routine aufrufen
> 3003 41 42 43 44 ;Auszugebende Zeichenfolge
> 3007 00          ;Endmarkierung des Textes
A 3008 NOP         ;folgender Programmtext
A 3009 BRK         ;zum Maschinensprachemonitor
```

Wenn Sie nun das Programm mit 'G 3000' starten, so sehen Sie, daß tatsächlich die Zeichenfolge 'ABCD' ausgegeben wird. Zur besseren Übersicht könnten wir jetzt noch den ASCII-Kode zum Bildschirmlöschen in unseren Text einbauen. Inklusive dieses ASCII-Kodes sähe unser Programm dann so aus:

```
A 3000 JSR $FF4F       ;PRIMM-Routine aufrufen
> 3003 93 41 42 43 44 ;Auszugebende Zeichenfolge
> 3008 00             ;Endmarkierung des Textes
A 3009 NOP            ;folgender Programmtext
A 300A BRK            ;zum Maschinensprachemonitor
```

Wenn Sie das Programm jetzt wieder mit 'G 3000' starten, so wird erst der Bildschirm gelöscht und der Cursor in die linke obere Ecke des Bildschirms gesetzt. Dann wird der Text 'ABCD' ausgegeben, der jetzt natürlich in der linken oberen Ecke des Bildschirms erscheint.

Außer dieser Routine PRIMM, gibt es noch eine Routine um einen String auszugeben. Dieser Routine wird nur die Adresse übergeben, ab der der ASCII-String steht. Da auch dieser String durch ein Nullbyte abgeschlossen werden muß, ist die Routine selbständig in der Lage, die Länge des Textes zu ermitteln und diesen String dann auszugeben. Die Adresse, an der der String beginnt, wird im Akku und im Y-Register übergeben. Der Akku enthält hierbei das LOW-Byte und das Y-Register das HIGH-Byte. Um also einen Text auszugeben, der an der Adresse $1234 beginnt, müßte man den Akkumulator mit dem LOW-Byte, also $34 und das Y-Register mit dem HIGh-Byte, also $12 laden.

Erst danach darf die Routine aufgerufen werden. Nach der Ausgabe des Textes wird wie bei jedem Unterprogrammaufruf zurückgesprungen. Der Text wird in der Programmausführung also nicht automatisch übersprungen!

Diese Routine mit dem Namen TXTOUT beginnt an der Adresse $9088 oder 37003. Wie gesagt, muß die Anfangsadresse des Strings im Akku und Y-Register übergeben und der String durch ein Nullbyte abschlossen werden. Dazu wieder ein Beispiel mit unsere Zeichenkette 'ABCD':

```
A 3000 LDA #$08          ;LOW-Byte Stringanfang
A 3002 LDY #$30          ;HIGH-Byte Stringanfang
A 3004 JSR $9088         ;Aufruf der Routine TXTOUT
A 3007 BRK               ;Zum Monitor
> 3008 93 41 42 43 44 ;Text
> 300D 00               ;Markierung für Textende
```

Diese Routine enthält übrigens wieder den ASCII-Kode 147, durch den ein Löschen des Bildschirms hervorgerufen wird. Aufgerufen wird dieses kleine Demonstrationsprogramm natürlich wieder durch:

```
G 3000 (RETURN-Taste)
```

Hier noch einmal zusammenfassend die beiden Routinen:

PRIMM: Mit dieser Routine ist es möglich, auf unkomplizierte Art und Weise einen Text auszugeben. Dieser Text wird in das Maschinenprogramm integriert und zwar direkt hinter dem Aufruf der Routine PRIMM. Nachdem der Text ausgegeben worden ist, wird die Programmausführung hinter dem Nullbyte, durch das der Text 4abgeschlossen wird, fortgesetzt.

Adresse: $FF4F oder 65359

TXTOUT: Auch mittels der Routine TXTOUT ist es möglich, Texte auszugeben. Auch der Text, der durch diese

Routine ausgegeben werden soll, muß durch ein Nullbyte abgeschlossen werden. Bevor diese Routine aufgerufen wird, muß im Gegensatz zur PRIMM-Routine der Akku und das Y-Register mit der Adresse des Stringanfangs geladen werden. Der Akku enthält hierbei das LOW- und das Y-Register das HIGH-Byte.

Adresse: $9088 oder 37003

3.3 Das Einbinden von Maschinenprogrammen in BASIC-Programme

Die Maschinensprache wird meist dort eingesetzt, wo es auf eine schnelle Programmausführung ankommt, oder ein gestelltes Problem in BASIC nicht zu realisieren ist. Der Einsatz von Maschinensprache bedeutet aber nicht, daß das ganze Programm nur in Maschinensprache geschrieben sein darf. Eine weit verbreitete Programmiertechnik ist daher auch, zeitaufwendige Programmteile in Maschinensprache zu schreiben, während der größte Teil des Programms noch in BASIC programmiert wird. Um aber Maschinenprogramme in BASIC-Programme einzubinden, benötigen wir eine Möglichkeit, Maschinenprogramme von BASIC-Programmen aus aufzurufen und Parameter an sie zu übergeben oder Ergebnisse zu erhalten. Dies für Sie zu ermöglichen ist der Sinn dieses Kapitels.

Wie schon angeführt, dient der BASIC-Befehl 'SYS' zum Aufrufen von Maschinenprogrammen von BASIC aus. Schauen wir uns dazu die SYS-Routine des BASIC-Interpreters etwas genauer an:

```
. A7B5  JSR $9DE1   ;Adresse nach $14/$15 holen
. A7B8  LDA #$A7    ;Rücksprungadresse-1 HIGH-Byte
. A7BA  PHA         ;Auf Stack legen
. A7BB  LDA #$CE    ;Rücksrungadresse-1 LOW-Byte
. A7BD  PHA         ;Auf Stack legen
. A7BE  LDA $07F5   ;Pseudostatusregister in Akku
. A7C1  PHA         ;Pseudostatusregister auf Stack
```

```
. A7C2  LDA $07F2   ;Pseudoakku in Akku
. A7C5  LDX $07F3   ;Pseudoxr in XR
. A7C8  LDY $07F4   ;Pseudoyr in YR
. A7CB  PLP         ;Pseudostatus von Stack in Status
. A7CC  JMP ($0014) ;Indirekt zur Adresse in $14/$15
. A7CF  PHP         ;Status auf Stack legen
. A7D0  STA $07F2   ;Akku in Pseudoakku
. A7D3  STX $07F3   ;XR in Pseudoxr
. A7D6  STY $07F4   ;YR in Pseudoyr
. A7D9  PLA         ;Status vom Stack holen
. A7DA  STA $07F5   ;Als Pseudostatus setzen
. A7DD  RTS         ;Wieder zum BASIC-Programm
```

Beschreibung:

Wie aus dem Disassemblerlisting der SYS-Routine ersichtlich ist, existieren vier sogenannte Pseudoregister. Diese Pseudoregister sind nichts anderes als normale Speicherstellen, deren Inhalte beim Aufruf eines Maschinenprogramms mit der SYS-Routine in die Prozessorregister geschrieben werden. Der umgekerte Prozess wird auch vollzogen, wenn das aufgerufene Maschinenprogramm beendet ist, Dann werden nämlich die Prozessorinhalte in diesen Pseudoregistern abgelegt.

Bevor dies allerdings geschieht, wird erst einmal dafür gesorgt, daß überhaupt in die SYS-Routine gesprungen wird, wenn im aufzurufenden Programm auf ein RTS gestoßen wird. Dies wird realisiert, indem die Adresse minus eins auf den Stack gelegt wird, zu der nach Beendigung des eigentlichen Programms gesprungen werden soll. Die Adresse wird minus eins auf den Stack gelegt, da der Prozessor bei einem RTS die Adresse, die er vom Stack holt, noch um eins erhöht. Im Endeffekt wird in der SYS-Routine bei einem RTS-Befehl also direkt hinter den indirekten JMP-Befehl verzweigt. Diese Prozedur simuliert in etwa einen indirekten JSR-Befehl, der ja leider im Befehlssatz des 7501-Prozessors nicht existiert.

Durch die oben schon angesprochenen Pseudoregister bietet sich uns nun schon eine Möglichkeit, Daten an ein Maschinenpro-

gramm zu übergeben und Daten von einem Maschinenprogramm zu empfangen. Versuchen wir doch einmal mittels der SYS- und der BSOUT-Routine ein Zeichen auf den Bildschim auszugeben:

Wie wir bereits wissen, muß der ASCII-Kode des Zeichens, das ausgegeben werden soll, im Akku an die Routine BSOUT übergeben werden. Wir müssen also nur den ASCII-Kode unseres Zeichens im Pseudoakku ablegen, der dann innerhalb der SYS-Routine in den richtigen Akku des Prozessors kopiert wird. Schauen wir uns noch einmal an, an welchen Adressen die Pseudoregister liegen:

```
Pseudoakkumulator:    $07F2 / 2034
Pseudox-register:     $07F3 / 2035
Pseudoy-register:     $07F4 / 2036
Pseudostatusregister: $07F5 / 2037
```

Um nun beispielsweise ein 'A' auszugeben, müßten wir vor dem SYS-Befehl mittels einen POKE-Befehls den ASCII-Kode 65 in den Pseudoakkumulator schreiben. Dazu geben Sie bitte den folgenden POKE-Befehl ein:

```
POKE 2034,65 : REM KODE FUER 'A' IN PSEUDOAKKU
```

Nachdem dies geschehen ist, können wir nun mittels des SYS-Befehl die Betriebssystemroutine BSOUT aufrufen. Machen wir dies, so geschieht folgendes:

1. Es werden alle Pseudoregister in die Prozessoregister kopiert. Damit enthält nun auch der richtige Akkumulator den ASCII-Kode für das Zeichen 'A'.

2. Dann wird die Routine BSOUT aufgerufen und das Zeichen ausgegeben, dessen ASCII-Kode im Akkumulator stand. In diesem Fall wird ein 'A' ausgegeben.

3. Es wird wieder in die SYS-Routine gesprungen und alle Prozessorregister in den Pseudoregistern abgespeichert.

Probieren Sie die oben beschriebene Prozedur jetzt durch die Eingabe von:

```
SYS DEC("FFD2")
```

und einem Druck auf die RETURN-Taste aus!

Wie erwartet, erscheint ein 'A' auf dem Bildschim. Noch eine Anmerkung zum SYS-Befehl:

Innerhalb des Maschinensprachemonitors haben wir unsere Maschinenprogramme immer durch einen BRK-Befehl abgeschlossen. Dieser Befehl bewirkte einen Sprung zum Maschinensprachemonitor. Da dies aber nicht unser Ziel ist, sondern wir vielmehr wieder dahin zurückspringen möchten, von wo der Sprung ausgegangen ist, müssen wir unsere Routinen jetzt durch einen RTS-Befehl abschließen. Vereinfacht kann man sich den SYS-Befehl als eine Art Unterprogrammaufruf vorstellen, da auch Unterprogramme immer mit einem RTS-Befehl abgeschlossen werden müssen.

Leider reichen drei Parameter manchmal nicht aus, und außerdem ist das Hantieren mit den POKE- und PEEK-Kommandos auch nicht ideal. Wenn wir uns die BASIC-Kommandos anschauen, erkennt man, daß fast alle Befehle auch Parameter benötigen, die meist durch ein Komma untereinander getrennt sind. Es muß folglich auch noch eine andere Möglichkeit geben, Parameter an ein Maschinenprogramm zu übergeben. Um diesem Geheimniß auf die Spur zu kommen, analysieren wir die Routine, die bei einem POKE-Befehl aufgerufen wird. Diese Routine sieht nun so aus:

```
. 9E12  JSR $9DD2   ;Parameter holen
. 9E15  TXA         ;2. Parameter in Akku
. 9E16  LDY #$00    ;Index auf Null
. 9E18  STA ($14),Y ;2. Parameter an $14/$15 speichern
. 9E1A  RTS         ;Fertig!
```

Was verbirgt sich nun hinter dieser ominösen Routine $9DD2, die ja scheinbar zum Holen der Parameter dient. Schnüffelt man

noch ein bischen im ROM herum, so findet man heraus, daß innerhalb dieser Routine eigentlich nur vier andere Routinen aufgerufen werden. Diese vier Routinen lauten:

```
FRMEVL: $9314 / 37652
GETADR: $9DE4 / 40420
CHKCOM: $9491 / 38033
GETBYT: $9D84 / 40324
```

Nun zur Beschreibung der einzelnen Funktionen:
Wahrscheinlich haben Sie auch schon festgestellt, daß Ihr Computer in der Lage ist (fast) beliebig verschachtelte Ausdrücke zu berechnen. Er kann sowohl einen einfachen:

```
POKE 32768,5
```

, als auch einen komplizierten Ausdruck berechnen:

```
POKE INT(SQR(5^SIN(2)))*10,5
```

Um solche Ausdrücke auszuwerten, existiert die Routine FRMEVL. Sie dient also zur Formelauswertung beliebiger Formeln.

Die Routine GETADR testet nun, ob das Ergebnis, das innerhalb der Formelauswertungsroutine berechnet worden ist, innerhalb des Bereichs von 0 bis 65535 liegt. Ist dies nicht der Fall, so wird die Fehlermeldung ?ILLEGAL QUANTITY ERROR ausgegeben. Liegt das Ergebniss der Formel allerdings im Bereich zwischen 0 und 65535, läßt es sich also durch zwei Bytes ausdrücken, so wird das LOW-Byte dieser Zahl in der Adresse $14, und das HIGH-Byte dieser Zahl in der Adresse $15 abgelegt. Da es sich um zwei hintereinanderliegende Zeropageadressen handelt, ist es möglich, diese beiden Speicherstellen direkt für die indirekt indizierte Adressierung zu benutzen, wie dies auch beim POKE-Befehl geschieht. Nach der GETADR-Routine wird nun die Routine mit dem Namen CHKCOM aufgerufen. CHKCOM steht für CHECK KOMMA und dient zum Testen auf ein Komma. Steht an der aktuellen Position im BASIC-Programm kein Komma, so wird die Fehlermeldung ?SYNTAX

ERROR ausgegeben. Andernfalls wird der BASIC-Programmzeiger um eine Position nach vorne verschoben und keine Fehlermeldung ausgegeben. War bis zu dieser Stelle alles korrekt, so wird die GETBYT-Routine aufgerufen. Innerhalb dieser Routine wird wieder die FRMEVL-Routine aufgerufen und dann getestet, ob das Ergebniss im Bereich vom 0 bis 255 liegt, also durch 8 Bit dargestellt werden kann. Ist diese Bedingung nicht erfüllt, so wird wieder mit einem ?ILLEGAL QUANTITY ERROR ausgestiegen und der BASIC-Programmablauf abgebrochen. Lag das Ergebniss allerdings im Bereich von 0 bis 255, so wird es im X-Register abgelegt.

Durch den Aufruf dieser vier Routinen, die natürlich auch einzeln aufgerufen werden können, steht nun die Adresse, auf die zugegriffen werden soll, in den Speicherstellen $14/$15 und der Wert, der in diese Speicherstelle geschrieben werden soll, im X-Register.

Schauen wir uns wieder unsere POKE-Routine an: Nachdem diese vier Routinen aufgerufen worden sind, wird mittels Befehls TXA der Inhalt des X-Registers in den Akkumulator transportiert. Der Akku enthält also nun den Wert, der in die angegebene Speicherstelle geschrieben werden soll. Danach wird das Y-Register für die indirekt indizierte Adressierung vorbereitet und der Inhalt des Akkumulators in die angegebene Speicherstelle geschrieben. Nachdem dies ausgeführt worden ist, ist der POKE-Befehl komplett abgearbeitet.

Wollen wir nun unseren SYS-Befehl zur Übergabe von mehreren Parametern benutzen, so müssen wir ananlog vorgehen. Schreiben wir hierzu eine Routine, die mittels des SYS-Befehls aufgerufen werden soll und den Cursor an den angegebenen Koordinaten positioniert. Die Parameterübergabe soll also so aussehen:

```
SYS Routinenadresse,X-Position,Y-Position
```

Als Voraussetzung sei noch gesagt, daß wir eine Routine des Betriebssystems benutzen wollen, die für Cursorpositionierung zuständig ist. Diese Routine hat den Namen PLOT und setzt den Cursor an die angegebenen Koordinaten, sofern das Carryflag

gelöscht ist. Die X-Koordinate muß hierbei im Y-Register und die Y-Koordinate im X-Register übergeben werden.

Nachdem unsere Routine über den SYS-Befehl aufgerufen worden ist, steht der BASIC-Programmzeiger auf dem Komma nach der Adresse unserer Routine. Da wir aber nicht unbedingt davon ausgehen können, daß dort tatsächlich ein Komma steht, müssen wir dies testen. Hierzu benutzen wir, wie auch der POKE-Befehl, die Routine CHKKOM. Ist kein Komma vorhanden, so wird eine BASIC-Fehlermeldung ausgegeben und unser Maschinenprogramm abgebrochen. Existiert allerdings ein Komma, so steht der BASIC-Programmzeiger auf dem ersten Wert hinter dem Komma. Diesen Wert müssen wir nun einlesen. Da die X-Koordinate nur im Bereich zwischen 0 und 39 liegen darf, eignet sich hierzu besonders die Routine GETBYT. In dieser Routine wird unter anderem auch die Routine zur Formelauswertung aufgerufen, so daß wir uns darum nicht mehr kümmern müssen. Der eingelesene 8-Bit-Wert wird von der Routine GETBYT im X-Register zurückgegeben. War kein Wert vorhanden, so wird der Wert 0 im X-Register übergeben. War dagegen ein ungültiger Wert angegeben worden oder es ist ein syntaktischer Fehler aufgetreten, so wird auch hier mit einer BASIC-Fehlermeldung abgebrochen.

Da die Betriebssystemroutine zum Cursorsetzen die X-Koordinate im Y-Register verlangt, müssen wir den Inhalt des X-Registers jetzt noch in das Y-Register übertragen. Leider wird durch die Routinen CHKKOM und GETBYT auch der Inhalt des Y-Registers verändert, so daß wir den Inhalt des X-Registers erst einmal auf den Stack legen und dann später wieder herunternehmen und in das Y-Register transferieren. Um das X-Register auf den Stack zu legen und später wieder herunterzuholen, wenden wir folgende Problemlösung an:

```
TXA ;X-Register in den Akku kopieren
PHA ;Akku auf den Stack legen
 .
 . ;anderer Programmtext CHKKOM etc.
 .
PLA ;Akku vom Stack holen
TAY ;Akku in Y-Register übertragen
```

Im Endeffekt steht nun unsere X-Koordinate sowohl im X-Register, als auch im Akku und im Y-Register. Nachdem wird nun die X-Koordinate eingelesen und verarbeitet haben, müssen wir nur noch die Y-Koordinate eingeben. Dies geschieht natürlich nach dem selben Prinzip. Also erst CHKKOM und dann GETBYT. Da die Y-Koordinate sowieso im X-Register erwartet wird, brauchen wir auch nichts mehr zu kopieren. Zuletzt müssen wir nur noch das Carryflag löschen (CLC) und die Betriebssystemroutine zum Cursorsetzen (PLOT) aufrufen. Komplett sieht unsere kleine BASIC-Erweiterung dann so aus:

```
A 03F7  JSR $9491 ;CHKKOM
A 03FA  JSR $9D84 ;GETBYT
A 03FD  TXA       ;XR in Akku übertragen
A 03FE  PHA       ;Akku bzw, XR auf Stack legen
A 03FF  JSR $9491 ;CHKKOM
A 0402  JSR $9D84 ;GETBYT
A 0405  PLA       ;XR vom Stack in Akku holen
A 0406  TAY       ;Akku in YR übertragen
A 0407  CLC       ;Carry löschen für Cursorsetzen
A 0408  JSR $FFF0 ;PLOT
A 040B  RTS       ;Zum BASIC-Interpreter
```

Wie oben schon gesagt, wird die Routine mittels des SYS-Befehls aufgerufen. Wie Ihnen sicherlich schon aufgefallen ist, liegt unsere Maschinenroutine diesmal nicht im Bereich ab $3000 sondern im Bereich ab $03F7. Dieser Bereich hat die Funktion einen RS-232C Puffers. Da diese Schnittstelle aber sowieso nur im Plus/4 eingebaut ist und wahrscheinlich auch nicht oft genutzt wird, legen wir unsere Maschinenroutine in diesem Bereich ab. Dies ist besonders nützlich, weil unsere Maschinenroutine hier keinen Platz des BASIC-Speichers benutzt. Der zweite

Vorteil dieses Speicherbereiches ist, daß man hier nicht den BASIC-Speicher einengen muß, was immer dann notwendig wird, wenn man BASIC-Programme zusammen mit Maschinenprogrammen betreiben will, die einen Teil des BASIC-Speichers belegen. Wenn möglich, sollten Sie daher Ihre Maschinenprogramme im Bereich von $03F7 bis $0472 ablegen. Oder Sie nutzen die anderen (gegebenenfalls) freien Speicherbereiche:

```
$0332 - $03F2 /  818 -  1010: Kassettenpuffer
$03F7 - $0472 / 1015 -  1138: RS-232C Puffer
$0567 - $05E6 / 1383 -  1510: Texte Funktionstasten
$065E - $06EB / 1630 -  1771: RAM für Sprachsynthesizer
$07B0 - $07CC / 1968 -  1996: Kassetten-Arbeitsbereich
```

Diese Speicherbereiche können Sie natürlich nur nutzen, wenn Sie auf die Funktionen verzichten, die eigentlich diesen Speicherbereichen zugeordnet sind. So müssen Sie z.B. folgende Speicherstellen ändern, um den Funktionstastenspeicher zu benutzen:

```
FOR I=0 TO 7:POKE 1375+I,0:NEXT I (RETURN-Taste)
```

Nun kennen wir bereits Möglichkeiten, Maschinenprogramme von BASIC aus aufzurufen und Parameter an sie zu übergeben, doch wie ist es möglich, Werte von Maschinenspracheprogrammen aus an BASIC-Programme zu übergeben. Die erste Möglichkeit die wir schon kennengelernt haben, war die Übergabe von Werten an BASIC-Programme über den sogenannten Pseudoregistersatz. Außer dieser Möglichkeit gibt es aber noch die sogenannte USR(X)-Funktion. Diese Funktion ist normalerweise nicht belegt und bewirkt einen ?ILLEGAL QUANTITY ERROR hervor wenn man sie aufruft. Dies liegt daran, daß der Vektor auf die Routine zeigt, die die Fehlermeldung ?ILLEGAL QUANTITY ERROR ausgibt. Ändert man diesen Vektor (siehe unten) auf eine eigene Routine, so ist es möglich, mittels der USR(X)-Funktion Werte an ein BASIC-Programm zu übergeben. Die USR(X)-Funktion hat übrigens die gleiche Syntax wie auch alle anderen Funktionen. Es reicht also nicht die bloße Eingabe

des Funktionsnamens, sondern es muß auch noch angegeben werden, was mit dem Ergebnis dieser Funktion geschehen soll. Dazu ein Beispiel:

Standhafte Anwendungen wären z.B.:

```
PRINT USR(X)
```

oder

```
A=USR(X)
```

Vorausgesetzt natürlich, der Vektor ist auf eine korrekte Routine umgebogen worden. Was man unter einem Vektor zu verstehen hat, werden wir gleich noch erklären, daher hier nur eine kurze Beschreibung:

Wird eine Routine über einen indirekten JMP-Befehl aufgerufen, so bezeichnet man die beiden Speicherstellen, in denen das Sprungziel steht, als Vektor. Für die USR(X)-Funktion wäre der Vektor die beiden Speicherstellen 1281 und 1282. Um nun herauszubekommen, wo das Ergebnis unserer Funktion stehen muß, schauen wir uns einmal die POS(X)-Funktion an. Die POS(X)-Funktion dient dazu, die Cursorspalte zu ermitteln. Der Ausdruck X ist hierbei ein Scheinparameter und hat keine Funktion. Hier nun das disassemblierte Listing der POS(X)-Funktion:

```
. 9A7D  SEC       ;Carry als Flag setzen
. 9A7E  JSR $FFF0 ;Plot, X-Koord. im Y-Reg.
. 9A81  LDA #$00  ;HIGH-Byte Integerzahl = 0
. 9A83  JMP $9471 ;Integer in Y/A nach FAC, RTS
```

Der Aufruf der Betriebssystemroutine PLOT mit gesetztem Carryflag, liefert die X-Koordinate des Cursors im Y-Register und die Y-Koordinate des Cursors im X-Register. Dann wird der Akku auf Null gesetzt und die Routine $9471 aufgerufen. Wie Sie dem disassemblierten Listing entnehmen können, dient diese Routine zum Wandeln einer 16-Bit Integerzahl in ein anderes Zahlenformat, das im FAC abgelegt werden soll. 'FAC' ist eine Abkürzung und bedeutet zu Deutsch 'Fließkommaakku-

mulator'. Unter einem solchen Fließkommaakkumulator hat man sich mehrere Speicherstellen vorzustellen, in denen Fließkommazahlen gespeichert werden. Fließkommazahlen sind Zahlen mit Nachkommastellen oder Exponentialdarstellung. Mittels der Fließkommadarstellung ist es möglich, mit einer höheren Genauigkeit zu rechnen, als dies mit den Integerzahlen möglich ist. Wie nun Fließkomma- oder Gleitkomma (gk) zahlen abgelegt werden, soll uns im Rahmen dieses Buches nicht interessieren. Viel wichtiger für uns ist die Tatsache, daß das Ergebniss einer Funktion wie z.B. der POS(X)- oder USR(X)-Funktion in diesem Fließkommaakkumulator verlangt wird und daß Routinen zum Wandeln einer Integerzahl in eine solche Fließkommazahl bestehen. Wollen wir also Funktionen programmieren, die eine Integerzahl als Ergebniss haben, so müssen wir diese Integerzahl in eine Fließkommazahl wandeln und über einen RTS-Befehl wieder in den BASIC-Interpreter (siehe unten) springen. Der Interpreter übernimmt dann alles weitere für uns. Welche Routinen alles zum Wandeln einer Integerzahl in eine Fließkommazahl existieren, finden Sie weiter unten. An dieser Stelle soll uns nur die Routine interessieren, die eine positive 16-Bit-Integerzahl in eine Fließkommazahl wandelt. Diese Routine hat folgendes Format und die folgende Aufrufadresse:

```
Integerzahl: YR/Akku (LOW/HIGH)

Adresse: $9471 / 38001
```

Wollen wir nun analog zur POS(X)-Funktion, mittels der USR(X)-Funktion eine Routine schreiben, die die Cursorzeile als Ergebniss liefert, so müssen wir wie bei der POS(X)-Routine vorgehen. Zuerst müssen wir also sowohl die X- als auch die Y-Koordinate des Cursors ermitteln. Dann müssen wir die Y-Koordinate des Cursors in das Y-Register übertragen (X-Register ins Y-Register). Ist dies geschehen, müssen wir nur noch den Akku mit Null laden und die Wandelroutine an $9471 mit einem JSR-Befehl aufrufen. Danach sollte unser Programm mit einem RTS wieder in den BASIC-Interpreter springen. Schreiben wir dazu jetzt, wie gerade beschrieben, das Maschinenprogramm, das wir am besten hinter unserer Cursorsetzroutine ablegen, um beide Routinen benutzen zu können:

```
A 040C  SEC        ;Carry als Flag setzen
A 040D  JSR $FFF0  ;Plot, Y-Koord. im X-Reg.
A 0410  TXA        ;X-Register in
A 0411  TAY        ;Y-Register übertragen
A 0412  LDA #$00   ;HIGH-Byte Integerzahl = 0
A 0414  JSR $9471  ;Integer in Y/A nach FAC
A 0417  RTS        ;Zum Interpreter
```

Gibt man jetzt voller Erwartung die Zeile:

```
PRINT USR(0) (RETURN-Taste)
```

ein, so wird man leider enttäuscht. Der Computer meldet sich immer noch mit einem ?ILLEGAL QUANTITY ERROR. Dies liegt daran, daß wir bis jetzt noch nicht den Vektor geändert haben. Wie oben schon gesagt, belegt dieser Vektor die beiden Speicherplätze 1281 und 1282. Schreiben wir nun das LOW-Byte der Startadresse unserer Routine in die Speicherstelle 1281 und das HIGH-Byte in die Speicherstelle 1282:

```
POKE 1281,DEC("0C") (RETURN-Taste)
POKE 1282,DEC("04") (RETURN-Taste)
```

Wenn Sie dies erledigt haben, können Sie ohne Bedenken noch einmal die Zeile:

```
PRINT USR(X)
```

gefolgt von der RETURN-Taste eingeben. Je nachdem, wo sich der Cursor gerade befand, wird eine Zahl zwischen 0 und 24 ausgegeben.

3.4 Der BASIC-Lader

Bis jetzt haben wir unsere Maschinenprogramme immer mit dem eingebauten Monitor eingegeben. Diese Möglichkeit war sehr übersichtlich und Fehlerquellen waren so gut wie ausgeschlossen. Will man jedoch Maschinenprogramme in BASIC-Programme einbinden, kann man den Anwender wohl kaum auffordern,

mittels des Maschinensprachemonitors die Maschinenroutinen einzugeben. Daher muß man nach einer anderen Lösung suchen, um Maschinenprogramme zu integrieren. Erinnern wir uns, wie wir die ersten Maschinensprachebefehle eingegeben haben:

Wir haben den Befehlskode und die Parameter mittels des POKE-Befehls in den Speicher geschrieben. Theoretisch könnten wir nun auch bei allen anderen Maschinenprogrammen so vorgehen. Es geht allerdings auch noch einfacher:

Wir legen unser Maschinenprogramm als Dezimalzahlen in sogenannten DATA-Zeilen ab. Zur Verarbeitung solcher DATA-Zeilen existieren die Befehle RESTORE, READ und DATA. Der RESTORE-Befehl setzt den DATA-Zeilen-Zeiger auf eine bstimmte DATA-Zeile, der READ-Befehl dient zum Lesen der Daten und der DATA-Befehl zum Ablegen der Daten. Eine typische DATA-Zeile könnte nun so aussehen:

```
1000 DATA 000,001,002,003,004,005
```

Um Maschinenprogramme in diese DATA-Zeilen zu übernehmen, bliebe uns theoretisch nichts anderes übrig, als mittels einer Schleife und der PEEK(X)-Funktion alle Speicherinhalte ausgeben zu lassen und diese dann in die DATA-Zeilen zu übernehmen. Doch zum Glück haben wir jemanden, der uns diese Arbeit abnimmt. Mit einigen Tricks ist es nämlich möglich, durch ein neues BASIC-Programm dieses BASIC-Programm zu ergänzen. Dies klingt vielleicht komisch, ist aber doch sehr effektiv und, wenn man den Trick kennt, recht einfach zu verstehen. Wie oben schon angeschnitten, hier das Programm mit dem es möglich ist, Speicherbereiche in DATA-Zeilen zu übernehemen. Speichern Sie den sogenannten DATA-Wandler aber auf jeden Fall vor dem Programmstart ab, da er sich sonst eventuell selber löscht. Außerdem sollten Sie noch darauf achten, das Sie die DATA's Ihres Maschinenprogramms nicht mitten in den DATA-Wandler schreiben. Um dies zu umgehen, dürfen Sie die DATA's frühstens ab Zeile 32 anlegen. Hier nun das BASIC-Programm zum DATA-Wandler:

```
0 rem *************************
1 rem * --- data-generator --- *
2 rem *************************
3 :
4 input "startadresse";sa
5 input "endadresse  ";ea
6 :
7 input "erste zeilennummer";zn
8 input "zeilenabstand      ";za
9 :
10 scnclr : print zn;"data ";
11 :
12 for sa=sa to sa+8
13 :   a$=str$(peek(sa))
14 :   a$=right$(a$,len(a$)-1)
15 :   a$=right$("00"+a$,3)
16 :   print a$;",";
17 next sa
18 :
19 print chr$(20)
20 :
21 zn=zn+za : if sa<ea then 24
22 scnclr : print "delete -31"
23 goto 29
24 :
25 print "zn=";zn;":za=";za;":sa=";sa
26 print : print
27 print "ea=";ea;":goto 10"
28 :
29 poke 1319,19 : poke 1320,13
30 poke 1321,13 : poke 1322,13
31 poke 239,4 : end
```

Bevor ich nun darauf eingehen will, wie man in DATA-Zeilen gespeicherte Maschinenprogramme wieder einliest, hier erst einmal die Beschreibung der Funktionsweise des DATA-Wandlers:

Das wichtigste Prinzip des DATA-Wandlers ist der sogenannte "programmierte Direktmodus". Obwohl dieser Programmiertrick eigentlich nichts mit der Programmierung in Maschinensprache

zu tun hat, zeigt sich doch die Chance auch von BASIC aus, die Möglichkeiten des Betriebssystems zu nutzen.

Wie Sie vielleicht schon während des Programmablaufs beobachtet haben, wird jedesmal in der obersten Zeile des Bildschirms eine komplette DATA-Zeile ausgegeben. Ein paar Zeilen darunter werden einigen Variablen Werte zugewiesen und dann mit einem GOTO-Befehl mitten ins BASIC-Programm gesprungen. Würden Sie nun im Direktmodus mit dem Cursor auf die erste Zeile fahren und einige Male die RETURN-Taste betätigen, so würde die ausgegebene DATA-Zeile in das Programm übernommen, die Variablen mit den angegebenen Werten definiert und anschließend das BASIC-Programm mit einem GOTO-Befehl gestartet werden. Im Prinzip geschieht in unserem Programm nichts anderes. Da es innerhalb eines laufenden Programms relativ kompliziert ist, dieses Programm zu ergänzen, beenden wir einfach unser Programm und springen damit in den Direktmodus. Dies allein reicht aber noch nicht aus, um Programmzeilen zu übernehmen. Vielleicht ist ihnen auch schon einmal aufgefallen, daß, wenn Sie während eines Programmablaufs ein paar Tasten betätigen, die diesen Tasten zugeordneten Zeichen erst bei einem INPUT-Befehl oder bei dem Programmende ausgegeben werden. Wenn nicht, so probieren Sie dies einmal aus, indem Sie während des Programmablaufs ein paar Tasten betätigen:

```
10 FOR I=1 TO 5000 : NEXT I : END (RETURN-Taste)
```

Wie Sie sehen, werden tatsächlich alle Zeichen, die Sie eingegeben haben, ausgegeben. Dies hängt damit zusammen, daß Ihr Computer einen sogenannten Tastaturpuffer besitzt. Werden nun Tasten gedrückt, wenn der Computer diese nicht verarbeiten kann, sei es, weil er gerade ein Programm abarbeitet oder anderseitig beschäftigt ist, so werden diese Tasten im Tastaturpuffer abgelegt. Ist der Computer wieder in der Lage, Tastendrücke zu verarbeiten, so holt er sich zuerst die Tasten aus dem Tastaturpuffer. Schreiben wir nun innerhalb unseres BASIC-Programms den ASCII-Kode der RETURN-Taste in den Tastaturpuffer, so werden diese ASCII-Kodes beim Programmende ausgeführt. Die ASCII-Kodes der RETURN-Taste haben dann

die gleiche Wirkung wie ein Druck der RETURN-Taste. Es wird also die DATA-Zeile ins Programm übernommen, die Variablen definiert und anschließend wieder ins Programm gesprungen.

Um nun selbst mit diesem Programmiertrick arbeiten zu können, hier die notwendigen Adressen:

```
1319 - 1328 / $0527 - $0530: Tastaturpuffer
    239 /        $00EF: Anzahl Zeichen
```

Wenn Sie z.B. dreimal den ASCII-Kode 13 im Tastaturpuffer ablegen wollen, müssen Sie auch in die Adresse 239 den Wert drei schreiben.

Nun zur Bedienung des DATA-Wandlers:

Bitte beachten Sie bei der Generierung von DATA-Zeilen, daß vom DATA-Wandler immer volle Zeilen erzeugt werden. Es wird also immer die gleiche Anzahl an Zahlen in einer Zeile abgelegt. Es kann daher vorkommen, daß mehr Bytes in den DATA-Zeilen abgelegt werden, als dies eigentlich nötig wäre. Wie man austestet, ob zuviel Bytes abgespeichert worden sind, werden wir gleich noch besprechen. Zuerst soll aber erst einmal geklärt werden, wie man Programme aus den DATA-Zeilen wieder in den Speicher lesen kann. Hierzu dienen hauptsächlich die beiden BASIC-Befehle RESTORE und READ. Mit dem RESTORE-Befehl setzt man den DATA-Zeilen-Zeiger auf die erste Zeile in der die DATA's des gewünschten Maschinenprogramms abgelegt sind. Existieren nur die DATA's eines Maschinenprogramms in Ihrem BASIC-Programm, so kann der RESTORE-Befehl entfallen, da der Zeiger sowieso immer auf die erste DATA-Zeile zeigt.

Zum Einlesen der Bytes in den RAM-Speicher gibt es nun zwei Möglichkeiten, die sich besonders hervortun:

Erstens: die Bytes mittels einer FOR - NEXT Schleife einlesen. Der Schleifenanfang sollte dann die Anfangsadresse des Maschinenprogramms im Speicher sein und die Endadresse die

Endadresse des Maschinenprogramms im RAM-Speicher. Mittels eines READ-Befehls wird dann innerhalb der Schleife ein Byte aus den DATA-Zeilen gelesen und mit einem POKE-Befehl in den Speicher geschrieben. Eine solche Schleife könnte z.B. so aussehen:

```
10 FOR I=4096 TO 8192 : REM Schleife mit Laufvariable 'I'
20     READ A         : REM Byte lesen
30     POKE I,A       : REM Byte in Speicher schreiben
40 NEXT I             : REM Schleifenende
.
.
.
1000 DATA 00,01,02,03,04,...
```

Hierbei werden nur die Bytes in den Speicher geschrieben, die auch wirklich ein Teil des Maschinenprogramms darstellen und nicht auch noch die Werte die zum Füllen einer DATA-Zeile dienten. Ist die Schleife beendet, kann man durch das mehrmalige Eingeben von

```
READ A (RETURN-Taste)
```

ermitteln, wieviel Bytes zuviel abgespeichert wurden. Konnte man z.B. dreimal 'READ A' eingeben, ohne das der Fehler ?OUT OF DATA ERROR auftrat, so heißt das: Man kann die letzten drei Bytes in der letzten DATA-Zeile löschen. Die zweite Möglichkeit, Maschinenprogramme einzulesen, die in DATA-Zeilen abgelegt wurden, ist, die letzte Datazeile mit einer Endmarkierung zu kennzeichnen. Es empfiehlt sich der Wert -1. Zur Verdeutlichung hier noch einmal, wie die letzte DATA-Zeile in Ihrem Programm dann aussehen sollte:

```
1000 DATA XXX,XXX,XXX,XXX,XXX,-1
```

Man liest dann innerhalb einer Schleife solange Werte aus den DATA-Zeilen ein und speichert sie im RAM, bis der gelesene Wert -1 beträgt. Ist dies der Fall, so sind bereits alle Bytes ein-

gelesen worden. Auch hier sollte man wieder darauf achten, daß auch nur die wirklich relevanten Bytes eingelesen werden und nicht die Bytes, die zum Füllen einer Zeile dienen.

Beispiel:

```
10 I=4096           : REM Zeiger im Speicher
20 READ A           : REM Byte lesen
30 IF A=-1 THEN 70 : REM Endmarkierung ?
40 POKE I,A         : REM Nein, in Speicher schreiben
50 I=I+1            : REM Zähler erhöhen
60 GOTO 20          : REM Zum Schleifenanfang
70 .
80 .
90 .
1000 DATA XXX,XXX,XXX,XXX,XXX,XXX,-1
```

Beim C16 oder C116 tritt aufgrund des geringen Speicherplatzes des öfteren der Fall ein, daß man es sich vom BASIC-Speicherplatz her nicht erlauben kann, Maschinenprogramme in DATA's abzulegen. In diesem Fall sollte man die Maschinenprogramme mittels der Betriebssystemroutinen direkt von Kassette oder Diskette in den Speicher laden und danach einen NEW-Befehl ausführen. Dies realisiert man am besten mit einem Ladeprogramm, das die einzelnen Maschinenprogramme vom Peripheriegerät in den Speicher lädt und anschließend das BASIC-Programm einlädt und dann startet. Für dieses Verfahren, das man auch als OVERLAY-Verfahren bezeichnet,eignet sich übrigens der programmierte Direktmodus recht gut (siehe oben).

4. Nutzung der ROMs

4.1 So funktioniert der Interpreter

Um die Funktionsweise eines Interpreters zu erklären, soll als erstes erläutert werden, was überhaupt ein Interpreter ist.
Wie Sie bereits wissen, kann man einen Prozessor und damit einen Computer nur in Maschinensprache programmieren. Damit unser Computer nun aber die Programmiersprache BASIC versteht, muß in ihm ein Programm existieren, das die BASIC-Befehle ausführt. Dieses Programm bezeichnet man als BASIC-Interpreter oder kurz Interpreter, weil es quasi die BASIC-Befehle so interpretiert, daß auf sie entsprechend reagiert werden kann.

Eine der wichtigsten Bestandteile des Interpreters ist die Eingabeschleife. Innerhalb dieser Eingabeschleife werden solange Zeichen von der Tastatur geholt und auf dem Bildschirm dargestellt, bis es sich um die RETURN-Taste handelt. Außerdem werden die Zeichen noch in dem sogenannten BASIC-Eingabepuffer abgelegt, der die Adressen 512 bis 600 belegt.

Handelt es sich bei der gedrückten Taste um die RETURN-Taste, so überprüft der Interpreter zuerst, ob es sich hierbei um ein Programm oder um einen direkt auszuführenden Befehl handelt. Handelt es sich um eine Programmzeile, die in den BASIC-Programmspeicher gebracht werden soll, so erkennt der Interpreter dies an der vorangestellten Zeilennummer. Handelt es sich dagegen um einen Befehl oder mehrere Befehle die durch einen Doppelpunkt getrennt sind, geschieht folgendes:

Als erstes wird der BASIC-Eingabepuffer tokenisiert. 'Tokenisieren' bedeutet, daß innerhalb des BASIC-Speichers z.B. das Befehlswort 'PRINT' nicht durch die einzelnen Buchstaben, sondern nur durch ein Byte dargestellt wird. Dies hat den Vorteil, daß BASIC-Programme schneller abgearbeitet werden können und weitaus weniger Platz benötigen. Bei der Ausführung eines BASIC-Befehls dient dieses sogenannte TOKEN dann als Zeiger auf eine Tabelle, aus der dann die Adresse der speziellen Routine ermittelt wird. Alle anderen Texte, bis auf die BASIC-

Befehlswörter und Funktionen, werden so übernommen, wie sie eingegeben worden sind. In der Regel unterscheiden sich die TOKENS von den normalen Zeichen dadurch, daß sie alle einen Wert größer als 128 besitzen. Hierbei existiert allerdings eine Ausnahme:

Aufgrund der extrem hohen Zahl an BASIC-Befehlen reichten die 128 freien Bytes nicht aus. Aus diesem Grund nimmt das TOKEN 254 eine besondere Stellung ein. Trifft des BASIC-Interpreter bei der Ausführung einer Programmzeile auf diesen Kode, so holt er auch noch das folgende Byte und benutzt dieses dann als Zeiger auf eine andere Tabelle. Einige BASIC-Befehle werden also durch zwei Bytes statt durch ein Byte dargestellt.

Was geschieht nun, nachdem unser BASIC-Eingabepuffer tokenisiert worden ist, also die Befehlsworte in die TOKENS übersetzt worden sind?

Der BASIC-Programmzeiger, der immer auf die aktuelle Position im BASIC-Text zeigt, also eine ähnliche Funktion besitzt wie der PC, wird auf den Anfang des BASIC-Eingabepuffers gesetzt. Nun wird mit der Ausführung der Befehle begonnen. Handelt es sich bei dem ersten Byte im Puffer um kein TOKEN, also um keinen gültigen BASIC-Befehl, so wird die Fehlermeldung ?SYNTAX ERROR ausgegeben. Handelt es sich allerdings um einen gültigen Kode, so wird dieser als Zeiger auf eine Tabelle benutzt und aus dieser dann die Adresse der Routine bestimmt. Handelte es sich z.B. um das TOKEN für den POKE-Befehl, so wird in die Routine ab 40466 verzweigt. In dieser Routine werden nun die Parameter geholt und der Befehl ausgeführt. Ist dies beendet, steht der BASIC-Programmzeiger auf dem nächsten Zeichen hinter dem Wert, der vom POKE-Befehl in den Speicher geschrieben werden soll. In diesem Fall handelt es sich entweder um eine 0, die den Puffer abschließt oder um einen Doppelpunkt, der zwei BASIC-Befehle voneinander trennt. Würde es sich um keinen der beiden Fälle handeln, so hätte schon die Routine GETBYT einen Fehler gemeldet (siehe oben). Die gerade beschriebene Prozedur wird nun für

alle BASIC-Befehle innerhalb des Puffers ausgeführt. Sind alle Befehle ausgeführt worden, so wird wieder in die Eingabeschleife gesprungen um die nächste Tastatureingabe zu erwarten.

Wäre das erste Zeichen im BASIC-Eingabepuffer allerdings eine Zahl gewesen, so wäre etwas anderes passiert:
Als erstes wäre die Zeile natürlich wieder tokenisiert worden, da die TOKENS die einzige Form der BASIC-Befehle sind, die bei der Programmausführung verarbeitet werden können. Vor dieser tokenisierung hat im Gegensatz zu eben, allerdings die Speicherung der Zeilennummer stattgefunden, so daß der Interpreter nun weiß, an welche Stelle im Programmtext er die Zeile einfügen oder anhängen soll. Die Zeilennummern im Programmtext werden nun aber auf eine ganz spezielle Art und Weise abgelegt:

Die ersten beiden Bytes einer Zeile stellen immer die sogenannte LINK-Adresse auf die nächste Zeile dar. D.h., sie zeigen immer auf das erste Byte der LINK-Adresse der nächsten Zeile. Nach diesen beiden Bytes folgt dann die Zeilennummer im LOW-HIGH-Format.

Nach dieser Zeilennummer folgt nun der tokenisierte Programmtext, der durch ein Nullbyte abgeschlossen wird. Geben Sie dazu bitte einmal das folgende Beispielprogramm ein:

```
10 FOR I=1 TO 100
20 PRINT "TESTPROGRAMM"
30 NEXT I:END
```

Wenn wir uns nun den entsprechenden Bereich des BASIC-Speoichers, der übrigens bei $1000 (4096) immer mit einem Nullbyte beginnt, anschauen, ergibt sich der folgende Ausdruck:

```
MONITOR

   PC  SR AC XR YR SP
; FF00 00 00 FF 00 F8

M 1000 1032 (RETURN-Taste)
```

```
>1000 00 11 10 0A 00 81 20 49 :..... I
>1008 B2 31 20 A4 20 31 30 30 :21 $ 100
>1010 00 26 10 14 00 99 20 22 :.&.... "
>1018 54 45 53 54 50 52 4F 47 :TESTPROG
>1020 52 41 4D 4D 22 00 30 10 :RAMM".0.
>1028 1E 00 82 20 49 3A 80 00 :... I:..
>1030 00 00 00 00 00 00 00 00 :........
```

Sehen wir uns jetzt die erste Zeile unseres BASIC-Programms an: Wie Sie sehen, steht in der Speicherstelle $1000 ein Nullbyte. Dieses Byte ist immer gleich 0 und gehört somit eigentlich gar nicht zum Programmtext. Die beiden Bytes danach stellen nun die LINK-Adresse auf die Zeile 20 dar. Wie wir sehen, zeigt diese Adresse auf die Speicherstelle $1011, also das erste Byte der LINK-Adresse der Zeile 20. Auf diese Adresse folgt nun wieder im LOW-HIGH-Format unsere Zeilennummer 10 ($0A). Hier schließt sich nun der tokenisierte BASIC-Text an. Als erstes erkennen wir den Kode $81. Dies ist das TOKEN für den BASIC-Befehl 'FOR'. Auf dieses TOKEN folgen nun die beiden Bytes $20 und $49. Wenn wir einmal auf die rechte Seite des Ausdrucks schauen, so sehen wir, daß dies die Hexadezimalkodes der Zeilenfolge ' I' sind.

Wo ist aber nun unser Gleichzeichen geblieben? Die Lösung dieser Frage ist recht einfach:

Arithmetische Operationen wie +, -, *, / ... stellen ebenfalls BASIC-Befehle dar und werden daher auch tokenisiert. Das TOKEN für das Gleichzeichen ist nun $B2 und genau dieses Byte finden wir auch in der folgenden Speicherstelle $1008. Analysieren wir unseren Programmtext der Zeile 10 nun weiter, so finden wir dort noch die Ziffer 1, eine Lücke, das TOKEN für 'TO', wieder eine Lücke und die drei Ziffern der Zahl 100. Hinter diesen Ziffern finden wir nun ein Nullbyte. Dieses Nullbyte kennzeichnet das Ende einer Zeile im BASIC-Programmspeicher und gleichzeitig natürlich auch den Anfang einer neuen Zeile, in diesem Fall die Zeile 20. Grundsätzlich ist die Zeile 20 natürlich genauso aufgebaut.

Einen besonderen Fall bietet nun aber die Zeile 30. Dort werden nämlich zwei Befehle durch einen Doppelpunkt getrennt und dort endet auch das BASIC-Programm.

Wie wir sehen, erscheint der Doppelpunkt tatsächlich als Doppelpunkt und nicht etwa als TOKEN. Das Zeilenende wird auch in diesem Fall dadurch festgelegt, daß die Zeile mit einem Nullbyte abgeschlossen wird. Die nun folgende LINK-Adresse zeigt aber auf die Adresse 0, die natürlich nicht existiert, da der BASIC-Speicher erst bei $1000 beginnt. Das Programm wird also durch zwei Nullbytes abgeschlossen. Wie der Interpreter ein Zeichen aus dem BASIC-Text liest und dies auswertet, erfahren Sie im folgenden Kapitel.

4.2 Die Routinen des Betriebssystems und des BASICs

Wie Sie mittlerweile sicherlich wissen, besitzt Ihr Computer bereits eigebaute Software. Es handelt sich hierbei um das sogenannte Kernal- und um das BASIC-ROM. Unter einem ROM versteht man einen Festwertspeicher, der nur gelesen werden kann. Im Falle des C16, C116 oder Plus/4 enthalten sowohl das Kernal- als auch das BASIC-ROM viele nützliche Routinen. Unter anderem Routinen zur Integermultiplikation oder Routinen für sämtliche Rechenarten in Fließkommaarithmetik (siehe unten). Diese Routinen kann man allerdings nur nutzen, wenn man weiß, wo die Einsprungpunkte in diese Maschinenroutinen liegen, was die Routinen machen und welche Übergabeparameter erwartet werden. Diese Informationen sollen Ihnen in den weiteren Kapiteln zur Verfügung gestellt werden. Die Routinen werden nacheinander besprochen, also in der Reihenfolge, in der sie auch im Speicher stehen. Am Ende dieses Kapitels finden Sie dann noch einmal eine Tabelle mit allen Routinen und ihren Einsprungpunkten. Die Fließkommaroutinen werden später noch besprochen. Soweit es notwendig erscheint, werden auch kleine Beispielprogramme angeführt. Die meisten Routinen rufen intern noch andere Routinen auf. Im Zweifelsfalle sollte man daher ein dokumentiertes ROM-Listing zur Hand nehmen.

Übrigens: Die Anschaffung eines ROM-Listings ist immer zu empfehlen. Diese Beschreibung der Routinen kann und soll kein ROM-Listing ersetzen. Hier nun die einzelnen Routinen mit Ein- bzw. Ausgabeparametern:

CHRGET

Zweck: Holen eines Zeichens aus dem BASIC-Text
Adresse: 1139 / $0473

Beschreibung: Diese Routine ist wohl die wichtigste Routine des BASIC-Interpreters. Sie dient nämlich dazu, ein Byte aus dem BASIC-Text zu holen. Bevor dies jedoch geschieht, wird der sogenannte Textzeiger (TXTPTR) um den Wert eins erhöht. Dieser Zeiger zeigt dann auf das zu holende Zeichen. Er belegt die Speicherstellen 59/60 ($3B/$3C). Ist diese Erhöhung geschehen, so wird der Interrupt verhindert und überall das RAM eingeblendet (Siehe auch BANKING). Dann wird schließlich durch die indirekt indizierte Adressierung ein Zeichen aus dem BASIC-Text geholt. Ist dies geschehen, so wird wieder die alte Speicherkonfiguration hergestellt und der Interrupt wieder freigegeben. Nach dieser Prozedur ist die CHRGET-Routine aber immer noch nicht abgeschlossen, denn es erfolgt noch eine Auswertung des gelesenen Zeichens. Wie dies im einzelnen geschieht, entnehmen Sie bitte dem dokumentierten Disassemblerlisting weiter unten. An dieser Stelle nur die Bedeutungen der einzelnen Flags nach dem Aufruf der Routine CHRGET:

Zero = 1:	Nullbyte (Zeilenende) oder Doppelpunkt (Befehlstrennung)
Zero = 0:	Irgend ein anderes Zeichen
Carry = 1:	ASCII-Kode ohne Ziffern von 0-9
Carry = 0:	Es wurde eine Ziffer gelesen (0-9)

Das gelesene Zeichen steht im Akku und kann dort weiterverwendet werden. Beachten Sie bitte, daß einige der weiter unten aufgeführten Routinen verlangen, daß der Akku das gelesene Zeichen enthalten muß. Außerdem muß auch das Statusregister

die entsprechenden Werte enthalten. Um diese Bedingungen zu erfüllen, reicht es, wenn Sie die weiter unten stehende Routine CHRGOT aufrufen. Diese Routine holt das Zeichen, auf das der Programmzähler gerade zeigt; es wird also nicht der TXTPTR erhöht.

Eingabeparameter: 59/60
Ausgabewerte: .A, Carry, Zero

CHRGOT

Zweck: Momentanes Zeichen aus BASIC-Text holen
Adresse: 1145 / $0479

Beschreibung: Diese Routine entspricht (fast) vollständig der gerade beschriebenen Routine CHRGET. Der einzige Unterschied ist, daß der BASIC-Programmzeiger TXTPTR nicht erst um den Wert eins erhöht wird.

Eingabeparameter: Siehe CHRGET
Ausgabewerte: Siehe CHRGET

Listing: CHRGET/CHRGOT:

```
. 0473  E6 3B    CHRGET INC $3B     ;TXTPTR-LOW um eins erhöhen
. 0475  D0 02           BNE $0479   ;Kein Überlauf, dann Sprung
. 0477  E6 3C           INC $3C     ;TXTPTR-HIGH um eins erhöhen
. 0479  78       CHRGOT SEI         ;Interrupt verhindern
. 047A  8D 3F FF        STA $FF3F  ;Gesamte RAM einblenden
. 047D  A0 00           LDY #$00   ;Index auf Null
. 047F  B1 3B           LDA ($3B),Y ;Zeichen in Akku holen
. 0481  8D 3E FF        STA $FF3E  ;Alte RAM/ROM Konfiguration
. 0484  58              CLI        ;Interrupt wieder freigeben
. 0485  C9 3A           CMP #$3A   ;Zeichen größer (Z=1)/gleich (0)':'?
. 0487  B0 0A           BCS $0493  ;Ja, dann fertig!
. 0489  C9 20           CMP #$20   ;Ist es ein Leerzeichen?
. 048B  F0 E6           BEQ $0473  ;Ja, dann nochmal CHRGET
. 048D  38              SEC        ;Nein, Carry für Subtraktion setzen
```

```
. 048E  E9 30     SBC #$30    ;$30 abziehen
. 0490  38        SEC         ;Carry für Subtraktion setzen
. 0491  E9 D0     SBC #$D0    ;$D0 abziehen - wieder Ausgangswert
. 0493  60        RTS         ;Carry = 1: Zeichen war Ziffer
```

LDA (...),Y AUS RAM

Zweck: Wert aus RAM holen
Adresse: 1172 / $0494

Beschreibung: Diese Routine holt ein Zeichen immer aus dem RAM, egal ob dort gerade ROM aktiv ist oder nicht. Wie dies geschieht entnehmen Sie bitte dem weiter unten stehenden Disassemblerlisting. Wenn Sie sich das Listing genauer anschauen, so werden Sie erkennen, daß das Programm sich selbst modifiziert. Dies ist zwar kein sauberer Programmmierstil, dafür aber ein schneller.

ACHTUNG!!!

Diesen Programmierstil sollten Sie sich auf keinen Fall aneignen, er führt dazu, daß Ihre Programme:

1. Äußerst fehleranfällig
2. Unübersichtlich
3. Nicht verschiebbar
4. Nicht im ROM ablegbar
5. Chaotisch

sind. Dieser Programmierstil kann nur in äußerst zeitkritischen Programmteilen vertreten werden. Beachten Sie dann aber bitte, daß Ihr Programm im RAM liegen muß. Nun aber zur Parameterübergabe der Routine:

Der Akku muß die Zeropageadresse enthalten, die die eigentliche Adresse enthält. Die Adresse muß schon vorher in diesen Zeropageadressen abgelegt worden sein und zwar wie üblich im

LOW-HIGH-Format. Das Offset wird, wie bei der normalen indirekt indizierten Adressierung auch, im Y-Register übergeben. Nach der Routine enthält der Akku das geholte Byte.

Eingabeparameter: Zeropage, .A, .Y
Ausgabewerte: .A

Listing: LDA (...),Y

```
. 0494  8D 9C 04 STA $049C   ;Zeropageadresse im Routine schreiben
. 0497  78       SEI         ;Interrupt verhindern
. 0498  8D 3F FF STA $FF3F   ;Überall RAM einblenden
. 049B  B1 ...   LDA (...),Y ;Byte holen (Mit Y-Offset)
. 049D  8D 3E FF STA $FF3E   ;Alle ROM/RAM-Konfiguration herstellen
. 04A0  58       CLI         ;Interrupt wieder freigeben
. 04A1  60       RTS         ;Fertig und Ende
```

Zweck: Warmstart ausführen
Adresse: 32778 / $800A

Beschreibung: Diese Routinen führen einen Warmstart des Computers aus. Es werden alle offenen Ein- Ausgabekanäle geschlossen, wobei eventuell offene Dateien aber nicht durch einen CLOSE-Befehl beendet werden. Als Eingabegerät wird wieder die Tastatur gesetzt und ebenfalls die Grafik ausgeschaltet. Hiernach springt die Routine dann zur Ausgabe von 'READY.'.

Eingabeparameter: Keine
Ausgabewerte: Keine

KALTSTART

Zweck: Ausführen eines Kaltstartes
Adresse: 32793 / $8019

Beschreibung: Diese Routine wird z.B. inerhalb der RESET-Routine aufgerufen. Es werden alle Vektoren eingerichtet und die weiter unten beschriebene Routine BASIC-RESET ausfgerufen. Nach dem Aufruf dieser Routine wird die Einschaltmeldung ausgegeben und getestet, ob ein Modul im Expansionsport aktiv ist. Wenn ja, wird dieses Modul gestartet. Andernfalls springt die Routine indirekt zur READY-Ausgabe.

Eingabeparameter: Keine
Ausgabewerte: Keine

BASIC-RESET

Zweck: Initialisieren des BASIC-Interpreters
Adresse: 32814 / $802E

Beschreibung: Innerhalb dieser Routine wird der Computer für den Betrieb des BASIC-Interpreters vorbereitet. Es werden sowohl die CHRGET-Routine als auch die Routinen zum RAM-Zugriff vom ROM in das RAM kopiert. Außerdem wird noch die USR-Adresse auf die Fehlermeldung ?ILLEGAL QUANTITY ERROR gesetzt und die Zeiger für die Speichergrenzen gesetzt. Die Startwerte für die RND-Funktion werden festgelegt und noch einige andere Adressen zurückgesetzt. Zuletzt werden durch den Aufruf dieser Routine auch die AUTO- und die TRACE-Funktion abgeschaltet und der Textmodus aktiviert.

Eingabeparameter: Keine
Ausgabewerte: Keine

EINSCHALTMELDUNG

Zweck: Ausgeben der BASIC-Einschaltmeldung
Adresse: 32962 / $80C2

Beschreibung: Es wird die BASIC-Einschaltmeldung auf den Bildschirm ausgegeben und anschließend der BASIC-Speicher gelöscht und die Variablezeiger neu gesetzt.

Eingabeparameter: Keine
Ausgabewerte: Keine

VEKTOREN EINRICHTEN

Zweck: Vektoren neu setzen
Adresse: 33047 / $8117

Beschreibung: Die Vektorentabelle von 33029-33046 ($8105-$8116) wird zur Adresse 768ff kopiert. Hierdurch werden alle Vektoren des Betriebssystems neu gesetzt.

Eingabeparameter: Keine
Ausgabewerte: Keine

CHRGET (KOPIE)

Zweck: Zeichen aus BASIC-Text holen
Adresse: Siehe oben; wird ins RAM kopiert

Beschreibung: Orginal der CHRGET-Routine. Dieser Teil des ROMs wird in den Systemspeicher kopiert.

Eingabeparameter: Siehe oben
Ausgabewerte: Siehe oben

FEHLERMELDUNG AUSGEBEN

Zweck: Ausgeben einer BASIC-Fehlermeldung
Adresse: 34438 / $8686

Beschreibung: Mittels dieser Routine kann eine BASIC-Fehlermeldung ausgegeben werden. Das X-Register muß hierzu die

Nummer der Fehlermeldung enthalten. Die Nummern der Fehlermeldungen laufen von 0 bis 36 und entsprechen den Werten, die nach dem Auftreten eines Fehlers in der BASIC-Systemvariablen 'ER' abgelegt werden. Die entsprechende Tabelle entnehmen Sie bitte Ihrem Computerhandbuch.

Eingabeparameter: .X
Ausgabewerte: Keine

BASIC-PROGRAMMZEILE LÖSCHEN

Zweck: Löschen einer Programmzeile
Adresse: 34619 / $873B

Beschreibung: Mittels dieser Routine ist es möglich einzelne Programmzeilen aus einem BASIC-Programm zu entfernen. Solch eine Routine kann sich z.B. dann als recht nützlich erweisen, wenn man selbst BASIC-Erweiterungen programmiert. Die Anfangsadresse der Zeile muß in den beiden Speicherstellen 95/96 ($5F/$60) stehen. Diese kann durch eine Routine ermittelt werden, die erst weiter unten beschrieben wird. Um nun die Adresse zu ermitteln, müssen Sie die Zeilennummer, die Sie löschen wollen, in die Speicherstellen 20/21 ($14/$15) schreiben. Rufen Sie jetzt die Routine an 35389 ($8A3D) auf, so wird die Adresse der Zeile berechnet und in den oben erwarteten Speicherstellen abgelegt.

Eingabeparameter: Zeilenadresse in 95/96
Ausgabewerte: Keine

Beispiel:

```
LDA #$0A  ;LOW-Byte Zeilennummer 10
LDY #$00  ;HIGH-Byte Zeilennummer 10
STA $14   ;LOW-Byte in 20
STY $15   ;HIGH-Byte in 21
JSR $8A3D ;Adresse der Zeile ermitteln
JSR $873B ;Zeile löschen
```

ZEILE EINFÜGEN

Zweck: Zeile in BASIC-Text einfügen
Adresse: 34706 / $8792

Beschreibung: Um eine Zeile in den BASIC-Programmtext einzufügen, muß die Adresse der folgenden Zeile bekannt sein. Diese Adresse kann man aber wieder auf die gleiche Weise ermittelt, wie beim Löschen von Programmzeilen. Nähere Informationen gibt hierzu die Beschreibung der Routine 'ZEILE SUCHEN', die weiter unten noch beschrieben wird. Die Adresse dieser folgenden Zeile muß ebenfalls wieder in den beiden Speicherstellen 95/96 ($5F/$60) abgelegt werden. Die Zeile, die eingefügt werden soll, muß tokenisiert im BASIC-Eingabepuffer stehen. Die Tokenisierung ist ebenfalls mit einer Routine möglich, die sich an 35155 ($8913) befindet.

Die tokenisierte Zeile muß durch ein Nullbyte abgeschlossen sein und die Länge dieser tokenisierten Zeile muß in der Speicherstelle 11 ($0B) stehen. Nähere Auskünfte hierzu entnehmen Sie bitte der Beschreibung zur Tokenisier-Routine.

Eingabeparameter: BASIC-Eingabepuffer, 11, 95/96
Ausgabewerte: Keine

Beispiel:

```
LDA #$0A  ;LOW-Byte Zeilennummer 10
LDY #$00  ;HIGH-Byte Zeilennummer 10
JSR $8A3D ;Adresse der folgenden Zeile nach 95/96
JSR $8792 ;Eingabepuffer in Programmtext einfügen
```

LINKADRESSEN ERZEUGEN

Zweck: Programmzeilen neu binden
Adresse: 34840 / $8818

Beschreibung: Die LINK-Adressen aller BASIC-Programmzeilen werden neu berechnet. Dies geschieht z.B. nach dem Einladen

eines BASIC-Programms, damit die Ausgabe des Programmlistings auch in anderen Speicherbereichen möglich ist. Dies ist notwendig, da die LIST-Routine des Interpreters sich anhand der LINK-Adressen vorantastet.

Eingabeparameter: Keine
Ausgabewerte: Keine

BLOCKVERSCHIEBEROUTINE

Zweck: Verschiebung von Speicherbereichen
Adresse: 35008 / $88C0

Beschreibung: Wollen Sie Speicherbereiche kopieren, so brauchen Sie nicht extra eine Kopierroutine schreiben, sondern Sie können die bereits vorhandene Blockverschieberoutine benutzen. Auch von BASIC aus kann diese Routine hervorragend angewendet werden. Bekannt sein muß lediglich die Adresse des neuen Blockendes, des Quellbereichanfangs und die Adresse des Quellbereichendes plus 1. Abgelegt werden die einzelnen Adressen in den folgenden Speicherstellen der Zeropage:

Neues Blockende: 49/50 ($31/$32)
Quellbereichende plus 1: 90/91 ($5A/$5B)
Quellbereichanfang: 95/96 ($5F/$60)

Eingabeparameter: 49/50, 90/91, 95/96
Ausgabewerte: Neuer Speicherbereich

TEST AUF PLATZ IM STACK

Zweck: Prüfen ob noch Platz im Stack ist
Adresse: 35077 / $8905

Beschreibung: Bevor man eine größere Anzahl an Daten auf dem Stack ablegt, sollte man testen, ob überhaupt noch genügend Speicherplatz zur Verfügung steht. Diese Aufgabe erledigt die hier vorgestellte Routine. Die Anzahl der benötigten Bytes wird

im Y-Register an diese Routine übergeben. Ist noch genügend Platz vorhanden, wird aus dieser Routine mit einem RTS ins Hauptprogramm zurückgekehrt. War die benötigte Anzahl an Bytes aber nicht mehr frei, so wird die BASIC-Fehlermeldung ?OUT OF MEMORY ERROR ausgegeben.

Eingabeparameter: .Y
Ausgabewerte: Keine

TOKENISIEREN

Zweck: BASIC-Befehle durch TOKENS ersetzen
Adresse: 35155 / $8953

Beschreibung: Mittels dieser Betriebsroutine werden die BASIC-Schlüsselwörter durch TOKENS ersetzt (siehe auch Kapitel 4.1). Der Zeiger an den Adressen 59/60 ($3B/$3C) gibt die Adresse an, an der der zu tokenisierende Text steht. Da die einzelnen Zeichen mittels der Routine CHRGET geholt werden, sollte vor dem Aufruf der Tokenisierroutine noch die Routine CHRGOT aufgerufen werden. Der Puffer muß durch ein Nullbyte abgeschlossen werden. Nach der Tokenisierung steht der neue Text wieder an der gleichen Stelle wie der alte Text. Da der Text aber insgesamt kürzer geworden ist, wird im Y-Register die neue Länge übergeben.

Eingabeparameter: 59/60
Ausgabewerte: .Y

ZEILE SUCHEN

Zweck: Adresse einer Zeile ermitteln, falls vorhanden
Adresse: 35389 / $8A3D

Beschreibung: Diese Routine ist eine der wichtigsten Routinen im BASIC-Interpreter. Sie ermittelt nämlich die Adresse des ersten Bytes einer Zeile. Sollte diese Zeile nicht vorhanden sein, wird die Adresse des ersten Bytes der folgenden Zeile in den

Speicherstellen 95/96 ($5F/$60) abgelegt. Als Unterscheidungsmerkmal wird außerdem noch das Carryflag gelöscht, falls die Zeile nicht vorhanden war. War sie allerdings vorhanden, so ist das Carryflag gesetzt. Die zu suchende Zeilennummer wird wieder in den Speicherstellen 20/21 ($14/$15) an die Routine übergeben.

Eingabeparameter: 20/21
Ausgabewerte: 95/96, Carry

NEW

Zweck: Löschen den BASIC-Speichers
Adresse: 35351 / $8A7B

Beschreibung: Diese Routine entspricht dem BASIC-Befehl NEW, nur mit der Ausnahme, daß nicht geprüft wird, ob noch ein Zeichen folgt. Wird diese Routine aufgerufen, so werden alle Zeiger, die den Anfang oder das Ende der Variabletabellen kennzeichnen, ebenfalls zurückgesetzt.

Eingabeparameter: Keine
Ausgabewerte: Keine

CLR

Zweck: Alle Variablen löschen
Adresse: 35482 / $8A9A

Beschreibung: Durch den Aufruf dieser Routine werden alle BASIC-Variablen gelöscht. Das BASIC-Programm bleibt hierbei natürlich, wie beim CLR-Befehl, erhalten.

Eingabeparameter: Keine
Ausgabewerte: Keine

LIST

Zweck: Auflisten eines BASIC-Programms
Adresse: 35586 / $8B02

Beschreibung: Mittels dieser Routine können Sie von Maschinensprache aus Teile eines BASIC-Programms auslisten. In der Speicherstellen 95/96 ($5F/$60) muß die Adresse der ersten zu zeigenden Zeile angegeben werden. Soll das ganze Programm gezeigt werden, ist dies die Adresse der Zeile 0. In den Speicherstellen 20/21 ($14/$15) wird dagegen die letzte zu zeigede Zeilennummer eingegeben. Soll das ganze Programm gezeigt werden, setzen Sie hier bitte die Zeilennummer 65535 ($FFFF) ein.

Eingabeparameter: 95/96, 20/21
Ausgabewerte: Keine

RUN OHNE ZEILENNUMMER

Zweck: Starten eines BASIC-Programms
Adresse: 35774 / $8BBE
Beschreibung: Mit dieser Routine können Sie BASIC-Programme von Maschinensprache aus aufrufen. Den Einsprungpunkt, um Programme bei einer bestimmten Zeilennummer zu starten, gibt es nicht, da von der RUN-Routine direkt in die GOTO-Routine des BASIC-Interpreters gesprungen wird.

Eingabeparameter: Keine
Ausgabewerte: Keine

RESTORE OHNE ZEILENNUMMER

Zweck: Positionieren des DATA-Zeigers
Adresse: 36017 / $8CB1

Beschreibung: Ruft man von Maschinensprache aus diese Routine auf, so wird der DATA-Zeiger auf das erste Element der ersten DATA-Zeile gesetzt.

Eingabeparameter: Keine
Ausgabewerte: Keine

GOTO

Zweck: Springen an eine bestimmte Zeilennummer
Adresse: 36176 / $8D50

Beschreibung: Mit dieser Routine können Sie in ein BASIC-Programm springen, ohne das die BASIC-Variablen gelöscht werden. Die Zeilennummer zu der gesprungen werden soll, wird in den Speicherstellen 20/21 ($14/$15) angegeben.

Eingabeparameter: 20/21
Ausgabewerte: Keine

STRING NACH INTEGER WANDELN

Zweck: Zahlenstring in 16-Bit Integerzahl
Adresse: 36414 / $8E3E

Beschreibung: Will man eine positive Integerzahl aus dem BASIC-Text holen, die in Form von einzelnen ASCII-Zeichen abgelegt ist, so bietet sich diese Routine an. Mittels CHRGET werden die einzelnen Zeichen aus dem BASIC-Text geholt und in eine 16-Bit-Integerzahl gewandelt. Diese Integerzahl muß im Bereich von 0 bis 63999 liegen und wird in den Speicherstellen 20/21 ($14/$15) abgelegt. Die Adresse des BASIC-Programmzeigers kann man wieder mittels der Speicherstellen 59/60 ($3B/$3C) ändern. Da die Zeichen mit CHRGET geholt werden, empfiehlt es sich, vor der Wandlung die Routine CHRGOT aufzurufen.

Eingabeparameter: 59/60
Ausgabewerte: 20/21

STRING AUSGEBEN (TXTOUT)

Zweck: Ausgeben eines Textes
Adresse: 37000 / $9088

Beschreibung: Mittels dieser Routine ist es möglich einen String auszugeben, der durch ein Nullbyte abgeschlossen ist. Die Adresse des ersten Zeichens muß im Akku und im Y-Register stehen. Der Akku enthält hierbei das LOW-Byte der Adresse, während das Y-Register das HIGH-Byte dieser Adresse enthält. Diese Routine eignet sich besonders dann, wenn in einer Routine auf ein Ereignis reagiert werden soll. Mit der Routine PRIMM müßte man für jeden Fall eine neue Ausgaberoutine programmieren, während es mit dieser Routine möglich ist, anhand einer Tabelle die Adresse eines Textes zu ermitteln.

Eingabeparameter: .A, .Y
Ausgabewerte: Keine

Beispiel:

```
    TYA              ;Nummer des Textes in Akku
    ASL              ;mit 2 multiplizieren
    TAY              ;Offset wieder ins Y-Register
    LDA TABELLE,Y    ;LOW-Byte des Textes aus Tabelle
    PHA              ;Auf Stack zwischenspeichern
    LDA TABELLE+1,Y ;HIGH-Byte holen
    TAY              ;HIGH-Byte in Y-Register
    PLA              ;LOW-Byte in Akku holen
    JSR $9088        ;String ausgeben
TABELLE .WORD TXT1,TXT2 ... ;Adressen der Texte LOW/HIGH
```

```
TXT1 .ASC "Text 1" .BYTE 0
TXT2 .ASC "Text 2" .BYTE 0
   .
   .
   .
```

FRMNUM

Zweck: Auswerten eines numerischen Ausdrucks
Adresse: 37652 / $9314

Beschreibung: Diese Routine ruft zuerst die Routine FRMEVL auf und testet anschließend, ob das Ergebnis eine Zahl war. War dies nicht der Fall, so wird die BASIC-Fehlermeldung ?TYPE MISMATCH ERROR ausgegeben. Das Ergebnis dieser numerischen Formelauswertung wird immer im Fließkommaakkumulator abgelegt, sofern es sich natürlich um ein numerisches Ergebnis handelt. Mittels der Fließkommaroutinen kann dieses Ergebnis dann in das Integerformat umgewandelt werden. Nähere Informationen hierzu gibt das Kapitel 5.2. Die einzelnen Zeichen des numerischen Ausdrucks werden wieder über die Routine GETBYT geholt. Aus diesem Grund ist der Zeiger wieder das Speicherstellenpaar 59/60 ($3B/$3C). Außerdem sollten Sie vor dem Aufruf noch die Routine CHRGOT aufrufen, damit der Akku wieder den Kode des ersten Zeichens enthält.

Eingabeparameter: 50/60
Ausgabewerte: FAC

CHKNUM

Zweck: Testen, ob die Formelauswertung eine Zahl ergab
Adresse: 37655 / $9317

Beschreibung: Diese Routine testet, ob das Ergebis der Formelauswertungsroutine (FRMEVL) eine numerische Zahl war. War

dies nicht der Fall, so wird eine BASIC-Fehlermeldung ausgegeben. Die Routine CHKNUM wird unter anderem auch in der Routine FRMNUM aufgerufen.

Eingabeparameter: FRMEVL
Ausgabewerte: Keine

CHKSTR

Zweck: Testen, ob die Formelauswertung einen String ergab
Adresse: 37658 / $931A

Beschreibung: Diese Routine testet analog zur Routine CHKNUM, ob das Ergebnis der Formelauswertungsroutine (FRMEVL) ein String war. War dies nicht der Fall, so wird analog zu CHKNUM eine BASIC-Fehlermeldung ausgegeben.

Eingabeparameter: FRMEVL
Ausgabewerte: Keine

FRMEVL

Zweck: Auswerten eines beliebigen Ausdrucks
Adresse: 37676 / $932C

Beschreibung: Diese Routine wertet jeden beliebigen Ausdruck aus. Dieser Ausdruck kann sowohl ein numerischer, als auch ein String-Ausdruck sein. War das Ergebnis ein numerischer Ausdruck, so steht das Ergebnis wie üblich im Fließkommaakku, war es dagegen ein String, so enthalten die Speicherstellen 98/99 ($62/$63) die Adresse des Strings und die Speicherstelle 97 ($61) die Länge des Strings. Die Speicherstellen 100/101 ($64/$65) enthalten die Adresse des Stringdeskriptors. Der Stringdeskriptor enthält ebenfalls die Daten des Strings. An der Adresse, die durch die Speicherstellen 100/101 festgelegt ist, steht zuerst die Stringlänge und dann die Adresse des Stringanfangs im LOW-HIGH-Format.

Da die Zeichen des Ausdrucks mittels der Routine CHRGET geholt werden, geben die Speichgerstellen 59/60 ($3B/$3C) wieder die Adresse des ersten Zeichens des Ausdrucks an. Vor dem Aufruf der Routine FRMEVL sollte man die Routine CHRGOT aufrufen, um den Kode des ersten Zeichens in den Akku zu laden, sofern dieser nicht noch im Akku enthalten ist.

Eingabeparameter: 59/60
Ausgabewerte: 97, 98/99, 100/101

16-BIT-INTEGERMULTIPLIKATION

Zweck: Multiplikation zweier Integerzahlen
Adresse: 39484 / $9A3C

Beschreibung: Mittels dieser Routine können zwei 16-Bit-Integerwerte miteinander multipliziert werden. Leider können diese Werte nur positiv sein. Auch darf das Ergebnis nicht den Bereich überschreiten, der sich durch eine 16-Bit-Zahl darstellen läßt. Geschieht dies doch, so wird die Fehlermeldung ?ILLEGAL QUANTITY ERROR ausgegeben. Der erste Faktor muß in den Speicherstellen 40/41 ($28/$29) und der zweite Wert in den Speicherstellen 113/114 ($71/$72) abgelegt werden. Das Ergenis dieser Integermultiplikation steht immer im Y- und im X-Register. Das Y-Register enthält hierbei das LOW- und das X-Register das HIGH-Byte des Ergebnisses.
Eingabeparameter: 40/41, 113/114
Ausgabewerte: .Y/.X

INTEGER (A,Y) NACH FAC

Zweck: Integerzahl im FAC ablegen
Adresse: 39570 / $9A92

Beschreibung: Die 16-Bit-Integerzahl im Akku und Y-Register wird im Fließkommaakkumulator abgelegt. Was man sich unter der Fließkommaarithmetik vorzustellen hat, wird im Kapitel 5.2 beschrieben.

Eingabeparameter: .A/.Y
Ausgabewerte: FAC

GETBYT

Zweck: Holen eines Bytes aus dem BASIC-Text
Adresse: 40321 / $9D81

Beschreibung: Diese Routine holt ein Byte aus dem BASIC-Text und legt es im X-Register ab. Innerhalb der Routine GETBYT wird die Routine FRMNUM aufgerufen. Daher wird die Fehlermeldung ?TYPE MISMATCH ERROR ausgegeben, wenn es sich nicht um einen numerischen Ausdruck handelt. Da innerhalb der Routine FRMNUM die einzelnen Zeichen des Ausdrucks durch die Routine CHRGET geholt werden, sollte CHRGOT aufgerufen werden, wenn der Akku und das Statusregister nicht mehr die richtigen Werte enthalten (Siehe CHRGET/CHRGOT).

Da die Zeichen durch CHRGET geholt werden, kann man die Adresse des ersten Zeichns wieder durch die Speicherstellen 59/60 ($3B/$3C) bestimmen.

Eingabeparameter: 59/60, .A, .ST
Ausgabewerte: .X

GETPAR

Zweck: Parameter holen (16-Bit,8-Bit)
Adresse: 40402 / $9DD2

Beschreibung: Innerhalb dieser Routine werden nacheinander die Routine GETADR (s.u.), CHKKOM (s.u.) und GETBYT (s.o.)

aufgerufen. Es wird also eine positive 16-Bitzahl geholt, in den Speicherstellen 20/21 ($14/$15) abgelegt, auf ein folgendes Komma getestet und eine 8-Bit-Zahl geholt, deren Wert im X-Register abgelegt wird.
Für genauere Informationen ziehen Sie bitte die Erklärungen der einzelnen Routinen zu Rate.

Eingabeparameter: 59/60
Ausgabewerte: 20/21, .X

GETADR

Zweck: Holen eines 16-Bit-Wertes aus dem BASIC-Text
Adresse: 40414 / $9DDE bzw. 40417 / $9DE1

Beschreibung: Diese Routine holt einen 16-Bit-Wert aus dem BASIC-Text und legt in den beiden Speicherstellen 20/21 ($14/$15) ab. Soll vorher noch auf ein Komma getestet werden, so ist der erste Einsprungpunkt anzuwenden; soll nur der Wert ohne Test auf ein Komma geholt werden, ist die Routine an der Adresse 40417 auzurufen. Innerhalb der Routine GETADR wird unter anderem auch die Routine FRMEVL aufgerufen. Daher sollte vor dem Aufruf der Routine GETADR bereits der Kode des ersten Zeichens im Akku enthalten sein. Ist er noch nicht enthalten, sollten Sie auf jeden Fall die Routine CHRGOT aufrufen. Da die Zeichen wieder über CHRGET geholt werden, kann man die Adresse, an der das erste Zeichen steht, in den Speicherstellen 59/60 ($3B/$3C) ändern.
Eingabeparameter: 59/60
Ausgabewerte: 20/21

FAC := ARG - FAC

Zweck: Fließkommasubtraktion
Adresse: 40583 / $9E87

Beschreibung: Der Inhalt des Fleißkommaakkumulators (FAC) wird vom Inhalt des zweiten Fließkommaakkumulators (ARG) abgezogen. Das Ergebnis wird wieder im FAC abgelegt.

Eingabeparameter: FAC, ARG
Ausgabewerte: FAC

FAC := KONST. (A,Y) AUS RAM + FAC

Zweck: Fließkommaakku um Konstante aus RAM erhöhen
Adresse: 40603 / $9E9B

Beschreibung: Zum Inhalt des ersten Fließkommaakkumulators (FAC) wird eine Konstante addiert. Die Adresse dieser Konstanten steht in dem Registerpaar Akku und Y-Register. Die Konstante wird immer aus dem RAM geholt. Das Ergebnis dieser Addition wird wieder im FAC abgelegt.

Eingabeparameter: FAC, .A/.Y
Ausgabewerte: FAC

FAC := ARG + FAC

Zweck: Fließkommaaddition
Adresse: 40606 / $9E9E

Beschreibung: Zum Inhalt des ersten Fließkommaakkumulators (FAC) wird der Inhalt des zweiten Fließkommaakkumulators addiert. Das Ergebnis dieser Addition steht wieder im FAC.

Eingabeparameter: FAC, ARG
Ausgabewerte: FAC

FAC := FAC + 0.5

Zweck: Der FAC wird um die Konstante 0.5 erhöht
Adresse: 41058 / $A062

Beschreibung: Der Inhalt des ersten Fließkommaakkumulators (FAC) wird um die Konstante 0.5 erhöht. Das Ergebnis dieser Addition wird ebenfalls wieder im ersten Fließkommaakkumulator (FAC) abgelegt.

Eingabeparameter: FAC
Ausgabewerte: FAC

FAC := KONST. (A,Y) AUS ROM - FAC

Zweck: FAC von Konstante aus ROM subtrahieren
Adresse: 41068 / $A06C

Beschreibung: Von der Fließkommakonstanten, die an der Adresse (A,Y) im ROM steht, wird der Inhalt des FAC subtrahiert. Das Ergebnis dieser Subtraktion steht wieder im FAC.

Eingabeparameter: .A/.Y, FAC
Ausgabewerte: FAC

FAC := KONST. (A,Y) AUS ROM / FAC

Zweck: Konstante aus ROM durch FAC dividieren
Adresse: 41074 / $A072

Beschreibung: Eine Konstante aus dem ROM, die an der Adresse steht die durch den Akku und das Y-Register gebildet wird, wird durch den Inhalt des FAC dividiert. Bei der Konstanten muß es sich natürlich wieder um eine Fließkommazahl handelt. Das Ergebnis der Division steht wieder im FAC.

Eingabeparameter: .A/.Y, FAC
Ausgabewerte: FAC

FAC := KONST. (A,Y) AUS RAM * FAC

Zweck: FAC mit Konstante multiplizieren
Adresse: 41080 / $A078

Beschreibung: Eine Konstante aus dem ROM an der Adresse, die durch den Akkumulator und das Y-Register gebildet wird, wird mit dem momentanen Ihalt des Fließkommaakkumulators (FAC) multipliziert. Das Ergebnis der Multiplikation steht wieder im FAC. Bei der Konstanten muß es sich ebenfalls um eine Fließkommazahl handeln.

Eingabeparameter: .A/.Y, FAC
Ausgabewerte: FAC

FAC := ARG * FAC

Zweck: Fließkommamultiplikation
Adresse: 41083 / $A07B

Beschreibung: Die beiden Fließkommaakkumulatoren werden miteinander multipliziert. Das Ergebnis der Multiplikation wird im FAC abgelegt.

Eingabeparameter: FAC, ARG
Ausgabewerte: FAC

KONST. (A,Y) AUS ROM NACH FAC

Zweck: Laden des FACs mit einer ROM-Konstante
Adresse: 41180 / $A0DC

Beschreibung: Mit dieser Routine ist es möglich den FAC mit einer Konstante zu laden, die im ROM steht. Die Adresse dieser

Konstanten muß, wie üblich, im Akku und im Y-Register stehen. Der Akku enthält hierbei das LOW- und das Y-Register das HIGH-Byte der Adresse.

Eingabeparameter: .A/.Y
Ausgabewerte: FAC

KONST. (A,Y) AUS RAM NACH ARG

Zweck: Laden des ARGs mit einer RAM-Konstante
Adresse: 41223 / $A107

Beschreibung: Mit dieser Routine ist es möglich, eine Konstante aus dem RAM in den zweiten Fließkommaakkumulator (ARG) zu laden. Die Adresse im RAM wird wieder durch den Akkumulator und das Y-Register bestimmt.

Eingabeparameter: .A/.Y
Ausgabewerte: ARG

FAC := FAC * 10

Zweck: Multiplizieren des FACs mit 10
Adresse: 41314 / $A162

Beschreibung: Der Inhalt des ersten Fließkommaakkumulators (FAC) wird mit dem Wert 10 multipliziert. Das Ergebnis dieser Multiplikation wird wird im FAC abgelegt. Der ARG enthält nach der Operation den ehemaligen Wert des FAC.

Eingabeparameter: Keine
Ausgabewerte: FAC, (ARG)

FAC := FAC / 10

Zweck: Dividieren des FACs durch 10
Adresse: 41347 / $A183

Beschreibung: Der Inhalt des FACs wird durch 10 dividiert. Das Ergebnis dieser Division wird wieder im FAC abgelegt. Der Inhalt des ARGs bleibt durch diese Routine nicht erhalten, er enthält nach dem Aufruf dieser Routine den ehemaligen Inhalt des FACs.

Eingabeparameter: Keine
Ausgabewerte: FAC, (ARG)

FAC := ARG / FAC

Zweck: Fließkommadivision
Adresse: 41364 / $A194 bzw. 41367 / $A197

Beschreibung: Der Inhalt des ARGs wird durch den Inhalt des FACs dividiert. Das Ergebnis dieser Division wird wieder im FAC abgelegt. Hat der FAC vor der Division den Inhalt 0, so wird die Fehlermeldung ?DIVISION BY ZERO ausgegeben. Soll der momentane Inhalt des ARGs durch den momentanen Inhalt des FACs dividiert werden, so muß die Routine über den zweiten Einsprungpunkt aufgerufen werden. Soll dagegen der ARG noch mit einer Konstanten aus dem RAM geladen werden, bevor die Division durchgeführt wird, ist der erste Einsprungpunkt zu wählen. Es wird dann nämlich noch die Routine an 41223 ($A107) aufgerufen.

Eingabeparameter: 1. ARG, FAC
2. .A/.Y, FAC
Ausgabewerte: FAC

KONST. (A,Y) AUS RAM NACH FAC

Zweck: FAC mit einer Konstanten aus dem RAM laden
Adresse: 41503 / $A21F

Beschreibung: Mittels dieser Routine können Fließkommakonstanten aus dem RAM in den FAC geladen werden. Die Adresse dieser Konstanten steht im Akku und im Y-Register. Im Akku steht das LOW- und im Y-Register das HIGH-Byte der Adresse.

Eingabeparameter: .A/.Y
Ausgabewerte: FAC

KONST. (A,Y) AUS ROM NACH FAC

Zweck: FAC mit einer Konstanten aus dem ROM laden
Adresse: 41505 / $A221

Beschreibung: Mittels dieser Routine können Konstanten aus dem ROM in den FAC geladen werden. Die Anfangsadresse dieser Fließkommakonstanten muß im Akku (LOW) und im Y-Register (HIGH) stehen.

Eingabeparameter: .A/.Y
Ausgabewerte: FAC

FAC NACH RAM-ADR. (X,Y)

Zweck: FAC im RAM-Speicher ablegen
Adresse: 41561 / $A259

Beschreibung: Zur Ablegung von Fließkommazahlen im Arbeitsspeicher (RAM) dient diese Unterroutine. Der FAC wird an der Adresse abgelegt, die im X- und Y-Register steht. Das X-Register enthält das LOW-Byte dieser Adresse, während das Y-Register wieder das HIGH-Byte der Adresse enthält.

Eingabeparameter: FAC, .X/.Y
Ausgabewerte: .X/.Y

ARG NACH FAC

Zweck: Übertragung des ARG in den FAC
Adresse: 41601 / $A281

Beschreibung: Der ARG wird in den FAC übertragen. Der Inhalt des FAC wird überschrieben. Um FAC und ARG auszutauschen, muß der FAC vorher zwischengespeichert werden.

Eingabeparameter: FAC, ARG
Ausgabewerte: FAC

FAC NACH ARG

Zweck: Übertragung des FAC in den ARG
Adresse: 41617 / $A291

Beschreibung: Der FAC wird in den ARG kopiert. Der Inhalt des ARGs geht hierbei verloren, sofern er nicht vorher sichergestellt worden ist.

Eingabeparameter: FAC, ARG
Ausgabewerte: ARG

FAC RUNDEN

Zweck: FAC runden
Adresse: 41632 / $A2A0

Beschreibung: Der FAC wird gerundet. Die Nachkommastellen werden allerdings nicht abgeschnitten, sondern 'echt' gerundet. Das Ergebnis dieser Rundung steht im FAC und kann als INTEGER-Zahl weiter verarbeitet werden.

Eingabeparameter: FAC
Ausgabewerte: FAC (gerundet)

VORZEICHEN VON FAC ERMITTELN

Zweck: Vorzeichen des FACs ermitteln
Adresse: 41648 / $A2B0

Beschreibung: Wenn Sie das Vorzeichen des ersten Fließkommaakkumulators (FAC) benötigen, so bietet sich diese Routine an. Ist der Inhalt des FACs negativ, so gibt die Routine den Wert 255 ($FF) im Akkumulator zurück. Ist der Inhalt des FACs dagegen positiv oder gleich Null, so übergibt die Routine den Wert 1 ($01) im Akku.

Eingabeparameter: Keine
Ausgabewerte: .A

FAC MIT KONST. (A,Y) AUS RAM VERGL.

Zweck: FAC mit Konstante vergleichen
Adresse: 41696 / $A2E0

Beschreibung: Mittels dieser Routine können Sie testen, ob der Inhalt des Fließkommaakkus kleiner oder größer als eine Vergleichszahl ist. Die Vergleichszahl muß auch eine Fließkommazahl sein. Sind beide Zahlen gleich, so wird das Zeroflag gesetzt. Ist die Konstante kleiner als der Inhalt des Fließkommaakkus, so wird der Akku mit dem Wert 1 ($01) zurückgegeben. Ist die Konstante allerdings größer als die Fließkommazahl, die im FAC steht, so wird im Akku der Wert 255 ($FF) zurückgegeben.

Eingabeparameter: FAC, .A/.Y
Ausgabewerte: Zero, .A

FAC NACH INTEGER WANDELN

Zweck: FAC in eine Integerzahl wandeln
Adresse: 41767 / $A327

Beschreibung: Beim Aufruf dieser Routine wird der FAC gerundet und das Ergebnis als 16-Bit-Integerzahl in den beiden Speicherstellen 100/101 ($64/$65) abgelegt.

Eingabeparameter: FAC
Ausgabewerte: 100/101

STRING NACH FAC WANDELN

Zweck: String in Fließkommazahl wandeln
Adresse: 41855 / $A37F

Beschreibung: Mittels dieser Routine ist es möglich einen String in eine Integerzahl zu wandeln. Ein gültiger String wären z.B. die folgenden Beispiele:

```
1E20
568234.124E-10
usw.
```

Der String wird dann als Fließkommazahl im FAC abgelegt. Da die einzelnen Zeichen über die Routine CHRGET geholt werden, kann man die Adresse des ersten Zeichens durch die Speicherstellen 59/60 ($3C/$3B) ändern. Man sollte dann aber nicht vergessen, die Routine CHRGOT aufzurufen, damit der Kode des ersten Zeichens bereits im Akku steht, wenn die Wandelroutine aufgerufen wird. Der String muß übrigens durch ein Nullbyte abgeschlossen werden.

Eingabeparameter: 59/60
Ausgabewerte: FAC

INTEGERZAHL AUSGEBEN (AXOUT)

Zweck: Ausgeben einer Integerzahl
Adresse: 42079 / $A45F

Beschreibung: Mit dieser Routine können Sie 16-Bit-Integerwerte ausgeben. Die Integerzahl muß dazu im Akku und im X-Register stehen. Der Akkumulator enthält das LOW-Byte der Zahl und das X-Register das HIGH-Byte der Zahl. Die Zahl wird immer an der momentanen Cursorposition ausgegeben. Besonders geeignet ist diese Routine, um z.B. den Punktestand in einem Spiel auszugeben.

Eingabeparameter: .A/.X
Ausgabewerte: Keine

FAC NACH ASCII WANDELN

Zweck: Fließkommazahl in einen String wandeln
Adresse: 42095 / $A46F

Beschreibung: Diese Routine stellt genau das Gegenteil zur oben beschriebenen Routine zum Wandeln eines ASCII-Textes in eine Fließkommazahl dar. In dieser Routine wird nämlich der FAC als ASCII-Text im Prozessorstack ab der Adresse 256 ($0100) abgelegt. Der Text wird durch ein Nullbyte abgeschlossen.

Eingabeparameter: FAC
Ausgabewerte: 256ff.

FAC := ARG ^ FAC

Zweck: Potenzieren des FAC
Adresse: 42478 / $A5EE

Beschreibung: Diese Routine dient zum Potenzieren des ARG. Der ARG ist die Basis und der FAC der Exponent der Potenz. Das Ergebnis der Potenzierung steht im FAC.

Eingabeparameter: ARG, FAC
Ausgabewerte: FAC

VORZEICHENWECHSEL DES FACs

Zweck: Umdrehen des Vorzeichens vom FAC
Adresse: 42535 / $A627

Beschreibung: Das Vorzeichen des FAC wird umgedreht. War es positiv, so wird der FAC negativ und umgekehrt.

Eingabeparameter: FAC
Ausgabewerte: FAC

RENUMBER

Zweck: Umnummerieren eines BASIC-Programms
Adresse: 44045 / $AC0D

Beschreibung: Mittels dieser Routine können BASIC-Programme umnummeriert werden. Die neue Anfangszeilennummer steht in den beiden Speicherstellen 3/4 ($03/$04), die alte Anfangszeile in den Speicherstellen 90/91 ($5A/$5B) und die Schrittweite zwischen den einzelnen Zeilen in den Speicherstellen 5 und 6 ($05/$06).

Eingabeparameter: 3/4, 90/91, 5/6
Ausgabewerte: Keine

DELETE

Zweck: Löschen von BASIC-Programmteilen
Adresse: 44645 / $AE65
Beschreibung: Diese Routine entfernt bestimmte Programmteile aus BASIC-Programmen. Die Parameterübergabe entspricht der der LIST-Routine. Die LINK-Adresse der ersten zu löschenden Zeile muß also in den beiden Speicherstellen 95/96 ($5F/$60) stehen, während die letzte Zeilennummer in den Speicherstellen 20/21 ($14/$15) stehen muß. Es wird dann der Programmteil von der Zeile, auf die der Zeiger in 95/96 zeigte, bis zur Zeile,

die in 20/21 angegeben ist, inclusive dieser Zeile gelöscht. Ein Wiederherstellen des Programms ist nach dem Ausführen einer solchen Löschoperation ausgeschlossen.

Eingabeparameter: 95/96, 20/21
Ausgabewerte: Keine

HELP

Zweck: Anzeigen des fehlerhaften Teils einer BASIC-Programmzeile
Adresse: 46824 / $B6E8

Beschreibung: Diese Routine entspricht dem gleichnamigen BASIC-Befehl, sowohl in der Parameterübergabe (keine Parameter) als auch in der Funktion. Es werden nämlich ab dem fehlerhaften Teil einer BASIC-Programmzeile bis zu deren Ende alle Zeichen blinkend dargestellt.

Eingabeparameter: Keine
Ausgabewerte: Keine

SOUND

Zweck: Ton spielen (Stimme,Frequenz,Dauer)
Adresse: 47201 / $B861

Beschreibung: Diese Routine entspricht dem BASIC-Befehl SOUND. Mit dieser Routine ist es möglich, einen Ton in einer bestimmten Tonhöhe, mit einer bestimmten Dauer und auf einer bestimmten Stimme zu spielen.

Hier die Tabelle mit den entsprechenden Speicherstellen:

Soundtyp:	128 / $80	0-2
LOW-Tonhöhe:	126 / $7E	0-255
HIGH-Tonhöhe:	127 / $7F	0-3
Tondauer:	.A/.Y	

Für die Tonhöhe und die Tondauer ziehen Sie bitte das Handbuch zu Rate. Beim Austesten dieser Routine beachten Sie, daß die Lautstärke nicht gleich Null sein darf, da sonst kein Ton zu hören ist!

Eingabeparameter: 128, 126/127, .A/.Y
Ausgabewerte: Keine

LAUTSTÄRKE SETZEN (VOL)

Zweck: Setzen der Lautstärke
Adresse: 47296 / $B8C0

Beschreibung: Diese Routine erfüllt die Anforderung, daß die Lautstärke nicht den Wert Null haben darf. Sie setzt nämlich die Lautstärke auf den Wert, der im X-Register an diese Routine übergeben worden ist. Dieser Wert muß sich im Bereich von 0 bis 8 bewegen.

Eingabeparameter: .X
Ausgabewerte: Keine

PAINT

Zweck: Abgeschlossene Fläche in der HGR füllen
Adresse: 47327 / $B8DF

Beschreibung: Diese Routine bezieht sich auf die hochauflösende Grafik (HGR). Sie dient zum Füllen abgeschlossener Flächen durch eine bestimmte Farbquelle. Als Parameter muß ein Koordinatenpaar angegeben werden, das innerhalb der zu füllenden Fläche liegt, außerdem natürlich noch die Farbquelle. Zuletzt kann als Option noch ein Parameter angegeben werden, der dar-

über entscheidet, ob die Fläche durch die gleiche Farbe abgeschlossen sein muß, mit der diese auch gefüllt werden soll, oder ob sie durch jede beliebige andere Farbe, außer der Hintergrundfarbe, abgeschlossen sein kann.

Im weiteren entspricht diese Routine dem PAINT-Befehl des BASICs. Für nähere Informationen ziehen Sie daher bitte das Handbuch zum C16/C116/Plus/4 oder das ebenfalls im DATA-Becker Verlag erschienene Grafikbuch zu diesen Computern hinzu.

Die Parameter der PAINT-Routine werden in den folgenden Speicherstellen bzw. Registern abgelegt:

132 / $0084:	Farbquelle
689 / $02B1:	X-Koordinate LOW-Byte
690 / $02B2:	X-Koordinate HIGH-Byte
691 / $02B3:	Y-Koordinate LOW-Byte
692 / $02B4:	Y-Koordinate HIGH-Byte
X-Register:	Modus 0=gleicher Farbe wie in 132
	Modus 1=irgend eine Nicht-Hintergrundfarbe

Eingabeparameter:	132, 689/690, 691/692, .X
Ausgabewerte:	Keine

BOX

Zweck:	Rechteck in der HGR zeichnen
Adresse:	47867 / $BAFB

Beschreibung: Genau wie auch der BASIC-Befehl BOX, dient auch diese Routine zum Zeichnen eines Rechtecks innerhalb der hochauflösenden Grafik. Als Parameter werden die Koordinaten der linken oberen und der rechten unteren Ecke verlangt. Weiterhin noch die Farbquelle, der Drehwinkel und der Modus, der darüber entscheidet, ob das Rechteck ausgefüllt werden soll oder nicht. Hier die Speicherstellen, in denen die Parameter der

BOX-Routine abgelegt werden. Für die genaue Anwendungsweise nehmen Sie bitte wieder das Handbuch oder das Grafikbuch zur Hilfe:

132 / $0084: Farbquelle
716 / $02CC: X-Koordinate linke obere Ecke; LOW-Byte
717 / $02CD: X-Koordinate linke obere Ecke; HIGH-Byte
718 / $02CE: Y-Koordinate linke obere Ecke; LOW-Byte
719 / $02CF: Y-Koordinate linke obere Ecke; HIGH-Byte
720 / $02D0: Drehwinkel; LOW-Byte
721 / $02D1: Drehwinkel; HIGH-Byte (0-1)
728 / $02D8: X-Koordinate rechte untere Ecke; LOW-Byte
729 / $02D9: X-Koordinate rechte untere Ecke; HIGH-Byte
730 / $02DA: Y-Koordinate rechte untere Ecke; LOW-Byte
731 / $02DB: Y-Koordinate rechte untere Ecke; HIGH-Byte
X-Register: Zeichenmodus (0-1)

Eingabeparameter: 132, 716/717, 718/719, 720/721, 728/729, 730/731
Ausgabewerte: Keine

STRECKE ZEICHNEN (DRAW)

Zweck: Linie in der HGR zeichnen
Adresse: 49370 / $C0DA
Beschreibung: Diese Routine dient zum Zeichnen einer Linie zwischen zwei Punkten. Der Farbkode, mit dem gezeichnet werden soll, wird wie immer in der Adresse 132 abgelegt. Die Koordinaten der beiden Punkte sind wie folgt im Speicher abgelegt:

132 / $0084:	Farbkode
685 / $02AD:	X-Koordinate Anfang; LOW-Byte
686 / $02AD:	X-Koordinate Anfang; HIGH-Byte
687 / $02AE:	Y-Koordinate Anfang; LOW-Byte
688 / $02AF:	Y-Koordinate Anfang; HIGH-Byte
689 / $02B0:	X-Koordinate Ende; LOW-Byte

690 / $02B1: X-Koordinate Ende; HIGH-Byte
691 / $02B2: Y-Koordinate Ende; LOW-Byte
692 / $02B3: Y-Koordinate Ende; HIGH-Byte

Eingabeparameter: 132, 685/686, 687/688, 689/690, 691/692
Ausgabewerte: Keine

PUNKT TESTEN

Zweck: Grafikpunkt testen
Adresse: 49651 / $C1F3

Beschreibung: Diese Routine testet, ob an einer Position im Grafikspeicher ein Punkt gesetzt ist oder nicht. Ist in der Speicherstelle 139 ($8B) das 7. Bit gesetzt (1) (MULTICOLORMODUS), so wird entweder der Wert 0 ($00) oder der Wert 255 ($FF) im X-Register übergeben. Der Wert Null bedeutet, daß ein Punkt gesetzt war und der Wert 255, daß kein Punkt gesetzt war. War alledings das 7. Bit in der Speicherstelle 139 ($8B) gleich 0 (NORMALMODUS), dann wird der Zeichenmodus des Punktes im Akku zurückgegeben. Diese Werte können sich im Bereich von 0 bis 3 bewegen und haben folgende Bedeutung:

Wert	Funktion
0	Kein Punkt gesetzt
1	Normaler Punkt gesetzt
2	Multicolorfarbe 1
3	Multicolorfarbe 2

Die Koordinaten des Punktes, der getestet werden soll, werden in den folgenden Speicherstellen abgelegt:

685 / $02AD: X-Koordinate; LOW-Byte
686 / $02AE: X-Koordinate; HIGH-Byte
687 / $02AF: Y-Koordinate; LOW-Byte
688 / $02B0: Y-Koordinate; HIGH-Byte

Eingabeparameter: 685/686, 687/688
Ausgabewerte: .X bzw. .A

PUNKT SETZEN

Zweck: Punkt im Grafikspeicher setzen
Adresse: 49603 / $C1C3

Beschreibung: Diese Routine setzt oder löscht einen Punkt im Grafikspeicher. Die Farbquelle wird wie immer in der Speicherstelle 132 ($84) angegeben. Die Koordinaten ebenfalls wieder in den Speicherstellen von 685 ($02AD) bis 688 ($02B0):

132 / $0084: Farbquelle
685 / $02AD: X-Koordinate; LOW-Byte
686 / $02AE: X-Koordinate; HIGH-Byte
687 / $02AF: Y-Koordinate; LOW-Byte
688 / $02B0: Y-Koordinate; HIGH-Byte

Eingabeparameter: 132, 685/686, 687/688
Ausgabewerte: Keine

ADRESSE ÜBERNEHMEN

Zweck: Adresse aus BASIC-Text holen
Adresse: 50063 / $C38F

Beschreibung: Diese Routine dient - ähnlich wie die Routine GETADR - zum Holen einer 16-Bit-Zahl aus dem BASIC-Text; allerdings mit einer Erweiterung:

War nämlich keine Zahl vorhanden, sondern folgte nur ein Trennzeichen, so wird zwar der Wert Null in den Speicherstellen 20/21 ($14/$15) abgelegt, es wird aber außerdem noch das Carryflag gelöscht. Wurde dagegen eine Zahl gefunden, wird das Carryflag gesetzt. Durch das Testen des Carryflags nach Aufruf dieser Routine kann man also feststellen, ob überhaupt eine Zahl angegeben war.

Eingabeparameter: 59/60
Ausgabewerte: 20/21

SCNCLR

Zweck: Löschen des aktiven Bildschirms
Adresse: 50535 / $C567

TESTEN, OB GRAFIKMAP VORHANDEN

Zweck: Testen, ob Grafikspeicher vorhanden
Adresse: 51135 / $C7BF

Beschreibung: Diese Routine testet, ob schon ein Grafikspeicher eingerichtet worden ist. War dies nicht der Fall, so wird die BASIC-Fehlermeldung ?NO GRAPHICS AREA ausgegeben und der Programmablauf abgebrochen.
Eingabeparameter: Keine
Ausgabewerte: Keine

CLEAR SCREEN

Zweck: Löschen des aktuellen Windows
Adresse: 55435 / $D88B

Beschreibung: Das aktuell aktivierte Window wird gelöscht. Ist kein Window aktiv, wird des gesamte Bildschirm gelöscht. Innerhalb dieser Routine wird auch die Routine HOME aufgerufen.

Eingabeparameter: Keine
Ausgabewerte: Keine

HOME

Zweck: Cursor in die linke, obere Ecke setzen
Adresse: 55450 / $D89A

Beschreibung: Der Cursor wird in die linke, obere Ecke des aktuellen Windows gesetzt. Wird diese Routine zweimal hintereinander aufgerufen, so wird ein aktuelles Window nicht inaktiviert!

Eingabeparameter: Keine
Ausgabewerte: Keine

Außer den eben beschriebenen Routinen gibt es noch eine Anzahl Routinen, die zu den elementarsten Routinen Ihres Rechners gehören. Eben wegen dieser besonderen Wichtigkeit sind die Einsprungadressen aller dieser Routinen in einer sogenannten Sprungtabelle zusammengefasst. Von hier aus wird dann in die eigentlichen Routinen verzweigt. In Ihren Programmen sollten Sie **immer** die Einsprünge über diese Tabelle wählen, da diese bei allen Commodorerechnern in (fast) derselben Art und Weise angelegt ist. Ihre Programme können dadurch einfacher umgeschrieben werden und sind natürlich auch übersichtlicher. Hier nun die Routinen der KERNAL-Sprungtabelle:

CINT

Zweck: Initialisieren des Editors und des TED
Adresse: 65409 / $FF81

Beschreibung: Der Editior wird komplett neu initialisiert. Unter anderem wird in dieser Routine auch der Bildschirm gelöscht, ein eventuell vorhandenes Window zurückgesetzt und der Gänsefüßchenmodus inaktiviert. Weiterhin werden noch die Funktionstasten mit den Standardtexten belegt und der Tastaturpuffer gelöscht. Auch die Routine CLRCH wird aufgerufen.

Eingabeparameter: Keine
Ausgabewerte: Keine

IOINIT

Zweck: Auslösen eines RESET am IEC-BUS
Adresse: 65412 / $FF84

Beschreibung: Durch diese Routine wird die RESET-Leitung auf dem seriellen Bus (IEC-BUS) aktiviert. Alle Geräte, die an diesem angeschlossen sind, werden in den Ausgangszustand versetzt, wie nach dem Einschalten.

Eingabeparameter: Keine
Ausgabewerte: Keine

RAMTAS

Zweck: Ausführen eines BASIC-Warmstartes
Adresse: 65415 / $FF87
Beschreibung: Diese Routine initialisiert das RAM; es werden also die Speicherobergrenze (MEMTOP) und die Speicheruntergrenze (MEMBOT) festgelegt. Außerdem wird noch die Zeropage neu initialisiert und die Zeiger für den Kassetten- und den RS-232C-Puffer zurückgesetzt.

Eingabeparameter: Keine
Ausgabewerte: Keine

RESTOR

Zweck: Systemvektoren initialisieren
Adresse: 65418 / $FF8A

Beschreibung: Diese Routine kopiert die Vektorentabelle von 62187 ($F2EB) bis 62218 ($F30A) an die Adresse 786ff ($0312). Sie sollte aufgerufen werden, wenn Sie an den Vektoren herummanipuliert haben und wieder den Ausgangszustand herstellen wollen. Außerdem ruft sie die Routine VECTOR mit gelöschtem Carryflag auf.

Eingabeparameter: Keine
Ausgabewerte: 786ff

VECTOR

Zweck: Vektorentabelle kopieren
Adresse: 65421 / $FF8D

Beschreibung: Diese Routine kopiert die Vektorentabelle ab 786ff ($0312) an die Adresse, die im X- und Y-Register festgelegt ist, sofern das Carryflag gesetzt, also gleich 1 ist. Ist das Carryflag gleich 0, so wird der gleiche Prozess in umgekehrter Richtung vollzogen; die Vektorentabelle wird also ab der Adresse, die im X- und im Y-Register definiert ist, an die Adresse 786ff ($0312) kopiert.

Eingabeparameter: .X/.Y
Ausgabewerte: 786ff

SETMSG

Zweck: Ausgabe der DOS-Meldungen ein-/ausschalten
Adresse: 65424 / $FF90

Beschreibung: Innerhalb dieser Routine wird der Inhalt des Akkumulators in der Speicherstelle 154 ($9A) abgelegt. Ist das 7.

Bit in dieser Speicherstelle gesetzt, so ist die Ausgabe der DOS-Meldungen erlaubt; andernfalls werden diese Meldungen unterdrückt.

Eingabeparameter: .A
Ausgabewerte: Keine

SECOND

Zweck: Sekundäradresse nach LISTEN senden
Adresse: 65427 / $FF93

Beschreibung: Die an ein Peripheriegerät zu übergebende Sekundäradresse wird im Akku an diese Routine übergeben. Sie gibt die Sekundäradresse dann auf den seriellen Bus aus.

Eingabeparameter: .A
Ausgabewerte: Keine

TKSA

Zweck: Sekundäradresse nach TALK senden
Adresse: 65430 / $FF96

Beschreibung: Nach einem TALK-Signal kann mittels dieser Routine der Akkumulatorinhalt als Sekundäradresse an das durch TALK angesprochene Gerät gesendet werden.
Eingabeparameter: .A
Ausgabewerte: Keine

MEMTOP

Zweck: Setzen/Holen der Speicherobergrenze
Adresse: 65433 / $FF99

Beschreibung: Wird diese Routine mit gesetztem Carryflag aufgerufen, so wird die momentane Speicherobergrenze im X- und

im Y-Register (LOW/HIGH) übergeben. War das Carryflag gelöscht, wird die Speicherobergrenze mit dem Inhalt des X- und des Y-Registers (LOW/HIGH) belegt.

Eingabeparameter: .X/.Y (Carry gelöscht)
Ausgabewerte: .X/.Y (Carry gesetzt)

MEMBOT

Zweck: Setzen/Holen der Speicheruntergrenze
Adresse: 65436 / $FF9C

Beschreibung: Analog zur MEMTOP-Routine ist es mit dieser Routine auf die gleiche Art und Weise - wie auch schon oben beschrieben - möglich, die Speicheruntergrenze festzulegen bzw. diese zu ermitteln.

Eingabeparameter: .X/.Y (Carry gelöscht)
Ausgabewerte: .X/.Y (Carry gesetzt)

SCNKEY

Zweck: Tastaturabfrage
Adresse: 65439 / $FF9F

Beschreibung: Diese Routine ermittelt den ASCII-Kode einer eventuell gedrückten Taste und fügt diesen dem Tastaturpuffer an.

Eingabeparameter: Keine
Ausgabewerte: 1319ff

ACPTR

Zweck: Holen eines Bytes vom IEC-Bus
Adresse: 65445 / $FFA5

Beschreibung: Diese Routine holt ein Byte vom IEC-Bus. Das geholte Byte wird im Akku abgelegt. Die Routine bezieht sich immer auf ein mit TALK angesprochenes Gerät, von dem dann das Byte empfangen wird. Das Statusbyte an der Speicherstelle 144 ($90) wird entsprechend dem Ergebnis der Routine beeinflusst.

Eingabeparameter: Keine
Ausgabewerte: .A, 144

CIOUT

Zweck: Ausgabe eines Zeichens auf den IEC-Bus
Adresse: 65448 / $FFA8

Beschreibung: Diese Routine gibt ein im Akkumulator enthaltenes Byte auf den IEC-Bus aus. Das Byte wird hierbei an das mit LISTEN angesprochene Gerät übergeben. Auch durch diese Routine wird das Statusbyte (144/$90), entsprechend dem Ausgang der Routine, beeinflusst.

Eingabeparameter: .A
Ausgabewerte: 144

UNTLK

Zweck: UNTALK auf IEC-Bus ausgeben
Adresse: 65451 / $FFAB

Beschreibung: Diese Routine dient zum Schließen oder Umlegen eines mit TALK geöffneten Eingabekanals. Die Verbindung zwischen Computer und Periepheriegerät wird gewissermaßen abgebrochen.

Eingabeparameter: Keine
Ausgabewerte: Keine

UNLSN

Zweck: UNLISTEN auf IEC-Bus ausgeben
Adresse: 65454 / $FFAE

Beschreibung: Diese Routine dient, analog zur UNTLK-Routine, zum Schließen oder Umlegen eines Ausgabekanals. Die Verbindung zwischen Computer und dem empfangenden Gerät wird unterbrochen.

Eingabeparameter: Keine
Ausgabewerte: Keine

LISTN

Zweck: LISTEN auf den IEC-Bus ausgeben
Adresse: 65457 / $FFB1

Beschreibung: Mittels dieser Routine wird ein am IEC-Bus angeschlossenes Gerät aufgefordert,Daten vom Computer zu empfangen (LISTEN=Zuhören). Dazu wird die Geräteadresse des anzusprechenden Gerätes plus $20 im Akku übergeben. Ist die Verbindung zwischen dem Computer und dem empfangenden Gerät hergestellt, so können mittels der Routine CIOUT Daten an dieses Gerät übergeben werden. Eine Ausgabe der Daten durch BSOUT ist nicht möglich!

Da immer nur ein Gerät gleichzeitig angesprochen werden kann, können auch die Betriebssystemroutinen OPEN, CLOSE, BASIN, BSOUT etc. verwendet werden. Diese Routien übernehmen dann die Verwaltung mehrerer Geräte und senden auch die Signale LISTEN, TALK usw.

Die Verbindung zwischen Computer und LISTENER (empfangendes Gerät) kann durch die oben beschriebene Routine UNLSN abgebrochen werden.

Eingabeparameter: .A
Ausgabewerte: Keine

TALK

Zweck: TALK auf den IEC-Bus ausgeben
Adresse: 65460 / $FFB4

Beschreibung: Durch diese Routine wird ein Gerät am seriellen Bus aufgefordert zu senden (TALK=Sprechen). Die Geräteadresse plus $60 wird im Akku an diese Routine übergeben. Nachdem die Verbindung zwischen dem Computer und dem sendenden Gerät hergestellt ist, können mittels der Routine ACPTR Daten von diesem Gerät empfangen werden. Beendet wird die Übertragung durch das Ausführen der Routine UNTLK.

Eingabeparameter: .A
Ausgabewerte: Keine

READST

Zweck: Holen des I/O-Statusbytes
Adresse: 65463 / $FFB7
Beschreibung: Der aktuelle Systemstatus wird im Akku übergeben. War die RS-232C aktiv, so wird das Statusbyte übergeben und im Speicher gelöscht. Gegebenenfalls sollte es daher zwischengespeichert werden.

Eingabeparameter. Keine
Ausgabewerte: .A

SETLFS

Zweck: Fileparameter setzen
Adresse: 65466 / $FFBA

Beschreibung: Zusammen mit den SETNAM und der OPEN-Routine dient diese Routine zum Öffnen von Files. Diese Routine dient speziell zum Festlegen der Filenummer, der Geräteadresse und der Sekundäradresse. Die Filenummer muß im

Akkumulator, die Geräteadresse im X-Register und die Sekundäradresse im Y-Register übergeben werden.

Eingabeparameter: .A, .X, .Y
Ausgabewerte: Keine

SETNAM

Zweck: Parameter des Filenamens festlegen
Adresse: 65469 / $FFBD

Beschreibung: Mittels dieser Routine wird der Filenamen, mit dem ein File eröffnet werden soll, genau definiert. Wird kein Filenamen benötigt, muß für die Länge des Filenamens der Wert Null eingesetzt werden. Die Länge des Namens wird im Akku und die Anfangsadresse des Strings im X- und Y-Register abgelegt. Diese Routine wird immer im Zusammenhang mit der Routine SETFLS und OPEN angewendet.

Eingabeparameter: .A, .X/.Y
Ausgabewerte: Keine

OPEN

Zweck: Eröffnen einer Datei
Adresse: 65472 / $FFC0

Beschreibung: Diese Routine eröffnet ein File mit den Parametern, die durch die Routinen SETFLS und SETNAM festgelegt worden sind. Daher ist es notwendig, diese beiden Routinen immer vor der Routine OPEN aufzurufen. Die Parameter des zu öffnenden Files werden in der Tabelle der logischen Filenummern aufgenommen. Wegen dieser Tabelle ist es auch möglich, mehrere Files auf einmal geöffnet zu haben, daher sind diese Routinen auch den Routine TALK und LISTEN vorzuziehen.

Eingabeparameter: Keine
Ausgabewerte: Keine

CLOSE

Zweck: Schließen einer Datei
Adresse: 65475 / $FFC3

Beschreibung: Eine durch OPEN geöffnete Datei wird geschlossen. Sämtliche Angaben über dieses zu schließende File werden aus der Tabelle der logischen Filenummern entfernt. Die Nummer des Files, das geschlossen werden soll, wird im Akku übergeben. Traten Schwierigkeiten beim Schließen der Datei auf, erkennt man dies an einem gesetzten Carryflag.

Eingabeparameter: .A
Ausgabewerte: Carry

CHKIN

Zweck: File als Eingabedatei definieren
Adresse: Adresse: 65478 / $FFC6

Beschreibung: Im X-Register wird die Nummer des Files übergeben, von dem aus jetzt die Eingabe erfolgen soll. Das File, auf das sich die Routine CHKIN beziehen soll, muß natürlich schon existieren. Ist diese Bedingung erfüllt, so wird beim Aufruf der Routine BASIN nicht mehr ein Zeichen von der Tastatur, sondern vom angegebenen File geholt. Nach dem Aufruf der Routine CLRCH ist wieder die Tastatur das Eingabegerät. Auch bei dieser Routine zeigt ein gesetztes Carryflag an, daß ein Fehler aufgetreten ist.

Eingabeparameter: .X
Ausgabewerte: Carry

CKOUT

Zweck: File als Ausgabedatei definieren
Adresse: 65481 / $FFC9

Beschreibung: Analog zur Routine CHKIN erfolgt die Ausgabe nach dem Aufruf dieser Routine nicht mehr auf den Bildschirm, sondern auf die Datei, deren logische Filenummer im X-Register übergeben worden ist. Prinzipiell entspricht diese Routine also dem BASIC-Befehl CMD, da alle Zeichen, die mit BSOUT ausgegeben werden, in die Datei mit der Nummer X geschrieben werden. Tritt bei diesem Schreibvorgang ein Fehler auf, wird dies durch ein gesetztes Carryflag signalisiert.

Eingabeparameter: .X
Ausgabewerte: Carry

CLRCH

Zweck: Ein-/Ausgabe wieder zurückstellen
Adresse: 65484 / $FFCC

Beschreibung: Diese Routine setzt wieder die Tastatur als Eingabedatei und den Bildschirm als Ausgabedatei ein. Die eventuell vorher geöffneten Ein-/Ausgabekanäle werden nicht geschlossen!

Eingabeparameter: Keine
Ausgabewerte: Keine

BASIN

Zweck: Zeichen von der Eingabedatei holen
Adresse: 65487 / $FFCF

Beschreibung: Es wird ein Zeichen von der momentan als Eingabedatei definierten Datei geholt und im Akku übergeben. Im Normalfall ist die Eingabedatei die Tastatur. Man kann dies aber durch die Routine CHKIN ändern.

Eingabeparameter: Keine
Ausgabewerte: .A

BSOUT

Zweck: Zeichen auf Ausgabedatei ausgeben
Adresse: 65490 / $FFD2

Beschreibung: Es wird das Zeichen im Akku auf die durch CKOUT definierte Ausgabedatei ausgegeben. Ist CKOUT nicht aufgerufen worden, geschieht die Ausgabe auf den Bildschirm, wobei der ASCII-Kode in den POKE-Kode übersetzt wird.

Eingabeparameter: .A
Ausgabewerte: Keine

LOAD

Zweck: Einladen eines Speicherbereiches
Adresse: 65493 / $FFD5

Beschreibung: Bevor die Routine LOAD aufgerufen werden kann, muß die Datei mittels der Routinen SETFLS und SETNAM definiert werden. Die Routine OPEN muß nicht aufgerufen werden. Die Startadresse, an die das Programm geladen werden soll, wird im X- und im Y-Register an die LOAD-Routine übergeben. Das Programm wird dann eingeladen, wobei auch die Routine CLOSE automatisch aufgerufen wird. Die angegebene Filenummer ist beliebig.

Eingabeparameter: .X/.Y
Ausgabewerte: Keine

SAVE

Zweck: Speichern eines Speicherbereiches
Adresse: 65496 / $FFD8

Beschreibung: Bevor die Routine SAVE aufgerufen werden kann, müssen vorher alle Angaben zur Datei mittels der Routinen SETFLS und SETNAM gemacht worden sein. Danach kann die Routine SAVE aufgerufen werden. Der Akku enthält hierbei die Adresse der Zeropageadresse, in der die Anfangsadresse des Speicherbereichs steht. X- und Y-Register enthalten dann die Endadresse plus 1.

Eingabeparameter: .A, .X/.Y
Ausgabewerte: Keine

SETTIM

Zweck: Setzen der Systemuhr TI
Adresse: 65499 / $FFDB
Beschreibung: Diese Routine dient zum Stellen der Systemuhr TI. Die drei Werte, die benötigt werden, werden im Akku, im X-Register und im Y-Register übergeben, wobei das Y-Register das höherwertige Byte der 24-Stunden-Uhr enthält. Weil die Systemuhr vom Betriebssystem gesteuert wird, ist sie nicht sehr genau.

Eingabeparameter: .A/.X/.Y
Ausgabewerte: Keine

RDTIM

Zweck: Auslesen der Systemuhr TI
Adresse: 65502 / $FFDE

Beschreibung: Diese Routine dient zum Auslesen der Systemuhr. Die drei Bytes werden im Akku, im X-Register und im Y-Register abgelegt. Der Akkumulator enthält das niederwertigste Byte, während das Y-Register das höchstwertige Byte enthält.

Eingabeparameter: Keine
Ausgabewerte: .A/.X/.Y

STOP

Zweck: Auf STOP-Taste testen
Adresse: 65505 / $FFE1

Beschreibung: Wurde seit der letzten Tastaturabfrage die STOP-Taste betätigt, wird aus dieser Routine mit einem gesetzten Zeroflag zurückgesprungen und die Routine CLRCH ausgeführt. Wurde die STOP-Taste nicht gedrückt, ist auch das Zeroflag nach dem Aufruf dieser Routine nicht gesetzt.

Eingabeparameter: Keine
Ausgabewerte: Zero

GETIN

Zweck: Zeichen aus Tastaturpuffer oder RS-232C holen
Adresse: 65508 / $FFE4

Beschreibung: Diese Routine holt ein Zeichen aus dem Tastaturpuffer oder dem RS-232C-Puffer. Wurde kein Zeichen eingegeben, so wird der Wert Null im Akku übergeben; andernfalls der ASCII-Kode der Taste.

Eingabeparameter: Keine
Ausgabewerte: .A

CLALL

Zweck: Schließen aller offenen Dateien
Adresse: 65511 / $FFE7

Beschreibung: Durch diese Routine werden alle logischen Dateien gelöscht. Hierbei wird allerdings kein CLOSE durchgeführt, so daß eine Datei auf Diskette auch nach dem Aufruf von

CLALL noch als offene Datei in der Direktory erscheint. In der Routine CLALL wird ferner noch die Routine CLRCH aufgerufen.

Eingabeparameter: Keine
Ausgabewerte: Keine

UDTIM

Zweck: Systemuhr hochzählen
Adresse: 65514 / $FFEA

Beschreibung: Diese Routine zählt die Systemuhr TI um eine Einheit hoch. Sie wird nur in der IRQ-Routine aufgerufen. Daher ist die Systemuhr auch nicht sehr genau.

Eingabeparameter: Keine
Ausgabewerte: Keine

SCREEN

Zweck: Ermittlung der momentanen Fenstergröße
Adresse: 65117 / $FFED

Beschreibung: Diese Routine übergibt die maximale Spaltenzahl im X-Register und die maximale Zeilenzahl im Y-Register. Zusätzlich enthält der Akkumulator noch die maximal mögliche Spaltenzahl.

Eingabeparameter: Keine
Ausgabewerte: .A, .X, .Y

PLOT

Zweck: Setzen bzw. Holen der Cursorposition
Adresse: 55353 / $D839

Beschreibung: Wird diese Routine mit einem gelöschten Carryflag aufgerufen, wird der Cursor an die Position gesetzt, die im Y- und im X-Register festgehalten ist. Das Y-Register muß hierbei die X-Koordinate des Cursors und das X-Register die Y-Koordinate des Cursors beinhalten. Ist das Carryflag beim Aufruf dieser Routine allerdings gesetzt, so wird die aktuelle X-Koordinate im Y-Register und die aktuelle Y-Koordinate im X-Register an das Hauptprogramm zurückgegeben.

Eingabeparameter: .Y, .X bzw. Keine
Ausgabewerte: Keine bzw. .Y, .X

IOBASE

Zweck: Beginn des I/O-Bereiches ermitteln
Adresse: 65123 / $FFF3
Beschreibung: Diese Routine gibt die Anfangsadresse des I/O-Bereichs im X- (LOW) und Y-Register (HIGH) zurück. Diese Routine existiert nur aus Kompatibilitätsgründen und liefert natürlich immer den Wert 64768 ($FD00).

Eingabeparameter: Keine
Ausgabewerte: .X/.Y

ADRESSEN ALLER ROUTINEN:

01139 / $0473:	CHRGET
01145 / $0479:	CHRGOT
01172 / $0494:	LDA (...),Y
32878 / $800A:	Warmstart
32793 / $8019:	Kaltstart
32814 / $802E:	BASIC-Reset
32962 / $80C2:	Einschaltmeldung ausgeben
33047 / $8117:	Vektoren einrichten
34438 / $8686:	Fehlermeldung ausgeben
34619 / $873B:	BASIC-Programmzeile löschen
34706 / $B792:	Zeile einfügen

34840 / $8818: LINK-Adressen erzeugen
35008 / $88C0: Blockverschieberoutine
35077 / $8905: Test auf Platz im Stack
35155 / $8953: Tokenisieren
35389 / $8A3D: Zeile suchen
35351 / $8A7B: NEW
35482 / $8A9A: CLR
35586 / $8B02: LIST
35774 / $8BBE: RUN ohne Zeilennummerangabe
36017 / $8CB1: RESTORE ohne Zeilennummerangabe
36176 / $8D50: GOTO
35414 / $8E3E: String nach Integer wandeln
37000 / $9088: String ausgeben (TXTOUT)
37652 / $9314: FRMNUM
37655 / $9317: CHKNUM
37658 / $931A: CHKSTR
37676 / $932C: FRMEVL
39484 / $9A3C: Integermultiplikation
39570 / $9A92: Integer (A,Y) nach FAC
40321 / $9D81: GETBYT
40402 / $9DD2: GETPAR
40414 / $9DDE: GETADR mit Test auf Komma
40417 / $9DE1: GETADR ohne Test auf Komma
40583 / $9E87: FAC := ARG - FAC
40603 / $9E9B: FAC := Konst. (A,Y) aus RAM + FAC
40606 / $9E9E: FAC := ARG + FAC
40747 / $9F2B: FAC := 0
41058 / $A062: FAC := FAC + 0.5
41068 / $A06C: FAC := Konst. (A,Y) aus ROM - FAC
41074 / $A072: FAC := Konst. (A,Y) aus ROM / FAC
*41080 / $A078: FAC := Konst. (A,Y) aus RAM * FAC*
*41083 / $A07B: FAC := ARG * FAC*
41180 / $A0DC: Konst. (A,Y) aus ROM nach FAC
41223 / $A107: Konst. (A,Y) aus RAM nach ARG
*41314 / $A162: FAC := FAC * 10*
41347 / $A183: FAC := FAC / 10
41364 / $A194: FAC := Konst. (A,Y) aus RAM / FAC
41503 / $A21F: FAC := Konst. (A,Y) aus RAM
41505 / $A221: FAC := Konst. (A,Y) aus ROM
41561 / $A259: FAC nach RAM-Adresse (X,Y)

41601 / $A281: *ARG nach FAC*
41617 / $A291: *FAC nach ARG*
41632 / $A2A0: *FAC runden*
41648 / $A2B0: *Vorzeichen von FAC ermitteln*
41696 / $A2E0: *FAC mit Konst. (A,Y) aus RAM vergleichen*
41767 / $A327: *FAC nach Integer wandeln*
41855 / $A37F: *String nach FAC wandeln*
42079 / $A45F: *Integerzahl ausgeben (AXOUT)*
42095 / $A46F: *FAC nach ASCII wandeln*
42478 / $A5EE: *FAC := ARG ^ FAC*
42535 / $A627: *Vorzeichenwechsel des FACs*
44045 / $AC0D: RENUMBER
44645 / $AE65: DELETE
46824 / $B6E8: HELP
47201 / $B861: SOUND
47296 / $B8C0: VOL
47327 / $B8DF: PAINT
47867 / $BAFB: BOX
49370 / $C0DA: Strecke zeichnen (DRAW)
49651 / $C1F3: Punkt testen
49603 / $C1C3: Punkt setzen
50063 / $C38F: Adresse übernehmen
50535 / $C567: SCNCLR
51135 / $C7BF: Testen ob Grafikmap vorhanden
55435 / $D88B: Clear Screen
55450 / $D89A: HOME
65409 / $FF81: CINT
65412 / $FF84: IOINIT
65415 / $FF87: RAMTAS
65418 / $FF8A: RESTOR
65421 / $FF8D: VECTOR
65424 / $FF90: SETMSG
65427 / $FF93: SECOND
65430 / $FF96: TKSA
65433 / $FF99: MEMTOP
65436 / $FF9C: MEMBOT
65439 / $FF9F: SCNKEY
65445 / $FFA5: ACPTR
65448 / $FFA8: CIOUT

65451 / $FFAB:	UNTLK
65454 / $FFAE:	UNLSN
65457 / $FFB1:	LISTN
65460 / $FFB4:	TALK
65463 / $FFB7:	READST
65466 / $FFBA:	SETLFS
65469 / $FFBD:	SETNAM
65472 / $FFC0:	OPEN
65475 / $FFC3:	CLOSE
65478 / $FFC6:	CHKIN
65481 / $FFC9:	CKOUT
65484 / $FFCC:	CLRCH
65487 / $FFCF:	BASIN
65490 / $FFD2:	BSOUT
65493 / $FFD5:	LOAD
65496 / $FFD8:	SAVE
65499 / $FFDB:	SETTIM
65502 / $FFDE:	RDTIM
65505 / $FFE1:	STOP
65508 / $FFE4:	GETIN
65511 / $FFE7:	CLALL
65514 / $FFEA:	UDTIM
65517 / $FFED:	SCREEN
65520 / $FFF0:	PLOT
65123 / $FFF3:	IOBASE

Alle arithmetischen Routinen erkennen Sie an der Kursivschrift!

4.3 Die Vektoren des Betriebssystems

Um Änderungen der wichtigsten Betriebssystemroutinen leicht zu ermöglichen ohne gleich ganze Computerbauteile austauschen zu müssen, springen die wichtigsten Routinen über Vektoren. Als Vektoren bezeichnet man nichts anderes als zwei Speicherstellen, in denen die eigentliche Zieladresse enthalten ist. Über diese Vektoren wird dann mittels eines indirekten JMP-Befehls gesprungen. Durch die Änderung dieser Vektoren können wir die unmöglichsten Dinge möglich machen. Im C16 bzw. C116 bzw. Plus/4 liegt die Vektorentabelle im unteren Systemspeicher

und zwar von der Speicherstelle 754 ($02F2) bis zur Speicherstelle 817 ($0331). Welche Routinen über die einzelnen Vektoren springen, entnehmen Sie bitte der Zeropage im Anhang. Einen sehr wichtigen Vektor werden wir noch im Zusammenhang mit der Interruptprogrammierung kennenlernen.

4.4 Ein-/Ausgabe von Maschinenprogrammen aus

Wie Sie vielleicht schon der Beschreibung der ROM-Routinen entnommen haben, gibt es mehrere Prinzipien, Daten auf ein Gerät auszugeben (der Monitor und die Tastatur sind auch Geräte!). Als erstes gibt es dort die Möglichkeit, Daten mittels der Routinen TALK, LISTEN etc. auszugeben. Dies funktioniert allerdings nur am IEC-Bus und es kann immer nur ein Gerät gleichzeitig angesprochen werden. Aus diesem Grund gehen wir auch auf diese Möglichkeit der Datenausgabe nicht weiter ein.

Eine zweite Möglichkeit der Datenausgabe kennen Sie wahrscheinlich schon vom BASIC Ihres Rechners. Gemeint sind die sogenannten Files oder Dateien. Maximal können zehn Dateien gleichzeitig geöffnet sein. Dateien können zum Lesen und zum Schreiben geöffnet werden und sowohl den Bildschirm, als auch die Tastatur oder die Floppy (Recorder) ansprechen. Ein File wird immer durch eine sogenannte Filenummer gekennzeichnet. Anhand dieser Nummer weiß der Computer dann sofort, welches File gemeint ist. Praktisch können Sie sich dies so vorstellen:

Wenn Sie zu Ihrer Bank gehen und etwas von Ihrem Konto abheben wollen, geben Sie auch nicht Ihren Namen und Ihre Anschrift an, sondern einfach Ihre Kontonummer an. Anhand dieser Kontonummer kann nun sofort ermittelt werden, wie Sie heißen und wo Sie wohnen. Dadurch wird das gesamte Verfahren erheblich vereinfacht und auch schneller.

Analog hierzu können Sie sich auch die Fileoperationen vorstellen. Die Filenummer würde dann der Kontonummer entsprechen. Anhand dieser Nummer können dann alle anderen Daten des Files ermittelt werden. Diese anderen Daten, die Aufschluß

über die genaue Art und Weise des Files angeben, brauchen daher nur einmal, nämlich beim Anlegen des Files, angegeben werden.

Um ein File zu öffnen, benötigt der Computer die folgenden Informationen:

1. Die Filenummer
2. Die Gerätenummer
3. Die Sekundäradresse
4. Die Länge des Filenamens
5. Die Adresse des Filenamens

Der Filenamen wird nur benötigt, wenn Sie eine Datei auf Kassette oder Diskette anlegen wollen. Er entspricht dann dem Namen, den Sie auf dem Speichermedium ansprechen wollen.

Zum Übergeben der ersten drei Informationen dient die Routine SETFLS. Hiernach muß die Routine SETNAM aufgerufen werden, um den Namen des Files zu bestimmen. Wird kein Name benötigt, geben Sie einfach die Länge Null an (nähere Beschreibung der Routinen im Kapitel 4.2).

Gehen wir einmal davon aus, daß Sie auf einer Diskette ein File mit dem Namen "TEST" eröffnen wollen. Weiterhin soll das File zum Schreiben geöffnet werden. Wir gehen jetzt folgendermaßen vor:

```
LAB .ASC "TEST,S,W" ;Filenamen in ASCII-Kodes
                    ;',S,W' ist eine Floppyanweisung
                    ;und hat hiermit nichts zu tun!


LDA #$01    ;Akku mit Filenummer laden
LDX #$08    ;Geräteadresse der Floppy ist 8
LDY #$02    ;Sekundäradresse ist 2
JSR SETFLS  ;Die Parameter werden festgelegt
LDA #$08    ;Länge des Namens "TEST"
```

```
LDX #LO-LAB ;X-Register mit LOW-Byte der Adresse laden
LDY #HI-LAB ;Y-Register mit HIGH-Byte der Adresse laden
JSR SETNAM  ;Namen setzen
```

Nun haben Sie sämtliche Parameter für Ihr File festgelegt. Geöffnet ist es damit aber noch nicht. Dazu müssen Sie die Routine OPEN aufrufen:

```
JSR OPEN
```

Jetzt können Sie in die Datei schreiben. Das geschieht diesmal mit der Routine BSOUT. Aber Moment, BSOUT diente doch zur Ausgabe von Zeichen auf den Bildschirm?

Da dies tatsächlich so ist, müssen wir dem Computer also vorher noch angeben, daß er die Zeichen nicht mehr auf den Bildschirm, sondern auf unser File #1 ausgeben soll. Wenn Sie das Kapitel 4.2 aufmerksam gelesen haben, werden Sie sich vielleicht noch an eine Routine mit dem Namen CKOUT erinnern. Diese Routine erfüllt unsere oben geforderte Aufgabe, nämlich die Bildschirmausgabe auf ein File umzulenken. Als Parameter braucht nur die Filenummer übergeben zu werden:

```
LDX #$01  ;Filenummer
JSR CKOUT ;Ausgabe auf File 1
```

So, nun können wir wirklich Zeichen mit BSOUT in unsere Datei schreiben, auch die Routine TXTOUT können Sie verwenden, da diese zur eigentlichen Ausgabe auch BSOUT aufruft. Also:

```
    .
    .
    .
JSR BSOUT
    .
    .
    .
```

Sind Sie fertig und wollen die Datei abschließen, so rufen Sie die Routine CLOSE auf. Die Filenummer muß hierzu allerdings im Akku übergeben werden. Das Ganze schaut nun so aus:

```
LDA #$01  ;Filenummer
JSR CLOSE ;File schließen
```

Haben Sie dies ausgeführt, existiert kein File mit der Nummer 1 mehr, denn Sie haben es ja abgeschlossen und damit angezeigt, daß Sie nichts mehr in dieses File schreiben wollen. Um die Ausgabe nun wieder auf den Bildschirm zu lenken, und wieder die Tastatur als Eingabegerät zu setzen, gibt es die Routine CLRCH. Zuletzt müssen Sie also nur folgendes ausführen:

```
JSR CLRCH
```

Und fertig! Wollen Sie aus einer bereits bestehenden Datei lesen, so müssen Sie die folgenden Änderungen vornehmen:

CKOUT wird zu CHKIN
BSOUT wird zu BASIN

und, falls Sie mit Floppy arbeiten, wird das 'W' im Programmnamen zu einem 'R' für Lesen (READ). Ähnlich müssen Sie auch vorgehen, wenn Sie mit einem Recorder arbeiten. Dieser hat dann die Geräteadresse 1. Was speziell beim Recorder die einzelnen Sekundäradressen bewirken, entnehmen Sie bitte dem Handbuch zu diesem, oder dem Cassettenbuch, das im DATA-Becker Verlag erschienen ist.

Hier eine Tabelle mit den Geräteadressen:

0	Tastatur
1	Recorder
2	RS 232C (Falls vorhanden)
3	Bildschirm
4	Drucker (IEC-Bus)
5	Drucker (alternativ-Adresse)
6 - 15	Andere Geräte am IEC-Bus

z.B. 8: Floppy
Die Parameterübergabe bei den Routinen LOAD und SAVE erfolgt genauso wie oben beim Öffnen einer Datei. Es entfällt lediglich der Aufruf der OPEN und der CLOSE-Routine, auch CHKIN/CHOUT und BASIN/BSOUT können natürlich entfallen, da die beiden Routinen die Datenein-/ausgabe selbständig übernehmen.

5. Fortgeschrittene Programmierung

5.1 Interruptprogrammierung

Die Möglichkeit der Interruptprogrammierung ist einer der weiteren Vorzüge des 7501 Mikroprozessors. Um zu verstehen, wie dies funktioniert, hier erst einmal die Beschreibung dessen, was wir mit 'Interrupt' bezeichnet haben:

Unter einem Interrupt versteht man die gesteuerte Programmunterbrechung an einer beliebigen Stelle, verursacht durch irgendein Ereignis. Ist eine solche Unterbrechung ausgelöst worden, springt der Computer sofort über einen Vektor in die sogenannte Interruptroutine. Nach Beendigung dieser Interruptroutine springt er zum eigentlichen Programm zurück und fährt dort in gewohnter Weise fort. Ein Interrupt ist also eine gewollte und programmtechnisch vorgesehene Unterbrechung des Programms, wobei nach der Abarbeitung der Interruptroutine wieder an der alten Programmstelle fortgefahren wird. Der 7501-Prozessor kennt folgende vier Interruptmöglichkeiten:

NMI (Non Maskable Interrupt: nicht programmtechnisch auszuschaltender Interrupt)
BRK (Break)
IRQ (Interrupt Request)
Reset

a) NMI

Ein NMI wird ausgelöst durch einen Impuls am NMI-Eingang des Prozessors. Er kann softwaremäßig nicht unterdrückt werden, wie dies z.B. beim IRQ durch den Maschinensprachebefehl SEI möglich ist. Leider wird beim Auslösen eines NMI nicht über einen Vektor gesprungen, sondern direkt in die RESET-Routine des Rechners verzweigt. Die Konsequenz ist die gleiche

wie beim Drücken des RESET-Tasters. Den NMI können Sie also wieder vergessen, da im Betriebssystem bedauerlicherweise seine Programmierung nicht vorgesehen wurde.

b) BRK

Dieser Interrupt wird nicht hardwaremäßig gesteuert, sondern durch den Assemblerbefehl BRK ausgelöst. Der Prozessor fährt dann mit seiner Arbeit in der IRQ-Routine fort, in der getestet wird, ob es sich um einen IRQ oder um einen BRK handelt. Ist dies der Fall, wird über den Vektor 790/791 ($0316/$0317) gesprungen. Im Normalfall zeigt dieser Vektor auf den BREAK-Einsprung des Monitors.

c) IRQ

Der IRQ stellt in etwa die hardwaremäßige Parallele zum BRK Interrupt dar, nur daß er maskierbar ist im Gegensatz zum BRK. Wird er ausgelöst (z.B. durch TED), wird indirekt über die Vektoren 786/787 ($0314/$0315) in die Interruptroutine gesprungen. Maskiert werden kann der IRQ durch die Befehle SEI und CLI. Bei SEI wird ein eventuell auftretendes IRQ-Signal ignoriert.

d) Reset

Der Reset Interrupt kann nicht softwaremäßig unterdrückt werden und wird nach dem Einschalten des Computers oder nach dem Betätigen der Reset-Taste ausgelöst. Es wird, wie bei der NMI-Routine, direkt in die RESET-Routine gesprungen, daher können wir auch diesen Interrupt nicht programmieren. Die beiden Maschinensprachebefehle SEI und CLI haben keine Wirkung für diese Interruptart.

Wie oben zu lesen war, ist bedauerlicherweise nur der IRQ für uns programmierbar. Dieser IRQ wird circa alle sechzigstel Se-

kunde durch einen der Timer im TED aufgerufen. Innerhalb dieser Routine wird die SOUND-Warteschlange abgespielt, die Tastatur abgefragt, TI erhöht usw. Eventuell wird die IRQ-Routine durch einen Interrupt des Bildschirmcontrollers aufgerufen. Dies ist zwar wieder der TED, aber es besteht auch die Möglichkeit einen Interrupt auslösen zu lassen, wenn gerade eine bestimmte Rasterzeile durchlaufen wird. Dieser Interrupt wird z.B. bei der gemischten Grafik (Text/Bildschirm) angewendet. Nähere Auskünfte hierüber gibt das GRAFIKBUCH ZUM COMMODORE 128 (DATA-BECKER-VERLAG).

In die normale Interruptroutine kann man natürlich auch eigene Routinen mit einschleusen, indem man den oben genannten Vektor auf eine eigene Routine verbiegt und zum Abschluß dieser Routine wieder die eigentliche Interruptroutine an 52750 / $CE0E aufruft.

ACHTUNG!!!

Die gesamte Interruptroutine darf nicht länger als eine sechzigstel Sekunde dauern, da sonst innerhalb der Interruptroutine wieder ein Interrupt auftreten würde. Dieser würde zwar innerhalb der Interruptroutine noch nicht registriert werden, da mittels SEI das Interruptflag gesetzt wird. Nachdem der Interrupt allerdings wieder freigegeben worden ist (CLI), wird sofort ein Interrupt ausgelöst. Die Konsequenz: Nichts geht mehr!

Am besten betätigen Sie in einem solchen Fall die RUN/STOP-Taste und gleichzeitig den RESET-Knopf. Sie landen dann im Maschinensprachemonitor. Sollte es Ihnen nicht gelingen, den Monitor zu verlassen, so bleibt Ihnen nur noch der Griff zum Ausschalter.

Noch ein TIP: Während Sie den Vektor ändern, über den in die IRQ-Routine gesprungen wird, sollten Sie unbedingt den IRQ mittels SEI verhindern. Die Konsequenz, wenn ein Interrupt ausgelöst würde, wenn Sie gerade das LOW-Byte geändert hätten, wäre so peinlich wie wirkungsvoll. Ihr Computer springt in die Wüste. Hiermit ist natürlich nicht die Sahara oder die Wüste

Gobi gemeint, sondern die Programmwüste. Diese ist allerdings programmtechnisch mit einer normalen Wüste zu vergleichen: Es passiert äußerst selten, daß der Prozessor sie wieder verläßt, also wieder die oben beschrieben Notlösung.

Dieser Sprung in die Wüste oder der Absturz liegt übrigens meist daran, daß der Prozessor auf nicht definierte Maschinenbefehle stößt, also Befehle, die er nicht interpretieren kann. Man spricht dann auch davon, daß sich der Computer aufgehängt hat. Sie sehen, die Computerfachsprache strotzt nur so von "Fachbegriffen".

Geben Sie nun bitte das folgende Programm ein:
Es ändert die Bildschirmfarbe zyklische:

```
A 3000  SEI       ;Interrupt verhindern
A 3001  LDA #$0D  ;LOW-Byte neue IRQ-Routine
A 3003  LDY #$30  ;HIGH-Byte neue IRQ-Routine
A 3005  STA $0314 ;Als neuen Vektor speichern
A 3008  STY $0315 ;s.o.
A 300B  CLI       ;Interrupt wieder freigeben
A 300C  RTS       ;Wieder zurück zum BASIC
```

Neue IRQ-Routine:

```
A 300D  INC $FF19 ;Rahmenfarbe erhöhen
A 3010  JMP $CE0E ;Zur alten IRQ-Routine
```

Nun können Sie die neue Interruptroutine mittels dieser Anweisung aktivieren:

SYS DEC ("3000") (RETURN-Taste)

5.2 Die Rechenroutinen des Interpreters

Wie Sie sich sicherlich denken können, benötigt der Computer die Möglichkeit, mit Zahlen zu rechnen. Als erstes bietet sich ihm dort die Zahlenverarbeitung mittels Integerzahlen. Mit Integerzahlen kann man aber nur ganze Zahlen darstellen. Will

man zwei Bytes pro Integerzahl belegen, so lassen sich maximal Zahlen im Bereich zwischen 0 und 65535 darstellen. Da diese Zahlenformat aber oft nicht ausreicht, existieren noch die sogenannten Fließkommazahlen. Eine Fließkommazahl ist wie folgt aufgebaut:

EX M1 M2 M3 M4 VZ

EX: Exponent
M1: Mantisse 1
M2: Mantisse 2
M3: Mantisse 3
M4: Mantisse 4
VZ: Vorzeichen

Genauer soll im Rahmen dieses Buches nicht auf die speicherinterne Verwaltung der Fließkommaarithmetik eingegangen werden, da auch ohne diese Kenntnisse die Fließkommaarithmetik recht gut genutzt werden kann. Wollen Sie sich aber weitergehend informieren, so kann ich Ihnen das Maschinensprachebuch für Fortgeschrittene zum Commodore 64 wärmstens empfehlen. Dort erfahren Sie dann die genauern Einzelheiten. Nun zu den Dingen, die Sie aber grundsätzlich doch wissen sollten:

Fließkommazahlen sind Zahlen wie z.B.:

1E24 oder 25243465.356873E-11

aber auch negative Zahlen:

-5 oder -5.234245E-11

Auch Nachkommastellen sind, wie Sie oben sehen können durchaus möglich. Der Zahlenbereich umfasst circa -1E36 bis +1E36, also schon eine ganze Menge. Um nun die im Kapitel 4.2 beschriebenen Routinen nutzen zu können, müssen Sie erst einmal wissen, was ein FAC und was ein ARG ist.

Der FAC (Fließkommaakkumulator) besteht aus mehreren Speicherstellen, in denen sich eine komplette Fließkommazahl speichern läßt. Der FAC ist praktisch das Hauptrechenregister für die Fließkommaarithmetik. Weiterhin existiert noch der ARG (Argument), der immer das Argument einer mathematischen Verknüpfung enthält. Mittels der Unzahl an Fließkommaroutinen können Sie mit Fließkomma oder auch Gleitkommazahlen hantieren, ohne direkt wissen zu müssen, was hierbei eigentlich vorgeht. Um z.B. eine Fließkommazahl im Speicher abzulegen, existiert eine Routine, die einen Zahlenstring (z.B. 1244.5634) in eine Fließkommazahl wandelt. Desweiteren gibt es auch Routinen zum Wandeln einer Fließkommazahl in eine Integerzahl. Durch die geschickte Anwendung aller dieser Routinen können Sie sich das Programmieren in Maschinensprache sehr vereinfachen.

5.3 Das Banking

Sind Sie der stolze Besitzer eines Commodore 16 oder 116, so werden Sie wahrscheinlich nie mit dem Banking zu tun haben, es sei denn, Sie erweitern Ihren Rechner z.B. auf 64K. Haben Sie dagegen einen Plus/4, so ist das unten stehende Kapitel schon wieder sehr viel interessanter. Aus diesem Grund wird weiter unten auch nur vom Plus/4 gesprochen, haben Sie allerdings einen erweiterten C16 oder C116, so gilt das Untenstehende natürlich auch für Sie:

Die Umschaltung zwischen RAM und ROM wird durch zwei Speicherstellen des TED-Chip vorgenommen. Schreiben Sie einen beliebigen Wert in die Speicherstelle 65342 ($FF3E), so wird ab 32768 ($8000) das ROM eingeschaltet. Schreibt man dagegen einen beliebigen Wert in die Speicherstelle 65343 ($FF3F), dann ist für den Prozessor dort nur RAM vorhanden (gilt auch für C16/116, falls erweitert).

Da der Plus/4 bereits eingebaute Software enthält, ist die Anzahl der ROM-Bereiche auch weitaus größer als beim C16 oder C116. Der gesamte Speicher läßt sich nun folgendermaßen in 16 verschiedene Konfigurationen einteilen:

0	BASIC	+ KERNAL
1	eingebaute Software LOW	+ KERNAL
2	Modul LOW	+ KERNAL
3	reserviert LOW	+ KERNAL
4	BASIC	+ eingebaute Software HIGH
5	eingebaute Software LOW	+ eingebaute Software HIGH
6	Modul LOW	+ eingebaute Software HIGH
7	reserviert LOW	+ eingebaute Software HIGH
8	BASIC LOW	+ Modul HIGH
9	eingebaute Software LOW	+ Modul HIGH
A	Modul LOW	+ Modul HIGH
B	reserviert LOW	+ Modul HIGH
C	BASIC	+ reserviert HIGH
D	eingebaute Software LOW	+ reserviert HIGH
E	Modul LOW	+ reserviert HIGH
F	reserviert LOW	+ reserviert HIGH

Um ein Byte aus einer anderen BANK zu holen, existiert die Routine LONG-FETCH. Die Parameterübergabe ist wie folgt:

Basis-Adresse des gesuchten Bytes:	$BE/$BF
Adresse-Offset:	.Y
Nr. Zielbank:	.X
Nr. momentane Bank:	.A

Um Routinen in einer anderen Bank aufzurufen, existiert ebenfalls eine Routine und zwar die LONG-JUMP-Routine. Die Parameterübergabe ist wie folgt:

Übergabe Akku:	$05F2
Übergabe X-Reg.:	$05F3
Übergabe Status:	$05F4

Sprungadresse: $05F0/$05F1
Nr. Zielbank: .X
Nr. momentane Bank: .A

Im Maschinensprachemonitor ist die Umschaltung zwischen ROM und RAM durch die Speicherstelle 2040 ($07F8) möglich. $00 = ROM; $80 = RAM

6. Beispielprogramme

6.1 Das Bildschirm-Kopier-Programm

Um in Maschinensprache zu programmieren reicht es nicht aus, nur den kompletten Befehlssatz des jeweiligen Prozessors zu kennen. Man benötigt weiter noch einen gewissen Fundus an Programmiertechniken, z.B. Schleifenkonstruktionen etc..

Einen Teil dieser Programmiertechniken haben wir schon besprochen und zwar haben wir mehrere Möglichkeiten der Zeichenausgabe kennengelernt. Hierbei lag der besondere Schwerpunkt auf den Schleifenkonstruktionen und den Adressierungsarten der Maschinensprachebefehle. Damit Sie nun einmal sehen können wie Maschinenprogramme für eine bestimmte Problemlösung aussehen können, werden ich Ihnen im folgenden zwei Programm vorstellen, die über einen gewissen Lehreffekt hinaus auch noch recht praktisch sind.

Es handelt sich hierbei zuerst um ein Programm, das den Inhalt des normalen Textschirms (Buchstaben, Grafikzeichen usw.) in den Grafikspeicher (Einzelpunktspeicher) kopiert. Hierbei werden auch die Zeichenfarben und Helligkeitswerte mit berücksichtigt. Bevor Sie allerdings die weiter untenstehende Maschinenroutine aufrufen, müssen Sie erst mittels des Befehls

GRAPHIC 1,1

den Grafikspeicher einrichten und löschen. Dies ist notwendig, da hierbei unter anderem die Hintergrundfarbe für die nicht gesetzten Punkte festgelegt wird. Diese wird durch die unten stehende Routine nämlich nicht berücksichtigt. Weiterhin werden auch möglicherweise bestehende Windows ignoriert, es kann also immer nur der ganze Textschirm kopiert werden.

Sicherlich werden Sie sich jetzt fragen wozu ein solches Programm wohl dienen kann:

Wollen Sie in der höchauflösenden Grafik ein Bild erstellen, so sind hierzu eine Unzahl an Grafikbefehlen notwendig, während man mit der Textgrafik wesentlich einfacher umgehen kann. Allerdings gibt es Anwendungsbereiche, wo man beides gerne miteinander kombinieren will. Hier bleibt einem dann nur der Griff zum 'CHAR'-Befehl oder dem weiter untenstehenden Utility. Als erstes hier das disassemblierte Listing unseres kleinen Hilfsprogrammes. Weiter unten finden Sie dann auch noch einen DATA-Lader. Sie brauchen also nicht das Mnemoniclisting eingeben, wenn Sie hierzu keine Lust haben.

Disassemblerlisting:

```
A 0332  JSR $C7BF   ;Testen ob Grafikspeicher vorhanden
A 0335  LDA #$00    ;LOW-Byte Text-Farbspeicher
A 0337  LDY #$08    ;HIGH-Byte Text-Farbspeicher
A 0339  STA $D8     ;in $D8/$D9 für indirekt indizierte
A 033B  STY $D9     ;Adressierung speichern
A 033D  LDY #$1C    ;HIGH-Byte HGR-Farbspeicher
A 033F  STA $DA     ;in $DA/$DB für indirekt indizierte
A 0341  STY $DB     ;Adressierung speichern
A 0343  LDY #$18    ;HIGH-Byte HGR-Lunminanzspeicher
A 0345  STA $DC     ;in $DC/$DD für indirekt indizierte
A 0347  STY $DD     ;Adressierung speichern
A 0349  LDX #$04    ;Zähler für äußere Schleife (4 mal)
A 034B  LDY #$FA    ;Zähler für innere Schleife (250 mal)
                    ;250 * 4 = 1000
A 034D  LDA ($D8),Y ;Farbe + Lum. aus Textfarbspeicher holen
A 034F  PHA         ;Farbe + Luminanz auf Stack legen
A 0350  ASL         ;untersten vier Bits um vier Positionen
A 0351  ASL         ;nach links verschieben
A 0352  ASL         ;Es steht dann nur noch die Farbe im
A 0353  ASL         ;Akku. Diese belegt die Bits 4-7
A 0354  STA $DE     ;Farbe zwischenspeichern
A 0356  PLA         ;Farbe + Lum. vom Stack holen
A 0357  AND #$7F    ;Flash-Bit ausfiltern
A 0359  LSR         ;Bits 4-6 um vier Positionen nach links
A 035A  LSR         ;verschieben. Die Luminanz belegt dann
A 035B  LSR         ;die Bits 0-2
```

```
A 035C  LSR          ;
A 035D  STA $DF      ;Luminanz zwischenspeichern
A 035F  LDA ($DA),Y  ;Vorder- und Hintergrundfarbe für einen
                     ;8*8-Block der HGR holen
A 0361  AND #$0F     ;Nur Farbe der 0-Bits zulassen
A 0363  ORA $DE      ;Farbe der 1-Bits mittels OR zufügen
A 0365  STA ($DA),Y  ;im Farbspeicher der HGR ablegen
A 0367  LDA ($DC),Y  ;Vorder- und Hintergrundluminanz für
                     ;einen 8*8-Block der HGR holen
A 0369  AND #$F0     ;Nur Luminanz der 0-Bits zulassen
A 036B  ORA $DF      ;Luminanz der 1-Bits mittels OR zufügen
A 036D  STA ($DC),Y  ;im Luminanzdspeicher der HGR ablegen
A 036F  DEY          ;Inneren Schleifenzähler minus 1
A 0370  CPY #$FF     ;Innere Schleife fertig?
A 0372  BNE $034D    ;Nein, dann weiter in innerer Schleife
A 0374  CLC          ;Ja, dann Carry für Addition vorbereiten
A 0375  LDA $DA      ;LOW-Byte HGR-Farbspeicher
A 0377  ADC #$FA     ;250 addieren
A 0379  STA $DA      ;wieder abspeichern
A 037B  BCC $0380    ;Überlauf? Nein, dann Sprung
A 037D  INC $DB      ;Ja, dann HIGH-Byte erhöhen
A 037F  CLC          ;Carry für Addition vorbereiten
A 0380  LDA $DC      ;LOW-Byte HGR-Luminanzspeicher
A 0382  ADC #$FA     ;250 addieren
A 0384  STA $DC      ;wieder abspeichern
A 0386  BCC $038B    ;Überlauf? Nein, dann Sprung
A 0388  INC $DD      ;Ja, dann HIGH-Byte erhöhen
A 038A  CLC          ;Carry für Addition vorbereiten
A 038B  LDA $D8      ;LOW-Byte Text-Farbspeicher
A 038D  ADC #$FA     ;250 addieren
A 038F  STA $D8      ;wieder abspeichern
A 0391  BCC $0395    ;Überlauf? Nein, dann Sprung
A 0393  INC $D9      ;Ja, dann HIGH-Byte erhöhen
A 0395  DEX          ;Äußeren Schleifenzähler minus eins
A 0396  BNE $034B    ;Ungleich Null, dann Sprung
A 0398  TXA          ;Akku auf Null setzen
A 0399  LDY #$20     ;HIGH-Byte Grafikspeicher
A 039B  STA $D8      ;in $D8/$D9 für indirekt indizierte
A 039D  STY $D9      ;Adressierung speichern
A 039F  LDY #$0C     ;HIGH-Byte Text-Videoram
```

```
A 03A1  STA $DA      ;in $DA/$DB für indirekt indizierte
A 03A3  STY $DB      ;Adressierung speichern
A 03A5  LDY #$00     ;Index für indirekt indizierte
                     ;Adressierung auf Null setzen
A 03A7  STY $DD      ;HIGH-Byte des Zeichens im CHAR-ROM
A 03A9  LDA ($DA),Y  ;Zeichen aus Text-Videoram holen
A 03AB  STA $DE      ;in $DE zwischenspeichern
A 03AD  AND #$7F     ;Reversflag ignorieren
A 03AF  STA $DC      ;LOW-Byte des Zeichens im CHAR-ROM
A 03B1  ASL $DC      ;
A 03B3  ROL $DD      ;
A 03B5  ASL $DC      ;Adresse mit 8
A 03B7  ROL $DD      ;multiplizieren
A 03B9  ASL $DC      ;Carry bleibt auf Null, da das HIGH-Byte
A 03BB  ROL $DD      ;vor der Verschiebung gleich 0 war
A 03BD  LDA $DD      ;HIGH-Byte der Adresse des Zeichens im
                     ;CHAR-ROM (0-1024)
A 03BF  ADC #$D0     ;HIGH-Byte der CHAR-ROM-BASIS-Adresse
                     ;addieren
A 03C1  STA $DD      ;als neues HIGH-Byte speichern
A 03C3  LDY #$07     ;Bytezähler beim Kopieren des Zeichens
                     ;vom CHAR-ROM in den Punktspeicher
A 03C5  LDA ($DC),Y  ;Bitmuster aus CHAR-ROM holen
A 03C7  LDX $DE      ;Zeichen aus Videoram
A 03C9  BPL $03CD    ;Wenn nicht revers, dann Sprung
A 03CB  EOR #$FF     ;Revers, dann Bitmuster invertieren
A 03CD  STA ($D8),Y  ;Bitmuster in Punktspeicher schreiben
A 03CF  DEY          ;Bytezähler erniedrigen
A 03D0  BPL $03C5    ;Größer als Null? Ja, dann Sprung
A 03D2  CLC          ;Carry für Addition vorbereiten
A 03D3  LDA $D8      ;LOW-Byte HGR-Punktspeicher
A 03D5  ADC #$08     ;8 addieren
A 03D7  STA $D8      ;als neues LOW-Byte setzen
A 03D9  BCC $03DD    ;Überlauf? Nein, dann Sprung
A 03DB  INC $D9      ;Ja, dann HIGH-Byte erhöhen
A 03DD  INC $DA      ;Zeiger im Text-Videoram-LOW plus eins
A 03DF  BNE $03E3    ;Überlauf? Nein, dann Sprung
A 03E1  INC $DB      ;Ja, dann auch HIGH-Byte erhöhen
A 03E3  LDA $DB      ;HIGH-Byte Text-Videoram
A 03E5  CMP #$0F     ;Schon letzte Page?
```

```
A 03E7  BNE $03A5   ;Nein, dann weiter machen
A 03E9  LDA $DA     ;Ja, dann LOW-Byte testen
A 03EB  CMP #$E8    ;LOW-Byte Text-Videoram schon $E8?
A 03ED  BNE $03A5   ;Nein, dann weiter machen
A 03EF  RTS         ;Ja, dann fertig. Wieder zum BASIC
```

Um die Funktionsweise des Programmes zu verstehen muß man zunächst in einige Hardwaregrundlagen eingeweiht sein. Diese Hardwaregrundlagen können an dieser Stelle leider nicht ausführlich besprochen werden, das sie den Rahmen eines Maschinensprachebuches sprengen würden. Wollen Sie also genauere Informationen über die hochaufgelöste Grafik erhalten, so kann ich Ihnen nur das Grafikbuch zu Ihrem Computer empfehlen, das ebenfalls im DATA-Becker-Verlag erschienen ist.

Wie Sie wissen hat der Textschirm eine Auflösung von 40*25 gleich 1000 Zeichen. Im Gegensatz hierzu besitzt die HGR eine Auflösung von 320*200 gleich 64000 Punkten, von denen jeweils 8 durch ein Byte angeordnet sind. Für ein Zeichen im Textspeicher ist die Farbe und die Helligkeit (Luminanz) in einem speziellen Speicher, dem Textfarbspeicher untergebracht. Die Zeichenfarbe belegt hierbei die untersten 4 Bits (0-3), die Helligkeit die Bits 4-6 und das Flashflag (Blinkflag), das 7. Bit. Bei der HGR ist die Sache schon etwas komplizierter:

Hier liegen nicht alle Speicherstellen auch auf dem Bildschirm nebeneinander sondern jeweils acht untereinander. Für dieses Achterpäckchen existiert nun eine spezifische Speicherstelle im HGR-Farbram und im HGR-Luminanzram. Die untersten vier Bits dieser beiden Speicherstellen geben immer die Farbe bzw. Luminanz des 0-Bits, also des gelöschten Punktes an, während die restlichen Bits für die 1-Bits, also die gesetzten Punkte zuständig sind. Ist man mit diesem Vorwissen ausgerüstet, versteht man auch die Notwendigkeit der verschiedenen Verschiebeoperationen, die dazu dienen die verschiedenen Speicherprizipien einander anzugleichen.

Zum Abschluß dieses Kapitels hier noch der versprochene DATA-Lader:

```
100 rem ************************
110 rem * screen-copy-program *
120 rem * ------------------- *
130 rem * c16/c116/plus 4     *
140 rem * d.vuellers mai  '86 *
150 rem ************************
160 :
170 for i=818 to 1007
180 :   read a
190 :   ps=ps+a
200 :   poke i,a
210 next i
220 :
230 if ps<>27745 then print "fehler in datas !!!":end
240 :
250 print "ok! programmstart nit sys 818"
260 :
270 data 32,191,199,169,0,160,8,133
280 data 216,132,217,160,28,133,218,132
290 data 219,160,24,133,220,132,221,162
300 data 4,160,250,177,216,72,10,10
310 data 10,10,133,222,104,41,127,74
320 data 74,74,74,133,223,177,218,41
330 data 15,5,222,145,218,177,220,41
340 data 240,5,223,145,220,136,192,255
350 data 208,217,24,165,218,105,250,133
360 data 218,144,3,230,219,24,165,220
370 data 105,250,133,220,144,3,230,221
380 data 24,165,216,105,250,133,216,144
390 data 2,230,217,202,208,179,138,160
400 data 32,133,216,132,217,160,12,133
410 data 218,132,219,160,0,132,221,177
420 data 218,133,222,41,127,133,220,6
430 data 220,38,221,6,220,38,221,6
440 data 220,38,221,165,221,105,208,133
450 data 221,160,7,177,220,166,222,16
460 data 2,73,255,145,216,136,16,243
470 data 24,165,216,105,8,133,216,144
480 data 2,230,217,230,218,208,2,230
490 data 219,165,219,201,15,208,188,165
500 data 218,201,232,208,182,96
```

6.2 REM-Killer

Wie schon weiter oben angesprochen, soll die Maschinensprache bei Ihnen die Programmierung in BASIC nicht völlig ablösen, sondern nur sinnvoll ergänzen. Daher wäre es doch ganz praktisch, ein Programm zu haben, das Ihre BASIC-Programme sozusagen "automatisch" optimiert. Unter optimieren verstehe ich in diesem Zusammenhang das Entfernen von Kommentaren, LET-Befehlen und überflüssigen Leerzeichen. Das sich diese Optimierung allerdings nur sehr schlecht mit der allgemein geforderten Übersichtlichkeit von Programmen vereinbaren läßt, sollte die Optimierung immer der letzte Schritt bei der Programmierung sein.

Ein solches Programm wie es oben beschrieben wurde, finden Sie nun weiter unten. Es erfüllt alle Aufgaben, die eben bei der Optimierung genannt wurden. Hierbei sollten allerdings folgende Punkte beachtet werden:

Springen Sie niemals mit einem Sprungbefehl zu einer Zeile, die nur einen REM-Befehl enthält. Diese Zeile wird nämlich bei der Optimierung völlig gelöscht; daher findet der BASIC-Interpreter sie auch nicht mehr und tut dies auch entsprechend kund.
Im wesentlichen werden im untenstehenden Programm nur Routinen verwendet die bereits besprochen wurden. Zur Auffrischung Ihrer Kenntnisse sollten Sie sich vielleicht noch einmal die entsprechenden Kapitel zu Gemüte führen. Ansonsten hoffe ich, ist die Dokumentierung des Maschinenprogramms aufschlußreich genug.

Hier nun das Disassemblerlisting:

```
A 0332  LDA $2B     ;Programmanfang-LOW (MEMBOT)
A 0334  LDY $2C     ;Programmanfang-HIGH (MEMBOT)
A 0336  STA $DA     ;Speichern in $DA/$DB für indirekt
A 0338  STY $DB     ;indizierte Adressierung
```

```
A 033A  LDY #$00    ;Index für indirekt indizierte
                    ;Adressierung aus RAM gleich 0
A 033C  JSR $040A   ;Byte mittels indirekt indizierter
                    ;Adressierung aus dem RAM holen (s.u.)
                    ;Geholtes Byte ist LINK-Adresse-LOW
A 033F  STA $DE     ;Byte in $DE speichern
A 0341  INY         ;Index erhöhen (1)
A 0342  JSR $040A   ;LINK-Adresse-HIGH holen
A 0345  ORA $DE     ;und mit LINK-Adresse-LOW Oderver-
                    ;knüpfen. Waren beide Null ist das
                    ;Ergebnis anschließend auch Null
A 0347  BNE $034C   ;Bei ungleich Null weiter machen
A 0349  JMP $800A   ;LINK-Adresse war Null, also Ende der
                    ;BASIC-Programms erreicht. Daher
                    ;Programm mittels Sprung zum Warmstart
                    ;beenden da bei Zeileneinfügen der
                    ;Stack neu gesetzt wird und daher die
                    ;Rücksprungadresse verloren geht
A 034C  INY         ;Index erhöhen (2)
A 034D  JSR $040A   ;Byte holen. Zeilennummer-LOW
A 0350  STA $14     ;Speichern für Adressenberechnung
A 0352  INY         ;Index erhöhen (3)
A 0353  JSR $040A   ;Byte holen. Zeilennummer-HIGH
A 0356  STA $15     ;Speichern für Adressenberechnung
A 0358  LDX $14     ;Zeilennummer-LOW ins XR für XAOUT
A 035A  LDA $15     ;Zeilennummer-HIGH in Akku für XAOUT
A 035C  JSR $A45F   ;XAOUT. 16-Bit-Integerzahl ausgeben
A 035F  LDA #$0D    ;ASC 13, Carriage-Return
A 0361  JSR $FFD2   ;Über BSOUT ausgeben
A 0364  LDA $DA     ;Zeiger auf LINK-Adresse-LOW
A 0366  LDY $DB     ;Zeiger auf LINK-Adresse-HIGH
A 0368  STA $DC     ;Beide speichern, für den Fall, daß die
A 036A  STY $DD     ;gesamte Zeile gelöscht wird. Der Zeiger
                    ;muß für diesen Fall dann nämlich nicht
                    ;neu berechnet werden
A 036C  CLC         ;Carry für Addition vorbereiten
A 036D  LDA $DA     ;Zeiger auf LINK-Adresse-LOW
A 036F  ADC #$04    ;Vier addieren. Zeiger zeigt nun auf
                    ;den eigentlichen Programmtext, da die
                    ;LINK-Adresse und die Zeilennummer
```

```
                        ;übersprungen werden
A 0371  STA $DA         ;Neue LINK-Adresse-LOW speichern
A 0373  BCC $0377       ;Sprung wenn kein Überlauf auftrat
A 0375  INC $DB         ;Überlauf, daher LINK-Adresse-HIGH + 1
A 0377  LDA #$01        ;Korrekturflag zurücksetzen. Das Flag
                        ;ist LOW-Aktiv. D.h. wenn es 0 ist, ist
                        ;eine Korrektur erfolgt, wenn es 1 ist,
                        ;ist keine Korrektur erfolgt
A 0379  STA $D8         ;Flag speichern
A 037B  LDY #$00        ;Index für LDA (...),Y zurücksetzen
A 037D  STY $D9         ;Gänsefüßchenflag löschen
A 037F  LDX #$00        ;Zeiger im BASIC-Eingabepuffer auf Null
A 0381  JSR $040A       ;Zeichen aus Programmzeile holen
A 0384  CMP #$88        ;Ist es das TOKEN für LET?
A 0386  BNE $0390       ;Nein, dann Sprung
A 0388  PHA             ;Ja, dann Akku sichern
A 0389  LDA #$00        ;Flag für 'Korrektur erfolgt'
A 038B  STA $D8         ;Flag setzen
A 038D  PLA             ;Akku wieder vom Stack holen ($88)
A 038E  BNE $03BC       ;Unbedingter Sprung da Akku = $88
A 0390  CMP #$8F        ;War das Byte das TOKEN für REM?
A 0392  BNE $0398       ;Nein, dann Sprung
A 0394  LDA #$00        ;Ja, dann Korrekturflag setzen und
A 0396  STA $D8         ;gelesenes Zeichen = 0 (Zeilenende)
A 0398  CMP #$22        ;War das Byte ein SPACE?
A 039A  BNE $03A4       ;Nein, dann Sprung
A 039C  PHA             ;Ja, dann Akku sichern
A 039D  LDA $D9         ;Gänsefüßchenflag holen
A 039F  EOR #$01        ;und umdrehen
A 03A1  STA $D9         ;Jetzt wieder abspeichern
A 03A3  PLA             ;Akku wieder vom Stack holen
A 03A4  ROR $D9         ;Gänsefüßchenflag nach rechts rotieren,
                        ;Dadurch wird der Inhalt des 0. Bits
                        ;ins Carryflag geschoben
A 03A6  PHP             ;Statusregister incl. Carry auf Stack
A 03A7  ROL $D9         ;Gänsefüßchenflag nach links rotieren
A 03A9  PLP             ;Statusregister vom Stack holen
                        ;C=0: Gänsefüßchenmodus inaktiv
                        ;C=1: Gänsefüßchenmodus aktiv
A 03AA  BCS $03B8       ;Sprung falls aktiv (Text)
```

```
A 03AC  CMP #$20     ;Inaktiv, also auf SPACE testen
A 03AE  BNE $03B8    ;Kein SPACE, dann Sprung
A 03B0  PHA          ;SPACE: Akku sichern
A 03B1  LDA #$00     ;Flag für 'Korrektur erfolgt' setzen
A 03B3  STA $D8      ;und speichern
A 03B5  PLA          ;Akku wieder vom Stack holen
A 03B6  BNE $03BC    ;Unbedingter Sprung, da Akku = $20
A 03B8  STA $0200,X  ;Gelesenes Zeichen im BASIC-
                     ;Eingabepuffer ablegen
A 03BB  INX          ;Zeiger im Eingabepuffer um 1 erhöhen
A 03BC  INC $DA      ;Zeiger-LOW im Programmtext plus eins
A 03BE  BNE $03C2    ;Kein Überlauf, dann Sprung
A 03C0  INC $DB      ;Überlauf, dann auch Zeiger-HIGH plus 1
A 03C2  CMP #$00     ;War gelesenes Zeichen Endmarkierung?
A 03C4  BNE $0381    ;Nein, dann Sprung
A 03C6  LDA $D8      ;Korrekturflag testen
A 03C8  BNE $0407    ;Keine Korrektur erfolgt, dann Sprung
A 03CA  STX $0B      ;Pufferlänge an ROM-Routine übergeben
A 03CC  LDA #$E1     ;Adresse an der das Programm fortgesetzt
A 03CE  LDY #$03     ;werden soll in Akku und YR laden
A 03D0  STA $0302    ;Als neue Adresse in den Vektor
A 03D3  STY $0303    ;schreiben über den am Ende der Routine
                     ;zum Einfügen und Löschen von Programm-
                     ;zeilen gesprungen wird
A 03D6  LDA #$00     ;Adresse-LOW des BASIC-Eingabepuffers
A 03D8  LDY #$02     ;Adresse-HIGH des BASIC-Eingabepuffers
A 03DA  STA $3B      ;In Speicherstelle $3B/$3C schreiben
A 03DC  STY $3C      ;Über diese Speicherstellen wird inner-
                     ;halb der ROM-Routine ein Zeichen aus
                     ;dem BASIC-Eingabepuffer geholt
A 03DE  JMP $8736    ;BASIC-Zeile löschen und evtl. einfügen
A 03E1  LDA #$12     ;Alten Inhalt des Vektors-LOW
                     ;Zu dieser Adresse wird über den Vektor
                     ;gesprungen
A 03E3  LDY #$87     ;Alten Inhalt des Vektors-HIGH
A 03E5  STA $0302    ;Vektor wieder auf alten Wert setzen
A 03E8  STY $0303    ;
A 03EB  LDA $0200    ;Erstes Zeichen aus Eingabepuffer holen
A 03EE  BNE $03FA    ;Ungleich Null, dann Sprung
A 03F0  LDA $DC      ;Gleich Null, dann wurde die Zeile ge-
```

```
A 03F2  LDY $DD     ;löscht und die alte Adresse kann bei-
A 03F4  STA $DA     ;behalten werden (s.o.)
A 03F6  STY $DB     ;
A 03F8  BNE $0407   ;Unbedingt, da YR immer größer $10
A 03FA  CLC         ;Carry für Addition vorbereiten
A 03FB  LDA $5F     ;LOW-Byte der Anfangsadresse ab der die
                    ;korrigierte Zeile eingefügt wurde
A 03FD  ADC $0B     ;Zeilenlänge addieren
A 03FF  STA $DA     ;Und als neuen Zeiger setzen
A 0401  LDA #$00    ;Akku auf Null setzen, aber Carry
                    ;beibehalten
A 0403  ADC $60     ;HIGH-Byte zum Carry addieren
A 0405  STA $DB     ;und speichern
A 0407  JMP $033A   ;Das ganze nochmal für die nächste Zeile
A 040A  LDA #$DA    ;Routine zum Laden eines Wertes aus dem
A 040C  JMP $0494   ;RAM über die Adresse $DA. Es wird
                    ;hierbei die Routine LDA (...),Y aufge-
                    ;rufen wobei für die Punkte der Wert
                    ;$DA eingesetzt wird
```

Ich hoffe die Dokumentation dieses Disassemblerlistings reicht aus, um alle entstehenden Fragen zu klären. Trotzdem will ich aber noch auf einige Punkte genauer eingehen. Wie Ihnen bei dem Studium des Listings sicherlich aufgefallen ist, arbeitet das Programm mit zwei Flags. Das eine Flag ist das sogenannte Korrekturflag und das andere das Gänsefüßchenflag. Das Korrekturflag hat die Aufgabe, festzuhalten, ob überhaupt eine Änderung vorgenommen worden ist. Wenn dies nicht der Fall war, kann das zeitaufwendige Austauschen der Programmtexte entfallen. Das zweite Flag ist das Gänsefüßchenflag. Dieses Flag hat die Aufgabe festzuhalten, ob gerade ein Text (z.B. hinter einer Printanweisung) oder ein ganz normaler BASIC-Programmtext eingelesen wird. Im ersten Fall dürfen die Leerzeichen nämlich nicht entfernt werden, da sie für den korrekten Programmablauf sehr wichtig sind. Im zweiten Fall können Sie allerdings entfallen, da Sie den Programmablauf nur verzögern.

Für alle, denen das Abtippen des Mnemonic-Listings zu mühselig ist, hier der DATA-Lader:

```
100 rem ************************
110 rem * rem-let-space-killer *
120 rem * -------------------- *
130 rem * c16/c116/plus 4      *
140 rem * d. vuellers juni '86 *
150 rem ************************
160 :
170 for i=818 to 1038
180 :   read a
190 :   ps=ps+a
200 :   poke i,a
210 next i
220 :
230 if ps<>24854 then print "fehler in datas !!!":end
240 :
250 print "ok! programmstart mit sys 818"
260 :
270 data 165,43,164,44,133,218,132,219
280 data 160,0,32,10,4,133,222,200
290 data 32,10,4,5,222,208,3,76
300 data 10,128,200,32,10,4,133,20
310 data 200,32,10,4,133,21,166,20
320 data 165,21,32,95,164,169,13,32
330 data 210,255,165,218,164,219,133,220
340 data 132,221,24,165,218,105,4,133
350 data 218,144,2,230,219,169,1,133
360 data 216,160,0,132,217,162,0,32
370 data 10,4,201,136,208,8,72,169
380 data 0,133,216,104,208,44,201,143
390 data 208,4,169,0,133,216,201,34
400 data 208,8,72,165,217,73,1,133
410 data 217,104,102,217,8,38,217,40
420 data 176,12,201,32,208,8,72,169
430 data 0,133,216,104,208,4,157,0
440 data 2,232,230,218,208,2,230,219
450 data 201,0,208,187,165,216,208,61
460 data 134,11,169,225,160,3,141,2
470 data 3,140,3,3,169,0,160,2
```

```
480 data 133,59,132,60,76,54,135,169
490 data 18,160,135,141,2,3,140,3
500 data 3,173,0,2,208,10,165,220
510 data 164,221,133,218,132,219,208,13
520 data 24,165,95,101,11,133,218,169
530 data 0,101,96,133,219,76,58,3
540 data 169,218,76,148,4
```

Anhang A

Die Zeropage und andere Systemadressen

$0000	0000	Datenrichtungsregister des 7501-Prozessorports
$0001	0001	Ein-/Ausgabe des 7501-Prozessors
$0002	0002	Schleifenflag
$0003-0004	0003-0004	Neue Startzeilennummer (RENUMBER)
$0005-0006	0005-0006	Schrittweite (RENUMBER)
$0007	0007	Suchzeichen
$0008	0008	1=Gänsefüßchenmodus
$0009	0009	Spaltenzähler bei TAB-Fkt.
$000A	0010	LOAD(0) / VERIFY(1)
$000B	0011	Anzahl der Elemente im Eingabepuffer
$000C	0012	Standart-DIM (10)
$000D	0013	Datentyp: $00=Fließkomma $FF=String
$000E	0014	Datentyp: $00=Fließkomma $80=Integer
$000F	0015	Flag für DATA/LIST
$0010	0016	Flag: Element/FNx Flag
$0011	0017	Flag: $00=Input, $40=Get, $98=Read
$0012	0018	Vorzeichen bei TAN(X)
$0013	0019	Flag für '?' bei Input
$0014-0015	0020-0021	Speicher für Zeilennummer
$0016	0022	Zeiger auf Stringstack
$0017-0018	0023-0024	Zeiger auf letzte Stringadresse
$0019-0021	0025-0033	Temorärer Stringstack
$0022-0023	0034-0035	Zeiger für diverse Zwecke
$0024-0025	0036-0037	Zeiger für diverse Zwecke
$0026-002A	0038-0042	Bereich für Ergebnis bei Multiplikation
$002B-002C	0043-0044	Zeiger auf BASIC-Programmstart
$002D-002E	0045-0046	Zeiger auf Anfang der Variablen
$002F-0030	0047-0048	Zeiger auf Anfang der Arrays
$0031-0032	0049-0050	Zeiger auf Ende der Arrays (+1)
$0033-0034	0051-0052	Zeiger auf Beginn der Strings

$0035-0036	0053-0054	Hilfzeiger für Strings
$0037-0038	0055-0056	Zeiger auf Ende des BASIC-Speichers
$0039-003A	0057-0058	Augenblickliche BASIC-Zeilennummer
$003B-003C	0059-0060	BASIC-Programmzeiger (TXTPTR)
$003D-003E	0061-0062	Zeiger auf nächstes BASIC-Statement für CONT
$003F-0040	0063-0064	Zeilennummer der aktuellen Datazeile
$0041-0042	0065-0066	Zeiger auf nächstes DATA-Element
$0043-0044	0067-0068	Zeiger auf Herkunft der Eingabe (z.B. INPUT)
$0045-0046	0069-0070	Variablenname im ASCII-Kode
$0047-0048	0071-0072	Variablenadresse
$0049-004A	0073-0074	Zeiger auf Variable für FOR...NEXT
$004B-004C	0075-0076	Zwischenspeicher für BASIC-Zeiger z.B. Programmzeiger TXTPTR
$004D	0077	Maske für Vergleichsoperationen
$004E-004F	0078-0079	Zeiger für FN
$0050-0052	0080-0082	Stringdescriptor
$0053	0083	Hilfsregister
$0054-0056	0084-0086	Sprungvektor für Funktionen $0054 enthält $4C (JMP) Vektor in $0055 und $0056
$0057-0060	0087-0096	Zwischenspeicher für Fließ-kommazahlen (z.B. FAC oder ARG) Bei FAC, ARG incl. Vorzeichen, Rundungsbyte etc.
$0061	0097	FAC: Exponent
$0062	0098	FAC: Mantisse 1
$0063	0099	FAC: Mantisse 2
$0064	0100	FAC: Mantisse 3
$0065	0101	FAC: Mantisse 4
$0066	0102	FAC: Vorzeichen
$0067	0103	FAC: Zähler für Polynomauswertung
$0068	0104	ARG: Exponent
$0069	0105	ARG: Mantisse 1
$006A	0106	ARG: Mantisse 2
$006B	0107	ARG: Mantisse 3
$006C	0108	ARG: Mantisse 4

$006D	0109	ARG: Vorzeichen
$006E	0110	ARG: Zähler für Polynomauswertung
$006F	0111	Vorzeichenvergleich: FAC mit ARG
$0070	0112	Rundungsstelle (niederw. Stelle)
$0071-0072	0113-0114	Zeiger auf Kassettenpuffer
$0073-0074	0115-0116	'AUTO'-Modus; 0=Aus sonst Schrittweite
$0075	0117	'GRAPHIC'-Flag: 0 = Textgrafik (mit JSR $C7C9) 32 = hochauflösende Grafik 96 = hochaufl. Grafik mit Text 160 = multicolor Grafik 224 = multicolor Grafik mit Text
$0076	0118	Funktionstastennr, bei Definierung
$0077	0119	KEY-Textlänge bei Definierung
$0078	0120	
$0079-007B	0121-0123	Stringdescriptor für DS$
$007C-007D	0124-0125	BASIC-(Pseudo)-Stackpointer
$007E-008F	0126-143	Arbeitsbereich SOUND (Tonhöhe)
$0090	0144	I/O-Statusbyte (auch RS-232C)
$0091	0145	Flag für STOP und RVS Taste(n)
$0092-0093	0146-0147	
$0094	0148	Flag für IEC-Ausgabe
$0095	0149	Ausgabepuffer für IEC-Bus
$0096	0150	
$0097	0151	Anzahl der offenen Files
$0098	0152	Eingabegerät (Tastatur = 0)
$0099	0153	Ausgabegerät (Bildschirm = 3)
$009A	0154	Flag: $80=Direktmodus $00=Programm-Modus
$009B-009C	0155-0156	Zeiger für Kassettenpuffer und Scrolling
$009D-009E	0157-0158	Zeiger auf Prg.ende bei LOAD/SAVE
$009F-00A0	0159-0160	Zeitkonstanten Bandtiming
$00A1-00A2	0161-0162	Temporärer Datenspeicher
$00A3-00A5	0163-0165	Echtzeituhr: TI
$00A6	0166	Register für IEC-Bus
$00A7	0167	Zeichenpuffer für TAPE
$00A8	0168	Zeichenpuffer für IEC-Bus
$00A9	0169	Temporärer Farbvektor

$00AA	0170	Bitzähler für Band
$00AB	0171	Länge des Filenamen
$00AC	0172	Logische Filenummer
$00AD	0173	Sekundäradresse
$00AE	0174	Geräteadresse
$00AF-00B0	0175-0176	Zeiger auf Filenamen
$00B1	0177	Checksumme
$00B2-00B3	0178-0179	I/O Startadresse
$00B4-00B5	0180-0181	Basis-Ladeadresse LOAD/SAVE
$00B6-00B7	0182-0183	Ladeendadresse von TAPE
$00B8-00B9	0184-0185	
$00BA-00BB	0186-0187	Zeiger im Kassettenpuffer
$00BC-00BD	0188-0189	
$00BE-00BF	0190-0191	BASIS-Adresse bei LONG-FETCH
$00C0-00C1	0192-0193	Register für Scrolling
$00C2	0194	REVERS-Falg für SCREEN
$00C3	0195	Ende der Zeile (INPUT)
$00C4	0196	Cursorkoordinate X (INPUT)
$00C5	0197	Cursorkoordinate Y (INPUT)
$00C6	0198	Nummer der gedrückten Taste
$00C7	0199	Flag für Eingabe von Tastatur oder Bildschirm
$00C8-00C9	0200-0201	Zeiger auf akt. Bildschirmzeile
$00CA	0202	Cursor: X-Koordinate
$00CB	0203	Flag für Hochkommamodus 0=Aus
$00CC	0204	Länge akt. Zeile
$00CD	0205	Cursor: Y-Koordinate
$00CE	0206	Letztes Zeichen bei Ein-/Ausgabe
$00CF	0207	Anzahl einzuf. Zeichen (INSERT)
$00D0-00D7	0208-0215	Reserviert für Sprachsynthesizer
$00D8-00E8	0216-0232	Reserviert für Anwendungssoftware
$00E9	0233	Speicher für CIRCLE (Kreissegment)
$00EA-00EB	0234-0235	Cursorfarbe incl. Lum.
$00EC-00EE	0236-0238	Arbeitsbereich für Bildschirm
$00EF	0239	Anzahl Zeichen im Tastaturpuffer
$00F0	0240	
$00F1-00F2	0241-0242	Register für Monitor
$00F3-00F9	0243-0249	Register für Ein-/Ausgabe
$00FA	0250	X-Register bei STOP-Tastentest
$00FB	0251	Bankkonfiguration

$00FC-00FD	0252-0253	
$00FE	0254	Arbeitsregister für EDITOR
$00FF	0255	
$0100-01FF	0256-0511	Prozessorstack
$0200-0258	0512-0600	BASIC-Eingabepuffer
$0259-025C	0601-0604	BASIC-Puffer
$025D	0605	DOS-Schleifenzähler
$025E-026D	0606-0621	Bereich für Filenamen
$026E	0622	Länge Filenamen 1
$026F	0623	Laufwerk 1
$0270-0271	0624-0625	Adresse Filenamen 1
$0272	0626	Länge Filenamen 2
$0273	0627	Laufwerk 2
$0274-0275	0628-0629	Adresse Filenamen 2
$0276	0630	DOS: Filenummer
$0277	0631	DOS: Geräteadresse
$0278	0632	DOS: Sekundäradresse
$0279-027A	0633-0634	DOS: Disketten ID
$027B	0635	DOS: ID-Test Flag
$027C	0636	DOS: Ausgabepuffer
$027D-02AC	0637-0684	DOS: Arbeitsbereich
$02AD-02B0	0685-0688	Grafik Cursor
$02B1-02B4	0689-0692	Register: Grafik Cursor
$02B5-02CB	0693-0715	Grafik Register
$02CC-02E3	0716-0739	Grafik-Arbeitsbereich und PRINT USING
$02E4	0740	HIGH-ADR: CHARACTER-ROM
$02E5-02EA	0741-0746	
$02EB	0747	'TRACE'-Modus-Flag
$02EC-02EF	0748-0751	
$02F0	0752	Anzahl Grafikparameter
$02F1	0753	Relative(1) / Absolute(0) Koordinaten bei Grafikbefehl
$02F2-02F3	0754-0755	VEKTOR: Fließkomma
$02F4-02F5	0756-0757	VEKTOR: Integer
$02F6-02FF	0758-0767	
$0300-0301	0768-0769	VEKTOR: Fehlermeldung
$0302-0303	0770-0771	VEKTOR: BASIC-Warmstart
$0304-0305	0772-0773	VEKTOR: Token Generierung
$0306-0307	0774-0775	VEKTOR: Keywort erzeugen

$0308-0309	0776-0777	VEKTOR: Hauptschleife
$030A-030B	0778-0779	VEKTOR: Eval
$030C-030D	0780-0781	VEKTOR: USER-Token Gener.
$030E-030F	0782-0783	VEKTOR: Keywort erzeugen
$0310-0311	0784-0785	VEKTOR: USER-Token bearbeiten
$0312-0313	0786-0787	VEKTOR: IRQ (UHR)
$0314-0315	0788-0789	VEKTOR: IRQ
$0316-0317	0790-0791	VEKTOR: BREAK-Interrupt
$0318-0319	0792-0793	VEKTOR: OPEN
$031A-031B	0794-0795	VEKTOR: CLOSE
$031C-031D	0796-0797	VEKTOR: CHKIN
$031E-031F	0798-0799	VEKTOR: CKOUT
$0320-0321	0800-0801	VEKTOR: CLRCH
$0322-0323	0802-0803	VEKTOR: INPUT
$0324-0325	0804-0805	VEKTOR: OUTPUT
$0326-0327	0806-0807	VEKTOR: STOP-Taste testen
$0328-0329	0808-0809	VEKTOR: GETIN
$032A-032B	0810-0811	VEKTOR: CLALL
$032C-032D	0812-0813	VEKTOR: Monitor Break
$032E-032F	0814-0815	VEKTOR: LOAD
$0330-0331	0816-0817	VEKTOR: SAVE
$0332-03F2	0818-1010	Kassettenpuffer
$03F3-03F4	1011-1012	Datenzähler für Schreiben
$03F5-03F6	1013-1014	Datenzähler für Lesen
$03F7-0436	1015-1078	RS-232C-INPUT-Puffer
$0437-0472	1079-1138	
$0473-0493	1139-1171	CHRGET-Routine
$0479	1145	CHRGOT-Einsprungpunkt
$0494-04A1	1172-1185	LDA (...),Y aus RAM
$04A2-04E6	1186-1254	Andere Bankswitchingroutinen
$04E7-04EA	1255-1258	PRINT-USING-Parameter (PUDEF)
$04EB-04F1	1259-1262	
$04EF	1263	Fehlerkode bei Fehler
$04F0-04F1	1264-1265	Zeilennr. des Fehlers
$04F2-04F3	1266-1267	Zeilennr. für Traping ($FFFF=Aus)
$04F4	1268	Zwischenspeicher bei Fehler
$04F5-04F6	1269-1270	Adresse des Fehlers (für TXTPTR)
$04F7-04FF	1271-1279	
$0500	1280	$4C JMP-Kode für USR(X)-Fkt.
$0501-0502	1281-1282	Adresse USR-Funktion (LOW/HIGH)

$0503-0507	1283-1287	Referenzwert für RND (Fließkomma)
$0508	1288	Flag für Warm- oder Kaltstart
$0509-0512	1289-1298	Tabelle der log. Filenummern
$0513-051C	1299-1308	Tabelle der Gerätenummern
$051D-0526	1309-1318	Tabelle der Sekundäradressen
$0527-0530	1319-1328	Tastaturpuffer
$0531-0532	1329-1330	Anfang BASIC-Speicher
$0533-0534	1331-1332	Ende BASIC-Speicher
$0535-0538	1333-1336	
$0539	1337	Zeiger im Kassettenpuffer
$053A	1338	Type des Kassettenfiles
$053B-053F	1339-1343	
$0540	1344	Tastenwiederholung: $80 = Alle Tasten; $40 = Aus $00 = INST/DEL, CURSOR und SPACE
$0541-0542	1345-1346	Zähler für Tastenwiederholung
$0543	1347	Flag: SHIFT (1), C= (2), CTRL (3)
$0544	1348	
$0545-0546	1349-1350	VEKTOR: Keylog
$0547	1351	Flag: Text/Grafik
$0548	1352	SCROLL-Flag
$0549-054A	1353-1354	
$054B-0551	1355-1361	Arbeitsbereich für TEDMON
$0552-0557	1362-1367	CPU-Register für TEDMON
$0558-055C	1368-1372	Arbeitsbereich für TEDMON
$055D	1373	Index für Funktionstasten
$055E	1374	Zeiger: Text der Funktionstasten
$055F-0566	1375-1382	Länge der Funktionstexte
$0567-05E6	1383-1510	Texte der Funktionstasten
$05E7-05EB	1511-1515	
$05EC-05EF	1516-1519	Adresstabelle für Banking
$05F0-05F1	1520-1621	LONG JUMP (Adresse)
$05F1	1522	LONG JUMP (Akku)
$05F2	1523	LONG JUMP (X-Reg.)
$05F3	1524	LONG JUMP (Status)
$05F5-065D	1525-1629	Arbeitsbereich Bankswitching
$056E-06EB	1630-1771	Arbeitsbereich Sprach-Synthesizer
$06EC-07AF	1772-1967	BASIC-Pseudo-Stack
$07B0-07CC	1968-1996	Kassetten-Arbeitsbereich
$07CD-07D0	1997-2000	RS-232C-Arbeitsbereich

$07D1	2001	RS-232C: Zeiger auf Anfang im Eingabepuffer
$07D2	2002	RS-232C: Zeiger auf Ende im Eingabepuffer
$07D3	2003	Anzahl Zeichen im Eingabepuffer
$07D4-07E4	2004-2020	
$07E5	2021	Window: Unterer Rand
$07E6	2022	Window: Oberer Rand
$07E7	2023	Window: Linker Rand
$07E8	2024	Window: Rechter Rand
$07E9	2025	
$07EA	2026	Auto-Insert $FF=Ein
$07EB-07ED	2027-2029	
$07EE-07F1	2030-2033	Bittabelle für Grafik
$07F2	2034	Pseudo-Akkumulator (SYS)
$07F3	2035	Pseudo-X-Register (SYS)
$07F4	2036	Pseudo-Y-Register (SYS)
$07F5	2037	Pseudo-Statusregister (SYS)
$07F6	2038	Register für Tastaturabfrage
$07F7	2039	CTRL-S: $00=Auf $06=Zu
$07F8	2040	TEDMON: $00=ROM $80=RAM
$07F9-07FB	2041-2043	
$07FC	2044	TAPE: Motorsteuerung
$07FB-07FF	2045-2047	
$0800-0BE7	2048-3047	Text-Farbspeicher
$0BE8-0BFF	3048-3071	
$0C00-0FE7	3072-4073	Text-Bildschirmspeicher
$0FE8-0FFF	4074-4095	

Grafik wurde noch nicht aufgerufen:

$1000-FCFF	4096-64767	BASIC-RAM Plus/4
$1000-3FFF	4096-16383	BASIC-RAM C-16/116

Grafik wurde schon einmal aufgerufen:

$1000-17FF	4096-6143	BASIC-RAM C-16/116
$1800-1BFF	6144-7167	Grafik-Luminanztabelle

$1C00-1FFF	7168-8191	Grafik-Farbtabelle
$2000-3FFF	8192-16383	Grafikbildschirm
$1000-17FF	4096-6143	Freier Speicher bei Plus/4
$4000-FCFF	16384-64776	BASIC-RAM Plus/4

Anhang B

Befehlscodes

			Absolut			Zeropage				
Mnemonic	Akku	Direkt		X	Y		X	Y	Indiziert Indirekt	Indirekt Indiziert
ADC	–	$69	$6D	$7D	$79	$65	$75	–	$71	$61
AND	–	$29	$2D	$3D	$39	$25	$35	–	$31	$21
ASL	$0A	–	$0E	$1E	–	$06	$16	–	–	–
BIT	–	–	$2C	–	–	$24	–	–	–	–
CMP	–	$C9	$CD	$DD	$D9	$C5	$D5	–	$D1	$C1
CPX	–	$E0	$EC	–	–	$E4	–	–	–	–
CPY	–	$C0	$CC	–	–	$C4	–	–	–	–
DEC	–	–	$CE	$DD	–	$C6	$D6	–	–	–
EOR	–	$49	$4D	$5D	$59	$45	$55	–	$51	$41
INC	–	–	$EE	$FD	–	$E6	$F6	–	–	–
LDA	–	$A9	$AD	$BD	$B9	$A5	$B5	–	$B1	$A1
LDX	–	$A2	$AE	–	$BE	$A6	–	$B6	–	–
LDY	–	$A0	$AC	$BC	–	$A4	$B4	–	–	–
LSR	$4A	–	$4E	$5E	–	$46	$56	–	–	–
ORA	–	$09	$0D	$1D	$19	$05	$15	–	$11	$01
ROL	$2A	–	$2E	$3E	–	$26	$36	–	–	–
ROR	$6A	–	$6E	$7E	–	$66	$76	–	–	–
SBC	–	$E9	$ED	$FD	$F9	$E5	$F5	–	$F1	$E1
STA	–	–	$8D	$9D	$99	$85	$95	–	$91	$81
STX	–	–	$8E	–	–	$86	–	$96	–	–
STY	–	–	$8C	–	–	$84	$94	–	–	–

```
BPL  BMI  BVC  BVS  BCC  BCS  BNE  BEQ
-----------------------------------
$10  $30  $50  $70  $90  $B0  $D0  $F0

TXA  TAX  TAY  TYA  TSX  TXS
---------------------------
$8A  $AA  $98  $A8  $BA  $9A

PHP  PLP  PHA  PLA
-----------------
$08  $28  $48  $68

BRK  JSR  RTI  RTS  JMP  JMP  NOP
--------------------------------
$00  $20  $40  $60  $4C  $6C  $EA

CLC  SEC  CLI  SEI  CLV  CLD  SED
--------------------------------
$18  $38  $58  $78  $B8  $D8  $F8

DEY  INY  DEX  INX
-----------------
$88  $C8  $CA  $E8
```

Anhang C

Umrechnungstabelle

000	0000	00000000	001	0001	00000001
002	0002	00000010	003	0003	00000011
004	0004	00000100	005	0005	00000101
006	0006	00000110	007	0007	00000111
008	0008	00001000	009	0009	00001001
010	000A	00001010	011	000B	00001011
012	000C	00001100	013	000D	00001101
014	000E	00001110	015	000F	00001111
016	0010	00010000	017	0011	00010001
018	0012	00010010	019	0013	00010011
020	0014	00010100	021	0015	00010101
022	0016	00010110	023	0017	00010111
024	0018	00011000	025	0019	00011001
026	001A	00011010	027	001B	00011011
028	001C	00011100	029	001D	00011101
030	001E	00011110	031	001F	00011111
032	0020	00100000	033	0021	00100001
034	0022	00100010	035	0023	00100011
036	0024	00100100	037	0025	00100101
038	0026	00100110	039	0027	00100111
040	0028	00101000	041	0029	00101001
042	002A	00101010	043	002B	00101011
044	002C	00101100	045	002D	00101101
046	002E	00101110	047	002F	00101111
048	0030	00110000	049	0031	00110001
050	0032	00110010	051	0033	00110011
052	0034	00110100	053	0035	00110101
054	0036	00110110	055	0037	00110111
056	0038	00111000	057	0039	00111001
058	003A	00111010	059	003B	00111011
060	003C	00111100	061	003D	00111101
062	003E	00111110	063	003F	00111111
064	0040	01000000	065	0041	01000001
066	0042	01000010	067	0043	01000011

068	0044	01000100	069	0045	01000101
070	0046	01000110	071	0047	01000111
072	0048	01001000	073	0049	01001001
074	004A	01001010	075	004B	01001011
076	004C	01001100	077	004D	01001101
078	004E	01001110	079	004F	01001111
080	0050	01010000	081	0051	01010001
082	0052	01010010	083	0053	01010011
084	0054	01010100	085	0055	01010101
086	0056	01010110	087	0057	01010111
088	0058	01011000	089	0059	01011001
090	005A	01011010	091	005B	01011011
092	005C	01011100	093	005D	01011101
094	005E	01011110	095	005F	01011111
096	0060	01100000	097	0061	01100001
098	0062	01100010	099	0063	01100011
100	0064	01100100	101	0065	01100101
102	0066	01100110	103	0067	01100111
104	0068	01101000	105	0069	01101001
106	006A	01101010	107	006B	01101011
108	006C	01101100	109	006D	01101101
110	006E	01101110	111	006F	01101111
112	0070	01110000	113	0071	01110001
114	0072	01110010	115	0073	01110011
116	0074	01110100	117	0075	01110101
118	0076	01110110	119	0077	01110111
120	0078	01111000	121	0079	01111001
122	007A	01111010	123	007B	01111011
124	007C	01111100	125	007D	01111101
126	007E	01111110	127	007F	01111111
128	0080	10000000	129	0081	10000001
130	0082	10000010	131	0083	10000011
132	0084	10000100	133	0085	10000101
134	0086	10000110	135	0087	10000111
136	0088	10001000	137	0089	10001001
138	008A	10001010	139	008B	10001011
140	008C	10001100	141	008D	10001101
142	008E	10001110	143	008F	10001111
144	0090	10010000	145	0091	10010001
146	0092	10010010	147	0093	10010011

148	0094	10010100	149	0095	10010101
150	0096	10010110	151	0097	10010111
152	0098	10011000	153	0099	10011001
154	009A	10011010	155	009B	10011011
156	009C	10011100	157	009D	10011101
158	009E	10011110	159	009F	10011111
160	00A0	10100000	161	00A1	10100001
162	00A2	10100010	163	00A3	10100011
164	00A4	10100100	165	00A5	10100101
166	00A6	10100110	167	00A7	10100111
168	00A8	10101000	169	00A9	10101001
170	00AA	10101010	171	00AB	10101011
172	00AC	10101100	173	00AD	10101101
174	00AE	10101110	175	00AF	10101111
176	00B0	10110000	177	00B1	10110001
178	00B2	10110010	179	00B3	10110011
180	00B4	10110100	181	00B5	10110101
182	00B6	10110110	183	00B7	10110111
184	00B8	10111000	185	00B9	10111001
186	00BA	10111010	187	00BB	10111011
188	00BC	10111100	189	00BD	10111101
190	00BE	10111110	191	00BF	10111111
192	00C0	11000000	193	00C1	11000001
194	00C2	11000010	195	00C3	11000011
196	00C4	11000100	197	00C5	11000101
198	00C6	11000110	199	00C7	11000111
200	00C8	11001000	201	00C9	11001001
202	00CA	11001010	203	00CB	11001011
204	00CC	11001100	205	00CD	11001101
206	00CE	11001110	207	00CF	11001111
208	00D0	11010000	209	00D1	11010001
210	00D2	11010010	211	00D3	11010011
212	00D4	11010100	213	00D5	11010101
214	00D6	11010110	215	00D7	11010111
216	00D8	11011000	217	00D9	11011001
218	00DA	11011010	219	00DB	11011011
220	00DC	11011100	221	00DD	11011101
222	00DE	11011110	223	00DF	11011111
224	00E0	11100000	225	00E1	11100001
226	00E2	11100010	227	00E3	11100011

228	00E4	11100100	229	00E5	11100101
230	00E6	11100110	231	00E7	11100111
232	00E8	11101000	233	00E9	11101001
234	00EA	11101010	235	00EB	11101011
236	00EC	11101100	237	00ED	11101101
238	00EE	11101110	239	00EF	11101111
240	00F0	11110000	241	00F1	11110001
242	00F2	11110010	243	00F3	11110011
244	00F4	11110100	245	00F5	11110101
246	00F6	11110110	247	00F7	11110111
248	00F8	11111000	249	00F9	11111001
250	00FA	11111010	251	00FB	11111011
252	00FC	11111100	253	00FD	11111101
254	00FE	11111110	255	00FF	11111111

Anhang D

Flagbeeinflussungen

Mnemonic	N	V	B	D	I	Z	C	-
ADC	X	X	-	-	-	X	X	-
AND	X	-	-	-	-	X	-	-
ASL	X	-	-	-	-	X	X	-
BCC	-	-	-	-	-	-	-	-
BCS	-	-	-	-	-	-	-	-
BEQ	-	-	-	-	-	-	-	-
BIT	M	M	-	-	-	X	-	-
BMI	-	-	1	-	1	-	-	-
BNE	-	-	-	-	-	-	-	-
BPL	-	-	-	-	-	-	-	-
BRK	-	-	-	-	-	-	-	-
BVC	-	-	-	-	-	-	-	-
BVS	-	-	-	-	-	-	-	-
CLC	-	-	-	-	-	-	0	-
CLD	-	-	-	0	-	-	-	-
CLI	-	-	-	-	0	-	-	-
CLV	-	0	-	-	-	-	-	-
CMP	X	-	-	-	-	X	X	-
CPX	X	-	-	-	-	X	X	-
CPY	X	-	-	-	-	X	X	-
DEC	X	-	-	-	-	X	-	-
DEX	X	-	-	-	-	X	-	-
DEY	X	-	-	-	-	X	-	-
EOR	X	-	-	-	-	X	-	-
INC	X	-	-	-	-	X	-	-
INX	X	-	-	-	-	X	-	-
INY	X	-	-	-	-	X	-	-
JMP	-	-	-	-	-	-	-	-
JSR	-	-	-	-	-	-	-	-
LDA	X	-	-	-	-	X	-	-
LDX	X	-	-	-	-	X	-	-
LDY	X	-	-	-	-	X	-	-
LSR	0	-	-	-	-	X	X	-

NOP	-	-	-	-	-	-	-	-
ORA	X	-	-	-	-	X	-	-
PHA	-	-	-	-	-	-	-	-
PHP	-	-	-	-	-	-	-	-
PLA	X	-	-	-	-	X	-	-
PLP	X	X	X	-	X	X	X	X -
ROL	X	-	-	-	-	X	X	-
ROR	X	-	-	-	-	X	X	-
RTI	X	X	X	X	X	X	X	-
RTS	-	-	-	-	-	-	-	-
SBC	X	X	-	-	-	X	X	-
SEC	-	-	-	-	-	-	1	-
SED	-	-	-	1	-	-	-	-
SEI	-	-	-	-	1	-	-	-
STA	-	-	-	-	-	-	-	-
STX	-	-	-	-	-	-	-	-
STY	-	-	-	-	-	-	-	-
TAX	X	-	-	-	-	X	-	-
TAY	X	-	-	-	-	X	-	-
TSX	X	-	-	-	-	X	-	-
TXA	X	-	-	-	-	X	-	-
TXS	-	-	-	-	-	-	-	-
TYA	X	-	-	-	-	X	-	-

M Das Flag wird bei diesem Befehl je nach dem Operanten gesetzt oder gelöscht
X Das Flag wird je nach dem Ergebnis beeinflusst
1 Das Flag wird gesetzt
0 Das Flag wird gelöscht

Anhang E

Literaturnachweise

Commodore Plus/4 ROM-Listing
ISBN 3-89133-006-5
Commodore Sachbuchreihe Band 6

C16 Tips & Tricks
Baloui
ISBN 3-89011-168-8
Data Becker GmbH

Commodore 64 Intern
Angerhausen, Brückmann, Englisch, Gerits
ISBN 3-89011-000-2
Data Becker GmbH

Effektiv & Kreativ
Froitzheim, Kausmann
ISBN 3-89011-073-8
Data Becker GmbH

Das Maschinensprachebuch für Fortgeschrittene zum C-64
Lothar Englisch
ISBN 3-89011-022-3
Data Becker GmbH

Commodore 128 Intern
Gerits, Schieb, Thrun
ISBN 3-89011-098-3
Data Becker GmbH

Das große Grafikbuch zum C-128
Durben, Löffelmann, Plenge, Vüllers
ISBN 3-89011-154-8
Data Becker GmbH

Bücher zu C16, C116 und Plus/4

C16 Tips und Tricks bietet eine hochinteressante Sammlung von Anregungen, Ideen und fertigen Lösungen zur Programmierung und Anwendung Ihres C16/116. Eine wahre Fundgrube für jeden, der auf dem Commodore 16 eigene Programme schreiben will!

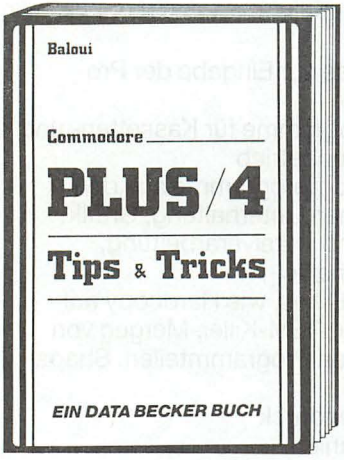
Aus dem Inhalt:

- Hinweise zur Eingabe der Programme
- Alle Programme für Kassetten- und Diskettenbetrieb
- Anwenderprogramme aus den Bereichen Unterhaltung, Grafik, Text- und Dateiverarbeitung, Mathematik
- Viele Utilities, wie Hardcopy auf Drucker, REM-Killer, Mergen von einzelnen Programmteilen, Shape-Editor
- Etikettendruck
- Lottozahlen
- Datumsberechnung
- Die wichtigsten Tips & Tricks
- Von BASIC zu Maschinensprache
- Wichtige Zeropageadressen
- Betriebssystemroutinen
- Routinen des BASIC-Interpreters

Baloui
C16 Tips & Tricks
201 Seiten, DM 29,–
ISBN 3-89011-168-8

Bücher zu C16, C116 und Plus/4

Plus/4 Tips & Tricks ist eine Sammlung von Anregungen, Ideen und fertigen Lösungen zur Programmierung und Anwendung Ihres Plus/4.
Alles, was Sie brauchen, um eigene Programme zu schreiben.



Aus dem Inhalt:

- Hinweise zur Eingabe der Programme
- Verwendung der Datasette
- Anwenderprogramme
  Spiele
  Graphik
  Text- und Dateiverarbeitung
  Kfz-Überwachung
  Mathematik
  Hardcopy
  Merge
  Shapeeditor
- Die wichtigsten Tips & Tricks
- Von BASIC bis ASSEMBLER
- Die wichtigsten ZEROPAGE-ADRESSEN
- Routinen des Betriebssystems und des BASIC-Interpreters
- Tips & Tricks für Fortgeschrittene

Baloui
Plus/4 Tips & Tricks
222 Seiten, DM 29,–
ISBN 3-89011-203-X

Bücher zu C16, C116 und Plus/4

Ein Team von Grafikspezialisten deckt wirklich alle Geheimnisse des C16, C116 und Plus/4 auf. Von den grundlegenden Grafikbefehlen bis hin zu nützlichen Utilities enthält dieses Buch alles, was Sie schon immer über Computergrafik wissen wollten.

Aus dem Inhalt:

- theoretische Grundlagen
- Linienberechnung in Assembler
- Einführung in die Hardware
- hochauflösende Grafik
- komfortabler Charactereditor
- Computer Aided Design
- Statistik
- Funktionsplotter
- Hardcopyroutinen
- Posterhardcopy auf allen Star-Druckern
- Shape-Editor

Plenge, Löffelmann
Grafikbuch zu C16, C116 und Plus/4
232 Seiten, DM 29,–
ISBN 3-89011-205-6

Bücher zu C16, C116 und Plus/4

Die BASIC-Programmierung auf Ihrem C16, C116, Plus/4 wird mit diesem Buch zum Kinderspiel! Der Autor führt detailliert in die BASIC-Programmierung ein und gibt eine Vielzahl von Tips und Tricks für den BASIC-Newcomer. Eine gut gegliederte Befehlsübersicht des C16-, C116-, Plus/4-BASIC erleichtert die Arbeit am Computer. Werden auch Sie zum BASIC-Profi!

Aus dem Inhalt:

- Tastatur und Editior
- BASIC-Kurs
- Demo-Programme
- Rechnen mit dem C16, C116, Plus/4
- Variablen
- Steuerung des Programmablaufs
- Arrays
- Graphik-Programmierung
- Malprogramm
- Multicolor-Graphik
- Shapes
- Graphiken speichern und laden
- Musik-Programmierung
- Datei- und Adressenverwaltung
- Window-Technik
- BASIC-Befehlsübersicht

Baloui
Das große BASIC-Buch C16, C116, Plus/4
ca. 250 Seiten, DM 29,–
ISBN 3-89011-204-8

DAS STEHT DRIN:

Viele Computerbenutzer, die bislang die Hemmschwelle zur Maschinensprache nicht überwunden haben, erhalten hier die perfekte Einführung in den Befehlssatz des 7501-Mikroprozessors. Neben leichtverständlicher Beschreibung aller 7501-Befehle werden viele Betriebssystemroutinen unter die Lupe genommen.

Aus dem Inhalt:

- Von BASIC zu Maschinensprache
- Der Maschinensprachemonitor
- Der 7501-Prozessor und sein Befehlssatz
- Ein- und Ausgabe von Zeichen in Maschinensprache
- Einbinden von Maschinenprogrammen in BASIC-Programme
- Der BASIC-Lader
- Routinen des Betriebssystems richtig genutzt
- Die Vektoren des Betriebssystems
- Interruptprogrammierung
- Beispielprogramme mit trickreichen Utilities
- Zeropage-, Befehlscode- und Umrechnungstabellen

UND GESCHRIEBEN HAT DIESES BUCH:

Dieter Vüllers hat bereits als Koautor des Grafikbuches zum C128 überzeugend seine Kenntnisse in der Assemblerprogrammierung unter Beweis gestellt. Auch in dem vorliegenden Buch ist es ihm gelungen, sein Know-how didaktisch hervorragend aufzubereiten.

ISBN 3-89011-206-4