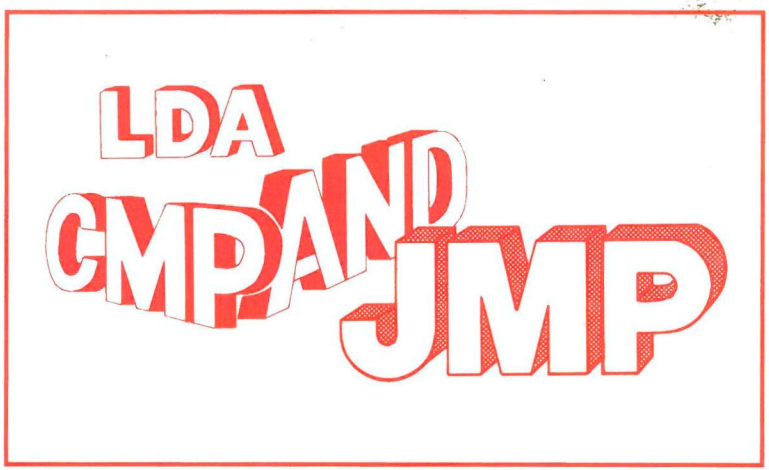
Lothar Englisch

Das Maschinensprache Buch

zum C64 & C128



EIN DATA BECKER BUCH

DAS STEHT DRIN:

Dieses Buch bietet eine leicht verständliche Einführung in die Maschinenspracheprogrammierung für alle, denen BASIC nicht mehr ausreicht. Sie lernen Aufbau und Arbeitsweise des 6510 Mikroprozessors (beziehungsweise des kompatiblen 8502) kennen und erfahren alles über Monitorprogramme und Assembler.

Aus dem Inhalt:

- Befehle und Adressierungsarten des 6510
- Eingabe von Maschinenprogrammen
- Der Assembler
- Der Einzelschrittsimulator für den 6510
- Maschinenprogramme auf dem C-64 & C 128
- Benutzung von Betriebssystemroutinen
- BASIC-Ladeprogramme
- Professionelle Hilfsmittel zur Erstellung von Maschinenprogrammen
- Umrechnungs- und Befehlstabellen
- Der Monitor des Commodore 128

UND GESCHRIEBEN HAT DIESES BUCH:

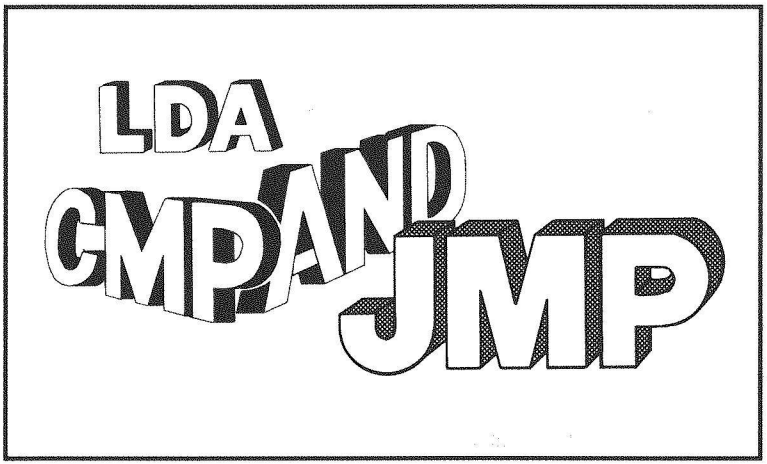
Lothar Englisch ist Systemprogrammierer in der DATA BECKER Entwicklungsabteilung. Vor allem aber ist er Autor vieler weiterer DATA BECKER Bücher (Maschinensprache für Fortgeschrittene, 64 Tips & Tricks, 64 Intern ...) und seine Bücher anerkannte Standardwerke.

ISBN 3-89011-008-8

Lothar Englisch

Das Maschinensprache Buch

zum C64 & C128



EIN DATA BECKER BUCH

ISBN 3-89011-008-8

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

Mit der Programmierung in Maschinensprache und Assembler ist das so eine Sache. Jeder würde es gern können (Maschinensprache ist extrem schnell und vielseitig), viele versuchen es (ein Versuch kann ja nicht schaden), die meisten davon geben schnell wieder auf (war wohl doch zu kompliziert) und nur wenige schaffen es wirklich (die sind dann zu beneiden).

Mit dem vorliegenden Buch wollen wir möglichst vielen Commodore 64 Anwendern den Weg in die Maschinensprache ermöglichen. Als Autor konnten wir hierfür Lothar Englisch gewinnen. Er hat bisher nicht nur an vielen unserer Bücher mitgewirkt, sondern gilt auch als ausgefuchster Kenner des Commodore Betriebssystems und der Programmierung aller Commodore Rechner in Maschinensprache und Assembler.

So stammen z.B. die beiden beliebten und sehr leistungsfähigen Programmpakete ProfiMon und ProfiAss aus seiner Feder.

Der Unterzeichner dieses Vorworts hat sich früher selbst häufig vergeblich an der Maschinensprache versucht und dabei in der Regel auch schnell frustriert aufgegegben. Geholfen hat mir schließlich das Manuskript dieses Buches und, das möchte ich nicht unerwähnt lassen, natürlich auch etwas mehr Geduld und Durchhaltevermögen.

Bei schwierigen Dingen muß man halt etwas stärker ran, aber - und das werden Sie nach erfolgreichem Durcharbeiten dieses Buches sicher bestätigen - es lohnt sich bestimmt.

Viel Spaß bei der Lektüre dieses Buches und viel Erfolg bei Ihren eigenen Maschinenspracheprogrammen.

Dr. Achim Becker

Inhaltsverzeichnis

1. Einführung

Warum überhaupt Maschinensprache? - Vor- und Nachteile der Programmierung in Maschinensprache gegenüber BASIC

Wenn Sie heute einen Personal- oder Homecomputer wie z.B. den Commodore 64 oder den Commodore 128 kaufen, so verfügen diese Geräte alle über die Programmiersprache BASIC. Mit BASIC lassen sich fast alle Aufgaben auf einem Homecomputer lösen. Die Programmierung des Computers ist leicht zu erlernen. Warum soll man sich da noch mit der Maschinensprache befassen. Ist dies nicht nur noch ein Relikt aus den Anfangstagen der Computerei? Vergleichen wir einmal BASIC mit der Maschinensprache.

Die meisten von Ihnen werden sicher BASIC beherrschen und bestätigen, daß diese Programmierung nicht schwer zu erlernen ist. Lassen Sie sich aber in diesem Buch überzeugen, daß das Programmieren in Maschinensprache ebenso einfach und schnell zu erlernen ist. Dabei kommt Ihnen zugute, daß Sie bereits BASIC beherrschen. Die grundsätzliche Arbeitsweise ist bei der Maschinensprache nicht viel anders. Welche Vorteile gegenüber BASIC lassen es nun gerechtfertigt erscheinen, sich eine neue Programmiersprache anzueignen?

1. Ihr Commodore 64 hat die Programmiersprache BASIC. Diese Sprache ist eine Abkürzung für 'Beginners All Purpose Symbolic Instruction Code', was soviel heißt wie symbolische Vielzweck-Programmiersprache für Anfänger und ist trotz leichter Erlernbarkeit durch großen Befehlsvorrat recht leistungsfähig. BASIC zählt zu den sogenannten höheren Programmiersprachen wie z.B. auch FORTRAN, PASCAL oder COBOL. Diese Sprachen werden oft auch als problemorientierte Sprachen bezeichnet, da sie sich an den zu lösenden Problemen, z.B. mathematischen Berechnungen oder kommerziellen Anwendungen, orientieren. Dem gegenüber gibt es sogenannte maschinenorientierte Sprachen wie z.B. FORTH, die sich an der Hardware des Rechners orientieren. Dazu gehört als Extremfall natürlich auch die Maschinensprache des Prozessors selbst.

2. BASIC verstehen Sie also gut, sonst würden Sie sich jetzt wohl kaum an die Programmierung in Maschinensprache herantrauen. Da sieht es nämlich mit Ihrem Commodore 64 schon ganz anders aus. Der versteht nämlich BASIC überhaupt nicht. Wie kommt es dann, so werden Sie zu recht fragen, daß er BASIC-Befehle dann so bereitwillig und schnell ausführt? Dafür sorgt der im Betriebssystem Ihres Commodore 64 enthaltene BASIC-Interpreter. Dieser Interpreter präsentiert Ihnen BASIC im Klartext. Geht es jedoch an die Ausführung der von Ihnen geschriebenen Programme, so muß jeder einzelne Befehl von Ihrem Commodore 64 erst interpretiert werden, d.h. in entsprechende, von ihm ausführbare Einzelschritte übertragen werden. Das braucht Sie eigentlich nicht weiter zu stören, denn schließlich erledigt der Interpreter seine Aufgabe doch souverän und für Sie kaum merkbar.

Schauen wir uns an einem einfachen Beispiel an, wie die Arbeit des BASIC-Interpreters vor sich geht:

```
PRINT "HALLO"
```

Wenn Sie diesen Befehl eingeben und RETURN drücken, wird der Befehl Zeichen für Zeichen vom Interpreter gelesen. Sobald er das erste Wort gelesen hat, geht der Interpreter hin und vergleicht dieses Wort mit allen Worten aus seiner Befehlstabelle (z.B. GOTO, FOR, INPUT usw.) und prüft, ob es in der Tabelle enthalten ist. Hat er es gefunden, so merkt er sich, an wievielter Stelle dieses Wort in der Tabelle steht. Diese Position braucht der Interpreter um feststellen, an welcher Stelle innerhalb des Interpreters der PRINT-Befehl steht. Nun kann der eigentliche PRINT-Befehl ausgeführt werden. Auch hier wird wieder zeichenweise gelesen. An den Anführungszeichen merkt der Interpreter, daß Sie einen Text drucken wollen. Er gibt Zeichen für Zeichen aus bis er mit dem nächsten Anführungszeichen das Ende des Wortes entdeckt. Nun prüft er, ob noch weiterer Text folgt. Ist das nicht der Fall, so ist der Befehl ausgeführt und der Interpreter meldet sich mit READY wieder.

Sicherlich werden Sie jetzt sagen: Das ist aber umständlich, da muß es doch was besseres geben. Stimmt! Und genau deshalb wollen wir uns jetzt einmal ansehen, welche Vorteile es hat, direkt in Maschinensprache zu programmieren.

Maschinensprache ist erheblich schneller

Welchen Geschwindigkeitsvorteil bringt die Maschinensprache gegenüber BASIC und warum ist das so? Damit Ihr Rechner BASIC überhaupt versteht, muß er einen sogenannten BASIC-Interpreter haben, der die einzelnen BASIC-Befehle erkennt und ausführt. Dieser Interpreter ist selbst in Maschinensprache geschrieben. Um z.B. den Befehl POKE 1024,10 auszuführen, muß zuerst der Befehl von Interpreter erkannt werden. Danach können die Argumente geholt und die 10 in die Speicherstelle 1024 geschrieben werden. Das Ganze dauert etwa 2 Millisekunden, das sind 2 tausendstel Sekunden; nicht viel Zeit könnte man meinen. Wie sieht das Ganze jetzt in Maschinensprache aus? Hier sind zwei Maschinenbefehle erforderlich:

```
LDA #10
STA 1024
```

Diese beiden Befehle brauchen nur 6 Mikrosekunden, das sind 6 millionstel Sekunden, das ist nicht einmal der dreihunderste Teil davon.

Man kann davon ausgehen, daß ein reines Maschinenprogramm ca. 10 bis 1000 mal schneller ist als ein BASIC-Programm für die gleiche Aufgabe.

Besonders zeitintensive Aufgaben sind z.B. umfangreiche mathematische Berechnungen und besonders auch das Sortieren von Daten. Bei umfangreichen Datenbeständen kann dies in BASIC leicht Stunden dauern. Hier kommt man oft ohne die Maschinensprache nicht aus.

Während man hier mit BASIC lediglich Zeit verschenkt, lassen sich andere Aufgaben ohne Maschinensprache prinzipiell nicht

lösen. Dazu gehört z.B. die Programmierung von Ein/Ausgabebausteinen zur Übertragung von Daten oder Routinen, die im Interrupt ausgeführt werden, eine Technik, die in BASIC nicht zur Verfügung steht. Interrupt heißt zu deutsch Unterbrechung und bedeutet, daß z.B. Bausteine des Rechners oder externe Geräte die Arbeit des Rechners jederzeit unterbrechen können und den Rechner zwingen, diese Geräte entsprechend zu bedienen.

Generell gilt, daß man alle Möglichkeiten seines Rechners nur mit Maschinensprache voll ausschöpfen kann. Dies gilt beim Commodore 64 speziell für die hochauflösende Graphik und den Synthesizer. Dies gilt auch für alle Sachen, die in 'Echtzeit' programmiert werden müssen.

Ein weiterer wichtiger Punkt ist die Speicherplatzausnutzung. Ein gut geschriebenes Maschinenprogramm kann durchaus zehnmal kürzer sein als ein entsprechendes Programm in BASIC. Ein BASIC-Programm von 1 K kann man nicht als sehr groß bezeichnen, für ein Maschinenpropgramm ist dies jedoch schon eine ansehnliche Größe.

Das gleiche gilt auch für die Datenspeicherung. Nur in Maschinensprache läßt sich eine so kompakte Datenspeicherung erreichen, wie sie in BASIC meist nicht zu realisieren. Haben Sie z.B. eine Tabelle, so belegt im günstigsten Fall jedes Feldelement zwei Bytes wenn Sie Integerfelder benutzen, obwohl Sie nur Zahlen zwischen null und hundert abspeichern müssen. In Maschinensprache ist es kein Problem, hier nur ein Byte pro Element zu benutzen. Sie sparen die Hälfte an Speicherplatz. In Maschinensprache können Sie also die jedem Problem optimal angepaßte Datenstruktur wählen.

Nun müssen wir aber fairerweise auch fragen, welche Nachteile wir uns durch die Maschinenspracheprogrammierung einhandeln. Nun - erst müssen wir das Programmieren in Maschinensprache einmal lernen. Wenn Sie jedoch BASIC beherrschen, bringen Sie bereits die Grundvoraussetzungen dafür mit. Außerdem brauchen wir geeignete Werkzeuge, mit denen die Erstellung von

Maschinenprogrammen ähnlich komfortabel vor sich gehen kann, wie Sie dies von BASIC her gewöhnt sind. Dazu finden Sie in diesem Buch einen Assembler, den Sie zur Erstellung von Maschinenprogrammen benutzen können. Ein Nachteil von Maschinenprogrammen ist, daß sie prinzipiell nur auf Maschinen laufen, die den gleichen Prozessortyp enthalten und auch auf diesen von Gerät zu Gerät größere Anpassungen erforderlich sein können. Dem kann man jedoch schon bei der Programmierung Rechnung tragen. Auch ist das Austesten von Maschinenprogrammen ohne entsprechende Werkzeuge nicht so einfach wie in BASIC. Dafür stellen wir Ihnen jedoch ein Simulator-Programm zur Verfügung, das Ihnen hilft, Fehler in Ihren Programmen zu finden.

Zusammenfassend können wir sagen, daß die Maschinenprogrammierung durchaus ihre Berechtigung hat. Viele Aufgaben lassen sich ohne Maschinenprogrammierung nicht lösen und nur mit ihr kann man 'das Letzte' aus seinem Rechner herausholen. Oft wird man auch seine Grundprogramme weiterhin in BASIC schreiben und lediglich Teilprobleme in Maschinensprache formulieren.

Diese Vorgehensweise ist sehr beliebt und begegnet Ihnen dauernd in Form von Programmierhilfen oder BASIC-Erweiterungen. Hier wurde für Sie in Maschinensprache programmiert, damit Sie mit BASIC komfortabler arbeiten können!

Haben Sie erst Ihr erstes eigenes Maschinenprogramm geschrieben, so werden auch Sie feststellen, daß es gar nicht so schwierig ist. Wir hoffen, daß Ihnen die Lektüre dieses Buches dabei behilflich sein kann und Sie zu eigenen Problemlösungen in Maschinensprache inspiriert.

2. Der 6510 - Mikroprozessor

Bevor wir uns mit der Programmierung in Maschinensprache befassen, wollen wir erst einmal den Prozessor selber näher kennenlernen, denn auf ihn bezieht sich später ja unsere ganze Programmierung. Dabei versuchen wir dann auch grundsätzliche Begriffe wie Register, Daten, Adressen sowie Bits und Bytes zu klären. Fangen wir also beim Aufbau des Prozessors an.

Der Mikroprozessor 6510 gehört ebenso zur Familie der 65XX-Prozessoren wie der 8502, der im Commodore 128 seinen Dienst tut. Prozessoren dieser Familie sind sehr verbreitet und finden sich z.B. in allen Commodore-Computern. Der Prozessor hat intern eine Reihe von Registern, in denen alle Operationen ablaufen. Was können wir uns nun unter Registern vorstellen? Ein Mikroprozessor arbeitet digital, d.h. er kann intern nur zwischen zwei Zuständen unterscheiden, die wir uns als EIN und AUS oder als 1 und 0 denken können. Solch ein Schalter kann also nur zwei verschiedene Zustände darstellen. Damit kann man noch nicht viel anfangen. Deshalb faßt man intern jeweils acht solcher Schalter zu einem Register zusammen. Man bezeichnet einen Schalter auch als ein Bit, während acht Bits als ein Byte bezeichnet werden. Ein Register zu 8 Bit kann man sich so vorstellen:

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

In der oberen Reihe sehen Sie die Nummerierung der einzelnen Bits, die von null bis sieben geht. Darunter steht der Inhalt des jeweiligen Bits; entweder eine 0 oder eine 1. Während ein Bit also zwei Zustände repräsentieren kann, können wir mit 8 Bit schon mehr darstellen. Sehen wir uns das einmal genauer an:

Vergleichen wir das einmal mit unseren normalen Dezimalzahlen.

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

Auch hier haben wir die Stellen wieder nummeriert, diesmal von 0 bis drei. Wie erhalten wir nun den Wert dieser Zahl? Nun, jede Ziffer hat einen Wert zwischen null und neun und die nächste Stelle hat den jeweils zehnfachen Wert:

$$4 + 2*10 + 7*10*10 + 5*10*10*10 = 5724$$

Ganz analog können wir jetzt mit unserem Registerinhalt vorgehen. Solche Zahlen nennt man im Gegensatz zu den üblichen Dezimalzahlen Binärzahlen, da jede Stelle nicht 10, sondern nur zwei Werte annehmen kann. Dementsprechend hat die nächsthöhere Stelle auch nicht den zehnfachen, sondern nur den doppelten Wert. Jetzt können wir auch den Wert unseres Registerinhalts berechnen:

$$1 + 0*2 + 0*2*2 + 1*2*2*2 + 0*2*2*2*2 + \\ + 1*2*2*2*2*2 + 1*2*2*2*2*2*2 + 0*2*2*2*2*2*2*2$$

$$= 1 + 0 + 0 + 8 + 0 + 32 + 64 + 0 = 105$$

Wir erhalten also den Wert 105 für unseren Registerinhalt. Diese etwas mühselige Rechnung können wir uns etwas vereinfachen, wenn wir uns die Werte jeder Stelle erst einmal ausrechnen, ähnlich wie wir die Werte der Dezimalstellen ja auch im Kopf haben.

Stelle	Wert
0	$2^0 = 1$
1	$2^1 = 2$
2	$2^2 = 4$
3	$2^3 = 8$
4	$2^4 = 16$
5	$2^5 = 32$
6	$2^6 = 64$
7	$2^7 = 128$

Welchen Wert kann unser Registerinhalt nun maximal haben? Dies ist dann der Fall, wenn alle Stellen den Wert eins haben, es ergibt sich dann 1 + 2 + 4 + 8 + 16 + 32 + 64 + 128 = 255. Der größte mit acht Bit darstellbare Wert ist also 255. Es lassen sich somit insgesamt 256 = 2^8 verschiedene Werte (0 bis 255) darstellen. Diesen Wertebereich kennen Sie vielleicht schon von der PEEK-Funktion, doch darauf kommen wir noch später zurück.

Diese Binärzahlen sind jedoch umständlich zu handhaben. Deshalb hat man noch ein zweites Zahlensystem eingeführt, das zwar die Beziehung zu den Binärzahlen weiterhin erkennen läßt, jedoch mit viel weniger Stellen zur Darstellung auskommt. Teilt man nämlich eine 8-Bit-Binärzahl in zwei 4-Bit-Binärzahlen auf, so kann jede 4-Bitzahl 16 verschiedene Werte annehmen. Man braucht jetzt nur noch ein Zahlensystem, was 16 verschiedene Ziffern hat, so kann man jede 8-Bit-Binärzahl durch zwei Ziffern dieses 16er Systems ausdrücken.

```
7 6 5 4  3 2 1 0
0 1 1 0  1 0 0 1
  6        9
```

Jedes Byte wird dazu in zwei Halbbytes, auch Nibbles genannt, zerlegt. Da diese Nibbles jedoch Werte von 0 bis 15 annehmen können, unser dezimales System jedoch nur die Ziffern 0 bis 9 zur Verfügung stellt, muß man sich für die Werte von 10 bis 15 etwas anderes einfallen lassen. Man nimmt dazu einfach die

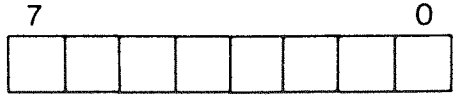
Buchstaben von A bis F. Dieses Zahlensystem nennt man übrigens Hexadezimalsystem. Unsere Wertetabelle sieht dann so aus:

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

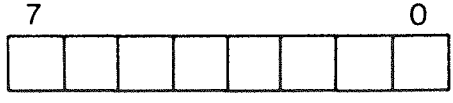
Unser Registerinhalt hat also den hexadezimalen Wert 69. Um die verschiedenen Zahlensysteme zu unterscheiden, kennzeichnet man Hexadezimalzahlen gewöhnlich durch ein vorangestelltes Dollarzeichen '$', während Binärzahlen meist durch ein Prozentzeichen '%' zu erkennen sind. Im Anhang finden Sie eine Umrechnungstabelle für Binär, Hex- und Dezimalzahlen. Wir werden meist mit Hexadezimalzahlen arbeiten, da diese einerseits leicht darstellbar und sich andererseits leicht in die dem Prozessor nahe Binärdarstellung umwandeln lassen. Im Hexadezimalsystem lassen sich also 8-Bit-Werte durch nur 2 Ziffern darstellen.

Nach diesem kurzen Ausflug in die Zahlensysteme wollen wir uns nun weiter mit dem Prozessor beschäftigen.
Dazu sehen wir uns einmal alle Register des Prozessors an.

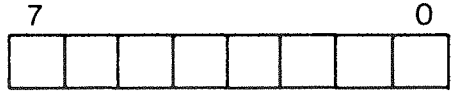
Akkumulator



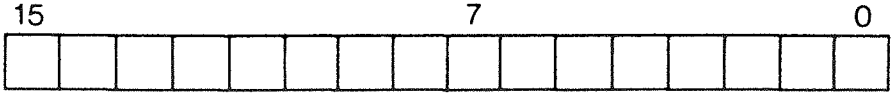
X-Register



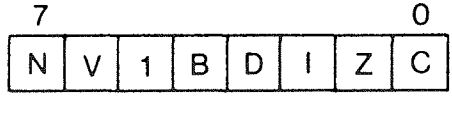
Y–Register



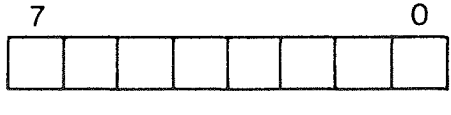
Programmzähler



Statusregister



Stapelzeiger



Wir erkennen insgesamt 6 Register, davon sind 5 Acht-Bit-Register, während das sechste ein 16-Bit-Register ist. Sehen wir uns die einzelnen Register einmal an.

Da ist zunächst der Akkumulator oder Akku, das wichtigste Register des Prozessors. Es ist das Universal-Arbeitsregister des Prozessors, in dem alle Operationen ablaufen. Dies bedeutet, daß alle arithmetischen und logischen Operationen, fast alle Vergleichsbefehle über den Akkumulator laufen. Die verschiedenen Arten der Adressierung bei den ganzen Befehlen lassen sich nur mit dem Akkuregister verwirklichen.

Das X-Register ist das zweite Register des Prozessors. Bei der Abarbeitung von Tabellen wird dieses Register in Zusammenarbeit mit dem Akkumulator immer als Zähler oder als Zeiger auf die einzelnen Tabellenelemente benutzt. Dieses Register wird deshalb auch Indexregister genannt.

Das Y-Register ist wie das X-Register ebenfalls ein Indexregister und dient ähnlichen Zwecken wie das X-Register.

Der Programmzähler ist ein 16-Bit-Register. Sein Inhalt bestimmt, aus welcher Speicherstelle der nächste Befehl geholt wird. Dieses Register wird vom Mikroprozessor selber verwaltet. Sie haben damit normalerweise direkt nichts zu tun.

Der Stapelzeiger zeigt auf den sogenannten Stapelspeicher oder Stack, der für Unterprogramme benutzt wird oder für die kurzfristige und schnelle Speicherung von Daten. Seine Bedeutung lernen wir bei der Programmierung von Unterprogrammen kennen.
Das Statusregister schließlich gibt Auskunft über das Ergebnis des letzten Befehls und ist Grundlage für Entscheidungen und bedingte Befehle. Von den 8 Bit des Statusregisters werden sieben Bits als sogenannte 'Flags' benutzt. 'Flags' oder 'Flaggen' sind direkt abfragbar. So gibt es zum Beispiel Sprungbefehle, die nur ausgeführt werden, wenn ein solches 'Flag' gesetzt oder nicht gesetzt

ist. Diese Flags können 'wahr' (=1) oder 'falsch' (=0) sein und signalisieren so eine bestimmte Bedingung. Das Statusregister ist folgendermaßen aufgebaut:

```
7 6 5 4 3 2 1 0
N V - B D I Z C
```

Die Buchstaben sind Abkürzungen für die Namen der Flags und haben folgenden Bedeutung:

C - Carry Das Carryflag zeigt an, ob bei einer Operation ein Überlauf aufgetreten ist, z.B. wenn beim Addieren zwei Zahlen das Ergebnis größer als 255 wird und sich nicht mehr mit 8 Bit darstellen läßt. In diesem Fall wird das Carry-Flag gesetzt.

Z - Zero Das Zero- oder Nullflag wird immer dann gesetzt, wenn das Ergebnis einer Operation Null ist.

I - Interrupt Disable Dieses Flag entscheidet, ob Interrupts im Programm erlaubt sind. Dieses Flag interessiert uns jedoch im Augenblick noch nicht.

D - Decimal Das Dezimalflag bestimmt, ob Addition oder Subtraktion im Dezimalmodus durchgeführt wird. Auch von diesem Flag werden wir zu Beginn noch keinen Gebrauch machen.

B - Break Das Breakflag zeigt eine Unterbrechung durch den BRK-Befehl an und hat für uns zunächst noch keine Bedeutung.

V - Overflow Das V-Flag wird nur benötigt, wenn wir mit vorzeichenbehafteten Zahlen rechnen. Es zeigt dann einen Überlauf an.

N - Negative Dieses Flag wird immer gesetzt, wenn das Ergebnis einer Operation einen Wert von größer als 127 ergibt, das siebente Bit also gesetzt ist. Die Bezeichnung Negative kommt daher, daß man Werte über $7F als negative Zahlen interpretieren kann.

Für den Anfang werden wir uns nur um das Carry-, das Zero- und das Negativ-Flag kümmern.
Um nun mit einem Prozessor arbeiten zu können, muß er ein Programm erhalten, das er ausführen kann. Ebenso muß er Daten, die er verarbeiten kann, von irgendwo her bekommen und auch wieder abspeichern können. Dazu dient der Arbeitsspeicher des Computers. Dieser Speicher wird nun in einzelne Zellen unterteilt, die jeweils 8 Bit umfassen, also genauso groß sind wie der Akkumulator oder X- und Y-Register. Damit der Prozessor nun auf den Speicher zugreifen kann, muß er die Möglichkeit haben, eine bestimmte Speicherstelle auszuwählen. Diese Auswahl nennt man Adressierung des Speichers. Man gibt jeder Speicherstelle also eine Nummer oder eine Adresse. Diese Adresse gibt der Prozessor wieder als Binärzahl aus. Besteht diese Binärzahl aus 8 Bits, so kann der Prozessor Adressen von 0 bis 255 erzeugen, er kann also 256 Speicherzellen adressieren. Dies ist jedoch für die meisten Anwendungen viel zu wenig. Deswegen hat man für die Adressen 16 Bit vorgesehen. Damit kann man nun 2^{16}= 65536 Speicherstellen adressieren. Man sagt, der Prozessor hat einen 16-Bit Adreßbus im Gegensatz zum 8-Bit Datenbus. Um es noch einmal deutlich zu machen: Der 6510-Prozessor kann 65536 Speicherstellen adressieren, wovon jede einen Wert von 0 bis 255 enthalten kann. Zur bequemeren Handhabung bezeichnet man 2^{10} = 1024 Bytes als ein Kilobyte oder auch 1 K. Der Prozessor kann also 64*1024 = 65536 Bytes oder 64 K adressieren. Beim Commodore 64 ist der komplette Adreßraum auch mit Speicherbausteinen bestückt.
Jetzt können wir auch die Bedeutung des Programmzählers verstehen. Der Programmzähler enthält einen 16-Bit Wert. Dieser 16-Bit-Wert ist die Adresse des nächsten Befehls, den der Mikroprozessor aus dem Speicher holt und dann ausführt. Ein Befehl für den Mikroprozessor kann demnach durch eine Zahl zwischen 0 und 255 ausgedrückt werden. Unser Mikroprozessor kann damit maximal 256 verschiedene Befehle haben. Beim 6510 haben jedoch nicht alle Kodes eine Bedeutung für den Prozessor, es existieren also weniger als 256 Befehle. Gemeint sind natürlich nicht BASIC-Befehle, sondern vom Prozessor direkt zu verstehende Maschinensprachebefehle.

3. Befehle und Adressierungsarten des 6510

Von den 256 möglichen 8-Bit-Kombinationen stellen 151 einen gültigen Befehl für den 6510 dar. Dies beinhaltet jedoch auch mehrere gleichartige Befehle, die sich nur durch die Adressierungsart unterscheiden. Unterschiedliche Befehle gibt es lediglich 56 beim 6510. Dieser Befehlssatz ist leicht zu erlernen. Wir wollen Ihnen jetzt die einzelnen Befehle gruppenweise vorstellen und gehen dabei auch auf die verschiedenen Adressierungsarten ein.

1. Die Ladebefehle

Diese Befehle dienen dazu, Daten aus dem Speicher in ein Register des Prozessors zu holen oder zu laden. Da es drei Arbeitsregister des Prozessors gibt, existieren auch die entsprechenden Ladebefehle dazu.

LDA lade Akkumulator
LDX lade X-Register
LDY lade Y-Register

Am Beispiel dieser Befehle wollen wir nun die verschiedenen Adressierungsarten kennenlernen. Die Adressierungsart gibt darüber Auskunft, wie der Prozessor an die Adresse der Daten im Speicher kommt. Wir geben zum Vergleich immer den entsprechenden BASIC-Befehl an, der dieser Adressierungsart entspricht.

Die unmittelbare Adressierung.

LDA #10
Diese Adressierungart wird durch das Doppelkreuz '#' vor dem Wert, der geladen werden soll gekennzeichnet. Unmittelbare Adressierung bedeutet, daß der Akkumulator direkt mit dem Wert 10 geladen wird. Der entsprechende BASIC-Befehl dazu ist

A = 10
Diese Adressierungsart benutzt man, wenn man ein Register mit

einer Konstanten laden will. Diese Adressierungsart gibt es auch beim X- und Y-Register.

LDX #$7F LDY #$AB

Hier wird das X-Register mit dem Wert $7F gleich 127 und das Y-Register mit dem Wert $AB gleich 171 geladen.

Bei dieser Adressierunsart ist also der Wert, der geladen wird, Bestandteil des Programms, wie es auch bei dem entsprechenden BASIC-Befehl der Fall ist. Im Speicher stehen der Befehl und der Wert in zwei aufeinander folgenden $peicherstellen. Wird ein Maschinenprogramm an dieser Adresse gestartet, so holt es sich den Inhalt dieser Adresse und interpretiert diesen als Befehl. Handelt es sich um $A9 oder dezimal 169, so weiß der Prozessor, daß es sich um den Befehl LDA # handelt. Er holt daraufhin den Inhalt der nächsten Adresse in den Akkumulator. Da dieser Befehl aus insgesamt zwei Bytes besteht - dem Befehl selbst und dem Wert, der geladen wird - erhöht der Prozessor den Programmzähler automatisch um 2. Der Programmzähler zeigt damit auf den nächsten Befehl, der nun vom Prozessor geholt werden kann.

Die absolute Adressierung.

Diese Adressierungsart wird benutzt, wenn ein Register nicht mit einem konstanten Wert, sondern mit dem Inhalt einer Speicherstelle geladen werden soll.
LDA $C0AF

Hier soll der Akkumulator mit dem Inhalt der Speicherzelle $C0AF geladen werden.Wie kann man nun diesen Befehl im Speicher ablegen? Die Adresse $C0AF ist eine 16-Bit-Zahl, eine Speicherzelle kann jedoch nur 8 Bit unterbringen. Hier geht man nun einfach hin, und teilt die 16-Bit-Zahl in zwei 8-Bit-Zahlen auf. Dabei benutzt man folgende Konvention: Nach dem Befehl kommt zunächst der niederwertige Teil der Adresse und dann der höherwertige. In unserem Beispiel also zunächst der Befehlskode $AD (173), dann $AF (175) und schließlich $C0

(192). Der entsprechende Befehl in BASIC wäre

A = PEEK($C0AF)

Die entsprechenden Befehle gibt es natürlich auch für das X- und Y-Register. Die Befehlskodes finden Sie in einer Tabelle im Anhang. Bei der Ausführung eines solchen Befehls weiß der Prozessor, das es sich um die absolute Adressierung handelt und er holt sich nacheinander erst den niederwertigen Adreßanteil, auch Lo-Byte genannt, danach den höherwertigen Anteil, Hi-Byte genannt. Dann wird der Inhalt dieser Adresse geholt und in den Akkumulator geladen. Nach diesem Befehl wird der Inhalt des Programmzählers und drei erhöht und dann der nächste Befehl geholt. Dieser Befehl ist ein 3-Byte-Befehl, während es sich bei der unmittelbaren Adressierung um einen 2-Byte-Befehl handelt. Der 6510-Prozessor kennt auch noch 1-Byte-Befehle, die nur den Prozessor intern betreffen und keine sogenannten Operanden benötigen.

An dieser Stelle wenden wir uns wieder dem Statusregister zu, das beim 6510 eine wichtige Rolle spielt und von fast allen Befehlen beeinflußt wird. Wir werden uns an dieser Stelle auf das Carry-, Zero- und Negative-Flag beschränken. Bei jedem Ladebefehl werden das Zero- und das Negative-Flag beeinflußt. Das Zero- oder Nullflag wird gesetzt, wenn der geladene Wert null ist und wird rückgesetzt, wenn der Wert ungleich null ist. Das Negativflag wird dann gesetzt, wenn der geladene Wert größer als $7F (127) war, das siebente Bit also gesetzt war. Anderenfalls wird das N-Flag gelöscht.

Eine weitere Adressierungsart ist die sogenannte *Zero-Page-Adressierung*, eine Besonderheit der 65XX-Prozessoren. Diese Adressierungsart kann benutzt werden, wenn der Inhalt einer Speicherstelle mit der Adresse zwischen 0 und $FF gleich 255 liegt. Diese Adresse läßt sich durch einen 8-Bit-Wert ausdrücken. Man kommt daher mit einem 2-Byte-Befehl im Gegensatz zum 3-Byte-Befehl bei der absoluten Adressierung aus. Der Befehl belegt also weniger Platz im Speicher und wird außerdem vom Prozessor noch schneller ausgeführt.

Nachteilig ist natürlich der eingeschränkte Adreßbereich von 0 bis 255. Die Bezeichnung Zero-Page kommt daher, daß man sich den Speicher in 256 Seiten oder Pages von je 256 Byte aufgeteilt denken kann, wobei die Adresse 0 bis 255 die Seite 0 bilden. Die Vorteile der Zeropageadressierung führen dazu, daß die Zeropage bei 65XX-System eine besondere Bedeutung hat. In ihr werden meist die Systemvariablen von Betriebssystem und BASIC-Interpreter abgespeichert. Unser Ladebefehl sieht dann so aus:

```
LDA $73
```

Dies wird als Zwei-Byte-Befehl $A5 (165) $73 (115) abgespeichert. In BASIC hieße das wieder

```
A = PEEK($73)
```

Eine weitere wichtige Adressierungsart ist die *indizierte Adressierung*. Hierbei spielen die Indexregister X und Y eine wichtige Rolle. Diese Adressierung benutzt man zur Programmierung von Schleifen.

```
LDA $25B8,X
```

Hierbei handelt es sich um diese absolute mit X indizierte Adressierung. Was passiert nun hierbei? Ganz einfach - Der Prozessor lädt den Akku nicht mit dem Inhalt des Speicherstelle $25B8, sondern er addiert zu diesem Wert erst den Inhalt des X-Registers hinzu. Enthält das X-Register z.B. $35 so findet folgende Rechnung statt:

```
$25B8 + $35 = $25ED
```

Der Akku wird also mit dem Inhalt der Speicherstelle $25DD geladen. Wird diese Programmstelle mehrfach mit verschiedenen X-Werten durchlaufen, so wird jedesmal ein anderer Wert geladen. Diese Adressierungsart ist sehr nützlich zur Programmierung von Schleifen und zur Bearbeitung von Tabellen. Wir werden später noch viele Beispiele sehen. In

BASIC könnten wir diese Adressierungsart so formulieren:

```
A = PEEK($25B8+ X)
```

Dabei bedeutet X den Inhalt des X-Registers.

Anstelle des X-Registers können wir auch das Y-Register zur indizierten Adressierung benutzen, z.B.

```
LDA $25B8,Y
```

Hier wird der Inhalt des Y-Registers zur Adresse addiert um die endgültige Adresse zu erhalten. Man hat damit zwei unabhängige Schleifenvariablen, z.B. zur Programmierung von verschachtelten Schleifen.

Auch bei der Zeropage-Adressierung gibt es die indizierte Adressierung; dabei werden die bekannten Vorteile der Zeropage-Adressierung auf die indizierte Adressierung übertragen. Ein entsprechender Befehl sieht dann so aus:

```
LDA $BA,X
```

Hierbei handelt es sich natürlich wieder um einen Zwei-Byte-Befehl.

Jetzt lernen wir eine Adressierungsart kennen, die nicht ganz so leicht zu verstehen ist, die jedoch eine sehr flexible Programmierung ermöglicht, die *indirekt indizierte Adressierung*. Die indizierte Adressierung haben wir schon kennengelernt, hier wir zusätzlich noch indirekt adressiert. Bei dieser Adressierungsart spielt die Zeropage wieder eine große Rolle. Bei der indirekt indizierten Adressierung bilden zwei aufeinander folgenden Speicherstellen in der Zeropage einen Zeiger auf die aktuelle Adresse. Dabei enthält die erste Speicherzelle das Lo-Byte und die nachfolgende das Hi-Byte der eigentlichen Adresse. Ein Beispiel macht das Ganze deutlich.

Nehmen wir an, die Zeropageadresse $70 enthält den Wert $20, die Adresse $71 enhält den Wert $C8. Diese beiden Speicherstellen bilden also einen Zeiger auf die Adresse $C820. Zusätzlich kommt jetzt noch die Indizierung durch das Y-Register hinzu. Enthält das Y-Register z.B. $B3, so wird dieser Wert zu $C820 dazu addiert und wir erhalten als effektive Adresse $C8D3. Sehen wir uns das nochmal bildlich an:

```
LDA ($70),Y        ($70)   =>   $20
                   ($71)   =>   $C8
                                $C820
                    (Y)    =>    $B3
                                $C8D3
 A = $4F          ($C8D3)  =>    $4F
```

In BASIC sieht das so aus:

```
A = PEEK ( PEEK($70) + 256 * PEEK($71) + Y )
```

Die indirekte Adressierung wird dadurch gekennzeichnet, daß der Operand in Klammern gesetzt wird. Diese Adressierungsart ist sehr leistungsfähig, denn man kann mit einem Zwei-Byte-Befehl auf den kompletten Speicher zugreifen. Auch diese Adressierung wird bevorzugt bei Tabellen und Schleifenverarbeitung benutzt. Sie ist flexibler als die einfache indizierte Adressierung, da man hier nicht nur auf eine Page, sondern auf den ganzen Speicher zugreifen kann. Dazu braucht lediglich der Inhalt des 2-Byte-Zeigers in der Zeropage geändert werden.

Als weitere Adressierungsart gibt es noch die *indiziert indirekte Adressierung*, die im Gegensatz zur oben beschriebenen indirekt indizierten Adressierung nicht mit dem Y-, sondern mit dem X-Register arbeitet. Auch hier bilden wieder zwei aufeinander folgende Adressen in der Zeropage einen Zeiger auf die eigentliche Adresse. Bei der Berechnung der Adresse wird jedoch zuerst der Index zu diesem Zeiger addiert und anschließend die Inhalte der daraus erhaltenen Adressen als Zeiger benutzt. Sehen wir uns das wieder am Beispiel an.

```
LDA ($70,X)          (X)    =>  $08
                =>   ($78)   =>  $40
                     ($79)   =>  $20
                                 $2040
                   ($2040)   =>  $A9
A = $A9
```

In BASIC sieht das so aus:

```
A = PEEK( PEEK($70 +X) + 256*PEEK($70+X+1) )
```

Hier wird also zuerst der Inhalt des X-Registers zum Operand dazu addiert und aus dem Inhalt der daraus erhaltenen Adresse sowie der folgenden . der Zeiger auf die effektive Speicherstelle geholt. Die indiziert indirekte Adressierung wird jedoch im Gegensatz zur oben beschriebenen indirekt indizierten kaum angewandt. Sie brauchen sich diese Adressierung daher für den Anfang nicht zu merken.

Hier noch einmal alle Adressierungsarten und Befehlskode zusammengefaßt:

Adressierungsart	LDA	LDX	LDY
unmittelbar	$A9	$A2	$A0
absolut	$AD	$AE	$AC
zeropage	$A5	$A6	$A4
absolut x-indiziert	$BD	-	$BC
absolut y-indiziert	$B9	$BE	-
zeropage x-indiziert	$B5	-	$B4
zeropage y-indiziert	-	$B6	-
indirekt-indiziert	$B1	-	-
indiziert-indirekt	$A1	-	-

Die relative Adressierung und die Akkumulator-Adressierung werden wir erst bei den zugehörigen Befehlen kennenlernen.

2. Die Speicherbefehle

Das Gegenstück zu den Ladebefehlen sind die Speicherbefehle. Damit können wir den Inhalt eines Registers im Speicher ablegen. Die Befehle dazu heißen

STA
STX
STY

Damit wird der Inhalt des Akkumulators bzw. des X- oder Y-Registers an die durch den Operand gegebene Speicherstelle abgelegt. Als Adressierungsarten stehen die gleichen wie bei den Ladebefehlen zur Verfügung, mit Ausnahme der unmittelbaren Adressierung natürlich, da beim Abspeichern immer eine Adresse angegeben werden muß an die der Registerinhalt gespeichert wird. Da beim Speichern von Registerinhalten sich an den Registern selbst nichts ändert, werden auch keine Flags beeinflußt.

Hier wieder alle Befehlskodes und Adressierungsarten:

Adressierungsart	STA	STX	STY
absolut	$8D	$8E	$8C
zeropage	$85	$86	$84
absolut x-indiziert	$9D	-	-
absolut y-indiziert	$99	-	-
zeropage x-indiziert	$95	-	$94
zeropage y-indiziert	-	$96	-
indirekt-indiziert	$91	-	-
indiziert-indirekt	$81	-	-

Den entsprechenden BASIC-Befehl zu den Speicherbefehlen kennen Sie sicher: Es ist der Poke-Befehl, der den Inhalt einer Variablen an die spezifizierte Adresse in den Speicher schreibt.

```
STA $8000          POKE $8000,A
```

```
STX $C020,Y        POKE $C020+Y,X
STY $F1            POKE $F1,Y
```

Die Speicherbefehle sind je nach Adressierungsart wieder 2- oder 3-Byte-Befehle.Bei den Adressierungsarten stehen Ihnen etwa die Gleichen wie bei den Ladebefehlen zur Verfügung. Beim Speichern von Registerinhalten werden jedoch keine Flags verändert.

Mit den Befehle zum Laden und Speichern haben wir bereits zwei wichtige Gruppen von Befehlen kennengelernt, die der Kommunikation des Prozessors mit dem angeschlossenen Speicher dienen.

3. Transferbefehle innerhalb des Prozessors

Der 6510 Mikroprozessor besitzt noch Befehle, die den Inhalt eines Registers in ein anderes Register kopieren. Damit können Sie z.B. den Inhalt des X-Registers in den Akkumulator holen oder umgekehrt. Dies ist enorm wichtig, da viele Befehle nur mit dem Akkumulator arbeiten. Dabei bleibt jeweils der Inhalt des Quellregisters unverändert; der Wert wird lediglich in das Zielregister kopiert. Bei den Transferbefehlen innerhalb des Prozessors ist bis auf eine Ausnahme immer der Akkumulator beteiligt; ein direktes Kopieren vom X- ins Y-Register oder umgekehrt ist nicht möglich.

Alle Transferbefehle sind Ein-Byte-Befehle, sie brauchen keinen Operanden. Durch den Befehlskode ist die Operation eindeutig bestimmt. Sehen wir uns die Befehle im einzelnen an. Zum besseren Verständnis sind wieder die entsprechenden Befehle in BASIC dazu geschrieben.

```
TAX        X = A
```

Der Inhalt des Akkumulators wird ins X-Register kopiert. Das Z- und das N-Flag werden beeinflußt. Der ursprüngliche Inhalt des Akkumulators bleibt erhalten.

```
TXA         A = X
```

Dies ist der umgekehrte Befehl zu oben. Der Inhalt des X-Registers wird in den Akku kopiert und Z- und N-Flag werden beeinflußt. Auch hier bleibt der Inhalt des X-Registers unverändert.

```
TAY         Y = A

TYA         A = Y
```

Dies sind die entsprechenden Befehle für das Y-Register. Die Funktionen entsprechen exakt den oben beschriebenen, lediglich ist hier das Y-Register anstelle des X-Registers betroffen.

Die nächsten beiden Transferbefehle betreffen den Stackpointer oder Stapelzeiger. Sie werden hier der Vollständigkeit halber erwähnt, obwohl wir die Bedeutung des Stapelzeigers erst später kennenlernen werden. Der Stapelzeiger kann nur mit dem X-Register ausgetauscht werden.

```
TSX         X = SP
```

Der Inhalt des Stapelzeigers wird ins X-Register gebracht. Entsprechend dem Wert werden wieder Z- und N-Flag gesetzt. Der Wert des Stapelzeigers bleibt bei dieser Operation natürlich unverändert.

```
TXS         SP = X
```

Dieser Befehl veranlaßt den umgekehrten Transport. Der Inhalt des X-Registers wird in den Stapelzeiger gebracht. Bei diesem Befehl werden keine Flags beeinflußt, da der Stapelzeiger kein normales Arbeitsregister des Prozessors ist. Der Inhalt des X-Registers bleibt natürlich erhalten.

In der Tabelle sind alle Transferbefehle zusammen mit ihren Befehlskodes enthalten.

Befehl	Befehlskode
TAX	$AA
TXA	$8A
TAY	$A8
TYA	$98
TSX	$BA
TXS	$9A

4. Die arithmetischen Befehle

Jetzt wollen wir den Prozessor rechnen lassen. Der 6510 beherrscht die Addition und die Subtraktion. Bei den arithmetischen Operationen wird folgendermaßen vorgegangen:

Jeder Rechenvorgang benötigt zwei Operanden, die miteinander verknüpft werden und liefert ein Ergebnis. Beim 6510 muß dazu der erste Operand im Akkumulator stehen, während der zweite Operand aus dem Speicher geholt wird. Dazu stehen wieder die verschiedenen Adressierungsarten zur Verfügung. Das Ergebnis wird immer im Akkumulator abgelegt. Dieses Prinzip gilt auch für die später besprochenen logischen Operationen. Der Vergleich mit den entsprechenden BASIC-Befehlen wird dies sicher deutlich machen.

Betrachen wir zunächst die Addition. Dabei wird der Inhalt der adressierten Speicherstelle zum Akkumulator dazu addiert und das Ergebnis wieder im Akkumulator abgelegt.

ADC #$3A A = A + $3A

Wenn wir zwei 8-Bit-Werte (0 bis 255) addieren, so kann es durchaus vorkommen, daß sich das Ergebnis nicht mehr durch eine 8-Bit-Zahl darstellen läßt. Es kann ein Übertrag auftreten. Sehen wir uns dazu die binäre Addition mal an zwei Beispielen an.

ADC #$3A, der Akku soll dabei $9E enthalten.

```
$9E    = %10011110
$3A    = %00111010
```

Die Addition sieht dann so aus:

```
      10011110
+     00111010
-------------

      11011000    = $08
```

Die binäre Addition wird dabei ganz analog zur dezimalen Addition durchgeführt. Dabei gibt es nur 4 Fälle zu unterscheiden:

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

Ein Übertrag wird wie bei der dezimalen Addition bei der nächsten Stelle mit berücksichtigt. In unserem Beispiel erhalten wir also %11011000 oder $D8 als Ergebnis. Das Ergebnis läßt sich noch mit acht Bit darstellen. Anders sieht es beim nächsten Beispiel aus.

ADC #$3A, der Akku soll diesmal $E4 enthalten.

```
$E4    = %11100100
$3A    = %00111010
```

Die Addition sieht dann so aus:

```
      11100100
+     00111010
-------------

     100011110    = $11E
```

In diesem Fall ist ein Übertrag über das 8. Bit hinaus entstanden; das Ergebnis ist %100011110 oder $11E. Der Akku

kann jedoch nur 8-Bit-Zahlen speichern. Für diesen Fall wird nun das Carry- oder Übertragsflag benutzt. Bei jeder Addition kommt der Übertrag ins Carryflag. Entsteht ein Übertrag, wird das Carryflag (auf 1) gesetzt, war kein Übertrag da, wird das Carryflag gelöscht (auf null gesetzt). Man könnte das Carryflag daher auch das neunte Bit des Akkumulators nennen. Will man nun Zahlen addieren, die sich nicht mehr mit 8 Bit darstellen lassen, so kann man die mehrfachgenaue Addition benutzen. Mit zwei 8-Bit-Zahlen lassen sich z.B. schon Zahlen zwischen 0 und 65535 darstellen; ein Wertebereich, der für viele Zwecke schon ausreicht. Um zwei solche Zahlen zu addieren, addiert man einfach nacheinander die unteren 8 Bit und danach die oberen 8 Bit, ganz analog wie es der Prozessor intern mit den 8 Bits nacheinander macht. Tritt jetzt ein Übertrag nach der ersten Addition auf, muß er bei der Addition der nächsten 8 Bit berücksichtigt werden. Um dies zu vereinfachen, wird bei jeder Addition das Carryflag mit addiert. Vor der ersten Addition muß daher das Carryflag gelöscht werden, da hierbei ja noch kein Übertrag berücksichtigt werden darf. Jetzt können wir auch den der Addition entsprechenden BASIC-Befehl angeben.

```
ADC #$3A A = A + $3A + C
```

Ein evtl. Übertrag läßt sich nach jeder Addition am Carryflag ablesen. Neben dem Carryflag werden noch das Zero- und das Negativ-Flag gesetzt, je nachdem ob das Ergebnis null oder das siebte Bit des Ergebnisses gesetzt ist. Ein weiteres Flag, das beeinfußt wird, ist das V-Flag. Dieses Flag wird nur bei vorzeichenbehafteter Arithmetik gebraucht und wird daher vorerst außer acht gelassen. Die Tabelle enthält die Befehlskodes für den ADC-Befehl in den unterschiedlichen Adressierungsarten.

Adressierungsart	ADC
unmittelbar	$69
absolut	$6D
zeropage	$65

```
absolut x-indiziert  $7D
absolut y-indiziert  $79
zeropage x-indiziert $75
indirekt-indiziert   $71
indiziert-indirekt   $61
```

Die Subtraktion geschieht analog der Addition. Hier wird vom Akkuinhalt das adressierte Byte abgezogen und das Ergebnis wieder im Akkumulator abgelegt. Auch hier gibt es die Möglichkeit, daß das Ergebnis nicht im Bereich von 0 bis 255 liegt. Bei der Subtraktion kann jedoch kein Überlauf, sondern nur ein Unterlauf vorkommen. Das Ergebnis wäre in solch einem Falle kleiner als Null. Auch hier nimmt man wieder das Carryflag zur Hilfe. Da Überlauf und Unterlauf jedoch gegensätzliche Bedeutung haben, wird hier ein Unterlauf durch ein gelöschtes Carryflag angezeigt. Ein gesetztes Carryflag bedeutet dann, daß kein Unterlauf aufgetreten ist. Entsprechend muß bei einer mehrstelligen Subtraktion zu Beginn das Carryflag gesetzt, um zu kennzeichnen, daß kein Unterlauf berücksichtigt werden muß. In BASIC könnten wir dies so formulieren:

```
SBC #$3A A = A - $3A - (1-C)
```

Die binäre Subtraktion wir ganz analog zur Addition durchgeführt. Dabei gibt es wieder nur 4 Fälle zu unterscheiden:

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

Enhält der Akku z.B. $7F, so sieht die binäre Darstellung so aus:

```
$7F    %01111111
$3A    %00111010
```

Bei gesetztem Carryflag sieht die Subtraktion dann so aus:

```
        01111111
-       00111010

        01000101
```

Das Ergebnis ist also %01000101 oder $45. Da kein Unterlauf vorkam, wird das Carryflag wieder gesetzt. Beim nächsten Beispiel sieht es etwas anders aus. Der Akkku soll diesmal $1E enthalten, das Carryflag wird wieder gesetzt.

```
$1E     %00011110
$3A     %00111010
```

Bei der Subtraktion ergibt sich nun folgendes:

```
        00011110
-       00111010

        11100100
```

Wir erhalten als Ergebnis %11100100 bzw. $E4. Da ein Unterlauf aufgetreten ist, wird das Carryflag gelöscht. Wie ist dieses Ergebnis nun zu interpretieren? Überlegen wir uns nun einmal, welches Ergebnis wir bei der normalen dezimalen Subtraktion erhalten würden. Dezimal lautet unsere Aufgabe 30 - 58. Wir erhalten die negative Zahl -28. Wir haben als Ergebnis $E4 gleich 228 erhalten. Haben diese Zahlen nun irgend etwas mit einander zu tun? Subtrahieren wir unser Ergebnis von 256, so erhalten wir 28. Das gelöschte Carryflag nach der Subtraktion sagt uns, daß wir das Ergebnis als negative Zahl interpretieren müssen. Man spricht von sogenannten 'Zweierkomplement' einer Zahl. Wir können den Wert der negativen Zahl bestimmen, indem wir sämtliche Bits umkehren und zu diesem Ergebnis dann eins addieren.

```
        11100100
gibt    00011011
```

```
plus 1  00011100
```

Wir erhalten %00011100 oder $1C gleich 28.

Bei den arithmetischen Operationen sollten wir uns merken, daß vor einer Addition das Carryflag gelöscht werden muß und ein Übertrag durch ein gesetzes Carryflag erkannt werden kann. Bei der Subtraktion muß das Carryflag gesetzt sein und ein Unterlauf wird durch ein gelöschtes Carryflag erkannt. In diesen Fall erhält man eine negative Zahl im Zweierkompement.

Die Tabelle enthält wieder die Befehlskodes für alle Adressierungsarten.

Adressierungsart	SBC
unmittelbar	$E9
absolut	$ED
zeropage	$E5
absolut x-indiziert	$FD
absolut y-indiziert	$F9
zeropage x-indiziert	$F5
indirekt-indiziert	$F1
indiziert-indirekt	$E1

5. Die logischen Befehle

Die logischen Befehle verknüpfen zwei Werte miteinander. Wie bei den arithmetischen Befehlen muß dabei ein Operand im Akkumulator stehen, während der zweite entsprechend der Adressierungsart aus dem Speicher geholt wird. Nach der Verknüpfung steht das Ergebnis wieder im Akku. Der 6510 kennt drei verschiedene Arten der logischen Verknüpfung.

Die UND-Verknüpfung

Bei der UND-Verknüpfung wird je ein Bit des Akkus mit dem entsprechenden Bit des Operanden verknüpft. Sind ein Bit des

Akkus UND ein Bit des Operanden gleich 1, so ist auch das entsprechende Bit des Ergebnisses eins.

0 AND 0 = 0
0 AND 1 = 0
1 AND 0 = 0
1 AND 1 = 1

An einem Beispiel soll die stellenweise Verknüpfung zwischen Akku und Operand deutlich werden.

AND #$37

Enthält der Akku z.B. $5D, sieht die Verknüpfung so aus:

```
$5D   01011101
$37 _ 00110111
$15   00010101
```

Als Ergebnis erhalten wir %00010101 oder $15. Dies entspricht exakt dem BASIC-Befehl AND:

A = A AND $37

in unserem Falle A = $5D AND $37 bzw. A = 93 AND 55 . Wir erhalten als Ergebnis 21 gleich $15. Bei der AND-Verknüpfung sind das N- und das Z-Flag betroffen. Ein Ergebnis von Null setzt das Z-Flag, Ergebnisse größer $7F (127) setzen das N-Flag.

Die Tabelle enthält die Befehlskodes zu jeder Adressierungsart.

Adressierungsart	AND
unmittelbar	$29
absolut	$2D
zeropage	$25
absolut x-indiziert	$3D

```
absolut y-indiziert  $39
zeropage x-indiziert $35
indirekt-indiziert   $31
indiziert-indirekt   $21
```

Die ODER-Verknüpfung

Bei der ODER-Verknüpfung wird je ein Bit des Akkus mit dem entsprechenden Bit des Operanden verknüpft. Ist ein Bit des Akkus oder ein Bit des Operanden gleich 1, so ist auch das Ergebnis gleich 1.

0 ORA 0 = 0
0 ORA 1 = 1
1 ORA 0 = 1
1 ORA 1 = 1

Aus der Wertetabelle ersehen Sie, daß es sich um ein sogenanntes 'inklusives' oder handelt. Das Ergebnis ist eins, wenn der erste Operand und/oder der zweite Operand eins sind, also nicht im Sinne eines entweder/oder. Bei der ODER-Verküpfung werden wieder Z- und N-Flag gesetzt.

ORA #$37

Enthält der Akku wieder $5D, sieht die Verknüpfung so aus:

```
$5D   01011101
$37 _ 00110111
$7F   01111111
```

Als Ergebnis erhalten wir %01111111 oder $7F. Dies entspricht exakt dem BASIC-Befehl OR:

A = A OR $37

in unserem Falle A = $5D OR $37 bzw. A = 93 OR 55 . Wir erhalten als Ergebnis 127 gleich $7F.

Die Tabelle enthält die Befehlskodes zu jeder Adressierungsart.

Adressierungsart	ORA
unmittelbar	$09
absolut	$0D
zeropage	$05
absolut x-indiziert	$1D
absolut y-indiziert	$19
zeropage x-indiziert	$15
indirekt-indiziert	$11
indiziert-indirekt	$01

Die Exklusiv-Oder-Verknüpfung

Bei der Exklusiv-Oder-Verknüpfung werden ebenfalls die entsprechenden Bits von Akku und Operand verknüpft. Das Ergebnis ist eins, wenn entweder das eine oder das andere Bit gleich eins ist, nicht jedoch wenn beide Bits eins sind. Die Wertetabelle sieht so aus:

0 EOR 0 = 0
0 EOR 1 = 1
1 EOR 0 = 1
1 EOR 1 = 0

Das Ergebnis der Verknüpfung ist also dann eins, wenn die beiden Bits unterschiedlich sind. Auch hierbei werden wieder entsprechend dem Ergebnis N- und Z-Flag gesetzt. Einen entsprechenden BASIC-Befehl dazu gibt es nicht; Sie können ihn ersetzen durch A EOR B := (A OR B) AND NOT (A AND B). Unser Beispiel sieht dann so aus:

```
EOR #$37
```

Enthält der Akku wieder $5D, sieht die Verknüpfung so aus:

```
$5D   01011101
$37 _ 00110111_
$6A   01101010
```

Als Ergebnis erhalten wir %01101010 oder $6A (106)

Die Tabelle enthält die Befehlskodes zu jeder Adressierungsart.

Adressierungsart	EOR
unmittelbar	$49
absolut	$4D
zeropage	$45
absolut x-indiziert	$5D
absolut y-indiziert	$59
zeropage x-indiziert	$55
indirekt-indiziert	$51
indiziert-indirekt	$41

Der BIT-Befehl

Eine Besonderheit der 65XX-Prozessoren ist der BIT-Befehl. Dieser Befehl verändert keine Registerinhalte sondern wirkt nur auf die Flags. Was dabei passiert ist folgendes: Es wird zwischen dem Akku und der adressierten Speicherstelle eine UND-Verknüpfung vorgenommen. Ist das Ergebnis null, so wird das Z-Flag gesetzt, ansonsten wird es gelöscht. Zusätzlich wird das 6. Bit der adressierten Speicherstelle in das V-Flag übernommen. Das 7. Bit kommt ins N-Flag. Damit kann man ohne Zerstörung von Registerinhalten diese beiden Bits einer Speicherstelle überprüfen. Sehen wir uns dazu folgendes Beispiel an:

```
BIT $1234
```

Im Akku soll $10 stehen, die Adresse $1234 soll $43 enthalten. Die UND-Verknüpfung liefert folgendes Ergebnis:

```
$10   % 00010000
$43 %   01000011
AND   % 00000000
```

Die AND-Verknüpfung liefert null, das Z-Flag wird also gesetzt. Das V-Flag wird gleich dem 6. Bit des Operanden, also 1, während das N-Flag gelöscht wird. Das Ergebnis ist also:

$$Z = 1;\ V = 1;\ N = 0.$$

Den BIT-Befehl gibt es in zwei Adressierungsarten:

Adressierungsart	BIT
zeropage	$24
absolut	$2C

6. Die Vergleichsbefehle

Die Gruppe der Vergleichsbefehle ermöglicht Vergleiche zwischen den Registerinhalten und den Inhalten von Speicherstellen. Diese Befehle verändern weder die Register noch die Speicherinhalte, sondern beeinflussen nur die Flags. Aufgrund dieser Flags kann man dann Entscheidungen treffen. Die zugehörigen Befehle werden im nächsten Kapitel besprochen.

Bei den Vergleichsbefehlen wird das adressierte Byte vom Register abgezogen. Dabei wird der Registerinhalt jedoch nicht verändert, sondern es werden nur das C-, N- und Z-Flag entsprechend dem Ergebnis gesetzt. Die Vergleichsbefehle gibt es für alle drei Arbeitsregister des Prozessors.

Der CMP-Befehl

Dieser Befehl vergleicht (to compare) den Inhalt des Akkus mit dem Inhalt der adressierten Speicherstelle. Dazu wird der

Inhalt des Operanden vom Akku abgezogen. Tritt dabei ein Unterlauf auf, so wird das Carryflag gelöscht, ansonsten wird es gesetzt. Ist das Ergebnis null, sind also beide Werte gleich, so wird das Z-Flag gesetzt, sonst wird es rückgesetzt. Ist das Ergebnis größer als $7F (127) wird das N-Flag gesetzt, ansonsten wird es gelöscht. Sehen wir uns das wieder an einem Beispiel an.

```
CMP #$30
```

Im ersten Fall soll der Akku $50 enthalten. Es wird also die Rechnung $50 - $30 durchgeführt. Das Ergebnis ist $20. Da dabei kein Unterlauf aufgetreten ist, wird das Carryflag gesetzt. Das Zeroflag wird gelöscht, da das Ergebnis ungleich null ist. Das N-Flag wird ebenfalls gelöscht, da die Zahl nicht größer als $7F ist. Wir erhalten also folgendes Ergebnis:

C = 1; Z = 0; N = 0

Enthält der Akku dagegen $30, so ergibt die Subtraktion null. Jetzt wird das Carryflag ebenfalls gesetzt, da noch kein Unterlauf aufgetreten ist. Da das Ergebnis jedoch null war, wird diesmal das Zeroflag gesetzt. Das N-Flag wird auch diesmal nicht gesetzt, da das Ergebnis nicht größer als $7F ist.

C = 1; Z = 1; N = 0

Im dritten Fall soll der Akku $10 enthalten. Die Subtraktion liefert diesmal $10 - $30 = $F0 und ein Unterlauf tritt auf. Das Carryflag wird also gelöscht, ebenso das Z-Flag, da das Ergebnis ungleich null ist. Das N-Flag wird diesmal gesetzt.

C = 0; Z = 0; N = 1

Für die Praxis kann man sich folgendes merken:

C = 1 bedeutet >= größer oder gleich

Z = 1 bedeutet = gleich
C = 0 bedeutet < kleiner

Zur Entscheidung auf 'größer' müssen daher zwei Flags geprüft werden:

Z = 0 und C = 1

Die Befehle zum Vergleichen ändern also nur die Flags und keine Registerinhalte; sie sind die Grundlagen für bedingte Befehle, die wir im nächsten Kapitel kennenlernen werden.

Die Tabelle enthält die Befehlskodes zu jeder Adressierungsart.

Adressierungsart	CMP
unmittelbar	$C9
absolut	$CD
zeropage	$C5
absolut x-indiziert	$DD
absolut y-indiziert	$D9
zeropage x-indiziert	$D5
indirekt-indiziert	$D1
indiziert-indirekt	$C1

Der CPX-Befehl

Ganz analog wie der CMP-Befehl arbeitet der CPX-Befehl. Hierbei wird jedoch nicht der Inhalt der Akkus mit dem Inhalt der adressierten Speicherstelle verglichen, sondern der Inhalt des X-Register. Auch hier wird der Inhalt des X-Registers nicht verändert. Das oben gesagt gilt entsprechend auch für den CPX-Befehl. Für den CPX-Befehl stehen jedoch nicht so viele Adressierungsarten zur Verfügung.

Adressierungsart	CPX
unmittelbar	$E0
absolut	$EC
zeropage	$E4

Der CPY-Befehl

Hier gilt wieder das gleiche wie oben; das Register mit dem verglichen wird ist jedoch diesmal das Y-Register. Auch hier stehen nur drei Adressierungsarten zur Verfügung.

Adressierungsart	CPY
unmittelbar	$C0
absolut	$CC
zeropage	$C4

7. Befehle zur bedingten Verzweigung

Jetzt lernen wir die Befehle kennen, mit denen man Entscheidungen treffen kann. Grundlage dieser Entscheidungen sind die Zustände der Flags, die wir schon ausführlich kennengelernt haben. Folgende vier Flags können als Entscheidungskriterium herangezogen werden: Das Z-Flag, das N-Flag, das C-Flag und das V-Flag. Für jedes Flag existieren zwei bedingte Sprungbefehle: Beim ersten wird verzweigt, wenn das Flag gesetzt ist, der zweite verzweigt, wenn das Flag rückgesetzt ist. Damit der Prozessor weiß, an welche Stelle er bei einer erfüllten Bedingung hin verzweigen soll, wird noch ein Operand benötigt, der die Zieladresse angibt. Das ist die Adresse, von der sich der Prozessor den Befehlskode holt. Man könnte jetzt als Operand einfach die 16-Bit-Adresse des Ziels der Verzweigung angeben. Dies würde in einem 3-Byte-Befehl resultieren. Da bedingte Verzweigungen jedoch meist nur über kurze Distanzen springen müssen, hat man sich eine andere Adressierungsart ausgedacht, die relative Adressierung. Dabei gibt man nicht die absolute Adresse an sondern nur die Distanz zum augenblicklichen Stand des Programmzählers. Für diese Distanz gibt man sich nun mit einem 8-Bit-Wert zufrieden.

Mit diesem 8-Bit-Wert kann man nun, wie wir bereits wissen, 256 verschiedene Zahlen darstellen. Man könnte also um maximal 256 Bytes nach vorne springen. Oft ist es jedoch erforderlich ein Stück zurück zu springen, z.B. an den Anfang einer Schleife. Dazu müßte man mit einer 8-Bit-Zahl auch negative Werte darstellen können. Erinnern wir uns dazu an den SBC-Befehl; die Subktraktion. Auch da konnten ja negative Werte auftreten. Man betrachtet nun einfach alle Zahlen, bei denen das 7. Bit gesetzt ist, als negative Zahlen im Zweierkomplement:

%10000000	$80	-128
%10000001	$81	-127
...	...	....
%11111110	$FE	-2
%11111111	$FF	-1
%00000000	$00	0
%00000001	$01	1
%00000010	$02	2
...	...	....
%01111110	$7E	126
%01111111	$7F	127

Sie sehen also, daß das oberste (7.) Bit entscheidet, ob eine Zahl negativ ist (daher auch die Bezeichnung des N-Flags). Sehen wir uns nun an, wie wir die Distanz oder den Offset bei einem relativen Sprung berechnen können. Dabei ist der Bezugspunkt die Adresse des nächsten Befehls hinter dem bedingten Sprung. Folgendes Beispiel: Der Sprungbefehl steht an Adresse $C47A und wir wollen nach SC4BF springen.

$C47A	Adresse des Sprungbefehls
$C47C	Adresse des nachfolgenden Befehls
$C4BF	Zieladresse

Jetzt bilden wir einfach die Differenz zwischen Ziel- und Ausgangsadresse:

$C4BF - $C47C = $43

Der Operand für unseren Sprungbefehl ist also $43. Wie sieht es nun bei einem Rückwärtssprung aus? Dazu wollen wir jetzt an die Adresse $C440 springen. Hier kann man auf verschiedene Weise den Offset berechnen. Die erste Methode bildet wieder einfach die Differenz, die diesmal negativ wird.

$C440 - $C47C = $FFC4 mit Unterlauf

Von dem Ergebnis brauchen wir lediglich das niederwertige Byte $C4 als Operand für den Befehl benutzen. Sie können jedoch auch die positive Differenz bilden und von dem Ergebnis das Zweierkomplement nehmen.

$C47C - $C440 = $3C

Davon nun das Zweierkomplement

```
    % 00111100
    % 11000011
+   ----------
                1
    % 11000100   =  $C4
```

Auch hier sind wir wieder auf den Offset von $C4 gekommen.

Was bringt nun diese relative Adressierung für Vorteile? Zuerst einmal werden nur zwei anstatt drei Bytes im Speicher belegt. Zusätzlich zur Speicherplatzersparnis wird so ein Zwei-Byte-Befehl vom Prozessor auch schneller ausgeführt. Der wichtigste Vorteil ergibt sich jedoch aus der relativen Adressierung - relativ zur Ausgangsposition. Der Befehl besagt ja schließlich nur, daß um soundsoviel Bytes nach vorne oder nach hinten gesprungen werden soll. Legt man solch einen Programmabschnitt an einer anderen Stelle im Speicher ab, so braucht am Programm nichts geändert werden - die Differenz zu der Sprungstelle bleibt ja gleich.

Wird eine Sprungadresse jedoch absolut angegeben, so muß eine solche Adresse bei einer Programmverschiebung immer mit geändert werden, ein Umstand, den wir bei der relativen Adressierung vermeiden. Ein Nachteil der relativen Adressierung ist jedoch der eingschränkte Adressbereich, den wir nur erreichen können. Lediglich um 129 Bytes nach vorne oder um 126 Bytes nach hinten, vom Sprungbefehl aus gerechnet, kann gesprungen werden. Dies hat sich jedoch in der Praxis als kein großes Hindernis erwiesen, da selten über größere Distanzen gesprungen werden muß.

Sollten Sie die Adreßberechnung bei der relativen Adressierung als sehr umständlich gefunden haben, so können wir Sie beruhigen: Es ging uns hierbei nur darum, daß Sie das Prinzip verstanden haben. Später wird der Assembler diese Arbeit für uns übernehmen; Sie brauchen ihm nur das Sprungziel anzugeben. Der Assembler macht Sie auch darauf aufmerksam, wenn Sie einen Sprung außerhalb der erlaubten Distanzen versucht haben. Doch nun zu den Verzweigungsbefehlen selbst.

Verzweigungen aufgrund des Zero-Flags

Eine Verzweigung ('branch') bei gesetztem Zero-Flag erfolgt mit dem Befehl 'branch on equal', abgekürzt BEQ. Soll bei rückgesetztem Zero-Flag gesprungen werden, heißt der Befehl 'branch on not equal', BNE.

Verzweigungen aufgrund des Carry-Flags

Hier heißen die Befehle 'branch on carry set' BCS für Verzweigungen bei gesetzem Carryflag und 'branch on carry clear' BCC bei gelöschtem Carryflag.

Verzweigungen aufgrund des Negative-Flags

Ist das N-Flag gesetzt, so wird beim Befehl 'branch on minus' BMI gesprungen, um bei gelöschtem Negative-Flag zu springen, müssen Sie den Befehl 'branch on plus' BPL verwenden.

Verzweigungen aufgrund des Overflow-Flags

Auch das V-Flag kann als Grundlage für bedingte Verzweigungen benutzt werden. Die entsprechenden Befehle sind 'branch on oVerflow set' BVS und 'branch on oVerflow clear' BVC. Entsprechend der untergeordneten Bedeutung des V-Flags werden auch diese beiden Befehle selten benutzt.

Die Tabelle faßt nochmal alle Befehle zur bedingten Verzweigung mit ihren Befehlskodes zusammen.

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

8. Sprungbefehle

Im Gegensatz zu den bedingten Sprungbefehlen (branch-Befehle), die wir oben kennengelernt haben, springt der unbedingte Sprungbefehl an eine absolut angegebene Adresse. Dieser Befehl ist von keiner Bedingung abhängig und wird immer durchgeführt. Beim Sprungbefehl gibt es neben der absoluten Adressierung, wie wir sie schon von den meisten Befehlen her kennen, noch die indirekte Adressierung, eine Besonderheit des Sprungbefehls. Bei diesem Befehl wird nicht an die angegebene Adresse gesprungen. Diese Adresse gibt nur an, wo die eigentliche Zieladresse zu holen. Dabei werden wieder zwei aufeinander folgende Adressen als Zeiger im Format Lo-Byte, Hi-Byte verwendet. Ein Beispiel soll dies deutlich machen.

JMP ($0302) indirekter Sprung über Adresse $0302

Die eigentliche Adresse wird jetzt aus den Speicherstellen $0302 und $0303 geholt. Steht dort z.B. $40 und $C8, so wird an die Adresse $C840 verzweigt. Diese Art der Adressierung ist nur beim JMP-Befehl verwirklicht. Sie können Adressen aus dem gesamten zur Verfügung stehenden Speicherbereich verwenden. Die Tabelle enthält die Befehlskodes für beide Adressierungsarten.

Adressierungsart	JMP
absolut	$4C
indirekt	$6C

Die Betriebssysteme von C 64 und C 128 machen von dieser Art der Adressierung reichlich Gebrauch. Von $300 bis $330 finden Sie nur Sprungvektoren. Dies hat den enormen Vorteil, daß Sie durch einfaches Ändern dieser Vektoren Routinen des Betriebssystems und des BASIC-Interpreters nach eigenen Erfordernissen ändern können.

9. Die Zählbefehle

Zur effektiven Programmierung von Schleifen und Zählern besitzt der 6510 Befehle, die Register- oder Speicherinhalte um eins erhöhen (inkrementieren) oder erniedrigen (dekrementieren) können. Diese Inkrementbefehle entsprechen zusammen mit der bedingten Verzweigung dem 'NEXT'-Befehl im BASIC. Mit den Dekrementbefehlen läßt sich der 'STEP-1'-Befehl simulieren.

INX

Der Inhalt des X-Registers wird um eins erhöht. Entsprechend dem Ergebnis werden das N- und Z-Flag gesetzt. In BASIC ließe sich der Befehl so formulieren:

X = X + 1

Wird ein Wert von $FF inkrementiert, so wird der Übertrag nicht berücksichtigt (das Carryflag wird nicht gesetzt). Der Inhalt wird dann zu null und das Z-Flag wird gesetzt.

INY

Dies ist der entsprechende Befehl, der das Y-Register um eins erhöht. Die Beeinflussung der Flags ist analog.

Einen Befehl um den Akkuinhalt um eins zu erhöhen oder zu erniedrigen gibt es beim 6510 nicht.

INC

Dieser Befehl erhöht den Inhalt einer Speicherstelle um eins. Dabei werden abhängig von Ergebnis wieder das Z- und das N-Flag gesetzt. Dieser Befehl unterscheidet sich von den bisherigen Befehlen dadurch, daß hier der Inhalt einer Speicherstelle zuerst gelesen, dann um eins erhöht und dann wieder abgespeichert wird (Read - Modify - Write). Die Befehle, die wir bisher kennengelernt haben, haben entweder nur eine Speicherstelle gelesen oder geschrieben, jedoch nie beides nacheinander. Beim INC-Befehl wird der Inhalt des Akkus nicht verändert.

In BASIC könnten wir das so formulieren:

POKE M, PEEK(M) + 1

Dabei ist M die adressierte Speicherstelle.

Nun die entsprechenden Befehle zur Erniedrigung von Register- bzw. Speicherinhalten.

DEX

Dieser Befehl vermindert den Inhalt des X-Registers um eins.

Beim Vermindern von $00 auf $FF wird kein Carryflag gesetzt. Das Z- und das N-Flag werden abhängig vom Ergebnis gesetzt. In BASIC könnte man dafür schreiben

X = X - 1

DEY

Dies ist der analoge Befehl, der den Inhalt des Y-Registers um eins vermindert. Die Flags werden analog dem DEX-Befehl gesetzt.

DEC

Mit diesem Befehl kann der Inhalt einer Speicherzelle vermindert werden, ohne daß der Inhalt des Akkumulators dabei verloren geht. Dabei gilt sinngemäß das Gleiche wie beim INC-Befehl.

Hier wieder die Tabelle mit den Befehlskodes

Befehl	Befehlskode
INX	$E8
INY	$C8
DEX	$CA
DEY	$88

Adressierungsart	INC	DEC
absolut	$EE	$CE
zerpage	$E6	$C6
absolute x-indiziert	$FE	$DE
zeropage x-indiziert	$F6	$D6

10. Befehle zur Beeinflussung der Flags

Außer durch die Befehle, die je nach Ergebnis die Flags beeinflussen, können die Flags vom Programmierer auch direkt gesetzt oder gelöscht werden. Dies ist ja z.B. vor Additionen und Subtraktionen erforderlich. Diese Befehle brauchen natürlich keine Operanden und sind daher Ein-Byte-Befehle.

Das Carry-Flag

Mit dem Befehle SEC ('set carry') wird das Carry-Flag gesetzt, mit CLC ('clear carry) wird das Flag gelöscht.

Der Befehl SEC muß vor jeder Subtraktion, der Befehl CLC vor jeder Addition angewendet werden, da sonst die Ergebnisse falsch werden können.

Das Dezimalflag

Dieses Flag entscheidet darüber, ob der Prozessor Additionen und Subtraktionen binär durchführt (bei gelöschtem Flag, so wie wir es bisher kennengelernt haben) oder ob dezimal addiert und subtrahiert wird. Dies ist bei gesetztem Dezimal-Flag der Fall. Der Prozessor arbeitet dann mit sogenannten BCD-Zahlen (binary coded decimals). Der Befehl SED ('set decimal') setzt das Flag, CLD ('clear decimal') löscht das Flag.

Das Interruptflag

Das I-Flag entscheidet darüber, ob der Prozessor bereit ist, einen Interrupt anzunehmen. Ist das I-Flag gesetzt mit SEI ('set interrupt disable'), wird kein Interrupt angenommen, ist es gelöscht mit CLI ('clear interrupt disable') kann der Prozessor einen Interrupt annehmen.

Das Overflowflag

Das V-Flag kann per Befehl nur gelöscht werden. Dazu dient der Befehl CLV ('clear oVerflow').

Die Tabelle enthält die Befehlskodes dieser Ein-Byte-Befehle.

Befehl	Befehlskode
CLC	$18
SEC	$38
CLD	$D8
SED	$F8
CLI	$58
SEI	$78
CLV	$B8

11. Die Verschiebebefehle

Der 6510-Prozessor kennt noch eine Gruppe von Befehlen, für die es kein direktes Äquivalent in BASIC gibt, die Verschiebebefehle. Dabei wird das Bitmuster des Akkumulators oder der adressierten Speicherstelle um eine Position nach links oder nach rechts verschoben. Bezieht sich dieser Befehl auf den Akkumulator, so spricht man auch von Akkumulator-Adressierung. Je nach Adressierungsart kann es sich also um Ein-, Zwei- oder Drei-Byte-Befehle handeln. Wird dabei eine Speicherstelle adressiert, so handelt es sich wie beim INC- und DEC-Befehl um ein Lesen mit anschließendem Schreiben. Der Akkuinhalt bleibt bei dieser Adressierungsart unverändert.

ASL

ASL bedeutet 'arithmetik shift left'. Dabei wird das adressierte Byte um eine Position nach links geschoben. In die frei werdende Position des 0. Bits kommt eine 0, das herausgeschobene 7. Bit kommt in das Carry-Flag. Sehen wir uns das am Beispiel des Akkus an.

ASL A Der Akku soll $47 enthalten.

```
$47  %01000111
     %10001110    $8E, C = 0
```

In unserem Falle erhalten wir das Ergebnis $8E und das Carry-Flag wird gelöscht, da der Akku in der 7. Position eine null enthielt. Vergleichen wir den Akkuinhalt vor und nach dem Verschieben, so stellen wir fest, daß der Inhalt des Akkus verdoppelt wurde. Würden wir eine normale Dezimalzahl um eine Position nach links schieben, so erhielten wir den zehnfachen Wert. Beim binären System hat die nächst höhere Stelle jedoch nur den doppelten Wert. Mit dem ASL-Befehl haben wir also eine einfache Möglichkeit gefunden, eine Zahl zu verdoppeln. Probieren wir jetzt noch ein weiteres Beispiel:

ASL A Der Akku soll nun $CD enthalten.

```
$CD  %11001101
      10011010    $9A, C = 1
```

Auch hier erhalten wir den doppelten Wert, das gesetzte Carry-Flag sagt jedoch, daß ein Übertrag aufgetreten ist. Das doppelte von $CD (205) ist also $19A (410).

LSR

Der LSR-Befehl ('logical shift right') entspricht dem ASL-Befehl; hier wird jedoch nach rechts verschoben. Dabei wird in das 7. Bit eine 0 geschoben und das 0. Bit kommt ins Carry. Ein Beispiel kann dies wieder deutlich machen.

LSR A Der Akku soll $CA enthalten.

```
$CA  %11001010
      01100101    $65, C = 0
```

Wir erhalten als Ergebnis $65 bei gelöschtem Carry. Wie Sie sicher bemerkt haben, bedeutet ein Verschieben nach rechts ein halbieren des Wertes. Das Carry-Flag gibt den Inhalt von

Bit 0 wieder. Anders ausgedrückt zeigt uns das Carry, ob beim Teilen durch 2 ein Rest aufgetreten ist. Damit läßt sich feststellen, ob eine Zahl ungerade ist. Der LSR-Befehl schiebt das unterste Bit ins Carry. Das Carry-Flag läßt sich dann mit BCC oder BCS testen. Wird mit dem LSR-Befehl eine Speicherstelle adressiert, bleibt der Inhalt des Akkus erhalten.

ROL

Mit dem ROL-Befehl ('rotate left') können wir eine Speicherstelle oder ein Register zyklisch nach links verschieben oder die Bits 'rotieren'. Dabei wird in das 0. Bit der Inhalt des Carry-Flags geschoben während der Inhalt des 7. Bits anschließend ins Carry kommt. Wir haben also ein zyklischen Verschieben über 9 Bit vor uns (8 Bit des Registers plus Carry-Flag). Ein Beispiel soll das wieder verdeutlichen.

ROL A Der Akku soll $4B enthalten,
das Carry-Flag ist gesetzt.

```
   $4B  %01001011    C=1
C = 0   %10010111    $97
```

Alle Bits wurden um eins nach links verschoben. Das Carry-Flag kommt in die Position des freigewordenen 0. Bits. Das herausgeschobene 7. Bit kommt ins Carry. Wir erhalten als Ergebnis $97 und ein gelöschtes Carry. Auch hierbei vordoppeln wir den Inhalt des Akkus; es kann jedoch ein evtl. Übertrag aus einem vorherigen Verschieben in Form des Carry dazuaddiert werden.

ROR

Der ROR-Befehl ('rotate right') ist das Gegenstück zum ROL-Befehl und verschiebt den Inhalt eines Registers zyklisch um eine Stelle nach rechts. Dabei wird der Inhalt des Carry-Flags in die frei werdende Position des 7. Bits

nachgeschoben während das herausgeschobene 0. Bit ins Carry kommt.

ROR A Der Akku soll $89 enthalten,
das Carry-Flag ist gelöscht.

```
$89  %10001001   C=0
$44   01000100   C=1
```

Aus $89 erhalten wir $44, das Carry ist gesetzt und zeigt einen Rest bei der Division durch zwei an. Bei sämtlichen Verschiebebefehle werden außer dem Carry-Flag noch die Flags Z und N gesetzt, jenachdem ob das Ergebnis 0 oder größer als $7F war.

Die Tabelle enthält die Befehlskodes zu allen Adressierungsarten.

Adressierungsart	ASL	LSR	ROL	ROR
akku	$0A	$4A	$2A	$6A
absolut	$0E	$4E	$2E	$6E
zerpage	$06	$46	$26	$66
absolut x-indiziert	$1E	$5E	$3E	$7E
zeropage x-indiziert	$16	$56	$36	$76

12. Die Unterprogrammbefehle

Eine sehr wichtige Programmiertechnik, die Sie von BASIC her sicher schon kennen, ist die Unterprogrammtechnik. In BASIC gibt dazu die Befehle GOSUB zum Aufruf eines Unterprogramms und RETURN zur Rückkehr aus einem Unterprogramm. Wodurch unterscheidet sich nun ein normaler Sprungbefehl wie GOTO bzw. JMP von einem Unterprogrammaufruf? Wird ein Unterprogramm aufgerufen, so muß der Prozessor oder der BASIC-Interpreter sich merken, von welcher Stelle das Unterprogramm aufgerufen wurde, damit er beim RETURN-Befehl wieder an die Stelle zurückverzweigen kann, die dem

Unterprogrammaufruf folgte. Der BASIC-Interpreter erledigt dies automatisch für uns; auch der 6510-Prozessor nimmt uns diese Arbeit ab. Trotzdem sollten wir wissen, wie dies funktioniert.

Damit der Prozessor weiß, an welche Stelle er beim RETURN-Befehl springen muß, wird beim Aufruf die augenblickliche Adresse abgespeichert. Dazu ist ein besonderer Speicherbereich reserviert, der sogenannte Stapelspeicher oder Stack. Dieser Stack liegt fest von Adresse S0100 bis $01FF (256 bis 511). Damit der Prozessor weiß, an welche Adresse des Stacks er eine Rücksprungadresse speichern kann, gibt es den Stapelzeiger oder Stackpointer. Dieses Prozessorregister haben wir bereits kurz kennengelernt. Sehen wir uns nun an, was beim Aufruf eines Unterprogramms passiert.

Der Prozessor nimmt die augenblickliche Adresse (+2) und zerlegt sie in Lo- und Hi-Byte. Das Hi-Byte wird an der Adresse $100 plus SP abgespeichert. Dann wird der Inhalt des Stackpointers SP um eins vermindert und das Lo-Byte im Stack abgespeichert (Adresse $100 + SP). Anschließend wird der Stackpointer noch einmal erniedrigt. Jetzt wird in das Unterprogramm verzweigt.

Stößt der Prozessor nun auf einen RETURN-Befehl, so läuft der umgekehrte Prozess ab: Der Inhalt des Stackpointers wird um eins erhöht um ein Byte vom Stack geholt (Adresse $100 plus SP). Dieses Byte wird als Lo-Byte des Programmzählers verwendet. Dann wird der Stackpointer nochmal um eins erhöht und das Hi-Byte des Programmzählers vom Stack geholt. Jetzt steht der Programmzähler wieder auf dem nächsten Befehl hinter dem Unterprogrammaufruf und das Programm wird dort fortgesetzt.

Schauen wir uns die Arbeitsweise des Stacks nochmal an. Werden Werte auf dem Stack abgelegt, so wird zuerst der Wert im Stack gespeichert und anschließend der Stackpointer um eins erniedrigt. Beim Zurückholen eines Bytes vom Stack wird

der Stackpointer anschließend wieder um eins erhöht. Der Stack wächst also von oben nach unten (von $1FF nach $100). Ein Beispiel soll die Vorgänge im Stack deutlich machen.

```
$C480   JSR  $2000      SP = $FA
                      $01FA = $C4   SP=SP-1
                      $01F9 = $82   SP=SP-1
                        SP = $F8
```

Nun wird an die Adresse $2000 verzweigt, an der für unser Beispiel direkt der RETURN-Befehl stehen soll.

```
$2000   RTS             SP = $F8
                        SP=SP+1   PCL = ($01F9) = $82
                        SP=SP+1   PCH = ($01FA) = $C4
                        SP = $FA
```

Der Programmzähler enthält nun $C482. Dieser Wert wird noch um eins erhöht und zeigt dann auf $C483, den nächsten Befehl hinter dem Unterprogrammaufruf an Adresse $C480.

Der Stack arbeitet nach dem Prinzip 'Last in - First out', das heißt der Wert, der zuletzt auf den Stack geschrieben wurde, wird als erstes wieder zurückgeholt. Durch dieses Prinzip ist es auch möglich, Unterprogramme zu schachteln. Wird nämlich von einem Unterprogramm ein weiteres Unterprogrammaufgerufen,sowirdbeimnächstenRETURN-Befehl die letzte Rücksprungadresse geholt und wieder in das nächst höhereUnterprgrammzurückgekehrt.DernächsteRETURN-Befehl führt dann wieder in das übergeordnete Hauptprogramm zurück.

Wenn Sie die Arbeitsweise des Stacks verstanden haben werden, können Sie den Stack auch zum Zwischenspeichern eigener Daten benutzen, so wie es im nächsten Abschnitt beschrieben ist.

Die Tabelle enthält die Befehlskodes für Unterprgrammaufruf und Rückkehr.

Befehl	Befehlskode
JSR	$20
RTS	$60

13. Die Stackbefehle

Der 6510 hat die Möglichkeit, den Inhalt des Akkumulators und des Statusregisters auf dem Stack abzuspeichern und von dort wieder zurückzuholen. Dabei wird der Stackpointer automatisch nach dem Schreiben um eins erniedrigt sowie vor dem Lesen automatisch um eins erhöht.

PHA

Der Befehl PHA ('push accu') speichert den Inhalt des Akkumulators auf den Stack und erniedrigt den Stackpointer anschließend um eins. Der Inhalt des Akkus bleibt dabei erhalten.

PHP

Mit dem PHP-Befehl ('push processor status') wird das komplette Statusregister auf dem Stack abgelegt und der Stackpointer um eins erniedrigt. Der Inhalt des Statusregisters bleibt dabei erhalten.

PLA

Der PLA-Befehl ('pull accu') ist das Gegenstück zu PHA. Der Stackpointer wird erhöht und ein Byte vom Stack wird in den Akku geholt. Entsprechend dem Wert werden N- und Z-Flag gesetzt.

PLP

Damit kann ein Byte vom Stack geholt werden und ins Statusregister übertragen werden. Der Befehl ist das Gegenstück zu PHP.

Die Tabelle enthält die Befehlskodes

Befehl	Befehlskode
PHA	$48
PHP	$08
PLA	$68
PLP	$28

14. Befehle zur Interruptbehandlung

Diese Befehle werden von uns vorerst nicht benutzt und sind nur der Vollständigkeit halber erwähnt. Der 6510 hat die Möglichkeit, ein Programm von außen zu unterbrechen. Dazu muß die sogenannte Interruptleitung (IRQ, interrupt request) des Prozessors aktiviert werden. Bei einem Interrupt passiert etwas ähnliches wie bei einem Unterprogrammaufruf. Der Prozessor unterbricht die Arbeit an seinem laufenden Programm und legt den Inhalt des Programmzählers auf den Stack. Zusätzlich wird noch der Inhalt des Statusregister auf den Stack gelegt, da die Unterbrechung ja an jeder beliebigen Stelle erfolgen kann. Jetzt wird an die Stelle verzweigt, auf die die Adressen $FFFE und $FFFF zeigen. Der Inhalt dieser Adressen wird als neuer Programmzähler verwendet.

Anstelle einer Unterbrechnung von außen gibt es beim 6510 noch die Möglichkeit, durch einen Befehl einen Interrupt per Programm auszulösen. Dazu dient der Befehl BRK ('break'). Dabei werden wie beim Interrupt Programmzähler und Statusregister auf den Stack gerettet.

Um aus einem Unterbrechungsprogramm wieder in das unterbrochene Programm zurückzukehren, gibt es einen Befehl ähnlich dem RTS-Befehl bei einem Unterprogramm. Der Befehl RTI ('return from interrupt') holt zusätzlich zum Programmzähler noch den Inhalt des Statusregisters vom Stack,

so daß das Programm ohne Veränderung der Flags seine Arbeit fortführen kann.

Die Tabelle enhält die Befehlskodes zu diesen Befehlen.

Befehl	Befehlskode
BRK	$00
RTI	$40

Abschließend soll noch ein Befehl nicht unerwähnt bleiben der gar nichts tun und auch so heißt: NOP ('no operation'). Dieser Befehl wird benutzt, um Anweisungen aus einem Programm zu entfernen ohne die restlichen Befehle zu verschieben sowie in Verzögerungsschleifen (auch dieser Befehl benötigt zur Ausführung eine gewisse Zeit).

Befehl	Befehlskode
NOP	$EA

4. Die Eingabe von Maschinenprogrammen

Nachdem wir nun alle Befehle des Prozessors und ihre Funktionen kennengelernt haben, wollen wir uns Gedanken darüber machen, wie man nun Programme in Maschinensprache schreibt und wie man solche Programme eingibt.

Wie wir bei der Beschreibung von Befehlen schon gesehen haben, besteht ein Maschinenprogramm einfach aus einer Folge von Befehlskodes und soweit erforderlich den zugehörigen Operanden. Als erstes einfaches Beispiel wollen wir ein Zeichen in den Bildschirmspeicher des Commodore 64 schreiben. Es entspricht dem einfachen POKE-Befehl in BASIC.

```
POKE 1024,1  :REM   BILDSCHIRMKODE FÜR A
POKE 55296,7 :REM   FARBKODE FÜR GELB
```

Wenn wir die beiden Befehle ausführen, erscheint in der linken oberen Bildschirmecke ein gelbes 'A'. Nun wollen wir sehen, ob wir diese beiden Befehle als Maschinenprogramm formulieren können. Dazu erinnern wir uns, daß wir den POKE-Befehle durch den Befehl 'STA' ersetzen können. Dieser Befehl speichert den Inhalt des Akkus an die angegebene Adresse. Deshalb muß der Akku erst einmal mit dem Wert geladen werden.

```
LDA #1
STA 1024
```

Wir haben also den Akku mit dem Wert 1 geladen und speichern nun den Akkuinhalt an die Adresse 1024. Analog können wir mit dem Wert für den Farbkode vorgehen.

```
LDA #7
STA 55296
```

Geben wir diese Befehle so ein, so erhalten wir nur einem '?SYNTAX ERROR'. Der Commodore 64 versteht ja direkt nur BASIC-Befehle. Wir müssen also anders vorgehen. Erinnern wir

uns daran, daß ein Maschinenprogramm ja nichts anderes ist als eine Folge von Befehlen und Adressen im Speicher. Wir ermitteln also zuerst die Befehlskodes jeden Befehls. Benutzen Sie dazu die Tabellen im Anhang. Für den LDA-Befehl mit der unmittelbaren Adressierung (wir wollen den Akku ja mit der Zahl 1 laden, nicht mit dem Inhalt der Speicherzelle 1) finden wir dort $A9. Als nächstes folgt der Operand, die 1. Beim STA-Befehl benutzen wir die absolute Adressierungsart. Der Befehlskode ist also $8D. Jetzt folgt der Operand. Der Operand ist in unserem Fall eine Speicheradresse, die als 16-Bit-Wert gespeichert wird. Dazu muß sie in zwei 8-Bit-Werte zerlegt werden. Es folgt dann zuerst das niederwertige Byte und dann das höherwertige Byte. Diese Zerlegung läßt sich nun mit Hexadezimalzahlen leichter durchführen. Dazu machen wir aus 1024 die Hexzahl $0400. Aus 55296 wird $D800. Formulieren wir unser Programm nun noch einmal mit Hexzahlen.

```
LDA #$01
STA $0400
LDA #$07
STA $D800
```

Das niederwertige Byte von $0400 ist $00, das höherwertige ist $04. Der Befehl STA $0400 wird nun durch die Folge $8D, $00, $04 ausgedrückt. Aus LDA #$07 STA $D800 wird dann $A9, $07, $8D, $00, $D8. Komplett sieht unser Programm nun so aus:

```
$A9, $01, $8D, $00, $04, $A9, $07, $8D, $00, $D8
```

Diese Bytefolge muß nun im Speicher abgelegt werden. Hier stoßen wir nun auf das nächste Problem: Wo soll unser Programm nun im Speicher abgelegt werden? Wir müssen uns dazu einen Bereich aussuchen, der von Betriebssystem und BASIC-Interpreter nicht benutzt wird. Beim Commodore 64 haben wir einen solchen Bereich von Adresse 49152 bis 53247 bzw. $C000 bis $CFFF. Dieser Bereich ist 4 KByte groß und reicht damit auch für sehr große Maschinenprogramm. Legen wir also unser Programm direkt ab Adresse 49152 ab. Das Ablegen können wir mit einem kleinen Programm in BASIC machen. Dazu wandeln wir die Hexzahlen erst in Dezimalzahlen um.

```
169, 1, 141, 0, 4, 169, 7, 141, 0, 216
```

```
100 FOR I=0 TO 9
110 READ A : POKE 49152+I, A
120 NEXT
130 DATA 169,1,141,0,4,169,7,141,0,216
```

Wenn wir dieses Programm nun mit RUN starten, wird unser Maschinenprogramm im Speicher ab Adresse 49152 abgelegt. Jetzt endlich könnten wir unser Programm ausführen. Dazu dient der SYS-Befehl in BASIC. Geben wir nach SYS die Startadresse 49152 an, so wird unser Programm als Unterprogramm von BASIC ausgeführt. Doch Vorsicht! Was passiert, wenn der Prozessor den Befehl STA $D800 ausfgeführt hat? Nun, es wird der Inhalt der folgenden Speicherstelle geholt und als Befehlskode interpretiert. Handelt es sich dabei jedoch nicht um einen gültigen Befehl, so kann der Prozessor in einen unkontrollierbaren Zustand geraten und 'abstürzen'. Sie müßten dann evtl. den Rechner aus- und wieder einschalten. Dabei geht jedoch das Programm verloren und Sie müßten wieder von vorne anfangen.

Wenn der Prozessor die 4 Befehle ausgeführt hat, soll er ja die Kontrolle an den BASIC-Interpreter zurückgeben. Der SYS-Befehl führt unser Programm ja als Unterprogramm durch. Wir brauchen unser Programm also lediglich mit einem RTS-Befehl abzuschließen. Mit

```
POKE 49152+10, 96
```

können wir den RTS-Kode ans Ende unseres Programms setzen. Jetzt endlich können wir unser Programm mit

```
SYS 49152
```

starten!

Augenblicklich erscheint das gelbe 'A' in der oberen Bildschirmecke und der Rechner meldet sich mit 'READY.' wieder.

War Ihnen nun diese ganze Prozedur zu umständlich? Dann sind Sie nicht der einzige, der dies meint. Man hat deshalb nach Wegen gesucht, diese Umwandlungen automatisch durchzuführen. Wozu hat man seinen Rechner schließlich!

Man braucht also ein Programm, das Maschinenbefehle wie 'LDA #1' annimmt und daraus automatisch die entsprechenden Befehlskodes im Speicher ablegt. Solche Programme gibt es; man nennt sie Assembler. Damit auch Sie gleich von Anfang an mit einem Assembler arbeiten können und nicht über langweiligem Umrechnen und Tabellennachschlagen die Lust verlieren, haben wir für Sie einen kompletten Assembler in BASIC geschrieben. Ehe wir Ihnen nun den Umgang mit diesem Assembler erklären, gehen wir noch kurz auf andere Dienstprogramme ein, die bei der Maschinenprogrammierung von Nutzen sein können.

Da ist in erster Linie das sogenannte Monitor-Programm. Dieser Monitor erlaubt den direkten Zugriff auf den Speicher sowie die Register des Prozessors. Mit dem Monitor kann man Speicher- und Registerinhalte kontrollieren und ändern. Außerdem kann man von einem Monitor aus Maschinenprogramme starten. Meistens erlauben Monitorprogramme auch das Abspeichern und Laden von Maschinenprogrammen auf Kassette und Diskette. Haben Sie einen Monitor zur Verfügung, so können Sie auch damit Ihre Maschinenprogramm im Hexcode eingeben. Bei kleinen Programmen oder Änderungen ist dies durchaus machbar. Oft enthalten Monitorprogramme auch einen sogenannten Disassembler. Solch ein Programm ist das Gegenstück zu einem Assembler. Der Disassembler liest ein Programm aus dem Speicher und gibt es in mnemonischer Form wieder aus; aus $A9, $01 macht er zum Beispiel wieder LDA #$01. Einen solchen Disassembler, in BASIC geschrieben, stellen wir Ihnen in diesem Buch zur Verfügung. Mit einem solchen Programm kann man nicht nur seine eigenen Programme

disassemblieren. Ebenso kann man sich Teile des Betriebssystems und des BASIC-Interpreters ansehen. Aus diesen Programmen kann man oft wertvolle Anregungen für seine eigene Programmierung erhalten.

Die Arbeitsweise und der Umgang mit einem Assembler

Wir wollen nun ein kleines Maschinenprogramm erstellen, das uns die Vorteile eines Assemblers demonstrieren soll. Dazu formulieren wir folgendes kleine Problem in Maschinensprache:

Wir wollen den gesamten Zeichensatz unseres Commodore 64 auf dem Bildschirm darstellen. Dazu überlegen wir uns zuerst, wie wir so etwas in BASIC lösen würden. Der Commodore 64 kann 256 verschiedene Zeichen auf dem Bildschirm darstellen; der Bildschirmkode kann von 0 bis 255 gehen. In BASIC würden wir dies mit einer Schleife lösen.

```
100 X = 0
110 A = X
120 POKE 1024+X, A  :REM BILDSCHIRMKODE
130 A = 1
140 POKE 55296+X, A :REM FARBKODE
150 X = X + 1
160 IF X <> 256 THEN 110
170 END
```

Lassen Sie dieses Programm laufen, so wird der ganze Zeichensatz Ihres Rechners dargestellt. Beachten Sie auch die Zeit, die das Programm dazu braucht: ca. 7 Sekunden. Das Programm ist bewußt so geschrieben, daß uns die Übertragung in Maschinensprache keine großen Schwierigkeiten machen dürfte. Gehen wir also ans Werk!

```
100 X = 0          =>    LDX #0
```

Wir benutzen für die Variable X das X-Register und für die Variable A den Akku.

```
110 A = X          =>    TXA
```

Der Inhalt des X-Registers wird in den Akku kopiert. Das X-Register bleibt dabei unverändert.

```
120 POKE 1024+X, A  =>    STA 1024,X
```

Hier soll der Inhalt des Akkus an die Speicherzelle 1024+X gespeichert werden. Wir benutzen also die indizierte Adressierung.

```
130 A = 1          =>    LDA #1
```

Der Akku wird mit dem Farbkode 1 (weiß) geladen.

```
140 POKE 55296+X,A  =>    STA 55296,X
```

und an die Speicherstelle 55296+X geschrieben.

```
150 X = X + 1      =>    INX
```

Um den Wert des X-Registers um eins zu erhöhen, können wir den Zählbefehl benutzen.

```
160 IF X <> 256 THEN 110    =>  ?
```

Dieser Befehl erfordert etwas Überlegung. Es soll dann nach Zeile 110 zurückgesprungen werden, wenn der Inhalt von X ungleich 256 ist. Nun kann aber das X-Register nur Werte bis maximal 255 enthalten. Was passiert, wenn das X-Register 255 enthält und der Befehl INX in Zeile 150 durchgeführt wird? Aus 255 ($FF) wird $100. Der Übertrag wird einfach ignoriert und wir erhalten $00 - null. Wie können wir diesen Fall nun erkennen? Dazu müssen wir uns an die Flags erinnern. Jedesmal wenn der Inhalt des X-Registers erhöht wird, werden auch das N- und das Z-Flag beeinflußt. Ist der erhaltene Wert größer als $7F (127) wird das N-Flag gesetzt, anderenfalls wird es gelöscht. Steht nach dem Inkrementieren der Wert null im Register, so wird das Z-Flag gesetzt, das den Wert von null

anzeigt. Ist das nicht der Fall, so wird das Zero-Flag gelöscht. Wir können also das Zeroflag zur Grundlage unserer Entscheidung machen. Ist es nicht gesetzt, so ist der Inhalt nicht 256 bzw. 0 und wir müssen nach Zeile 110 zurückspringen.

```
160 IF X <> 256 THEN 110    => BNE  Zeile 110
```

Jetzt taucht das nächste Problem auf: In der Maschinenprogrammierung können wir nicht sagen 'springe nach Zeile 110', sondern wir müssen die Speicheradresse angeben, an der der Befehl aus unserer Zeile 110 steht. Diese Adresse kennen wir jedoch nicht. Dazu müssen wir wissen, an welcher Adresse unser Programm startet und dann die Länge jeden Befehls beim Weiterzählen berücksichtigen. Wir wollen unsere Programme ab Adresse 49152 bzw. $C000 ablegen, da dieser Bereich von BASIC nicht genutzt wird.

```
100 $C000 LDX #0
110 $C002 TXA
120 $C003 STA $0400,X
130 $C006 LDA #1
140 $C008 STA $D800,X
150 $C00B INX
160 $C00C BNE $C002
170 $C00E RTS
```

Wir haben den Programmzähler also jeweils um die Länge des eines Befehls (ein, zwei oder drei Bytes) weitergezählt. Der Sprung in Zeile 160 muß nun nach Adresse $C002 gehen. Nehmen wir nun zum letzten Mal die Mühe auf uns, das Programm von Hand in Maschinenkode zu überführen.

```
100   $C000    A2 00       LDX #0
110   $C002    8A          TXA
120   $C003    9D 00 04    STA $0400,X
130   $C006    A9 01       LDA #1
140   $C008    90 00 D8    STA $D800,X
150   $C00B    E8          INX
```

```
160   $C00C    D0 ??       BNE $C002
170   $C00E    60          RTS
```

Beim Rücksprung in Zeile 160 nach Zeile 110 wollen wir noch den fehlenden Offset berechnen (Schauen Sie falls erforderlich noch einmal im Kapitel über bedingte Verzweigungen nach). Wir bilden die positive Differenz der Adressen und bilden das Zweierkomplement davon.

```
  $C00E
- $C002
  $000C

$0C  =  %00001100
        %11110011
    +           1
        %11110100   =  $F4
```

Der Offset ist also $F4. Wenn wir diesen Wert nun noch einsetzen und die Kodes in den Speicher bringen (z.B. mit einem Ladeprogramm in BASIC oder dem M-Befehl des Einzelschrittsimulators) können wir unser erstes Programm einmal ausprobieren. Gehen Sie dazu mit dem Cursor in die untere Bildschirmhälfte und geben Sie

SYS 49152

ein. Augenblicklich erscheint der komplette Zeichensatz auf dem Bildschirm. Das Programm, dessen Ausführung in BASIC noch mehr als 7 Sekunden brauchte. läuft jetzt in Sekundenbruchteilen ab. Es ist eine eindrucksvolle Demonstration, welche Geschwindigkeiten mit Maschinensprache erreicht werden können. Doch jetzt wollen wir uns dem Assembler zuwenden, der auch das Erstellen von Maschinenprogrammen schnell und komfortabel ermöglicht.

5. Der Assembler

Der Assembler ermöglicht es uns, Maschinenprogramme genauso wie ein BASIC-Programm einzugeben. Man kann nun genauso einfach wie in BASIC Zeilen ändern, löschen oder einfügen. Jede Zeile bekommt also eine Nummer; anschließend folgen die Assemblerbefehle. Eine Zeile besteht aus der Zeilennummer, dann kann eine Marke folgen. Nun kommt der Assemblerbefehl mit dem optionalen Operand. Damit Sie Ihre eigenen Programme auch später noch verstehen, kann durch Semikolon getrennt ein Kommentar folgen, der vom Assembler nicht beachtet wird, jedoch im Listing mit ausgegeben wird und zu Ihrer Information dient. Er entspricht also dem BASIC-Befehl REM. Eine komplette Assemblerzeile kann z.B. so aussehen:

```
100 TEXT LDA $70,X ; STARTWERT HOLEN
```

Dieses sogenannte Quell- oder Source-Programm kann nun auf Diskette abgespeichert werden. Diesem Assemblerprogramm müssen Sie zur Unterscheidung von dem dadurch erzeugten Maschinenprogramm die Endung '.SRC' im Namen geben. Jetzt kann der Assembler geladen werden. Wird er mit RUN gestartet, fragt er nach dem Namen des Programms, das er übersetzen oder assemblieren soll. Er liest nun dieses Programm von Diskette und erzeugt daraus den Maschinenkode, den er direkt im Speicher ablegt. Zusätzlich kann der Assembler auf Wunsch noch ein sogenanntes Assemblerlisting erzeugen, das neben Zeilennummern und Befehlen noch den erzeugten Maschinenkode in Hexdarstellung enthält. Dieses Assemblerlisting kann sowohl auf den Bildschirm als auch auf einen angeschlossenen Drucker gehen. Beim Assemblieren berechnet der Assembler auch automatisch die Adressen von Sprüngen. Sie als Programmierer brauchen Sprungziele daher nicht absolut als Adressen angeben, sondern können sie symbolisch in Form von Marken oder Labels angeben. Unser Beispielprogramm von vorhin sieht dann so aus:

```
100         LDX #0
110 MARKE   TXA
```

```
120         STA $0400,X
130         LDA #1
140         STA $D800,X
150         INX
160         BNE MARKE
170         RTS
```

Wir geben der Adresse, auf die wir uns später beziehen wollen, einfach einen Namen. Wenn der Assembler das Programm durcharbeitet und er auf solch eine Marke stößt, so merkt er sich den Namen und den Wert (des Programmzählers). In unserem Beispiel hatte der Programmzähler in Zeile 110 einen Wert von $C002. Diesen Wert weist der Assembler nun dem Symbol MARKE zu. Stößt er nun in Zeile 160 wieder auf das Symbol MARKE, so weiß er, das MARKE für den Wert $C002 steht. Aus dem augenblicklichen Wert des Programmzählers und dem Wert des Symbols kann er nun den Offset für den Branch-Befehl berechnen. Während der Assembler das Programm durcharbeitet, legt er automatisch den Kode für die Befehle und ihre Operanden im Speicher ab, so daß das Programm nach dem Assemblerlauf zur Ausführung bereit steht. Bei dieser Vorgehensweise kann es jedoch vorkommen, daß man sich auf ein Symbol bezieht, bevor es definiert wurde:

```
100         LDA $40
110         BEQ WEITER
120         LDX #$FF
130 WEITER  STX $0840
140         RTS
```

Hier bezieht man sich in Zeile 110 auf ein Symbol, das an dieser Stelle noch nicht definiert ist. Der Assembler kann hier noch nicht wissen, welchen Wert die Marke WEITER hat. Man behilft sich deshalb mit einem Trick. Der Assembler geht dazu einmal durch das ganze Programm und merkt sich alle Symbole. Nun wird wieder von vorne angefangen und die eigentliche Assemblierung durchgeführt. Kommt der Assembler nun wieder in Zeile 110, so kennt er den Wert von WEITER schon aus dem ersten Durchgang und kann den Offset für den

Sprungbefehl berechnen. Der Assembler benötigt also zwei Durchläufe oder 'Passes'; man spricht deshalb auch von einem 2-Pass-Assembler. Nach diesem Prinzip funktioniert auch unser Assembler. Damit man sehen kann, wie weit die Arbeit des Assemblers fortgeschritten ist, zeigt er jeweils die Nummer der gerade bearbeiteten Zeile an.

Wenn wir nun ein Assemblerprogramm in der angegeben Weise eingeben, so durchsucht der BASIC-Interpreter unsere Befehlszeile nach BASIC-Befehlsworten und ersetzt dieses, falls er welche findet, durch einen 1-Byte-Kode. Dadurch könnte unser Assembler also Worte, in denen BASIC-Befehlsworte enthalten sind, wie z.B. ON, TO oder aber auch = und * nicht mehr erkennen, da sie nicht als normaler Text gespeichert werden. Deshalb laden Sie bitte erst das nachfolgend abgedruckte BASIC-Programm und starten es mit RUN. Wenn Sie jetzt Programmzeilen eingeben, werden BASIC-Befehlsworte nicht mehr in die entsprechenden Abkürzungen umgewandelt, so daß der Assembler sie erkennen kann. Wollen Sie danach wieder normale BASIC-Programme eingeben, so müssen Sie erst mit dem Befehl

```
SYS 53181
```

wieder den ursprünglichen Zustand herstellen.

```
100 FOR I = 53100 TO 53191
110 READ X : POKE I,X : S=S+X : NEXT
120 DATA 169,119,160,207,141,  2,  3,140,  3,  3, 96, 32
130 DATA  96,165,134,122,132,123, 32,115,  0,170,240,243
140 DATA 162,255,134, 58,144,  6, 32,121,165, 76,225,167
150 DATA  32,107,169,160,  0,162,  0,189,  0,  2,232,201
160 DATA  32,240,248,201, 48,144,  4,201, 58,144,240,153
170 DATA   0,  2,201,  0,240,  7,189,  0,  2,200,232,208
180 DATA 242,200,200,200,200,200, 76,162,164,169,131,160
190 DATA 164,141,  2,  3,140,  3,  3, 96
200 IF S <> 11096 THEN PRINT "FEHLER IN DATAS !!" : END
210 SYS 53100 : PRINT "OK"
```

Sie sollten also dieses Programm auf jeder Diskette abspeichern, auf der Sie später Assemblerquellprogramme haben und daran denken, vor Eingabe von Assemblerprogrammen das obige Programm zu laden und auszuführen. Sollten Sie dies einmal vergessen haben, so genügt es auch nach der Ausführung dieses Programms das Quellprogramm zu laden und jede Zeile durch Drücken von RETURN neu zu übernehmen und dieses Programm dann neu abzuspeichern.

Geben Sie jetzt einmal unser Beispielprogramm von der letzten Seite ein und speichern Sie es unter dem Namen 'TEST.SRC' auf Diskette ab. Fügen Sie noch eine Zeile 180 ein, in der .EN steht. Dies bedeutet für den Assembler, daß Ihr Quellprogramm hier zu Ende ist. Vergessen Sie nicht, vorher das obige BASIC-Programm auszuführen. Nun können wir den Assembler laden und starten. Auf dem Bildschirm erscheint nun folgendes. Ihre Eingaben sind unterstrichen dargestellt.

```
          6510 - ASSEMBLER

SOURCE-FILE-NAME ? TEST
LISTING  J/N     ? J
DRUCKER  J/N     ? N
```

Nach kurzer Zeit erscheint 'PASS 1' auf dem Bildschirm und die Diskette läuft an. Dahinter erscheinen jetzt die bearbeiteten Zeilennummern von 100 bis 180. Im zweiten Pass erscheint das Listing auf dem Bildschirm.

```
PASS 2
    C000  A2 00       100        LDX   #0
    C002  8A          110 MARKE  TXA
    C003  9D 00 04    120        STA   $0400,X
    C006  A9 01       130        LDA   #1
    C008  9D 00 D8    140        STA   $D800,X
    C00B  E8          150        INX
    C00C  D0 F4       160        BNE   MARKE
    C00E  60          170        RTS
                      180        .EN
```

```
MARKE  C002
```

Jetzt fragt der Assembler, ob er das erzeugte Maschinenprogramm auf Diskette abspeichern soll. Antworten Sie mit J(a).

```
AUFZEICHNUNG  J/N     ? J
```

Das Programm wird unter dem Namen 'TEST.OBJ' auf Diskette abgespeichert (OBJ = Objektprogramm). Dann wird eine Statistik über den erzeugten Kode sowie eventuelle Fehler ausgegeben.

```
C000 / C00F / 000F
SOURCEFILE IST TEST2.SRC
 0  FEHLER
```

Nun bietet der Assembler noch die Möglichkeit, sämtliche Symbole und ihre Werte auszugeben.

```
SYMBOLTABELLE  J/N ? J
SORTIERT   J/N ? N

MARKE  C002
```

Dabei können Sie noch angeben, ob die Symboltabelle alphabetisch sortiert sein soll. Da wir nur ein Symbol definiert haben, erübrigt sich das für uns.

Das erzeugte Maschinenprogram steht nun sowohl unter dem Namen 'TEST.OBJ' auf Diskette als auch im Speicher und ist direkt zur Ausführung bereit. Überzeugen Sie sich davon, indem Sie

```
SYS 49152
```

eingeben und es erscheint wieder der Zeichensatz auf dem Bildschirm.

Nun geben wir Ihnen noch ein paar nähere Informationen über den Assembler. Jede Zeile des Quellprogramms besteht aus Zeilennummer, einem optionalen Symbol (auch Label genannt), dann immer dem Befehlswort, z.B. LDA, gefolgt von den Operanden (falls erforderlich) sowie durch Semikolon getrennt einem Kommentar. Der Kommentar kann natürlich auch fehlen, jedoch raten wir Ihnen, von dieser Möglichkeit reichlich Gebrauch zu machen und genau zu beschreiben, was Sie mit den Anweisungen bezwecken. Sollten Sie nämlich in die Verlegenheit kommen, eines Ihrer Assemblerprogramme später noch-einmal zu benötigen oder ändern zu müssen, werden Sie dafür sehr dankbar sein. Zu den Symbolen gibt es noch zu sagen, daß sie aus einem bis maximal 5 Buchstaben bestehen dürfen. Natürlich dürfen zwei unterschiedliche Symbole nicht gleich lauten. Zusätzlich zu der sogenannten impliziten Definition von Symbolen, wie wir Sie oben kennengelernt haben, ist es noch möglich und sinnvoll, Symbolen direkt einen Wert zuzuweisen und im Programm später nur noch die symbolische Schreibweise zu benutzen. Dies macht Programme bedeutend übersichtlicher und durchschaubarer. In unserem Beispiel könnten wir den Adressen für Farb- und Bildschirmspeicher ein Symbol zuweisen:

```
PASS 2
    0400                 70 VIDEO  =     $400
    D800                 80 FARBE  =     $D800
    C000                 90        *=    $C000
    C000  A2 00         100        LDX   #0
    C002  8A            110 MARKE  TXA
    C003  90 00 04      120        STA   VIDEO,X
    C006  A9 01         130        LDA   #1
    C008  9D 00 D8      140        STA   FARBE,X
    C00B  E8            150        INX
    C00C  D0 F4         160        BNE   MARKE
    C00E  60            170        RTS
                        180        .EN
```

Lassen wir uns nun die sortierte Symboltabelle ausgeben, so erhalten wir

```
FARBE  D800      MARKE  C002
VIDEO  0400
```

Dabei haben wir in Zeile 90 eine weitere neue Anweisung. Sie besagt, daß der Assembler dem Programmzähler den Wert $C000 zuweisen soll. Diese Anweisung sollten wir jedes mal vor die erste Anweisung setzen. Damit können Sie nämlich den Assembler anweisen, Ihren Kode an eine beliebige Stelle im Speicher zu legen. Eine weitere freie Stelle ist z.B. der Kassettenpuffer von $33C bis $3FB (828 bis 1019). Der Bereich des BASIC-Interpreters läßt sich jedoch nicht ohne weiters benutzen, da der Assembler diesen Bereich während seiner Arbeit benötigt.

Was ist nun der Vorteil dieser symbolischen Schreibweise? Da gibt es zwei Aspekte: Zum ersten kann durch geschickte Wahl der Namen die Bedeutung einzelner Speicherzelle bereits aus dem Namen erkannt werden, wie in unserem Beispiel. Zum zweiten ist solch ein Programm bedeutend änderungsfreundlicher. Haben wir z.B. den Bereich des Videorams in unserem Rechner verlegt, so genügt es, den Wert von VIDEO zu Beginn unseres Programm zu ändern. Sämtliche Bezüge auf diesen Namen werden dadurch mit geändert. Das wird natürlich um so interessanter, je häufiger so ein Name in einem Programm vorkommt. Sie sollten es sich daher von Anfang an angewöhnen, möglichst viel in einem Programm symbolisch zu bezeichnen.

Kommen wir jetzt noch zu einem weiteren Pseudo-Befehl, auch Assemblerdirektive genannt. Dieser Befehl ermöglicht es Ihnen, beliebige Werte innerhalb des Maschinenprogramms abzulegen. Damit können Sie z.B. irgendwelche Daten oder Texte innerhalb ihres Maschinenprogramms ablegen. Der Befehl dazu lautet

```
.BY
```

Danach muß ein Wert im Bereich von 0 bis 255 folgen, der an der augenblicklichen Stelle im Programm abgelegt wird. Dabei

können Sie außer Konstanten natürlich auch Symbole angeben, deren Wert jedoch nicht größer als 255 sein darf.

```
.BY 100
.BY $7F
.BY CR
```

Bei diesem Befehl gibt es noch eine zusätzliche Option. Oft ist es nämlich erforderlich, einen 16-Bit-Wert in zwei 8-Bit-Wert zu zerlegen. Dazu gibt es die Operatoren '>' und '<'. Das Größer-Zeichen kennzeichnet dabei das Hibyte (Bit 7 bis 15), während das Kleiner-Zeichen für das Lobyte steht (Bit 0 bis 7). Dazu folgendes Beispiel:

```
100  CONST   =   $AB3F
110  .BY  <CONST
120  .BY  >CONST
```

Dieser Programmabschnitt legt die Werte $3F und $AB nacheinander im Programm ab. Diese Operatoren können Sie auch bei der unmittelbaren Adressierung mit '#' benutzen, z.B.

```
130  LDA  #<CONST
140  LOY  #>CONST
```

Um die Zeropageadressierung benutzen zu können, müssen Sie dies dem Assembler durch einen Stern '*' vor dem Operanden mitteilen, ansonsten nimmt der Assembler die absolute Adressierung. Bei der indizierten Adressierung, die nur mit Zeropagedadressen arbeitet, ist dies nicht erforderlich. Sehen Sie dazu das nächste Beispiel.

```
00B0                 100 START  =     $B0
C000  AD B0 00       110        LDA   START
C003  A4 B0          120        LDY   *START
C005  8D 27 00       130        STA   $27
C008  84 60          140        STY   *$60
C00A  24 B0          150        BIT   *START
```

Sie sehen also, daß ohne den Stern '*' die absolute 3-Byte Form des Befehls erzeugt wird, wie in Zeile 110 und 130 zu sehen ist. Mit dem Stern vor dem Operand wird die Zeropage-Adressierung ausgewählt, die in 2-Byte Befehlen resultiert, wie Sie in Zeile 120, 140 und 150 sehen können.

Damit haben Sie alle Funktionen des Assemblers kennengelernt und wir können uns nun voll in die Programmierung stürzen. Auf den nächsten Seiten finden Sie das Listing des Assemblers sowie eine kurze Beschreibung der Funktionsweise und der verwendeten Variablen.

Damit Sie das Listing nicht nur stupide Abtippen müssen, sollten Sie beim Eingeben des Programms gleichzeitig die nachfolgende Funktionsbeschreibung lesen, und versuchen zu verstehen, was der Assembler macht und wie er so etwas macht. Sie können dadurch nicht nur etwas über die Arbeitsweise eines Assemblers lernen, sondern auch etwas über die Maschinensprache lernen, z.B. die unterschiedlichen Adressierungsarten.

```
100 REM 6510 ASSEMBLER  LE 12/83
110 PRINTCHR$(147):PRINT:PRINT:PRINT,"6510 - ASSEMBLER"
    :PRINT:DG=8
120 INPUT"SOURCE-FILE-NAME ";SN$
130 IFRIGHT$(SN$,4)=".SRC"THENSN$=LEFT$(SN$,LEN(SN$)-4)
140 DD$="0":REM DRIVENUMMER
150 INPUT"LISTING  J/N     ";A$:IFA$<>"J"THENPM=1:GOTO190
160 PF=4:PG=3
170 INPUT"DRUCKER  J/N     ";A$:IFA$="J"THENPG=4
180 OPENPF,PG
190 GOSUB5000:REM TABELLEN AUFBAUEN
200 A=0:AD=49152:PRINT:PRINT:PA=A
210 PRINT"PASS 1":GOSUB4000:PRINT"PASS 2":FF%=0:FE%=0
220 OP$=DD$+":"+SN$+".SRC"
230 OPEN8,DG,0,OP$
240 GET#8,A$,A$:REM STARTADRESSE
250 IFPM=1THENPRINTCHR$(145),,ZN$
260 F%=0:IFAD>65535THENPRINT:PRINT
    :PRINT"BEREICHSUEBERSCHREITUNG!":GOTO1000
270 A=AD:GOSUB3240:PR$=A$+"  ":GOSUB2000
    :IFLEFT$(X$,3)=".EN"THEN1000
280 XX$=LEFT$(X$,1):IFXX$="*"THENPR$="                    "
    :LN$="      "
290 IFXX$="."ORXX$="*"ORXX$="="THENGOSUB2900:GOTO380
295 IFXX$=""THENPR$=PR$+"           ":GOTO430
300 ONLM%GOTO320
310 SA=OF+AD:PA=AD:LM%=1
320 XX$=LEFT$(X$,3):FORJ=0TONN%:IFXX$=MN$(J)THEN350
330 NEXT
340 FL$(1)="A":A%=1:F%=1:GOSUB1520:GOTO370
350 GOSUB2400:F%=0:IFT%=5ANDT%(J,9)>0THENT%=9:REM RELATIV
360 ONT%+1GOSUB500,600,600,600,600,800,800,800,500,900
    ,600,600,800
370 POKEOF+AD,A
380 AD=AD+A%:IFLEFT$(X$,2)="*="THEN400
390 LX=AD
400 REM         ************  AUSGABE
410 IFF%=0THENIFFL$(0)=" "ANDFL$(1)=" "ANDFL$(2)=" "THEN430
420 BS%=BS%+1
```

```
430 ONPMGOTO250
440 Y$=LEFT$(Y$+"            ",11):FORI=1TO3:PRINT#PF,FL$(I);
    :NEXT
450 PRINT#PF,PR$ZN$LN$"  "LEFT$(X$+"     ",6)Y$" "RM$
460 GOTO250
500 REM EIN-BYTE-BEFEHLE
510 A%=1:A=T%(J,T%):IFA<OTHENFL$(2)="A":GOTO1510
520 GOSUB3240:PR$=PR$+RIGHT$(A$,2)+"         ":RETURN
600 REM ZWEI-BYTE-BEFEHLE
610 A=T%(J,T%):IFA<OTHENFL$(2)="A":GOTO1500
620 GOSUB3240:PR$=PR$+RIGHT$(A$,2)
630 YY$=YA$:IFLEFT$(YY$,1)="#"THENYY$=MID$(YY$,2)
640 IFLEFT$(YY$,1)="*"THENYY$=MID$(YY$,2)
650 A%=2:IFLEFT$(YY$,1)=">"ORLEFT$(YY$,1)="<"THEN
    :YY$=MID$(YY$,2)
660 A$=LEFT$(YY$,1):IFA$="$"ORA$>"/"ANDA$<":"THENA$=YY$
    :GOTO690
670 SL$=YY$:GOSUB4500
680 A$="$"+HE$
690 GOSUB3100
700 IFLEFT$(YA$,2)="#>"THENA=INT(A/HI)
710 IFLEFT$(YA$,2)="#<"THENA=A-INT(A/HI)*HI
720 IFA>LOTHENFL$(2)="O":F%=1:A=0
730 GOSUB3240:POKEOF+AD+1,AL%:PR$=PR$+" "+RIGHT$("00"+A$,2)
    +"     "
740 A=T%(J,T%):RETURN
800 REM DREI-BYTE-BEFEHLE
810 A%=3
820 A=T%(J,T%)
830 GOSUB3240:PR$=PR$+RIGHT$(A$,2)
840 A$=LEFT$(YA$,1):IFA$="$"ORA$>"/"ANDA$<":"THENA$=YA$
    :GOTO870
850 SL$=YA$:GOSUB4500
860 A$="$"+HE$
870 GOSUB3100:GOSUB3240:PR$=PR$+" "+RIGHT$("00"+A$,2)+" "
    +LEFT$(A$,2)+"  "
880 POKEOF+AD+1,AL%:POKEOF+AD+2,AH%
890 A=T%(J,T%):RETURN
900 REM RELATIV
```

```
910 A%=2
920 A=T%(J,T%):GOSUB3240:PR$=PR$+RIGHT$(A$,2)
930 A$=LEFT$(Y$,1):IFA$="$"ORA$>"/"ANDA$<":"THENA$=Y$
    :GOTO960
940 SL$=Y$:GOSUB4500
950 A$="$"+HE$
960 GOSUB3100:IFFL$(2)="U"THENA=AD+2
970 DF=A-(AD+2):IFDF<-128ORDF>127THENFL$(3)="R":F%=1:DF=0
980 A=DFANDLO:GOSUB3240
990 PR$=PR$+" "+RIGHT$(A$,2)+"      ":POKEOF+AD+1,A:A=T%(J,T%)
    :RETURN
1000 PR$="                  ":IFF%=0THEN1020
1010 BS%=BS%+1
1020 IFAE<AD+OFTHENAE=AD+OF
1030 ONPMGOTO1060
1040 FORI=0TO3:PRINT#PF,FL$(I);:NEXT
1050 PRINT#PF,PR$ZN$LN$"  "LEFT$(X$+"    ",6)Y$" "RM$
1060 CLOSE8:INPUT"AUFZEICHNUNG  J/N      ";A$
     :IFA$<>"J"THEN1130
1070 A$=DD$+":"+SN$+".OBJ":POKE186,DG
1080 A%=LEN(A$):POKE183,A%:POKE187,681ANDLO:POKE188,681/HI
1090 FORI=1TOA%:POKE680+I,ASC(MID$(A$,I)):NEXT:REM FILENAME
1100 A=SA:GOSUB3240:POKE251,AL%:POKE252,AH%:REM STARTADRESSE
1110 A=AE:GOSUB3240:POKE781,AL%:POKE782,AH%:REM ENDADRESSE
1120 POKE780,251:SYS65496:REM SAVE
1130 A=PA:GOSUB3240:PA$=A$:A=AD:GOSUB3240:AD$=A$:A=AD-PA
     :GOSUB3240
1140 BA$=A$:ONPMGOTO1180
1150 PRINT#PF:PRINT#PF,PA$" / "AD$" / "BA$
1160 PRINT#PF,"SOURCEFILE IST "SN$+".SRC"
1170 PRINT#PF,BS%" FEHLER":PRINT#PF
1180 INPUT"SYMBOLTABELLE J/N ";Z$:IFZ$<>"J"THEN1400
1190 MX=2:IFPG>3THENPRINT#PF,CHR$(12):MX=5
1200 INPUT"SORTIERT  J/N ";Z$:IFZ$="J"THEN1300
1210 REM
1220 M%=0:P$="":FORI=LL%TOUL%
1230 IFLB$(I)="     "THEN1290
1240 P$=P$+LB$(I)+"  "+HE$(I)+"     ":M%=M%+1
1250 IFM%<>MXTHEN1290
```

```
1260 ONPMGOTO1280
1270 PRINT#PF,P$
1280 P$="":M%=0:IFI>=UL%THEN1400
1290 NEXTI:IFP$<>""THEN1260
1300 HI$=CHR$(127)+CHR$(127)+CHR$(127)+CHR$(127)+CHR$(127)
     :F%=0:REM SORT
1310 M%=0:SL$=HI$:FORI=LL%TOUL%:IFLB$(I)="      "THEN1340
1320 IFLB$(I)<SL$THENSL$=LB$(I):M%=I+1
1330 UL%=I
1340 NEXTI:IFF%<MXTHEN1360
1350 F%=0:IFPM=0THENPRINT#PF
1360 IFM%=0THEN1400
1370 ONPMGOTO1390
1380 PRINT#PF,SL$"  "HE$(M%-1)"      ";
1390 LB$(M%-1)="      ":F%=F%+1:GOTO1310
1400 REM
1410 IFPG=4THENPRINT#PF,CHR$(12)
1420 CLOSEPF:END
1500 POKEOF+AD+2,0:REM NOP-FUELLER
1510 POKEOF+AD+1,0
1520 A=0:PR$=PR$+NP$(A%):RETURN
1600 IFLEFT$(LN$,1)="."THENI=-1:RETURN
1610 IFMID$(LN$,4,1)<>" "THENI=NN%+1:RETURN
1620 MN$=LEFT$(LN$,3):REM LABEL = MNEMONIC ?
1630 FORI=0TONN%:IFMN$<>MN$(I)THENNEXT
1640 RETURN
2000 GET#8,A$,B$:IFA$+B$=""THEN2290:REM LINKADRESSE
2010 GET#8,Z1$,Z2$
2020 ZN=ASC(Z1$+CHR$(0))+HI*ASC(Z2$+CHR$(0))
2030 ZN$=RIGHT$("   "+STR$(ZN),5)+" "
2040 GOSUB2300:IFFF%THENRETURN
2050 LN$="":X$="":Y$="":RM$="":X%=0
2060 FORI=0TO3:FL$(I)=" ":NEXTI:IFZ$="*"THEN2190
2070 IFZ$=";"THEN2280
2080 REM LABELNAME
2090 IFZ$=" "ORFF%THENLN$=LEFT$(LN$+"    ",5):GOTO2120
2100 LN$=LN$+Z$:IFLEN(LN$)=6THENX%=1:FL$(0)="L"
2110 GOSUB2300:GOTO2090
2120 GOSUB1600:IFI<=NN%THENX$=LN$:LN$="      ":GOTO2200
```

```
2130 X%=ASC(LN$):IFX%<65ORX%>90THENFL$(0)="S"
2140 REM OPERATION
2150 GOSUB2300:IFFF%THENRETURN
2160 IFZ$<>" "THEN2190
2170 GOTO2150
2180 GOSUB2300:IFFF%THENRETURN
2190 IFZ$<>" "THENX$=X$+Z$:GOTO2180
2200 IFFF%THENRETURN
2210 IFZ$=";"THEN2280
2220 IFZ$<>" "THEN2260:REM OPERAND
2230 GOSUB2300:IFFF%THENRETURN
2240 GOTO2200
2250 GOSUB2300:IFFF%THENRETURN
2260 IFZ$<>" "THENY$=Y$+Z$:GOTO2250
2270 GOSUB2300:IFFF%THENRETURN:REM BEMERKUNG
2280 RM$=RM$+Z$:GOTO2270
2290 X$=".EN":RM$="END ASSUMED":LN$="      ":Y$=""
     :ZN$="      ":RETURN
2300 GET#8,Z$:FF%=Z$="":RETURN
2400 REM ADRESSIERUNGSART ERMITTLEN
2410 IFY$=""THENT%=8:RETURN:REM IMPLIZIT
2420 YA$=Y$:IFLEFT$(YA$,1)="("THENYA$=MID$(YA$,2)
2430 IFRIGHT$(YA$,1)=")"THENYA$=LEFT$(YA$,LEN(YA$)-1)
2440 IFRIGHT$(YA$,3)="),Y"THENYA$=LEFT$(YA$,LEN(YA$)-3)
2450 IFRIGHT$(YA$,2)=",Y"ORRIGHT$(YA$,2)=",X"THEN
     :YA$=LEFT$(YA$,LEN(YA$)-2)
2460 Z$=Y$:K$=LEFT$(Y$,1)
2470 IFZ$="A"THENT%=0:RETURN:REM AKKU
2480 IFK$="#"THENT%=1:RETURN:REM IMMIDIATE
2490 IFK$="("THEN2600:REM INDIREKT
2500 ZP=K$="*":REM ZEROPAGE
2510 Z$=MID$(Y$,2+ZP)
2520 IFLEN(Z$)<2THEN2550
2530 K$=MID$(Z$,LEN(Z$)-1,1)
2540 IFK$=","THEN2570:REM INDIZIERT
2550 T%=5
2560 T%=T%+3*ZP:RETURN:REM ABSOLUT BZW. ZEROPAGE
2570 K$=RIGHT$(Z$,1):IFK$="X"THENT%=6:GOTO2560
2580 IFK$="Y"THENT%=7:GOTO2560
```

```
2590 T%=-1:RETURN:SYNTAX ERROR
2600 K$=RIGHT$(Z$,1):IFK$=")"THEN2630
2610 IFRIGHT$(Z$,2)<>",Y"THEN2590
2620 T%=11:RETURN
2630 IFMID$(Z$,LEN(Z$)-2,2)=",X"THENT%=10:RETURN
2640 T%=12:RETURN
2700 IFX$="="THEN2730:REM PSEUDO-OPS PASS 1
2710 IFLEFT$(X$,2)="*="THEN2780
2715 IFLEFT$(X$,3)=".BY"THENA%=1:RETURN
2720 A%=0:RETURN
2730 A%=0:IFY$="*"THENRETURN
2740 A%=ASC(LEFT$(LN$,1)):IFA%<65ORA%>90THENRETURN
2750 A$=LEFT$(Y$,1):IFA$<>"$"AND(A$<"0"ORA$>"9")THENRETURN
2760 A$=Y$:GOSUB3100:IFF%THENML$(HC%)=FL$(2):RETURN
2770 GOSUB3240:HE$(HC%)=RIGHT$("0000"+A$,4):RETURN
2780 A%=0:Y1$=LEFT$(Y$,1)
     :IFY1$="$"ORY1$>"/"ANDY1$<":"THEN2800
2790 RETURN
2800 A$=Y$:GOSUB3100:IFF%THENRETURN
2810 HA=A:GOSUB3240:X%=ASC(LEFT$(LN$+CHR$(0),1))
     :IFX%>64ANDX%<91THENHE$(HC%)=A$
2820 RETURN
2900 IFXX$="="THEN2940:REM PSEUDO-OPS PASS 2
2910 IFLEFT$(X$,2)="*="THEN2990
2915 IFLEFT$(X$,3)=".BY"THEN2991
2920 FL$(1)="S"
2930 A%=0:F%=1:PR$="                    ":RETURN
2940 A%=0
2950 A$=LEFT$(Y$,1)
2960 IFA$<>"*"ANDA$<>"$"AND(A$<"0"ORA$>"9")THENFL$(2)="S"
     :GOTO2930
2970 SL$=LN$:F%=0:GOSUB4500:IFF%THENFL$(0)=FL$(2):FL$(2)=" "
     :GOTO2930
2980 PR$=HE$+"              ":RETURN
2990 A%=0:YZ$=LEFT$(Y$,1)
     :IFYZ$="$"ORYZ$>"/"ANDYZ$<":"THEN3010
2991 YZ$=LEFT$(Y$,1):LH%=YZ$=">"ORYZ$="<":YA$=MID$(Y$,1-LH%)
2992 YZ$=LEFT$(YA$,1):IFYZ$="$"ORYZ$>"/"ANDYZ$<":"THENHE$=YA$
     :GOTO2994
```

```
2993 SL$=YA$:F%=0:GOSUB4500:HE$="$"+HE$:IFF%THENFL$(0)=FL$(2)
     :FL$(2)=" "
2994 A$=HE$:GOSUB3100:IFA>LOANDLH%=0THENA=0:FL$(1)="0"
2995 IFLEFT$(Y$,1)=">"THENA=INT(A/HI)
2996 IFLEFT$(Y$,1)="<"THENA=A-INT(A/HI)*HI
2998 POKEAD,A:A%=1:GOSUB3240
     :PR$=PR$+RIGHT$("00"+A$,2)+"         ":RETURN
3000 FL$(2)="S":F%=1:GOTO3030
3010 A$=Y$:GOSUB3100:IFF%THEN3030
3020 AD=A:GOSUB3240:PR$=A$+"  "
3030 PR$=PR$+"           ":RETURN
3100 REM WANDLUNG HEX -> DEC  A$ -> A
3110 Z$=LEFT$(A$,1):IFZ$="$"THENA$=RIGHT$(A$,LEN(A$)-1)
     :GOTO3150
3120 IFZ$<"0"ORZ$>"9"THENFL$(2)="S":F%=1:RETURN
3130 A=VAL(A$):IFA>65535ORA<0THENFL$(2)="0":F%=1
3140 RETURN
3150 A=0:L%=LEN(A$):IFL%>4THENF%=1:FL$(2)="L":RETURN
3200 FORI=1TOL%:AA%=ASC(MID$(A$,I))-48
3210 IFAA%<0ORAA%>9THENIFAA%<17ORAA%>22THENF%=1:FL$(2)="S"
     :RETURN
3220 IFAA%>9THENAA%=AA%-7
3230 A=A+AA%*16^(L%-I):NEXT:RETURN
3240 AH%=A/HI:AL%=A-AH%*HI:A$=A$(AH%/16)+A$(AH%AND15)+
     A$(AL%/16)+A$(AL%AND15)
3250 RETURN
4000 DIMLB$(349),HE$(349),ML$(349):HA=AD
     :REM ADRESSBUCH AUFBAUEN
4010 FORI=0TO349:LB$(I)="     ":HE$(I)="0000":ML$(I)=" ":NEXT
4020 OP$=DD$+":"+SN$+".SRC"
4030 OPEN8,DG,0,OP$
4040 GET#8,A$,A$:LL%=349
4050 1FST<>0THENCLOSE8:END
4060 GOSUB2000:PRINTCHR$(145),ZN$
     :IFLN$=""ORLEFT$(LN$,1)=" "THEN4210
4070 X%=ASC(LEFT$(LN$,1)):IFX%<65ORX%>90THEN4210
4080 GOSUB4100:GOTO4130
4090 LN$=LEFT$(LN$+"    ",5):REM HASH-CODE BILDEN
4100 HC=0:FORI=1TO5
```

```
4110 HC%=ASC(MID$(LN$,I,1))
     :HC=HC+(HC%/10-INT(HC%/10))*10^(6-I):NEXTI
4120 HC%=(HC/307-INT(HC/307))*300:RETURN
4130 A=HA:GOSUB3240
4140 IFLB$(HC%)<>"      "THEN4180
4150 LB$(HC%)=LN$:HE$(HC%)=A$:IFHC%>UL%THENUL%=HC%
4160 IFHC%<LL%THENLL%=HC%
4170 GOTO4210
4180 IFLB$(HC%)=LN$THENML$(HC%)="M":GOTO4210
4190 HC%=HC%+1:IFHC%<350THEN4140
4200 PRINT"SYMBOLTABELLE VOLL":CLOSE8:END
4210 IFX$=".EN"THENCLOSE8:RETURN
4220 XX$=LEFT$(X$,1):IFXX$="."ORXX$="*"ORXX$="="THEN
     :GOSUB2700:HA=HA+A%:GOTO4060
4230 F%=0:XX$=LEFT$(X$,3):FORJ=0TONN%:IFXX$<>MN$(J)THENNEXT
     :GOTO4270
4240 GOSUB2400
4250 IFT%(J,T%)>=0THEN4280
4260 IFT%=5ANDT%(J,9)>=0THENT%=9:GOTO4280
4270 F%=1:HA=HA+1:GOTO4060
4280 HA=HA+L%(T%):GOTO4060
4500 REM          ********* LABEL SUCHEN
4510 X%=ASC(LEFT$(SL$,1)):IFX%<65ORX%>90THENFL$(2)="S"
     :F%=1:HE$="0000":RETURN
4520 IFLEN(SL$)>5THENFL$(2)="L"
4530 SV$=LN$:LN$=SL$:GOSUB4090:SL$=LN$:LN$=SV$
4540 IFLB$(HC%)="      "ORHC%>UL%THENFL$(2)="U":F%=1
     :HE$="0000":RETURN
4550 IFLB$(HC%)<>SL$THEN4580
4560 HE$=HE$(HC%):IFML$(HC%)<>" "THENFL$(2)=ML$(HC%)
4570 RETURN
4580 HC%=HC%+1:GOTO4540
4590 Y1$="":Y2$="":I=1:REM ZERLEGEN VON Y$ IN Y1$ UND Y2$
4600 IFMID$(Y$,I,1)<>","THENY1$=Y1$+MID$(Y$,I,1)
4610 IFI>LEN(Y$)THENF%=1:RETURN
4620 IFMID$(Y$,I,1)<>","THENI=I+1:GOTO4600
4630 I=1+1:IFI>LEN(Y$)THENF%=1:RETURN
4640 Y2$=Y2$+MID$(Y$,1,1):IFI=LEN(Y$)THENF%=0:RETURN
4650 I=I+1:GOTO4640
```

```
5000 READNN%:HI=256:LO=255
5010 DIMA$(15),MN$(NN%),T%(NN%,12),L%(12),FL$(3),NP$(3)
5020 FORI=0TO15:READA$(I):NEXT
5030 NP$(1)="00          ":NP$(2)="00 00      "
     :NP$(3)="00 00 00  "
5040 FORI=0TO12:READL%(I):NEXT
5050 FORJ=0TONN%:READMN$(J):FORJJ=0TO12:READA$
     :IFA$="-1"THENA=-1:GOTO5070
5060 A=0:FORI=1TO2:X=ASC(RIGHT$(A$,I))-48:X=X+(X>9)*7
     :A=A+X*16^(I-1):NEXT
5070 T%(J,JJ)=A:NEXT:NEXT:RETURN
6000 DATA 55 :REM ANZAHL MNEMONICS
6010 DATA 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
6020 DATA 1,2,2,2,2,3,3,3,1,2,2,2,3
7000 DATA ADC,-1,69,65,75,-1,6D,7D,79,-1,-1,61,71,-1
7010 DATA AND,-1,29,25,35,-1,20,30,39,-1,-1,21,31,-1
7020 DATA ASL,0A,-1,06,16,-1,0E,1E,-1,-1,-1,-1,-1,-1
7030 DATA BCC,-1,-1,-1,-1,-1,-1,-1,-1,-1,90,-1,-1,-1
7040 DATA BCS,-1,-1,-1,-1,-1,-1,-1,-1,-1,B0,-1,-1,-1
7050 DATA BEQ,-1,-1,-1,-1,-1,-1,-1,-1,-1,F0,-1,-1,-1
7060 DATA BMI,-1,-1,-1,-1,-1,-1,-1,-1,-1,30,-1,-1,-1
7070 DATA BIT,-1,-1,24,-1,-1,2C,-1,-1,-1,-1,-1,-1,-1
7080 DATA BNE,-1,-1,-1,-1,-1,-1,-1,-1,-1,D0,-1,-1,-1
7090 DATA BPL,-1,-1,-1,-1,-1,-1,-1,-1,-1,10,-1,-1,-1
7100 DATA BRK,-1,-1,-1,-1,-1,-1,-1,-1,00,-1,-1,-1,-1
7110 DATA BVC,-1,-1,-1,-1,-1,-1,-1,-1,-1,50,-1,-1,-1
7120 DATA BVS,-1,-1,-1,-1,-1,-1,-1,-1,-1,70,-1,-1,-1
7130 DATA CLC,-1,-1,-1,-1,-1,-1,-1,-1,18,-1,-1,-1,-1
7140 DATA CLD,-1,-1,-1,-1,-1,-1,-1,-1,D8,-1,-1,-1,-1
7150 DATA CLI,-1,-1,-1,-1,-1,-1,-1,-1,58,-1,-1,-1,-1
7160 DATA CLV,-1,-1,-1,-1,-1,-1,-1,-1,B8,-1,-1,-1,-1
7170 DATA CMP,-1,C9,C5,D5,-1,CD,DD,D9,-1,-1,C1,D1,-1
7180 DATA CPX,-1,E0,E4,-1,-1,EC,-1,-1,-1,-1,-1,-1,-1
7190 DATA CPY,-1,C0,C4,-1,-1,CC,-1,-1,-1,-1,-1,-1,-1
7200 DATA DEC,-1,-1,C6,D6,-1,CE,DE,-1,-1,-1,-1,-1,-1
7210 DATA DEX,-1,-1,-1,-1,-1,-1,-1,-1,CA,-1,-1,-1,-1
7220 DATA DEY,-1,-1,-1,-1,-1,-1,-1,-1,88,-1,-1,-1,-1
7230 DATA EOR,-1,49,45,55,-1,4D,5D,59,-1,-1,41,51,-1
7240 DATA INC,-1,-1,E6,F6,-1,EE,FE,-1,-1,-1,-1,-1,-1
```

```
7250 DATA INX,-1,-1,-1,-1,-1,-1,-1,-1,E8,-1,-1,-1,-1
7260 DATA INY,-1,-1,-1,-1,-1,-1,-1,-1,C8,-1,-1,-1,-1
7270 DATA JMP,-1,-1,-1,-1,-1,4C,-1,-1,-1,-1,-1,-1,6C
7280 DATA JSR,-1,-1,-1,-1,-1,20,-1,-1,-1,-1,-1,-1,-1
7290 DATA LDA,-1,A9,A5,B5,-1,AD,BD,B9,-1,-1,A1,B1,-1
7300 DATA LDX,-1,A2,A6,-1,B6,AE,-1,BE,-1,-1,-1,-1,-1
7310 DATA LDY,-1,A0,A4,B4,-1,AC,BC,-1,-1,-1,-1,-1,-1
7320 DATA LSR,4A,-1,46,56,-1,4E,5E,-1,-1,-1,-1,-1,-1
7330 DATA NOP,-1,-1,-1,-1,-1,-1,-1,-1,EA,-1,-1,-1,-1
7340 DATA ORA,-1,09,05,15,-1,0D,1D,19,-1,-1,01,11,-1
7350 DATA PHA,-1,-1,-1,-1,-1,-1,-1,-1,48,-1,-1,-1,-1
7360 DATA PHP,-1,-1,-1,-1,-1,-1,-1,-1,08,-1,-1,-1,-1
7370 DATA PLA,-1,-1,-1,-1,-1,-1,-1,-1,68,-1,-1,-1,-1
7380 DATA PLP,-1,-1,-1,-1,-1,-1,-1,-1,28,-1,-1,-1,-1
7390 DATA ROL,2A,-1,26,36,-1,2E,3E,-1,-1,-1,-1,-1,-1
7400 DATA ROR,6A,-1,66,76,-1,6E,7E,-1,-1,-1,-1,-1,-1
7410 DATA RTI,-1,-1,-1,-1,-1,-1,-1,-1,40,-1,-1,-1,-1
7420 DATA RTS,-1,-1,-1,-1,-1,-1,-1,-1,60,-1,-1,-1,-1
7430 DATA SBC,-1,E9,E5,F5,-1,ED,FD,F9,-1,-1,E1,F1,-1
7440 DATA SEC,-1,-1,-1,-1,-1,-1,-1,-1,38,-1,-1,-1,-1
7450 DATA SED,-1,-1,-1,-1,-1,-1,-1,-1,F8,-1,-1,-1,-1
7460 DATA SEI,-1,-1,-1,-1,-1,-1,-1,-1,78,-1,-1,-1,-1
7470 DATA STA,-1,-1,85,95,-1,8D,9D,99,-1,-1,81,91,-1
7480 DATA STX,-1,-1,86,-1,96,8E,-1,-1,-1,-1,-1,-1,-1
7490 DATA STY,-1,-1,84,94,-1,8C,-1,-1,-1,-1,-1,-1,-1
7500 DATA TAX,-1,-1,-1,-1,-1,-1,-1,-1,AA,-1,-1,-1,-1
7510 DATA TAY,-1,-1,-1,-1,-1,-1,-1,-1,A8,-1,-1,-1,-1
7520 DATA TSX,-1,-1,-1,-1,-1,-1,-1,-1,BA,-1,-1,-1,-1
7530 DATA TXA,-1,-1,-1,-1,-1,-1,-1,-1,8A,-1,-1,-1,-1
7540 DATA TXS,-1,-1,-1,-1,-1,-1,-1,-1,9A,-1,-1,-1,-1
7550 DATA TYA,-1,-1,-1,-1,-1,-1,-1,-1,98,-1,-1,-1,-1
```

Beschreibung des 6510-Assembler und der wichtigsten Variablen.

100 - 190 Ausgabe des Titel, Eingabe des Quellprogramm namens nach SN$, nachfragen ob und falls ja wohin Assemblerlisting gehen soll. Abhängig von der Antwort wird die Gerätenummer DG für die Ausgabe gesetzt. Das Flag PM bestimmt, ob ein Listing gewünscht wird oder nicht. Anschließend wird die Routine zur Initialisierung der Felder mit GOSUB 5000 aufgerufen.

200 - 460 Hauptschleife des Programms. In Zeile 210 wird Pass 1 als Unterprogramm durchgeführt (GOSUB 4000). Dann wird das Quellprogramm als Datei zum Lesen auf der Diskette geöffnet. Falls kein Listing gewünscht wird, werden in Zeile 250 lediglich die Zeilenummern ZN$ ausgegeben. Anschließend wird auf Ende X$=".EN" sowie auf Pseudo-Befehle abgefragt, die gesondert behandelt werden. Die Variable zur Druckausgabe PR$ wird aufgebaut. In Zeile 320 wird geprüft, ob es sich um einen gültigen Befehl handelt. Wenn nicht wird ein Fehlerflag gesetzt und eine Null (BRK-Befehl) in den erzeugten Kode gesetzt. In Zeile 350 wird die Adressierungsart ermittelt (GOSUB 2400). Bei der relativen Adressierung wird eine entsprechende Korrektur vorgenommen. Abhängig von Adressierungsart und Länge des Befehls wird in Zeile 360 ein Unterprogramm aufgerufen, das die Erzeugung des Kodes sowie die Übernahme in den Druckstring übernimmt. In Zeile 370 wird der Befehlskode in den Speicher geschrieben. Zeile 380 zählt den Programmzähler weiter. Die Zeilen 400 bis 460 erledigen die Druckausgabe. Handelt es sich um eine fehlerhafte Anweisung, so wird der Zähler dafür in Zeile 420 erhöht. Zeile 450 schließlich gibt die komplette Assemblerzeile aus.

500 - 520 Abhandlung von Ein-Byte-Befehlen. Die Variable A% wird auf die Anzahl der Bytes gesetzt und mithilfe des Feld T% wird der Befehlskode ermittelt. Ein Wert kleiner null bedeutet dabei, daß dieser Befehl nicht mit dieser Adressierungsart existiert. In diesem Falle wird nach 1510 gesprungen, wo ein Nullbyte stattdessen geschrieben wird. Ansonsten wird der Opkode nach hex gewandelt und in den Druckstring eingebaut.

600 - 740 Abhandlung von Zwei-Byte-Befehlen. Zeile 610 ermittelt wieder den Opkode, in Zeile 620 wird er in den Druckstring eingebaut. Danach wird auf unmittelbare Adressierung '#' sowie die Operatoren '>' und '<' geprüft. Ebenso wird hier erkannt, ob Zeropageadressierung vorliegt (gekennzeichnet durch '*'). Zeile 660 stelt fest, ob es sich um einen numerischen Wert handelt (hex oder dezimal). Ist dies nicht der Fall, wird in Zeile 670 der Wert des Labels geholt (GOSUB 4500). Nun wird der Wert des Operanden nach hex gewandelt. Zeile 700 und 710 modifizieren den Wert entsprechend den Operatoren '>' und '<'. Zeile 720 prüft schließlich, ob der Wert nicht größer als 255 ist. Nun kann der Wert in den Speicher abgelegt sowie in den Druckstring eingebaut werden.

800 - 890 Hier machen wir das gleiche bei Drei-Byte-Befehlen. Die Vorgehensweise ist analog den Zwei-Byte-Befehlen. Der Wert des Operanden wird nun in der Reihenfolge lo, hi in der Speicher gepoket und in den Druckstring eingebaut.

900 - 990 Hier wird die relative Adressierung behandelt. Auch hier wird zunächste der Wert des Operanden geholt. Zeile 970 berechnet nun den Offset und prüft, ob er noch innerhalb des gültigen Adressbereichs liegt. Ist dies nicht der Fall

wird ein Fehler 'R' (Range) angezeigt und der Offset auf null gesetzt. Zeile 980 führt eine AND-Verknüpfung mit 255 durch, um die negativen Werte ins Zweierkomplement zu überführen. Nun kann der Wert in den Druckstring eingebaut werden sowie im Speicher abgelegt werden.

1000 - 1420 Hierhin wird gesprungen, wenn der Assemblerdurchlauf beendet ist. Die letzte Zeile wird ausgegeben. Nun wird gefragt, ob der erzeugte Kode auf Diskette abgespeichert werden soll. Wird dies bejaht, so werden Filenamen sowie Start- und Endadresse an den entsprechenden Stellen im Speicher abgelegt und die SAVE-Routine des Betriebssystem mit SYS aufgerufen. Nun wird der Bereich sowie die Länge des erzeugten Kodes ausgegeben. An dieser Stelle können Sie entscheiden, ob Sie die Labeltabelle ausgegeben haben wollen und falls ja ob dies in alphabetisch sortierter Form geschehen soll. Dies passiert in den Zeilen 1200 - 1400. Damit ist der Assemblerlauf beendet.

Ab jetzt folgen nun Unterroutinen die von den obigen Hauptroutinen aufgerufen werden und z.B. Zahlenumwandlungen durchführen.

1500 - 1520 Diese Routine gibt ein oder zwei Nullbytes aus, falls ein Fehler beim Assemblieren entdeckt wurde.

1600 - 1640 Diese Routine prüft, ob es sich bei einem Label um den Namen eines Befehlswortes handelt. Ist dies der Fall so enthält die Variable I einen Wert zwischen 0 und NN% (Anzahl der Befehlsworte) einschließlich. Damit kann entschieden werden, ob das erste Wort in einer Programmzeile als Label oder Befehlswort zu behandeln ist.

2000 - 2300 Diese Routine wird benutzt, um eine Programmzeile von Diskette zu lesen und die Variablen für Zeilennummer, Label, Befehlswort, Operand und Bemerkung die entsprechenden Werte zuzuweisen. Die Routine 2300 holt dabei ein Byte von Diskette in die Variable Z$. Das Flag FF% wird gesetzt, falls es sich um ein Nullbyte handelt (Erkennung des Zeilenendes). Die ersten beiden Bytes, die die Linkadresse enthalten, werden nicht benutzt. Sind beide jedoch null, so wurde das Programmende erreicht und es wird eine '.EN'-Anweisung erzeugt. Ansonsten wird aus den nächsten beiden Bytes die Zeilennummer geholt. Nun werden jeweils solange Zeichen in einer Variable summiert, bis man auf ein Leerzeichen oder das Zeilenende stößt. Wird ein Semikolon gefunden, wird der nachfolgende Text der Variablen für den Kommentar zugewiesen. Ansonsten steht anch Aufruf dieser Routine in der Variablen LN$ die Zeilennummer; LN$ enthält den Labelnamen, X$ steht für das Befehlswort, Y$ enthält den Operanden und RM$ enthält den Kommentar.

2400 -2640 Diese Routine ermittelt die Adressierungsart zu einem Befehl. Dazu wird geprüft, ob die Zeichen '(', ')' sowie Komma ',' und 'X' und 'Y' enthalten sind. Ebenso wird die unmittelbare Adressierung durch '#' erkannt. Die Zeropage-adressierung wird durch den vorangestellten Stern '*' erkannt (Variable ZP, Zeile 2510). Das Ergebnis dieser Routine ist die Adressierungart in der Variablen T%, die dann einen Wert zwischen null und zwölf enthält. Dies entspricht dem Aufbau des Feldes T% zur Bestimmung des Befehlskodes. Ein negativer Wert kennzeichnet eine ungültige Adressierungsart.

2700 - 2820 In dieser Routine werden die Pseudo-Befehle '=', '*=' und '.BY' abgehandelt. Diese Routine wird in

Pass 1 aufgerufen und z.B. zur Labeldefinition benutzt.

2900 - 2998 Diese Routine Routine handelt die gleichen Befehle wie die obige ab, jedoch für Pass zwei. Von dieser Routine wird z.B. der Kode des '.BY'-Befehls im Speicher abgelegt sowie für alle Befehle der Druckstring aufbereitet.

3000 - 3030 Diese Routine ruft die folgenden Routinen zur Zahlenumwandlung auf und wird benutzt, um den Druckstring mit der Adresse des Programmzählers zu initialisieren.

3100 - 3230 Hier wird eine Hexzahl in A$ in eine Dezimalzahl in A umgewandelt.

3240 - 3250 In dieser Routine wird aus einer Dezimalzahl in A wieder eine Hexzahl in A$ erzeugt. Nach Aufruf dieses Unterprogramms stehen in AL% und AH% noch Lo- und Hi-Byte zur Verfügung.

4000 - 4280 Dieses Unterprogramm übernimmt den kompletten Pass 1. Die Aufgabe besteht hauptsächlich im Heraussuchen aller Labels und der Zuweisung ihrer Werte. Von dieser Routine werden auch die laufenden Zeilenummern ausgegeben (Zeile 4060). Die Labels werden zum späteren schnelleren Wiederfinden nach einem Hashkode-Verfahren im Feld LB$() abgelegt, die entsprechenden Werte im Hexkode in HE$(). Dazu muß aus der Adressierungsart die Länge des jeweiligen Befehls ermittelt werden, damit den Labels der korrekte Wert zugewiesen werden kann. Ebenso muß auf doppelte Labels geprüft werden. Die Erhöhung des Programmzähler geschieht nach dem Ermitteln der Adressierungsart T% über das Feld L%(), daß zu jeder Adressierungsart die entsprechende Länge enthält.

4500 - 4650 Diese Routine wird in Pass 2 aufgerufen und ermittelt den Wert eines Labels, das in SL$ übergeben wird. Wird das Label nicht gefunden, wird ein Fehlerflag gesetzt, ansonsten wird in HE$ der Hexwert zurückgegeben.

5000 - 5070 Diese Routine übernimmt die Initialisierung aller Felder, die in den folgenden DATA-Statments abgelegt sind.

6000 - 7550 Hier stehen die DATA-Statements, die für die Umwandlung von Dezimal nach Hex erforderlich sind, die Befehlslängen zu allen Adressierungsarten sowie sämtliche Befehlsworte mit zugehörigen Befehlskodes und Adressierungsarten.

Nachfolgend finden Sie noch die Bedeutung der wichtigsten benutzten Variablen.

SN$ Diese Variable enthält den Namen des Quellprogramms (ohne die Endung '.SRC'). Das erzeugte Maschinenprogramm wird unter dem gleichen Namen mit der Endung '.OBJ' abgespeichert.

DD$ Diese Variable enthält die Drivenummer.

PG Gerätenummer des Ausgabegeäts für das Listing; 3 = Bildschirm, 4 = Drucker.

PM Flag für Druckausgabe unterdrücken (=1).

A aktueller Adresswert

AD augenblicklicher Programmzähler während der Assemblierung.

ZN$ laufende Zeilennummer

LN$	Labelname
X$	Befehlswort
Y$	Operand
RM$	Kommentar
T%	Adressierungsart (null bis zwölf)
OF	Offset bei der Abspeicherung des erzeugten Kode (nicht benutzt, 0)
A%	Länge des Befehls
A$	Hexdarstellung der aktuellen Adresse A
SL$	Suchlabel, muß bei Aufruf von 4500 den Namen des zu suchenden Labels enthalten.
HE$	Hexwert des Labels
LO	Konstante 255
HI	Konstante 256
DF	Adressendifferenz bei der relativen Adressierung.
BS%	Zähler für fehlerhafte Statements
MX	Anzahl der Labels pro Zeile zur Ausgabe der Symboltabelle.
HI$	'größter' Labelname für Sortierung
PR$	String zur Ausgabe einer Druckzeile im Listing
MN$	Mnemonic

Z$	Zeichen von Diskette
NN%	Anzahl der Mnemonics (Befehlskodes)
X%	ASCII-Kode
ZP	Flag für Zeropageadressierung
HA	Wert eines Labels (in Pass 1)
F%, FF%	Fehlerflags
HC, HC%	Hashkode
FL$(3)	Fehlerkodes
LB$(349)	Tabelle der Labels
HE$(349)	Tabelle der zugehörigen Werte der Labels im Hexkode
T%(55,12)	Tabelle der Opkodes und Adressierungsarten. Der erste Index bezeichnet das Befehlswort, der zweite Index kennzeichnet die Adressierungsart.
MN$(55)	Tabelle der Befehlsworte in alphabetischer Reihenfolge.

6. Der Einzelschrittsimulator für den 6510

In diesem Kapitel wollen wir Ihnen nun ein Programm vorstellen, das beim Austesten Ihrer Programme sowie bei der Fehlersuche sehr nützlich sein kann. Dieses BASIC-Programm simuliert die Arbeit des 6510-Prozessors. Wenn Sie das Programm mit RUN starten, erscheinen die Registerbezeichnungen des Prozessors sowie anschließend die Inhalte der Register auf dem Bildschirm:

```
 PC   AC XR YR SR SP  NV-BDIZC
0000  00 00 00 20 FF  00100000
```

Die Bedeutungen haben Sie sicher schon erkannt:

```
PC    Programmzähler (program counter)
AC    Akkumulator    (accu)
XR    X-Register
YR    Y-Register
SR    Status-Register
SP    Stapelzeiger   (stack pointer)

N     Negative-Flag
V     Overflow-Flag
B     Break-Flag
D     Dezimal-Flag
I     Interrupt-Flag
Z     Zero-Flag
C     Carry-Flag
```

Unter den jeweiligen Bezeichnungen sind die Inhalte der Register dargestellt. Die obigen Werte werden beim Starten als Anfangswerte benutzt.

Durch Drücken einer Taste können Sie nun die Inhalte einzelner Register ändern. Es erscheint dann unterhalb der Darstellung der alte Inhalt, und der Cursor blinkt. Sie können diesen Wert nun mit einer gültigen Hex-Zahl überschreiben und RETURN drücken. Der neue Wert wird nun in

die obige Anzeige übernommen während die Eingabezeile wieder gelöscht wird. Sie können sämtliche Registerinhalte bzw. Flags durch Drücken des Anfangsbuchstaben ändern:

E Es erscheint der alte Inhalt des Programmzählers. Nach dem Überschreiben durch einen neuen Wert wird er in die Registeranzeige übernommen. Gleichzeitig wird der Befehl, der an dieser Stelle stellt, disassembliert und unterhalb der Registeranzeige dargestellt.

A Der Inhalt des Akkumulators wird angezeigt und kann geändert werden. Nach Drücken von RETURN erscheint der neue Wert in der Registeranzeige.

X Durch Eingabe von X können Sie den Inhalt des X-Register ändern.

Y Mit Y können Sie analog den Wert des Y-Registers ändern.

S Der Inhalt des Stackpointers erscheint in der Eingabezeile. Nach Überschreiben mit einem neuen Wert und Drücken von RETURN erscheint der neue Wert des Stackpointers in der Registeranzeige.

Das Statusregister SR können Sie nicht direkt ändern. Dies geschieht über die einzelnen Flags, aus denen es zusammengesetzt ist. Wird ein Flag geändert, so ändert sich der Wert des Statusregisters in der Anzeige automatisch mit.

N Durch Eingabe von N wird der Wert des N-Flags umgekehrt: Aus 1 wird 0 und umgekehrt. Gleichzeitig wird wie bereits erwähnt auch der Inhalt des Statusregisters, das sich ja aus den einzelnen Flags zusammen setzt, entsprechend mitgeändert.

V Durch Drücken der Taste V wird der Inhalt des V-Flags umgekehrt.

B Mit der Taste B können Sie das Break-Flag beeinflussen.

D Das Dezimal-Flag können Sie mit der Taste D invertieren.

I Mit I können Sie das Interrupt-Flag setzen und löschen.

Z Die Taste Z ist schließlich zum Umkehren des Zero-Flags gedacht.

C Das Carry-Flag können Sie mit C setzen und löschen.

Die wichtigste Funktion unseres Simulators können Sie jedoch mit der Leertaste auslösen. Betätigen Sie diese Taste, so wird der Maschinenbefehl, auf den der Programmzähler gerade zeigt und der in disassemblierter Form unterhalb der Register zu sehen ist, ausgeführt. Wie der Name Simulator schon sagt, wird dieser Befehl nicht vom Prozessor direkt ausgeführt, sondern der Befehl wird per BASIC nachvollzogen. Dabei werden die Registerinhalte und die Flags so beeinflußt, wie dies auch durch den Prozessor geschehen würde. Nach Drücken der Leertaste ändern sich also evtl. Register- und Flag-Inhalte. Unterhalb der Registeranzeige sehen Sie den nächsten Befehl disassembliert, den der Prozessor als nächstes ausführen würde. Dies können Sie auch am Inhalt des Programmzählers sehen.

Sehen wir uns das einmal an einem konkreten Beispiel. Wir benutzen dazu der Einfachheit halber einen Programmabschnitt aus dem Betriebsystem. Dazu setzen wir den Programmzähler auf $A81D. Wir erhalten nun folgende Anzeige:

```
  PC    AC XR YR SR SP  NV-BDIZC
 A81D   00 00 00 20 FF  00100000

A81D 38       SEC
```

Drücken wir nun die Leertaste um die Ausführung dieses Befehls zu simulieren, so erhalten wir folgende Anzeige:

```
  PC    AC XR YR SR SP  NV-BDIZC
 A81E   00 00 00 21 FF  00100001
```

```
A81E A5 2B    LDA $2B
```

Der Befehl SEC wurde also ausgeführt, das Carry-Flag ist gesetzt. Der Wert des Statusregisters hat sich automatisch auf $21 mitgeändert. Der Programmzähler wurde um eins auf $A81E weitergezählt. Dort steht nun ein Ladebefehl. Auch diesen wollen wir nun durch Drücken der Leertaste ausführen.

```
  PC    AC XR YR SR SP  NV-BDIZC
 A820   01 00 00 21 FF  00100001

A820 E9 01    SBC #$01
```

Der Akku wurde also mit dem Inhalt der Speicherstelle $2B geladen, die den Wert 1 enthielt. Beachten Sie, daß Z- und N-Flag gelöscht bleiben, da der geladene Wert weder 0 noch größer als $7F war. Der Programmzähler steht nun auf $A820, also zwei Bytes weiter. Dort steht nun der Befehl SBC #$01. Vom Akkuinhalt soll also der Wert 1 abgezogen werden.

```
  PC    AC XR YR SR SP  NV-BDIZC
 A822   00 00 00 23 FF  00100011

A822 A4 2C    LDY $2C
```

Vom Wert eins im Akku wurde also eins abgezogen, das Ergebnis null steht nun im Akku. Bei den Flags hat sich nun aber etwas getan. Der Zero-Flag ist gesetzt, um anzuzeigen, daß das Ergebnis der Operation null ist. Das Carry-Flag ist weiterhin gesetzt. Damit können wir erkennen, daß bei der Subtraktion kein Unterlauf vorgekommen ist. Der nächste Befehl an Adresse $A822 heißt nun LDY $2C. Nach der Ausführung ergibt sich folgendes Bild:

```
  PC    AC XR YR SR SP  NV-BDIZC
 A824   00 00 08 21 FF  00100001

A824 B0 01    BCS $A827
```

Das Y-Register enthält nun $08 und das Zero-Flag ist gelöscht. Auf Adresse $A824 finden wir nun einen bedingten Sprungbefehl. Können Sie voraussagen, ob dieser Befehl ausgeführt wird? Nun, es soll verzweigt werden, falls das Carry-Flag gesetzt ist. Dies ist bei uns der Fall. Überzeugen Sie sich durch Drücken der Leertaste:

```
  PC    AC XR YR SR SP  NV-BDIZC
 A827   00 00 08 21 FF  00100001

A827 85 41    STA $41
```

Sie sehen, wir haben Recht gehabt. Der Programmzähler steht auf dem Wert $A287. Beachten Sie, daß sich durch den Sprungbefehl an den Flags nichts geändert hat. Der nächste Befehl speichert nun den Inhalt des Akkus in die Speicherzelle $41.

```
  PC    AC XR YR SR SP  NV-BDIZC
 A829   00 00 08 21 FF  00100001

A829 84 42    STY $42
```

Beim STA-Befehl-Befehl haben sich keine Flags verändert. Auch der nächste Befehl STY $42 ändert nichts an den Flags.

```
  PC    AC XR YR SR SP  NV-BDIZC
 A82B   00 00 08 21 FF  00100001

A82B 60       RTS
```

An dieser Stelle wollen wir unsere Simulation abbrechen. Der große Vorteil dieses Simulators ist, daß Sie genau sehen, was bei jedem Befehl passiert. Sie können vor der Ausführung jeden Befehls willkürlich Register- und Flag-Inhalte ändern, um zu sehen, wie der Prozessor darauf reagiert. Sie können natürlich auch nach der Ausführung eines Befehl den Programmzähler wieder auf diesen Befehl zurücksetzen und ihn mit geänderten Registern oder Flags noch einmal ausführen, bis der Befehl so arbeitet, wie Sie es erwartet haben.

Außer den bereits erwähnten Kommandos für den Einzelschritt-simulator stehen Ihnen noch weitere Möglichkeiten zur Verfügung.

Wollen Sie einen Befehl nicht ausführen, sondern nur den Programmzähler auf den nächsten Befehl setzen, so können Sie dies mit der Taste 'Cursor down'. machen. Wenn während Ihrer Simulation Befehle vorkommen, die Speicherinhalte ändern, z.B. STA oder INC, so kann es vorkommen, daß Sie dadurch wichtige Betriebssystemvariablen überschreiben und Ihr Rechner dadurch 'abstürzt'. Deshalb werden Befehle, die in den Speicher schreiben, nicht ausgeführt. Ist dies für Ihr Programm jedoch erforderlich, z.B. wenn eine Speicherstelle als Zähler dient, so können Sie das Programm anweisen, solche schreibenden Befehle auch wirklich auszuführen. Drücken Sie dazu die Taste E. Es erscheint die Zeile 'ECHTSIMULATION ? J' auf dem Bildschirm. Drücken Sie nun RETURN, so werden ab sofort alle schreibenden Befehle in den Speicher auch wirklich ausgeführt. Überschreiben Sie das J(a) jedoch durch N(ein), so können Sie diese Option wieder ausschalten.

Ein weiteres Kommando haben Sie durch Drücken von M zur Verfügung. Mit diesem Befehl können Sie sich den Inhalt einer Speicherstelle ansehen und ihn bei Bedarf ändern. Änderungen werden jedoch nur dann akzeptiert, wenn wir vorher mit E die Echtsimulation gewählt haben.

Das nächste Beispiel soll uns die Wirkungsweise des Stacks verdeutlichen. Dazu wollen wir einen Break-Befehl ausführen. Dazu wählen wir die Echtsimulation, damit wir den Inhalt einer Speicherstelle auf 0 ändern können, den Befehlskode des BRK-Befehls. Starten Sie also den Simulator mit RUN, geben E für Echtsimulation ein und setzen den Programmzähler auf $0002. Sie erhalten folgendes Bild:

```
 PC    AC XR YR SR SP  NV-BDIZC
0002   00 00 00 20 FF  00100000
```

```
0002 00       BRK
```

Um die Vorgänge später besser verfolgen zu können, schreiben wir in Akku, X- und Y-Register unterschiedliche Werte:

```
  PC    AC XR YR SR SP  NV-BDIZC
 0002   22 44 88 20 FF  00100000

0002 00       BRK
```

Sollte an der Adresse $0002 nicht der BRK-Befehl stehen, so drücken Sie M und geben die Adresse 0002 ein. Es erscheint nun der alte Inhalt der Speicherstelle 2, den Sie mit $00 überschreiben. Nun wird der BRK-Befehl angezeigt. Drücken Sie nun die Leertaste und beobachten Sie die Anzeige.

```
  PC    AC XR YR SR SP  NV-BDIZC
 FF48   22 44 88 34 FC  00110100

FF48 48       PHA
```

Das B- und das I-Flag sind gesetzt. Der Stackpointer wurde von $FF auf $FC um drei vermindert, da der Programmzähler (zwei Bytes) und das Statusregister auf den Stack gelegt wurde. Der Programmzähler wurde mit dem Inhalt der Adressen $FFFE und $FFFF geladen und zeigt auf $FF48. Dort steht der Befehl PHA, der den Inhalt des Akkus auf den Stack ablegt.

```
  PC    AC XR YR SR SP  NV-BDIZC
 FF49   22 44 88 34 FB  00110100

FF49 8A       TXA
```

Der Akkuinhalt wurde auf den Stack gelegt und der Stackpointer automatisch um eins erniedrigt. Nun wird der Inhalt des X-Registers in den Akku übertragen.

```
  PC    AC XR YR SR SP  NV-BDIZC
 FF4A   44 44 88 34 FB  00110100

FF4A 48       PHA
```

Die Flags werden nicht verändert, da nun im Akku weder eine null noch ein Wert größer $7F steht. Nun wird der Akkuinhalt wieder auf den Stack gelegt.

```
  PC    AC XR YR SR SP  NV-BDIZC
 FF4B   44 44 88 34 FA  00110100

FF4B 98       TYA
```

Der Stackpointer wurde wieder dekrementiert. Nun wird der Inhalt des Y-Registers in den Akku geholt. Jetzt sollte jedoch das N-Flag gesetzt werden, da der Wert des Y-Registers größer als $7F ist.

```
  PC    AC XR YR SR SP  NV-BDIZC
 FF4C   88 44 88 B4 FA  10110100

FF4C 48       PHA
```

Nun wird der Akkuinhalt wieder auf den Stack gelegt. Diese Prozedur hatte nun den Zweck, alle Registerinhalte auf dem Stack zwischenzuspeichern, damit sie vor der Rückkehr mit dem RTI-Befehl wieder rekonstruiert werden können.

```
  PC    AC XR YR SR SP  NV-BDIZC
 FF4D   88 44 88 B4 F9  10110100

FF4D BA       TSX
```

Nun wollen wir direkt an die Stelle verzweigen, an der die Registerinhalte wieder zurückgeholt werden. Vorher setzen wir jedoch die Registerwerte willkürlich auf null, um deutlich zu machen, wie die Registerinhalte wieder zurückgeholt werden. Setzen Sie dazu den Programmzähler auf $EA81.

```
  PC    AC XR YR SR SP  NV-BDIZC
 EA81   00 00 00 B4 F9  10110100

EA81 68       PLA
```

Hier wird nun ein Wert vom Stack in den Akku geholt.

```
  PC    AC XR YR SR SP  NV-BDIZC
 EA82   88 00 00 B4 FA  10110100

EA82 A8       TAY
```

Dieser Wert wird nun wieder ins Y-Register übertragen.

```
  PC    AC XR YR SR SP  NV-BDIZC
 EA83   88 00 88 B4 FA  10110100

EA83 68       PLA
```

Nun kann der nächste Wert von Stack geholt werden.

```
  PC    AC XR YR SR SP  NV-BDIZC
 EA84   44 00 88 34 FB  00110100

EA84 AA       TAX
```

Jedes Mal, wenn ein Wert von Stack geholt wird, erhöht sich der Stackpointer um eins. Nun wird der Akkuinhalt ins X-Register übertragen.

```
  PC    AC XR YR SR SP  NV-BDIZC
 EA85   44 44 88 34 FB  00110100

EA85 68       PLA
```

Nun kann der Inhalt des Akkus vom Stack geholt werden.

```
  PC    AC XR YR SR SP  NV-BDIZC
 EA86   22 44 88 34 FC  00110100

 EA86 40       RTI
```

Sie sehen, daß nun alle Registerinhalte wiederhergestellt sind und der Stackpointer wieder auf dem Wert steht, den er nach dem BRK-Befehl hatte. Wichtig ist nur, daß die Registerinhalte in der umgekehrten Reihenfolge wieder vom Stack geholt werden. Das 'Last in - First out'-Prinzip kennzeichnet diese Vorgehensweise. Nun wollen wir auch noch den RTI-Befehl ausführen, der uns wieder in unser unterbrochenes Programm zurückführt.

```
  PC    AC XR YR SR SP  NV-BDIZC
 0005   22 44 88 20 FF  00010000

0005 91 B3    STA ($B3),Y
```

Jetzt hat das Statusregister wieder seinen ursprünglichen Wert, und auch der Programmzähler steht nun hinter dem BRK-Befehl.

Der Simulator ist das ideale Werkzeug zum Austesten Ihrer Programme. Hier können Sie Schritt für Schritt sehen, ob der Prozessor wirklich das macht, was Sie in Ihrem Programm beabsichtigt haben. Die Fehlersuche, bei Maschinenprogrammen sonst immer ein heikles Thema, läßt sich mit dem Simulator leicht in den Griff bekommen. Besonders der Anfänger, der die Adressierungsarten noch nicht so genau kennt oder der beim Setzen der Flags noch Probleme hat, wird dies zu schätzen wissen. Auf den nächsten Seiten finden Sie nun das Listing des Programms; dahinter eine kurze Beschreibung der einzelnen Routinen und der Variablen.

Noch ein Hinweis zur Eingabe des Programms: Sollten Sie eine Programmzeile nicht in zwei Bildschirmzeilen unterbringen können, so müssen Sie die Befehlsworte abgekürzt eingeben, anstelle von 'goto' z.B. 'gO' (g shift-o).

```
100 PRINT"       6510 EINZELSCHRITT-SIMULATOR"
110 PRINT"      ----------------------------"
120 PRINT"     ┌─────┬──────────────────┬────────┐"
130 PRINT"     | PC  | AC XR YR SR SP   |NV-BBIZC|"
140 PRINT"     |     |                  |        |"
150 PRINT"     └─────┴──────────────────┴────────┘"
160 FF=255:HI=256:UL=2↑16:SC=2↑15-1:SP=FF
170 DIMMN$(FF),OP(FF),AD(FF),SP(FF),H$(15)
180 FORJ=0TO15:READH$(J):NEXT
190 FORJ=0TOFF:READMN$(J),OP(J),AD(J):NEXT
200 REM REGISTERANZEIGE
210 PRINT"";
215 IFPC>=ULTHENPC=PC-UL
220 A=PC:GOSUB2290:PRINT"";
230 A=AC:GOSUB2320:PRINT"";
240 A=XR:GOSUB2320:PRINT"";
250 A=YR:GOSUB2320:PRINT"";
255 GOSUB900:REM SR
260 A=SR:GOSUB2320:PRINT"";
270 A=SP:GOSUB2320:PRINT"";
280 PRINTCHR$(48+N);
290 PRINTCHR$(48+V);
300 PRINT"1";
310 PRINTCHR$(48+B);
320 PRINTCHR$(48+D);
330 PRINTCHR$(48+I);
340 PRINTCHR$(48+Z);
350 PRINTCHR$(48+C)
360 PRINT"                              ";
400 GETT$:IFT$=""THEN400
405 IFT$=" "THEN1100:REM SIMULATION
410 IFT$="P"THENPRINT"PC  ";:A=PC:GOSUB2290:INPUT"";A$
    :GOSUB2380:PC=A:GOTO1000
420 IFT$="A"THENT$="AC":A=AC:GOSUB540:AC=A:GOTO200
430 IFT$="X"THENT$="XR":A=XR:GOSUB540:XR=A:GOTO200
440 IFT$="Y"THENT$="YR":A=YR:GOSUB540:YR=A:GOTO200
450 IFT$="S"THENT$="SP":A=SP:GOSUB540:SP=A:GOTO200
460 IFT$="N"THENN=1-N:GOTO200
470 IFT$="V"THENV=1-V:GOTO200
```

```
480 IFT$="B"THENB=1-B:GOTO200
490 IFT$="B"THEND=1-D:GOTO200
500 IFT$="I"THENI=1-1:GOTO200
510 IFT$="Z"THENZ=1-Z:GOTO200
520 IFT$="C"THENC=1-C:GOTO200
525 IFT$="[■]"THENS=P:E=P:PC=P:GOTO1010
527 IFT$="M"THEN3000
528 IFT$="E"THEN3100
530 GOTO400
540 PRINTT$"   ";:GOSUB2320:INPUT"[■■■■]";A$:GOTO2380
900 SR=N*128+V*64+32+B*16+D*8+I*4+Z*2+C:RETURN
910 N=SGN(SRAND128):V=SGN(SRAND64):B=SGN(SRANB16)
    :D=SGN(SRAND8)
920 I=SGN(SRAND4):Z=SGN(SRAND2):C=SRAND1:RETURN
980 N=SGN(ACHND128):Z=1-SGN(AC):REM FLAGS
990 PC=PC+1+L
1000 S=PC:E=PC
1010 PRINT"[■■■■■■■■]":GOSUB2040:GOTO200
1100 A=OP(PEEK(PC)):L=0:IFA=0THEN990
1110 ONAGOTO1200,1210,1220,1230,1240,1250,1260,1270,1280
     ,1290,1300,1310,1320,1330
1115 A=A-14
1120 ONAGOTO1340,1350,1360,1370,1380,1390,1400,1410,1420
     ,1430,1440,1450,1460,1470
1125 A=A-14
1130 ONAGOTO1480,1490,1500,1510,1520,1530,1540,1550,1560
     ,1570,1580,1590,1600,1610
1135 A=A-14
1140 ONAGOTO1620,1630,1640,1650,1660,1670,1680,1690,1700
     ,1710,1720,1730,1740,1750
1150 GOTO200
1200 IFDTHEN1205:REM ADC
1201 GOSUB1900:V=1-SGN(ACAND128):AC=AC+OP+C:C=-(AC>FF)
1202 AC=ACANDFF:N=SGN(ACAND128):V=VANDN:GOTO980
1205 GOSUB1900:AC=VAL(H$(AC/16)+H$(ACANB15))
    :OP=VAL(H$(OP/16)+H$(OPAND15))
1206 AC=AC+OP+C:C=-(AC>99):IFAC>99THENAC=AC-100
1207 A$=MID$(STR$(AC),2):GOSUB2390:AC=A:GOTO980
1210 REM AND
```

```
1211 GOSUB1900:AC=ACANDOP:GOTO980
1220 REM ASL
1221 IFAD(PEEK(PC))=4THENAC=AC*2:C=-(AC>FF):AC=ACANBFF
     :GOTO980
1222 GOSUB1900:A=OP*2:C=-(A>FF):A=AANDFF:GOSUB1850
1223 M=SGH(OPANBFF):Z=1-SGN(OP):GOTO990
1230 REM BCC
1231 FL=1-C:GOTO1800
1240 REM BCS
1241 FL=C:GOTO1800
1250 REM BEQ
1251 FL=Z:GOTO1800
1260 REM BIT
1261 GOSUB1900:N=SGN(OPAHB128):V=SGN(0PHND64)
     :Z=1-SGN(OPANBAC):GOTO990
1270 REM BMI
1271 FL=N:GOTO1800
1280 REM BNE
1281 FL=1-Z:GOTO1800
1290 REM BPL
1291 FL=1-N:GOTO1800
1308 REM BRK
1301 PC=PC+2:1FPC>=ULTHENPC=PC-UL
1302 PH=INT(PC/HI):PL=PC-PH*HI:SP(SP)=PH:SP=SP-1ANDFF
     :SP(SP)=PL:SP=SP-1ANDFF
1303 B=1:I=1:GOSUB900:SP(SP)=SR:SP=SP-1ANDFF
     :PC=PEEK(UL-2)+HI*PEEK(UL-1):GOTO1000
1310 REM BVC
1311 FL=1-V:GOTO1800
1320 REM BVS
1321 FL=V:GOTO1800
1330 REM CLC
1331 C=0:GOTO990
1340 REM CLD
1341 D=0:GOTO990
1350 REM CLI
1351 I=0:GOTO990
1360 REM CLV
1361 V=0:GOTO990
```

```
1370 REM CMP
1371 GOSUB1900:A=AC-OP
1372 N=SGN(AANB128):Z=-(A=0):C=-(A>=0):GOTO990
1380 REM CPX
1381 GOSUB1900:A=XR-OP:GOTO1372
1390 REM CPY
1391 GOSUB1900:A=YR-OP:GOTO1372
1400 REM DEC
1401 GOSUB1900:A=OP-1AMDFF:GOSUB1850
1402 GOTO1442
1410 REM BEX
1411 XR=(XR-1)ANDFF:GOTO1452
1420 REM BEY
1421 YR=(YR-1)ANDFF:GOTO1462
1430 REM EOR
1431 GOSUB1900:AC=(ACOROP)ANDNOT(ACANDOP)
1432 GOTO980
1440 REM INC
1441 GOSUB1900:A=OP+1ANDFF:GOSUB1850
1442 H=SGN(AAND128):Z=1-SGH(A):GOTO990
1450 REM INX
1451 XR=(XR+1)ANDFF
1452 Z=1-SGN(XR):H=SGN(XRAND128):GOTO990
1460 REM IHY
1461 YR=(YR+1)ANDFF
1462 Z=1-SGN(YR):N=SGN(YRAND128):GOTO990
1470 REM JMP
1471 GOSUB1900:PC=AD:GOTO1000
1480 REM JSR
1481 A=PC+2:PH=INT(A/HI):PL=A-PH*HI:SP(SP)=PH
    :SP=SP-1ANDFF:SP(SP)=PL:SP=SP-1ANDFF
1482 PC=PEEK(PC+1)+PEEK(PC+2)*HI:GOTO1000
1490 REM LDA
1491 GOSUB1900:AC=OP:GOTO980
1500 REM LDX
1501 GOSUB1900:XR=OP:GOTO1452
1510 REM LDY
1511 GOSUB1900:YR=OP:GOTO1462
1520 REM LSR
```

```
1521 IFAD(PEEK(PC))<>4THEN1524
1522 AC=AC/2
1523 C=-(AC<>INT(AC)):AC=ACANDFF:GOTO980
1524 GOSUB1900:A=OP/2:C=-(A<>INT(A)):A=AANDFF:GOSUB1850
1525 GOTO1442
1530 REM NOP
1531 GOTO990
1540 REM ORA
1541 GOSUB1900:AC=ACOROP:GOTO980
1550 REM PHA
1551 SP(SP)=AC:SP=SP-1ANDFF:GOTO990
1560 REM PHP
1561 GOSUB900:SP(SP)=SR:SP=SP-1ANDFF:GOTO990
1570 REM PLA
1571 SP=(SP+1)ANDFF:AC=SP(SP):GOTO980:REM FLAOS SETZEN
1580 REM PLP
1581 SP=(SP+1)ANDFF:SR=SP(SP):GOSUB910:GOTO990
1590 REM ROL
1591 IFAD(PEEK(PC))=4THENAC=AC*2+C:GOTO1523
1592 GOSUB1900:A=OP*2+C:C=-(A>FF)
1593 A=AANDFF:GOSUB1850
1594 GOTO1442
1600 REM ROR
1601 IFAD(PEEK(PC))=4THENAC=AC/2+128*C:GOTO1523
1602 GOSUB1900:A=OP/2+128*C:C=-(A<>INT(A)):GOTO1593
1610 REM RTI
1611 SP=SP+1ANDFF:SR=SP(SP):GOSUB910:GOTO1621
1620 REM RTS
1621 SP=SP+1ANDFF:A=SP(SP):SP=SP+1ANDFF:PC=A+SP(SP)*HI
     :GOTO990
1630 IFDTHEN1635:REM SBC
1631 GOSUB1900:V=SGN(ACAND128):AC=AC-OP-1+C:C=-(AC>=0)
1632 AC=ACANDFF:N=SGN(ACAND128):V=VAND1-N:GOTO980
1635 GOSUB1900:AC=VAL(H$(AC/16)+H$(ACAND15))
     :OP=VAL(H$(OP/16)+H$(OPAND15))
1636 AC=AC-OP+C-1:C=-(AC>=0):IFAC<0THENAC=AC+100
1637 A$=MID$(STR$(AC),2):GOSUB2390:AC=A:GOTO980
1640 REM SEC
1641 C=1:GOTO990
```

```
1650 REM SED
1651 D=1:GOTO990
1660 REM SEI
1661 I=1:GOTO990
1670 REM STA
1671 GOSUB1900:A=AC:GOSUB1850
1672 GOTO990
1680 REM STX
1681 GOSUB1900:A=XR:GOSUB1850
1682 GOTO990
1690 REM STY
1691 GOSUB1900:A=YR:GOSUB1850
1692 GOTO990
1700 REM TAX
1701 XR=AC:GOTO1452
1710 REM TAY
1711 YR=AC:GOTO1462
1720 REM TSX
1721 XR=SP:GOTO1452
1730 REM TXA
1731 AC=XR:GOTO980
1740 REM TXS
1741 SP=XR:GOTO990
1750 REM TYA
1751 AC=YR:GOTO980
1800 REM BRANCH-BEFEHLE
1810 IFFL=0THENL=1:GOTO990
1820 GOSUB1985:GOTO1000
1850 REM POKE
1870 IFAD<HIORAD>HI+FFTHEN1880
1875 SP(AD-HI)=A:RETURN
1880 IFESTHENPOKEAD,A
1885 RETURN
1900 REM OPERAND HOLEN
1910 A=AD(PEEK(PC))
1920 ONAGOSUB1930,1935,1940,1945,1950,1955,1960,1965,1970
     ,1975,1980,1985,1990
1925 IFAB<HIORAD>HI+FFTHENRETURN
1927 OP=SP(AD-HI):RETURN
```

```
1930 AD=0:RETURN:REM IMPLIED
1935 AD=PC+1:OP=PEEK(AD):L=1:RETURN:REM #
1940 AD=PEEK(PC+1):OP=PEEK(AD):L=1:RETURN:REM ZEROPAGE
1945 AD=0:RETURN:REM A
1950 AD=PEEK(PC+1)+HI*PEEK(PC+2):OP=PEEK(AD):L=2:RETURN
     :REM ABSOLUT
1955 AD=PEEK(PC+1)+XRANDFF:OP=PEEK(AB):L=1:RETURN
     :REM ZEROPAGE,X
1960 AD=PEEK(PC+1)+YRANDFF:OP=PEEK(AD):L=1:RETURN
     :REM ZEROPAGE,Y
1965 AD=PEEK(PC+1)+HI*PEEK(PC+2)+XR:OP=PEEK(AD):L=2:RETURN
     :REM ABSOLUT,X
1970 AD=PEEK(PC+1)+HI*PEEK(PC+2)+YR:OP=PEEK(AD):L=2:RETURN
     :REM ABSOLUT,Y
1975 AD=PEEK(PEEK(PC+1))+HI*PEEK(PEEK(PC+1)+1ANDFF)+YR
     :OP=PEEK(AD):L=1:RETURN:REM INDIREKT Y
1980 AD=PEEK(PC+1)+XRANDFF:AD=PEEK(AD)+HI*PEEK(AD+1)
     :OP=PEEK(AD):L=1:RETURN:REM INDIREKT X
1985 A=PEEK(PC+1):A=A+HI*(A>127)+2+PC
1986 PC=INT(A/HI)*HI+((A+(A>SC)*UL)ANDFF):RETURN:REM RELATIV
1990 AD=PEEK(PC+1)+HI*PEEK(PC+2):AD=PEEK(AD)+HI*PEEK(AD+1)
     :OP=PEEK(AD):RETURN:REM INDIREKT
2040 FORP=STOE:PRINT" ";
2050 A=P:GOSUB2290:REM ADRESSE
2060 PRINT" ";:A=PEEK(P):GOSUB2320:PRINT" ";:J=PEEK(P)
     :OP=AD(J)
2070 ONOPGOSUB2350,2360,2360,2350,2370,2360,2360,2370,2370
     ,2360,2360,2360,2370
2080 PRINT" ";MN$(J)" ";
2090 ONOPG0SUB2110,2120,2130,2140,2150,2160,2170,2180,2190
     ,2200,2210,2220,2240
2100 PRINT"          ":NEXTP
2105 IFP>=ULTHENP=P-UL
2110 RETURN
2120 PRINT"#";:GOSUB2330:P=P+1:RETURN
2130 G0SUB2330:P=P+1:RETURN
2140 PRINT" A";:RETURN
2150 GOSUB2260:P=P+2:RETURN
2160 GOSUB2330:P=P+1:PRINT",X";:RETURN
```

```
2170 GOSUB2330:P=P+1:PRINT",Y";:RETURN
2180 GOSUB2260:P=P+2:PRINT",X";:RETURN
2190 GOSUB2260:P=P+2:PRINT",Y";:RETURN
2200 PRINT"(";:GOSUB2330:P=P+1:PRINT"),Y";:RETURN
2210 PRINT"(";:GOSUB2330:P=P+1:PRINT",X)";:RETURN
2220 A=PEEK(P+1):A=A+HI*(A>127)+2+P
2230 A=INT(A/HI)*HI+((A+(A>SC)*UL)ANDFF):PRINT"$";:GOSUB2290
     :P=P+1:RETURN
2240 PRINT"(";:GOSUB2260
2250 PRINT")";:P=P+2:RETURN
2260 PRINT"$";
2270 A=PEEK(P+1)+HI*PEEK(P+2)
2280 REM HEXADRESSE A
2290 HB=INT(A/HI):A=A-HI*HB
2300 PRINTH$(HB/16)H$(HBAND15);
2310 REM HEXBYTE A
2320 PRINTH$(A/16)H$(AAND15);:RETURN
2330 PRINT"$";
2340 A=PEEK(P+1):GOTO2320
2350 PRINT"      ";:RETURN
2360 GOSUB2340:PRINT"    ";:RETURN
2370 GOSUB2340:PRINT" ";:A=PEEK(P+2):GOTO2320
2380 IFASC(A$)=42THENEND
2390 A=0:FORJ=1TOLEN(A$):X=ASC(RIGHT$(A$,J))-48:X=X+(X>9)*7
     :A=A+X*(16↑(J-1)):NEXT:RETURN
3000 PRINT:PRINT"[RVS]":PRINT"ADRESSE:    $*****[RVS]";:INPUTA$
     :GOSUB2380
3010 PRINT"[CLR]",,:AD=A:OP=PEEK(A):GOSUB1925:A=OP:GOSUB2320
     :INPUT"[RVS]";A$:GOSUB23S0
3020 GOSUB1850:PRINT"[CLR]                                    "
     :IFAD=PCTHEN1000
3030 GOTO200
3100 INPUT"ECHTSIMULATION    J[RVS]";ES$:ES=ES$="J":GOTO200
10000 DATA0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
10010 DATA"BRK",11,1,"ORA",35,11,"???",0,1
10020 DATA"???",0,1,"???",0,1,"ORA",35,3
10030 DATA"ASL",3,3,"???",0,1,"PHP",37,1
10040 DATA"ORA",35,2,"ASL",3,4,"???",0,1
10050 DATA"???",0,1,"ORA",35,5,"ASL",3,5
```

```
10060 DATA"???",0,1,"BPL",10,12,"ORA",35,10
10070 DATA"???",0,1,"???",0,1,"???",0,1
10080 DATA"ORA",35,6,"ASL",3,6,"???",0,1
10090 DATA"CLC",14,1,"ORA",35,9,"???",0,1
10100 DATA"???",0,1,"???",0,1,"ORA",35,8
10110 DATA"ASL",3,8,"???",0,1,"JSR",29,5
10120 DATA"AND",2,11,"???",0,1,"???",0,1
10130 DATA"BIT",7,3,"AND",2,3,"ROL",40,3
10140 DATA"???",0,1,"PLP",39,1,"AND",2,2
10150 DATA"ROL",40,4,"???",0,1,"BIT",7,5
10160 DATA"AND",2,5,"ROL",40,5,"???",0,1
10170 DATA"BMI",8,12,"AND",2,10,"???",0,1
10180 DATA"???",0,1,"???",0,1,"AND",2,6
10190 DATA"ROL",40,6,"???",0,1,"SEC",45,1
10200 DATA"AND",2,9,"???",0,1,"???",0,1
10210 DATA"???",0,1,"AND",2,8,"ROL",40,8
10220 DATA"???",0,1,"RTI",42,1,"EOR",24,11
10230 DATA"???",0,1,"???",0,1,"???",0,1
10240 DATA"EOR",24,3,"LSR",33,3,"???",0,1
10250 DATA"PHA",36,1,"EOR",24,2,"LSR",33,4
10260 DATA"???",0,1,"JMP",28,5,"EOR",24,5
10270 DATA"LSR",33,5,"???",0,1,"BVC",12,12
10280 DATA"EOR",24,10,"???",0,1,"???",0,1
10290 DATA"???",0,1,"EOR",24,6,"LSR",33,6
10300 DATA"???",0,1,"CLI",16,1,"EOR",24,9
10310 DATA"???",0,1,"???",0,1,"???",0,1
10320 DATA"EOR",24,8,"LSR",33,8,"???",0,1
10330 DATA"RTS",43,1,"ADC",1,11,"???",0,1
10340 DATA"???",0,1,"???",0,1,"ADC",1,3
10350 DATA"ROR",41,3,"???",0,1,"PLA",38,1
10360 DATA"ADC",1,2,"ROR",41,4,"???",0,1
10370 DATA"JMP",28,13,"ADC",1,5,"ROR",41,5
10380 DATA"???",0,1,"BVS",13,12,"ADC",1,10
10390 DATA"???",0,1,"???",0,1,"???",0,1
10400 DATA"ADC",1,6,"ROR",41,6,"???",0,1
10410 DATA"SEI",47,1,"ADC",1,9,"???",0,1
10420 DATA"???",0,1,"???",0,1,"ADC",1,8
10430 DATA"ROR",41,8,"???",0,1,"???",0,1
10440 DATA"STA",48,11,"???",0,1,"???",0,1
```

```
10450 DATA"STY",50,3,"STA",48,3,"STX",49,3
10460 DATA"???",0,1,"DEY",23,1,"???",0,1
10470 DATA"TXA",54,1,"???",0,1,"STY",50,5
10480 DATA"STA",48,5,"STX",49,5,"???",0,1
10490 DATA"BCC",4,12,"STA",48,10,"???",0,1
10500 DATA"???",0,1,"STY",50,6,"STA",48,6
10510 DATA"STX",49,7,"???",0,1,"TYA",56,1
10520 DATA"STA",48,9,"TXS",55,1,"???",0,1
10530 DATA"???",0,1,"STA",48,8,"???",0,1
10540 DATA"???",0,1,"LDY",32,2,"LDA",30,11
10550 DATA"LDX",31,2,"???",0,1,"LDY",32,3
10560 DATA"LDA",30,3,"LDX",31,3,"???",0,1
10570 DATA"TAY",52,1,"LDA",30,2,"TAX",51,1
10580 DATA"???",0,1,"LDY",32,5,"LDA",30,5
10590 DATA"LDX",31,5,"???",0,1,"BCS",5,12
10600 DATA"LDA",30,10,"???",0,1,"???",0,1
10610 DATA"LDY",32,6,"LDA",30,6,"LDX",31,7
10620 DATA"???",0,1,"CLV",17,1,"LDA",30,9
10630 DATA"TSX",53,1,"???",0,1,"LDY",32,8
10640 DATA"LDA",30,8,"LDX",31,9,"???",0,1
10650 DATA"CPY",20,2,"CMP",18,11,"???",0,1
10660 DATA"???",0,1,"CPY",20,3,"CMP",18,3
10670 DATA"DEC",21,3,"???",0,1,"INY",27,1
10680 DATA"CMP",18,2,"DEX",22,1,"???",0,1
10690 DATA"CPY",20,5,"CMP",18,5,"DEC",21,5
10700 DATA"???",0,1,"BNE",9,12,"CMP",18,10
10710 DATA"???",0,1,"???",0,1,"???",0,1
10720 DATA"CMP",18,6,"DEC",21,6,"???",0,1
10730 DATA"CLD",15,1,"CMP",18,9,"???",0,1
10740 DATA"???",0,1,"???",0,1,"CMP",18,8
10750 DATA"DEC",21,8,"???",0,1,"CPX",19,2
10760 DATA"SBC",44,11,"???",0,1,"???",0,1
10770 DATA"CPX",19,3,"SBC",44,3,"INC",25,3
10780 DATA"???",0,1,"INX",26,1,"SBC",44,2
10790 DATA"NOP",34,1,"???",0,1,"CPX",19,5
10800 DATA"SBC",44,5,"INC",25,5,"???",0,1
10810 DATA"BEQ",6,12,"SBC",44,10,"???",0,1
10820 DATA"???",0,1,"???",0,1,"SBC",44,6
10830 DATA"INC",25,6,"???",0,1,"SED",46,1
```

```
10840 DATA"SBC",44,9,"???",0,1,"???",0,1
10850 DATA"???",0,1,"SBC",44,8,"INC",25,8
10860 DATA"???",0,1
```

Programmbeschreibung zum Einzelschrittsimulator

100 - 190 Aufbau der Registeranzeige, Initialisierung der Variablen und Felder.

200 - 360 Anzeige der Registerinhalte. Die Inhalte der Prozessorregister werden in Hexadezimaldarstellung gebracht und angezeigt. Bei den Flags geschieht dies einfach über die CHR$-Funktion, die abhängig vom Wert dann '0' oder '1' liefert.

400 - 530 Die Tastatureingabe wird ausgewertet. Ist die Leertaste gedrückt, wird nach Zeile 1100 zur Simulation verzweigt. Bei den Registern wird in eine Eingaberoutine verzweigt, die den alten Wert anzeigt und die Eingabe eines neuen Werts erwartet. Bei den Flags wird der Zustand einfach umgekehrt. Ist die Taste 'Cursor down' gedrückt, wird zur Disassemblerroutine verzweigt und der nächste Befehl angezeigt.

900 - 920 Berechnung des Wertes des Statusregisters SR aus den einzelnen Flags und umgekehrt.

980 Einsprung mit Setzen des N- und Z-Flags.

990 Einsprung mit Erhöhen des Programmzählers.

1000 - 1010 Disassemblieren der nächsten Anweisung.

1100 - 1150 Einzelschrittsimulation, abhängig vom Befehlswort wird zur entsprechenden Einzelroutine gesprungen.

1200 - 1751 Simulationsroutinen für alle 6510-Befehle. Die Routinen folgen den alphabetisch geordneten Befehlsworten und springen anschließend nach Zeile 990 wo der Programmzähler entsprechend der Befehlslänge erhöht wird. Beim Sprung nach Zeile 980 werden zusätzlich N- und Z-Flag entsprechend dem Wert im Akku gesetzt.

1800 - 1820 Hier werden alle Branchbefehle abgehandelt, nachdem der entsprechende Flagwert in die Variable FL gebracht wurde.

1850 - 1885 Diese Routine wird benutzt, um Werte in den Speicher zu schreiben. Dabei wird der Stackbereich von $100 bis $1FF gesondert behandelt. Die POKEs werden nur dann ausgeführt, wenn Echtsimulation gewünscht wurde (Variable ES).

1900 - 1990 Diese Routine holt die Operanden zu den Befehlen abhängig von der Adressierungsart. Nach Aufruf dieser Routine steht die Adresse des Operanden in AD, der Wert selbst in OP.

2040 - 2370 Diese Routine beinhaltet einen Disassembler, der nach jeden Einzelschritt aufgerufen wird und die nächste Zeile disassembliert. In Zeile 2070 werden je nach Adressierungsart die Operandenbytes ausgegeben. Zeile 2090 besorgt dies nocheinmal wobei hier für die der Adressierungart entsprechende Darstellung gesorgt wird. Enthält eine Speicheradresse keinen gültigen Befehlskode, so werden stattdessen drei Fragezeichen ausgegeben. Die nachfolgenden Routinen führen dann die obenangesprochenen Aufgaben sowie die Umwandlung von Dezimal nach Hex aus.

2380 - 2390 Hier werden sämtliche Eingaben von Hex nach dezimal umgerechnet. Geben Sie stattdessen einen Stern ein, so wird das Programm beendet.

3000 - 3030 Diese Routine dient zur Anzeige und Änderung von Speicherinhalten. Die Änderung ist nur möglich, wenn Echtsimulation gewählt wurde.

3100 Hier können Sie über die Echtsimulation entscheiden.

10000-10860 In diesen Zeilen sind die Befehlsworte, die zugehörigen Nummern und die Adressierungsarten abgelegt, die zu Beginn in die entsprechenden Felder eingelesen werden.

Es folgt nun eine Beschreibung der wichtigsten Variablen und ihrer Bedeutung.

FF Konstante 255

HI Konstante 256

UL Konstante 65536

SC Konstante 32767

MN$(255) Tabelle der 6510-Mnemonics

OP(255) Tabelle mit den zugehörigen Befehlsnummern für die Einzelschrittsimulation.

AD(255) Dieses Feld enthält zu jedem Befehl die Adressierungsart.

SP(255) Dieses Feld enthält den Stack

H$(15) Feld mit den Hexziffern

PC Programmzähler

AC Akkumulator

XR	X-Register
YR	Y-Register
SR	Statusregister
SP	Stackpointer
N	Negative-Flag
V	Overflow-Flag
B	Break-Flag
D	Dezimal-Flag
I	Interrupt-Flag
Z	Zero-Flag
C	Carry-Flag
T$	gedrückte Taste
L	Länge des Operanden
ES	Flag für Echtsimulation
OP	Operand

7. Maschinenspracheprogrammierung auf dem Commodore 64

Bei der Programmierung des Commodore 64 gibt es ein Gebiet, das sich für die Maschinensprache direkt anbietet: Die hochauflösende Grafik. Wir wollen in diesem Kapitel ausgehend von Lösungen in BASIC versuchen, die entsprechenden Routinen in Maschinensprache zu formulieren. Dabei werden wir nach und nach alle Programmiertechniken und Möglichkeiten des Prozessors kennenlernen und benutzen.

Gerade die Programmierung der Grafik, in BASIC meist nur eine Sammlung von undurchsichtigen POKEs und PEEKs, soll uns zeigen, wo die Stärken der Maschinenprogrammierung liegen. Gleichzeitig werden wir dabei lernen, wie man Programme in Maschinensprache mit BASIC-Programmen kombiniert und wie man die Stärken beider Sprachen ausnutzt. Wir gehen dabei nur soweit auf die Programmierung des Videocontroller-Baustein des Commodore 64 ein, wie es zur Lösung unser Probleme erforderlich ist. Wollen Sie sich näher mit Hardware und Betriebssystem des Commodore 64 befassen, was sicher früher oder später der Fall sein wird, so empfehlen wir Ihnen das Buch '64 intern'.

Ehe wir uns nun dem ersten Beispiel zuwenden, sehen wir uns erst einmal an, wie man von BASIC aus Maschinenprogramme benutzt und wie man Parameter zwischen beiden Programmen austauscht. Zum Aufruf von Programmen dient in erster Linie der SYS-Befehl. Dabei gibt man die Startadresse des Programms an, das man ausführen will. Trifft das Maschinenprogramm auf einen RTS-Befehl (return from subroutine), so kehrt es wieder zu BASIC zurück. Das BASIC-Programm wird anschließend fortgeführt. Soll ein Maschinenprogramm nur eine feste Aufgabe erfüllen, z.B. das Löschen des Grafikbildschirms, so brauchen wir ja keine weiteren Parameter und der einfache SYS-Befehl erledigt diese Aufgabe. Soll nun eine Routine einen Punkt in der Grafik setzen, so müssen wir dem Programm mitteilen, welchen Punkt wir meinen. Dazu gibt es mehrere Alternativen: Bei der sogenannten Briefkastenmethode hinterlegt man die Parameter einfach in einer oder mehreren

Speicherstellen, die vorher vereinbart wurden. Bei unserem Beispiel könnte also eine Speicherzelle die horizontale Koordinate und eine andere Zelle den Wert der vertikalen Koordinate enthalten. Von BASIC aus kann dies mit zwei POKE-Befehlen geschehen. Das Maschinenprogramm kann nun die Werte der Koordinaten sich aus diesen Speicherzellen holen und verarbeiten. Der BASIC-Interpreter bietet jedoch noch eine spezielle Methode. Wird ein SYS-Befehl ausgeführt, so besteht die Möglichkeit, dem Prozessor bestimmte Registerinhalte zu übergeben. Da wir von BASIC aus jedoch nicht direkt auf die Prozessorregister zugreifen können, wurden 4 Speicherzellen dafür reserviert. Beim Aufruf des SYS-Befehls wird nun deren Inhalt in die 4 Register übertragen, ehe in die Routine verzweigt wird.

780 => Akkumulator
781 => X-Register
782 => Y-Register
783 => Status-Register

Wollen wir also unsere Routine mit einem bestimmten Akku-Wert starten, genügt es vor dem Aufruf des SYS-Befehls den Wert mit einem POKE-Befehl in die Adresse 780 zu schreiben. Die Adressen 781 und 782 sind für X- und Y-Register zuständig. Beim Statusregister ist jedoch etwas Vorsicht geboten. Sie sollten besonders darauf achten, daß Sie nicht unbeabsichtigt das Dezimal- oder das Interrupt-Flag setzen, da dies zu Komplikationen führen kann. Nach dem Einschalten des Rechners steht in diesen Adressen null. Wenn Sie das Statusregister also nicht versorgen, werden also alle Flags zu Beginn der Routine gelöscht, was keine Probleme machen kann. Ist der SYS-Befehl abgearbeitet, so werden die augenblicklichen Registerinhalte wieder in diese Speicherstellen übertragen. Ihnen stehen also die Registerinhalte nach Ausführung der Routine in BASIC zur Verfügung. Auf diese Weise ist es also möglich, drei bzw. vier 8-Bit-Werte zwischen BASIC und Maschinenprogramm sowie umgekehrt zu übergeben. Für die meisten Anwendungen reicht das völlig aus, so daß auch wir darauf zurückgreifen werden. Sind mehr Parameter zu

übertragen, so müssen Sie also wie oben beschrieben, Speicherzellen zur Übergabe vereinbaren.

Es gibt nun noch eine weitere Methode, die recht komfortabel ist. Dabei benutzt man die Routinen des BASIC-Interpreters. Stößt der BASIC-Interpreter z.B. auf einen Befehl wie POKE 780,10, so ruft er nach Erkennen des POKE-Befehls eine universelle Routine auf, die einen Parameter aus dem BASIC-Programm holt. Diese Routine kann nun nicht nur Konstanten wie in unserem Beispiel, sondern beliebig komplizierte Ausdrücke auswerten. Der Befehl könnte z.B. auch POKEA+7.5*Z%(INT(SIN(X)*1000)),EXP(X)lauten.DieseRoutinen können wir uns zunutze machen und haben damit auch die Möglichkeit solche Ausdrücke zu benutzen. Wir werden später auf diese Routinen und ihre Benutzung noch näher eingehen. Für den Anfang nutzen wir jedoch die Möglichkeit, Parameter direkt den Registern zu übergeben.

Bevor wir uns nun auf die Programmierung der Grafik stürzen, noch ein paar grundsätzliche Anmerkungen zu deren Prinzip. Mit der hochauflösenden Grafik können Sie jeden einzelnen Bildpunkt gezielt ansprechen. Während Sie bei der nornmalen Zeichendarstellung immer nur komplette Zeichen setzen können, die aus 8 * 8 = 64 Bildpunkten bestehen, erlaubt uns die Einzelpunktgrafik den Zugriff auf jeden Punkt. Bei der normalen Einzelpunktdarstellung haben Sie also 25 * 40 Zeichen zur Verfügung; bei der hochauflösenden Grafik sind es in jeder Richtung achtmal so viel, also 200 * 320 Punkte. Beiden Darstellungsweisen gemeinsam ist dabei Abbildung des Bildschirms in einem sogenannten Videoram. Bei der normalen Bildschirmdarstellung entspricht jedem Zeichen ein Byte; bei der hochauflösenden Grafik wird für jeden Punkt nur ein Bit gebraucht - 0 für einen gelöschten Punkt und 1 für einen gesetzen Punkt. Für die normale Bildschirmdarstellung brauchen wir deshalb 25 * 40 * 1 Byte = 1000 Byte, während die hochauflösende Grafik 200 * 320 * 1 Bits = 64000 Bits oder 8000 Byte benötigt. Bei der normalen Zeichendarstellung liegt das Videoram von Adresse 1024 bis 2023 ($400 bis $7E8),

wie Sie wissen. Diese Startadresse des Videorams kann jedoch durch Programmierung des Videocontrollers in Schritten von 1 Kbyte ($400) verlegt werden, z.B. auf $800, $C00, $1000 usw. Bei der hochauflösenden Grafik braucht man einen Speicher von 8 KByte. Die Startadresse dieses Speichers kann wieder durch Programmierung des Videocontrollers festgelegt werden. Dabei ist jedoch zu beachten, daß der Videocontroller jeweils nur einen Bereich von 16 KByte adressieren kann. Welcher 16-K-Bereich ausgewählt ist, kann durch Programmierung eines I/O-Bausteins bestimmt werden. Wir wollen uns also jetzt einen geeigneten 8-K-Byte-Bereich für unsere hochauflösende Grafik aussuchen.

Auf den ersten Blick würde es sich anbieten, 8-KByte vom BASIC-Speicher abzuzweigen. Aber wir programmieren ja in Maschinensprache! Der Commodore 64 hat ja den gesamten Adreßbereich mit RAM-Speicher belegt. Wir benutzen einfach den RAM-Bereich, der 'unter' dem Betriebsysten von $E000 bis $FFFF (57344 bis 65535) liegt als Grafikram. Von BASIC aus ist dies ja nicht möglich, da wir ja beim Lesen dieses Speichers BASIC-Interpreter und Betriebssystem 'ausschalten' müssen. Das normale Videoram wird bei der hochauflösenden Grafik als Farbram benutzt. Da das Videoram sowieso im gleichen 16K-Bereich wie die hochauflösende Grafik liegen muß ($C000 - $FFFF), wählen wir der Einfachheit halber den Adreßbereich von $C000 bis $C3FF, der von BASIC ja auch nicht benutzt wird. Da wir für das Farbram der hochauflösenden Grafik nur 1 KByte zur Verfügung haben, wird daraus schon klar, das nicht jeder Bildpunkt eine eigene Farbe haben kann. Die Farbe gilt weiterhin für ein Feld von 8 * 8 Punkten, also in der Größe der früheren Zeichen.

Wir wollen nun verschiedene Hilfsroutinen in Maschinensprache schreiben, die wir später von BASIC aus benutzen können. Die erste Routine soll die Umschaltung von der normalen Zeichendarstellung auf die hochauflösende Grafik vornehmen. Dabei soll unsere Grafik in den oben beschriebenen Speicherbereich gelegt werden. Dies hat auch den zusätzlichen Vorteil, daß durch die Farbe der Grafikpunkte unser normaler

Bildschirminhalt nicht zerstört wird. Er bleibt nach dem Zurückschalten erhalten. Wir formulieren die Umschaltung zuerst in BASIC.

```
100 V = 53248 : REM STARTADRESSE VIDEOCONTROLLER
110 V1 = V+17 : REM UMSCHALTADRESSE GRAFIKMODUS
120 V2 = V+24 : REM FESTLEGUNG VIDEORAMADRESSE
130 CIA = $0000 : REM FESTLEGUNG 16K BEREICH
140 POKE V1,59
150 POKE V2,8
160 POKE CIA,0
170 END
```

Wollen wir dies in Maschinensprache übertragen, müssen wir uns erst überlegen, in welchen Speicherbereich wir unsere Routinen legen wollen. Da der Bereich von $C000 bis $C400 durch das Farbram belegt ist, nehmen wir einfach den Bereich ab $C400. Die Übertragung dieser Befehle dürfte uns keine Schwierigkeiten machen.

```
100 VIDEO = 53248 ; VIDEOCONTROLLER
110 V1 = 53265 ; ADRESSE FÜR GRAFIKMODUS
120 V2 = 53272 ; ADRESSE FÜR VIDEORAMADRESSE
130 CIA = $DD00 ; 16K - SELEKTION
140 *= $C400 ; BEGINN UNSERER ROUTINE
150 LDA #59
160 STA V1
170 LDA #8
180 STA V2
190 LDA #0
200 STA CIA
210 RTS
220 .EN
```

Lassen wir dieses Programm assemblieren, so erhalten wir folgendes Listing:

```
D000                 100 VIDEO  =    53248
D011                 110 V1     =    53265
D018                 120 V2     =    53272
DD00                 130 CIA    =    $DD00
                     140        *=   $C400
C400  A9 3B          150 EIN    LDA  #59
C402  8D 11 D0       160        STA  V1
C405  A9 08          170        LDA  #8
C407  8D 18 D0       180        STA  V2
C40A  A9 00          190        LDA  #0
C40C  8D 00 DD       200        STA  CIA
C40F  60             210        RTS
                     220        .EN
```

Ehe wir unsere Routine nun ausprobieren, wollen wir zuerst die Umschaltung auf die normale Darstellung programmieren. Wir müssen dazu die Register wieder mit den ursprünglichen Werten laden. Der Einfachheit halber hängen wir diese Routine direkt an oben an.

```
100 VIDEO = 53248 ; VIDEOCONTROLLER
110 V1 = 53265 ; ADRESSE FÜR GRAFIKMODUS
120 V2 = 53272 ; ADRESSE FÜR VIDEORAMADRESSE
130 CIA = $DD00 ; 16K - SELEKTION
140 *= $C400 ; BEGINN UNSERER ROUTINE
150 EIN LDA #59
160 STA V1
170 LDA #8
180 STA V2
190 LDA #0
200 STA CIA
210 RTS

220 ; AUSSCHALTEN
230 AUS LDA #27
240 STA V1
250 LDA #21
260 STA V2
270 LDA #3
```

```
280 STA CIA
290 RTS
300 .EN
```

Wenn wir das Programm nocheinmal assemblieren, lassen wir uns die Symboltabelle mit ausgeben. Können Sie sich vorstellen, weshalb wir die Symbole EIN und AUS definiert haben, obwohl wir uns im Programm darauf nicht beziehen? - Richtig, wir brauchen diese Adressen ja später für den Aufruf durch die SYS-Funktion.

```
   D000              100 VIDEO  =    53248
   D011              110 V1     =    53265
   D018              120 V2     =    53272
   DD00              130 CIA    =    $DD00
                     140        *=   $C400
   C400  A9 3B       150 EIN    LDA  #59
   C402  8D 11 D0    160        STA  V1
   C405  A9 08       170        LDA  #8
   C407  80 18 D0    180        STA  V2
   C40A  A9 00       190        LDA  #0
   C40C  8D 00 DD    200        STA  CIA
   C40F  60          210        RTS
   C410              220                  ; AUSSCHALTEN
   C410  A9 1B       230 AUS    LDA  #27
   C412  8D 11 D0    240        STA  V1
   C415  A9 15       250        LDA  #21
   C417  8D 18 D0    260        STA  V2
   C41A  A9 03       270        LDA  #3
   C41C  80 00 DD    280        STA  CIA
   C41F  60          290        RTS
                     300        .EN

C400 / C420 / 0020
SOURCEFILE IST BEISPIEL 1.SRC
 0  FEHLER

AUS    C410     CIA    DD00     EIN    C400
VIDEO  D000     V1     D011     V2     D018
```

Ehe wir unsere Routinen nun ausprobieren, rechnen wir noch die Startadressen EIN und AUS in dezimal um: $C400 ist gleich 50176, $C410 ist gleich 50192. Wir testen unsere Routinen mit einem kleinen BASIC-Programm:

```
100 SYS 50176 : REM GRAFIK EIN
110 GET A$ : IF A$="" THEN 110
120 SYS 50192 : REM GRAFIK AUS
```

Das Programm schaltet also auf Grafik um, wartet dann auf einen Tastendruck um dann wieder auf den normalen Modus zurückzuschalten. Probieren wir es einmal aus!

Starten Sie das Programm, so erscheint eine Gemisch aus verschiedenfarbigen Quadraten. Drücken Sie nun eine Taste, so sind Sie wieder in der normalen Zeichendarstellung. Was Sie da gesehen haben, sind die zufälligen Werte, die die sonst unbenutzen RAM-Bereiche des Commodore 64 nach dem Einschalten haben. Unsere nächste Aufgabe soll es nun sein, die hochauflösende Grafik sowie das zugehörige Farbram zu löschen. In BASIC würden wir dies mit einer POKE-Schleife lösen.

Wir wollen alle Punkte im Grafikspeicher löschen; dazu muß jedes Bit den Wert null haben und demzufolge auch jedes Byte. Die Schleife muß von Adresse $E000 bis $FFFF gehen (exakt nur bis $FF3F, da ja nicht 8192 sondern nur 8000 Bytes dargestellt werden).

```
FOR I = 57344 TO 65535 : POKE I,0 :NEXT
```

Dies ist zwar in BASIC schnell formuliert, dauert jedoch ca. 30 Sekunden. In Maschinensprache geht das Ganze bedeutend schneller.
Eine Schleife haben wir bereits programmiert, als wir den Zeichensatz unseres Rechners auf dem Bildschirm dargestellt haben. Diese Schleife ging jedoch nur über 256 Bytes; den Bereich, den die Indexregister X und Y überstreichen können. Wir müssen jedoch 8000 Bytes löschen. Probieren wir es doch

einmal mit zwei verschachtelten Schleifen. In BASIC könnten wir das so machen:

```
AD = 57344
FOR X = 0 TO 31
FOR Y = 0 TO 255
POKE AD+Y, 0
NEXT Y
AD = AD+256
NEXT X
```

Wir haben also den Bereich von 8192 Bytes in 32 Teilbereiche ('Pages') von je 256 Bytes unterteilt. Mit der Y-Schleife werden jeweils 256 Bytes gelöscht. Dann wird die Basisadresse AD um 256 erhöht und die nächsten 256 Bytes können gelöscht werden. Dies geschieht insgesamt 32 mal; die Schleifenvariable X zählt die Durchläufe. Wie Sie sehen, werden hier immer ganze Speicher-'Seiten' zu je 256 Bytes gelöscht, was jedoch bei unserer Anwendung nicht stört. Die zuviel gelöschten 192 Bytes haben ja keinerlei Funktion. Versuchen wir nun einmal die Formulierung in Maschinensprache.

```
100 AD = $E000
110 LDA #0 ; AKKU LÖSCHEN
120 LDX #0
130 LDY #0
140 STA AD,Y
150 INY
160 BNE SYMB1
170 ; AD = AD + $100
180 INX
190 ; IST X SCHON 31 ?
200 ; NEIN, DANN ZURÜCK NACH ZEILE 130
210 .EN
```

Mit unseren bisherigen Kenntnissen könnte der Ansatz etwa so aussehen. Auch die Marke SYMB1 werden Sie vielleicht schon richtig in Zeile 140 gesetzt haben. Aber die Erhöhung der Adresse AD scheint so nicht zu funktionieren. Erinnern wir

uns dazu der indirekt indizierten Adressierung. Dabei wird die aktuelle Adresse aus einen Zwei-Byte-Zeiger in der Zeropage sowie dem Y-Register gewonnen. Diesen Zeiger können wir dann später auch um $100 erhöhen, wie in Zeile 170 gefordert. Die normale indizierte Adressierung, wie Sie noch in Zeile 140 steht, kann ja nur einen Bereich von 256 Bytes überstreichen. Auch die Abfrage des X-Registers auf 31 in Zeile 190 und der Rücksprung in Zeile 200 bei Ungleichheit ist kein großes Problem:

```
100 AD = $E000
110 LDA #0 ; AKKU LÖSCHEN
120 LDX #0
130 SYMB2 LDY #0
140 SYMB1 STA (AD),Y
150 INY
160 BNE SYMB1
170 ; AD = AD + $100
180 INX
190 CPX #32
200 BNE SYMB2
210 .EN
```

Bei der indirekt indizierten Adressierung muß die Adresse AD jedoch ein Zwei-Byte-Zeiger in der Zeropage sein und nicht wie oben eine absolute Adresse. Wir können z.B. die Adresse $FA und $FB benutzen. Dieser Zeiger muß zu Beginn erst mit dem Wert $E000 geladen werden; und zwar das Lo-Byte ($00) nach $FA und das Hi-Byte ($E0) nach $FB. Jetzt ist auch die Erhöhung dieser Adresse um $100 kein Problem mehr: wir Erhöhen das Hi-Byte einfach um eins. Wenn wir schließlich noch den RTS-Befehl anfügen, sieht unser endgültiges Programm dann so aus:

```
 90 *= $C420
100 LDA #<$E000
102 STA $FA
104 LDA #>$E000
106 STA $FB
```

```
110 LDA #0 ; AKKU LÖSCHEN
120 LDX #31
130 SYMB2 LDY #0
140 SYMB1 STA (AD),Y
150 INY
160 BNE SYMB1
170 INC $FB
180 INX
190 CPX #32
200 BNE SYMB2
205 RTS
210 .EN
```

Zusätzlich haben wir den Programmstart noch auf $C420 gelegt; die nächste freie Adresse nach unseren ersten beiden Routinen. Jetzt können wir unser Programm abspeichern und assemblieren.

```
00FA                 90 AD     =     $FA
00FB                 95 AD1    =     $FB
C420                 97        *=    $C420
C420  A9 00         100        LDA   #<$E000
C422  80 FA 00      102        STA   AD
C425  A9 E0         104        LDA   #>$E000
C427  80 FB 00      106        STA   AD1
C42A  A9 00         110        LDA   #0
C42C  A2 00         120        LDX   #0
C42E  A0 00         130 SYMB2  LDY   #0
C430  91 FA         140 SYMB1  STA   (AD),Y
C432  C8            150        INY
C433  D0 FB         160        BNE   SYMB1
C435  EE FB 00      170        INC   A01
C438  E8            180        INX
C439  E0 20         190        CPX   #32
C43B  D0 F1         200        BNE   SYMB2
C43D  60            205        RTS
                    210        .EN

C420 / C43E / 001E
```

```
SOURCEFILE IST BEISPIEL 2.SRC
 0  FEHLER

AD     00FA     AD1    00FB     SYMB1  C430     SYMB2  C42E
```

Diese Version ist bereits lauffähig; wir wollen jedoch sehen, ob wir nicht manches noch besser und eleganter lösen können. Zuerst können wir bei den Adressen AD und AD1 die Zeropageadressierung benutzen. Dies erreichen wir durch den vorangestellten Stern. Ebenso können wir das Laden des Akkus mit null in der Zeile 110 sparen, da dies ja schon in Zeile 100 geschach. Dazu kehren wir die Reihenfolge der Anweisungen 100 bis 102 und 104 bis 106 um. Zudem lassen wir die X-Schleife nun von 32 bis null laufen; dadurch entfällt der Test in Zeile 190. Weiterhin käönnen wir in Zeile 130 das Y-Register einfacher dadurch mit null Laden, indem wir einfach den Akkuinhalt hineinkopieren. Mit diesen Verbesserungen wird unser Programm noch ein Stück kürzer.

```
00FA                 90 AD     =     $FA
00FB                 95 AD1    =     $FB
C420                 97        *=    $C420
C420  A9 E0         100        LDA   #>$E000
C422  85 FB         102        STA   *AD1
C424  A9 00         104        LDA   #<$E000
C426  85 FA         106        STA   *AD
C428  A2 20         110        LDX   #32
C42A  A8            120 SYMB2  TAY
C42B  91 FA         130 SYMB1  STA   (AD),Y
C42D  C8            140        INY
C42E  D0 FB         150        BNE   SYMB1
C430  E6 FB         160        INC   *AD1
C432  CA            170        DEX
C433  D0 F5         180        BNE   SYMB2
C435  60            190        RTS
                    200        .EN

C420 / C436 / 0016
SOURCEFILE IST BEISPIEL 2.SRC
```

```
 0  FEHLER

AD     00FA     AD1    00FB     SYMB1  C42B     SYMB2  C42A
```

Das Programm wird dadurch kürzer und schneller. Probieren wir unser Programm nun gleich aus:

```
100 SYS 50176 : REM GRAFIK EIN
110 GET A$ : IF A$="" THEN 110
120 SYS 50208 : REM GRAFIK LÖSCHEN
130 GET A$ : IF A$="" THEN 130
140 SYS 50192 : REM GRAFIK AUS
```

Nach dem Starten mit RUN wird wieder auf den Grafik-Modus umgeschaltet. Drücken wir nun eine Taste, so wird der Grafikbildschirm gelöscht. Dies geschieht praktisch augenblicklich innerhalb von Sekundenbruchteilen. Mit BASIC dauerte dies noch 30 Sekunden! Durch nochmaliges Drücken kommen wir wieder aus dem Grafikmodus raus. Jetzt dürfte es uns auch nicht schwer fallen, eine entsprechende Routine zur Initialisierung des Farbrams zu schreiben.

Beim Farbram wollen wir jedoch die Möglichkeit vorsehen, einen Wert zur Initialisierung zu übergeben. Dieser Wert ist sowohl für die Farbe des Hintergrundes (nicht gesetzte Punkte) als auch für die Farbe der gesetzten Punkte zuständig. Dabei bestimmt das untere Nibble die Farbe des Hintergrundes und das obere Nibble die Farbe der gesetzten Punkte. Jedes Bytes ist dabei wie gesagt für eine Gruppe von 8 * 8 Punkten zuständig. Nehmen wir z.B. $10 so ist das untere Nibble 0 und das obere Nibble 1. Der Hintergrund ist demnach schwarz, der Vordergrund ist weiß. Wir wollen unserer Routine die Information im Akku übergeben. Versuchen Sie nun selber das Problem zu lösen und vergleichen Sie dann Ihre Lösung mit unserem Programm. Als Startadresse wählen wir $C440.

```
00FA            90 AD    =    $FA
00FB            95 AD1   =    $FB
```

```
C440                 97        *=    $C440
C440  A0 C0         100        LDY   #>$C000
C442  84 FB         102        STY   *AD1
C444  A0 00         104        LDY   #<$C000
C446  84 FA         106        STY   *AD
C448  A2 04         110        LDX   #4
C44A  91 FA         130 SYMB1  STA   (AD),Y
C44C  C8            140        INY
C44D  D0 FB         150        BNE   SYMB1
C44F  E6 FB         160        INC   *AD1
C451  CA            170        DEX
C452  D0 F6         180        BNE   SYMB1
C454  60            190        RTS
                    200        .EN

C440 / C455 / 0015
SOURCEFILE IST BEISPIEL 3.SRC
 0  FEHLER

AD     00FA     AD1    00FB     SYMB1  C44A
```

So ähnlich könnte auch Ihr Programm aussehen. Gegenüber den vorigen Programm haben wir noch eine kleine Verbesserung durchgeführt: Der Rücksprung in Zeile 180 kann ebenfalls zu SYMB1 erfolgen, da zu diesem Zeitpunkt das Y-Register noch den Wert null enthält, so daß das erneute Laden mit null bei SYMB2 überflüssig ist. Sie sollten Ihre Version jedoch jetzt einmal ausprobieren. Die Startadresse ist $C440 bzw. 50240. Vorher sollten Sie jedoch das X-Register mit dem Farbwert versorgen:. z.B. POKE 780, 16. Da wir den Grafikspeicher jedoch vorher gelöscht haben, dürften Sie keine Reaktion sehen. Nach diesen Routinen zur Vorbereitung wollen wir uns jetzt an die wichtigste Routine für unsere hochauflösende Grafik herantrauen: Das Setzen und Löschen von Grafikpunkten. Bei dieser Routine lassen sich viele Programmiertechniken demonstrieren, z.B. die Benutzung der logischen Verknüpfungen. Dazu müssen wir jedoch erst einmal wissen, in welcher Reihenfolge das Bitmuster für die Grafikpunkte im Speicher abgelegt ist. Sehen wir uns das einmal an einer

kleinen Tabelle an. Dabei wird auch die Ähnlichkeit mit der Zeichendarstellung zu 40 Zeichen in 25 Spalten deutlich.

	Spalte 0	Spalte 1	....	Spalte 39
	0	8		312
	1	9		313
	2	10		314
Zeile 0	3	11	....	315
	4	12		316
	5	13		317
	6	14		318
	7	15		319
	320	328		632
	321	329		633
	322	330		634
Zeile 1	323	331	....	635
	324	332		636
	325	333		637
	326	334		638
	327	335		639
...	....	....	....	....
	7680	7688		7992
	7681	7689		7993
	7682	7690		7994
Zeile 24	7683	7691	....	7995
	7684	7692		7996
	7685	7693		7997
	7686	7694		7998
	7687	7695		7999

Der Bildschirm bleibt also weiterhin in 25 Zeilen zu 40

Spalten unterteilt, für jedes 'Zeichen' stehen jedoch 8 Bytes zur Darstellung zur Verfügung. Ein Byte ist dabei für eine Punktreihe zuständig. Die acht Bits jeden Bytes stehen für jeden einzelnen Punkt. Das Bit mit der höchsten Wertigkeit (Bit 7) ist dabei für den am weitesten links stehenden Punkt zuständig:

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

Enthält das Byte das Bitmuster %10001100 bzw. $8C, so bedeutet das, daß der erste, der 5. und der 6. Punkt von links aus gesehen gesetzt sind. Zur besseren Handhabung der Grafik soll jeder Punkt durch seine horizontale (X) und vertikale (Y) Koordinate angesprochen werden. Diese Koordinaten sollen von 0 bis 319 bzw. von 0 bis 199 laufen. Wir müssen also zunächst einen Algorithmus finden, der uns ausgehend von den Koordinaten die Nummer des Bytes im Videoram liefert sowie zusätzlich noch die Nummer des Bits innerhalb dieses Bytes. Versuchen wir dies erst einmal in BASIC.

Dazu nun einige Überlegungen. Jeweils die X-Positionen 0 bis 7, 8 bis 15, 16 bis23 usw. liegen innerhalb des gleichen Bytes im Grafikram. Bei diesen Zahlengruppen sind jeweils die untersten 3 Bits unterschiedlich während die übrigen Bits gleich sind. Diese Bits haben also bei der Berechnung der Bytenummer keine Bedeutung und können daher gelöscht werden. Dies kann mit der AND-Funktion geschehen. Die AND-Funktion gibt nur dann als Ergebnis ein 1-Bit, wenn beide Bits gleich 1 sind. Wir können also folgende AND-Verknüpfung benutzen:

```
X  AND  %11111000
```

Mit dieser Funktion werden die untersten drei Bits gelöscht, egal welchen Wert die Koordinate X hat. Probieren Sie das in BASIC einmal aus.

```
PRINT X AND 248
```

Wie wirkt sich nun die Y-Koordinate auf die Byteposition aus? Liegt der Wert zwischen 0 und 7, so kann die Y-Koordinate direkt dazu addiert werden. Ist sie größer als 8, so müssen wir 40mal den Wert der Y-Koordinate dazu addieren, wobei die untersten drei Bits unberücksichtigt bleiben. Bei der Übersetzung in Maschinensprache müssen wir noch berücksichtigen, daß der Wert der X-Koordinate größer als 255 sein, also nicht in einem Byte ausgedrückt werden kann. Wir teilen die X-Koordinate daher in XL und XH auf. XH kann dabei nur null oder eins sein, da der größte Wert für X 319 gleich $13F ist.

```
PRINT XH*256 + (XL AND 248) + (Y AND 7) + 40*(Y AND 248)
```

Diese Formel gilt es nun in Maschinensprache zu übersetzen. Legen wir erst einmal die Register fest, in denen wir die Werte übergeben:

Y => Y-Koordinate
A => XL-Koordinate
X => XH-Koordinate

Für die X-Koordinate benutzen wir einen 2-Byte Speicher, die Y-Koordinate belassen wir im Y-Register, während wir für die Bildung der Summe einen zweiten 16-Bit-Speicher benötigen.

```
100 XL = $FA
110 XH = $FB
120 SUML = $FC
130 SUMH = $FD
140 *= $C460
150 STA *XL
160 STX *XH ; X-KOORDINATE MERKEN
170 TYA ; Y-KOORDINATE
180 AND #$F8 ;
```

Wir versuchen also den letzten Term zuerst zu berechnen. Da der Prozessor jedoch keinen Befehl zur Multiplikation hat, müssen wir uns etwas anderes einfallen lassen. Wir erinnern

uns, daß durch Verschieben nach links der Wert verdoppelt wurde. Versuchen wir deshalb, die Multiplikation mit 40 in mehrere Verdopplungen zu unterteilen.

A * 40 => A *2 *2 +A *2 *2 *2

Führt man dies der Reihe nach durch, so erhält man erst den doppelten, dann den vierfachen sowie durch Addition des ursprünglichen Werts das fünffache. Durch dreimalige Verdopplung erhält man daraus endlich den 40fachen Wert.

```
190 STA *$FE ; WERT FÜR SPÄTER MERKEN
200 STA *SUML
210 LDA #0
220 STA *SUMH ; HIBYTE LÖSCHEN
230 ASL *SUML ; VERSCHIEBUNG NACH LINKS
240 ROL *SUMH ; BEI HIBYTE CARRY BERÜCKSICHTIGEN
250 ASL *SUML
260 ROL *SUMH ; NOCHMALS VERSCHIEBEN
```

Da die Verschiebung über 16 Bit geschieht, muß beim Hibyte der ROL-Befehl benutzt werden, da hier ein eventueller Übertrag (das heraus geschobene 7. Bit) über das Carry als Bit null ins Hibyte übernommen werden muß. Jetzt kann die Addition des ursprünglichen Werts geschehen.

```
270 CLC ; ÜBERTRAG LÖSCHEN
280 LDA *SUML
290 ADC *$FE
300 STA *SUML ; ERGEBNIS WIEDER ABSPEICHERN
310 LDA *SUMH
320 ADC #0
330 STA *SUMH
```

Was soll nun die Addition von Null zu SUMH ? Dies geschieht aus dem gleichen Grunde wie oben: Falls bei der Addition von SUML ein Überlauf ins Carry aufgetreten ist, wird er bei der nächsten Addition mit berücksichtigt.

Jetzt müssen wir das Ergebnis noch dreimal verdoppeln.

```
340 ASL *SUML
350 ROL *SUMH
360 ASL *SUML
370 ROL *SUMH ; DREIMAL VERDOPPELN
380 ASL *SUML
390 ROL *SUMH
```

Damit haben wir den ersten und schwierigsten Term schon abgehandelt. Nun können wir den zweiten Ausdruck addieren.

```
400 TYA ; Y-KOORDINATE IN AKKU
410 AND #7
420 CLC
430 ADC *SUML
440 STA *SUML
450 LDA *SUMH
460 ADC #0
470 STA *SUMH
```

Auch hier haben wir wieder eine 16-Bit-Addition mit Berücksichtigung des Übertrags durchgeführt. Nun müssen wir nur noch den X-Wert AND 248 addieren.

```
480 CLC
490 LDA *XL
500 AND #$F8
510 ADC *SUML
520 STA *SUML
530 LDA *XH
540 ADC *SUMH
550 STA *SUMH
```

Da unser Grafikspeicher jedoch nicht bei Adresse 0, sondern bei $E000 beginnt, muß dieser Wert noch dazu addiert werden.

```
560 CLC
570 LDA #<$E000
```

```
580 ADC *SUML
590 STA *SUML
600 LDA #>$E000
610 ADC *SUMH
620 STA *SUMH
```

Nun endlich haben wir in SUML/SUMH die Adresse des Bytes im Grafikspeicher. Nun müssen wir noch die Bitposition innerhalb dieses Bytes bestimmen. Dies können wir wieder aus den untersten drei Bits der X-Koordinate bestimmen. Wir kommen zu folgender Wertetabelle:

```
X   Bitposition
0  =>   7
1  =>   6
2  =>   5
3  =>   4
4  =>   3
5  =>   2
6  =>   1
7  =>   0
```

Sehen wir uns dies binär an, so erkennen wir, daß dabei die Werte der einzelnen Bits umgekehrt sind. Dies kann die Exklusiv-Oder Funktion für uns erledigen.

```
630 LDA *XL
640 AND #7
650 EOR #7
```

Wir haben jetzt die Adresse des Bytes sowie die Nummer des Bits das wir setzen wollen. Aus der Nummer müssen wir jetzt den dieser Position entsprechenden Wert erzeugen. Dazu schieben wir den Wert '1' (Bitposition 0) so oft nach links, wie es die Position angibt.

```
660 TAX ; POSITION NACH X
670 LDA #1
680 SHIFT DEX
```

```
690 BMI OK
700 ASL A  ; BIT NACH LINKS VERSCHIEBEN
710 BNE SHIFT
720 OK ...
```

Wir verschieben so oft nach links, wie der Zähler im X-Register angibt. In Zeile 690 wird dazu der Inhalt des X-Register erniedrigt, um geprüft, ob er schon negativ (kleiner als null) ist. Falls ja, sind wir schon fertig; sonst verschieben wir um eine Position nach links. Der Sprung in Zeile 710 wird immer ausgeführt, da das Ergebnis des Verschiebens immer ungleich null ist. Nun haben wir also im Akku den der Bitposition entsprechenden Wert und wir könnten ihn an die oben berechnete Adresse speichern. Dabei ist jedoch zu beachten, daß bereits ein anderer Punkt innerhalb dieses Bytes gesetzt sein kann, den wir durch Abspeichern des neuen Werts überschreiben und damit löschen würden. Wir müssen also den alten Wert erhalten und den neuen nur hinzufügen. Auch dafür haben wir schon eine logische Funktion kennengelernt, die ODER-Funktion. Wir ODERn also den ursprünglichen Wert dazu und speichern das Ergebnis wieder an die gleiche Stelle ab. Dazu benutzen wir wieder die indirekt indizierte Adressierung. Den Adreßzeiger in der Zeropage haben wir ja vorher schon berechnet. Da es diese Adressierungsart jedoch nur indiziert mit Y gibt, müssen wir das Y-Register vorher löschen.

```
720 OK LOY #0
730 ORA (SUML),Y
740 STA (SUML),Y
750 RTS
```

Damit ist unser neuer Punkt gesetzt und wir sind fertig. Eine Kleinigkeit haben wir jedoch noch übersehen:
Mit der ODER-Funktion in Zeile 730 lesen wir den Inhalt einer Speicherzelle im Bereich von $E000 bis $FFFF. Der Commodore 64 gibt nun im Normalfall beim Lesen den Inhalt des Betriebsystem-ROMs wieder, wenn wir ihm nicht mitteilen, daß wir auf das RAM zugreifen wollen. Dies geschieht über den

Prozessorport an der Adresse 1. Während dieses Zugriffs müssen wir zusätzlich noch Interrupts verhindern, da die Interruptroutinen sich im zu diesem Zeitpunkt nicht erreichbaren ROM befinden. Danach stellen wir die alte Speicherkonfiguration wieder her und geben den Interrupt wieder frei.

```
730 LDX #$34 ; KONFIGURATION RAM
740 SEI ; INTERRUPT VERHINDERN
750 STX *1
760 ORA (SUML),Y
770 STA (SUML),Y ; PUNKT SETZEN
780 LDX #$37 ; KONFIGURATION ROM
790 STX *1
800 CLI ; INTERRUPT FREIGEBEN
810 RTS
820 .EN
```

Sehen wir uns nun das Assemblerlisting an.

```
00FA             100 XL     =    $FA
00FB             110 XH     =    $FB
00FC             120 SUML   =    $FC
00FD             130 SUMH   =    $FD
C460             140        *=   $C460
C460  85 FA      150        STA  *XL
C462  86 FB      160        STX  *XH  ; X-KOORDINATE MERKEN
C464  98         170        TYA       ; Y-KOORDINATE
C465  29 F8      180        AND  #$F8
C467  85 FE      190        STA  *$FE
C469  85 FC      200        STA  *SUML
C46B  A9 00      210        LDA  #0
C46D  85 FD      220        STA  *SUMH
C46F  06 FC      230        ASL  *SUML
C471  26 FD      240        ROL  *SUMH
C473  06 FC      250        ASL  *SUML
C475  26 FD      260        ROL  *SUMH
C477  18         270        CLC       ; ÜBERTRAG LÖSCHEN
C478  A5 FC      280        LDA  *SUML
```

```
C47A  65 FE     290       ADC   *$FE
C47C  85 FC     300       STA   *SUML
C47E  A5 FD     310       LDA   *SUMH
C480  69 00     320       ADC   #0
C482  85 FD     330       STA   *SUMH
C484  06 FC     340       ASL   *SUML
C486  26 FD     350       ROL   *SUMH
C488  06 FC     360       ASL   *SUML
C48A  26 FD     370       ROL   *SUMH
C48C  06 FC     380       ASL   *SUML
C48E  26 FD     390       ROL   *SUMH
C490  98        400       TYA           ; Y-KOORDINATE
C491  29 07     410       AND   #7
C493  18        420       CLC
C494  65 FC     430       ADC   *SUML
C496  85 FC     440       STA   *SUML
C498  A5 FD     450       LDA   *SUMH
C49A  69 00     460       ADC   #0
C49C  85 FD     470       STA   *SUMH
C49E  18        480       CLC
C49F  A5 FA     490       LDA   *XL
C4A1  29 F8     500       AND   #$F8
C4A3  65 FC     510       ADC   *SUML
C4A5  85 FC     520       STA   *SUML
C4A7  A5 FB     530       LDA   *XH
C4A9  65 FD     540       ADC   *SUMH
C4AB  85 FD     550       STA   *SUMH
C4AD  18        560       CLC
C4AE  A9 00     570       LDA   #<$E000
C4B0  65 FC     580       ADC   *SUML
C4B2  85 FC     590       STA   *SUML
C4B4  A9 E0     600       LDA   #>$E000
C4B6  65 FD     610       ADC   *SUMH
C4B8  85 FD     620       STA   *SUMH
C4BA  A5 FA     630       LDA   *XL
C4BC  29 07     640       AND   #7
C4BE  49 07     650       EOR   #7
C4C0  AA        660       TAX
C4C1  A9 01     670       LDA   #1
```

```
C4C3  CA          680 SHIFT  DEX
C4C4  30 03       690        BMI   OK
C4C6  0A          700        ASL   A
C4C7  DO FA       710        BNE   SHIFT
C4C9  AO 00       720 OK     LDY   #0
C4CB  A2 34       730        LDX   #$34
C4CD  78          740        SEI
C4CE  86 01       750        STX   *1
C4D0  11 FC       760        ORA   (SUML),Y
C4D2  91 FC       770        STA   (SUML),Y
C404  A2 37       780        LDX   #$37
C4D6  86 01       790        STX   *1
C4D8  58          800        CLI
C4D9  60          810        RTS
                  820        .EN

C460 / C4DA / 007A
SOURCEFILE IST BEISPIEL 3.SRC
 0  FEHLER

OK      C4C9      SHIFT  C4C3      SUMH   OOFD      SUML   OOFC
XH      OOFB      XL     OOFA
```

Probieren wir jetzt mal unsere Routinen im Zusammenhang aus.

```
100 SYS 50176 : REM GRAFIK EINSCHALTEN
110 SYS 50208 : REM GRAFIK LÖSCHEN
120 POKE 780,16 : REM SCHWARZ/WEISS
130 SYS 50240 : REM FARBE INITIALISIEREN
140 FOR X=0 TO 319
150 POKE 780, X AND 255 : REM X-LO
160 POKE 781, X / 256   : REM X-HI
170 POKE 782, X * 0.625 : REM Y
180 SYS 50272 : REM PUNKT SETZEN
190 NEXT
200 GET A$ : IF A$="" THEN 200
210 SYS 50192 : REM AUSSCHALTEN
```

Lassen Sie dieses Programm laufen, so wird eine diagonale

Linie von links oben nach rechts unten gezogen. Durch Drücken einer Taste kommen wir wieder in den Normalmodus zurück. Lassen Sie uns jetzt überlegen, wie wir einen Punkt der Grafik wieder löschen können. Es ist klar, daß die Routinen zur Berechnung der Position im Grafikram und innerhalb des Bytes die gleichen sind wie beim Setzen eines Punktes. Wir müssen jedoch den Punkt nicht setzen wie in Zeile 760 sondern löschen. Schauen wir nochmal genau was beim Setzen eines Punktes mit ORA geschieht.

```
Ehemaliges Bitmuster     % 01001000
zu setzender Punkt       % 00010000
Ergebnis mit ORA         % 01011000
```

Durch die ODER-Funktion wurde der neue Punkt gesetzt. Jetzt wollen wir den gleichen Punkt jedoch löschen. Dies können wir wie mit der AND-Funktion bewerkstelligen.

```
Ehemaliges Bitmuster     % 01011000
zu löschender Punkt      % 00010000
Ergebnis mit AND         % 00010000
```

Hier ist noch was schief gelaufen! Es sind alle anderen mit Ausnahme des zu löschen Punkt gelöscht worden. Bei dem Bitmuster zur AND-Verknüpfung darf nicht das zu löschende Bit gesetzt sein, sondern alle anderen. Mit anderen Worten müssen alle Bitwerte vorher umgekehrt werden. Dies können wir mit der EOR-Funktion machen.

```
zu löschender Punkt      % 00010000
alle Bits umdrehen       % 11111111
ergibt neues Muster      % 11101111
```

Jetzt sind alle Bit mit Ausnahme des zu löschenden gesetzt und die AND-Verknüpfung mit den ursprünglichen Muster führt zum Erfolg.

```
Ehemaliges Bitmuster     % 01011000
mit neuen Muster verkn.  % 11101111
löscht gewünschtes Bit   % 01001000
```

Wir wollen jetzt die Löschfunktion noch einbauen. Damit unsere Routine unterscheiden kann, ob der angesprochene Punkt gelöscht oder gesetzt werden soll, benutzen wir das Carryflag als Entscheidungskriterium dafür. Ist das Carryflag gelöscht, soll auch der Punkt gelöscht werden. Dazu müssen wir uns den Zustand des Carryflags zu Beginn merken. Wir benutzen dazu den Befehl PHP, der das Statusregister auf dem Stack zwischenspeichert und fügen ihn als Zeile 145 ein. In Zeile 735 holen wir die Flags mit PLP wieder von Stack. Den Rest des Programms ersetzen wir dann durch folgende Anweisungen:

```
760 BCC LÖSCH
770 ORA (SUML),Y
780 BCS OK2
790 LÖSCH EOR #$FF ; UMDREHEN
800 AND (SUML),Y
810 OK2  STA (SUML)
820 LDX #$37
830 STX *1
840 CLI
850 RTS
860 .EN
```

Ist das Carryflag also gelöscht, springen wir nach Zeile 790, wo die Bits mit EOR #$FF umgedreht werden, dann die AND-Verknüpfung durchgeführt wird sowie das Ergebnis wieder abgespeichert wird. War das Carryflag dagegen gesetzt, setzen wir mit ORA wie vorher das Bit und verzweigen dann an die Stelle, wo der neue Wert wieder abgespeichert wird. Dazu nehmen wir den BCS-Befehl, da das Carryflag durch den ORA-Befehl nicht beeinflußt wird.

```
00FA            100 XL      =     $FA
00FB            110 XH      =     $FB
00FC            120 SUML    =     $FC
00FD            130 SUMH    =     $FD
C460            140         *=    $C460
```

```
C460  08        145       PHP
C461  85 FA     150       STA   *XL
C463  86 FB     160       STX   *XH ; X-KOORDINATE
C465  98        170       TYA       ; Y-KOORDINATE
C466  29 F8     180       AND   #$F8
C468  85 FE     190       STA   *$FE
C46A  85 FC     200       STA   *SUML
C46C  A9 00     210       LDA   #0
C46E  85 FD     220       STA   *SUMH
C470  06 FC     230       ASL   *SUML
C472  26 FD     240       ROL   *SUMH
C474  06 FC     250       ASL   *SUML
C476  26 FD     260       ROL   *SUMH
C478  18        270       CLC           ; ÜBERTRAG LÖSCHEN
C479  A5 FC     280       LDA   *SUML
C47B  65 FE     290       ADC   *$FE
C47D  85 FC     300       STA   *SUML
C47F  A5 FD     310       LDA   *SUMH
C481  69 00     320       ADC   #0
C483  85 FD     330       STA   *SUMH
C485  06 FC     340       ASL   *SUML
C487  26 FD     350       ROL   *SUMH
C489  06 FC     360       ASL   *SUML
C48B  26 FD     370       ROL   *SUMH
C48D  06 FC     380       ASL   *SUML
C48F  26 FD     390       ROL   *SUMH
C491  98        400       TYA           ; Y-KOORDINATE
C492  29 07     410       AND   #7
C494  18        420       CLC
C495  65 FC     430       ADC   *SUML
C497  85 FC     440       STA   *SUML
C499  A5 FD     450       LDA   *SUMH
C49B  69 00     460       ADC   #0
C49D  85 FD     470       STA   *SUMH
C49F  18        480       CLC
C4A0  A5 FA     490       LDA   *XL
C4A2  29 F8     500       AND   #$F8
C4A4  65 FC     510       ADC   *SUML
C4A6  85 FC     520       STA   *SUML
```

```
C4A8  A5 FB       530         LDA   *XH
C4AA  65 FD       540         ADC   *SUMH
C4AC  85 FD       550         STA   *SUMH
C4AE  18          560         CLC
C4AF  A9 00       570         LDA   #<$E000
C4B1  65 FC       580         ADC   *SUML
C4B3  85 FC       590         STA   *SUML
C4B5  A9 E0       600         LDA   #>$E000
C4B7  65 FD       610         ADC   *SUMH
C4B9  85 FD       620         STA   *SUMH
C4BB  A5 FA       630         LDA   *XL
C4BD  29 07       640         AND   #7
C4BF  49 07       650         EOR   #7
C4C1  AA          660         TAX
C4C2  A9 01       670         LDA   #1
C4C4  CA          680 SHIFT   DEX
C4C5  30 03       690         BMI   OK
C4C7  0A          700         ASL   A
C4C8  D0 FA       710         BNE   SHIFT
C4CA  A0 00       720 OK      LDY   #0
C4CC  A2 34       730         LDX   #$34
C4CE  28          735         PLP
C4CF  78          740         SEI
C4D0  86 01       750         STX   *1
C4D2  90 04       760         BCC   LÖSCH
C4D4  11 FC       770         ORA   (SUML),Y
C4D6  80 04       780         BCS   OK2
C4D8  49 FF       790 LÖSCH   EOR   #$FF
C4DA  31 FC       800         AND   (SUML),Y
C4DC  91 FC       810 OK2     STA   (SUML),Y
C4DE  A2 37       820         LDX   #$37
C4E0  86 01       830         STX   *1
C4E2  58          840         CLI
C4E3  60          850         RTS
                  860         .EN

      C460 / C4E4 / 0084
      SOURCEFILE IST BEISPIEL 4.SRC
       0  FEHLER
```

```
LOSCH  C4D8     OK     C4CA     OK2    C4DC     SHIFT  C4C4
SUMH   00FD     SUML   00FC     XH     00FB     XL     00FA
```

Modifizieren wir unser Beispiel von vorhin nun etwas.

```
100 SYS 50176 : REM GRAFIK EIN
110 SYS 50208 : REM GRAFIK LÖSCHEN
120 POKE 780,16
130 SYS 50240 : REM FARBE SETZEN
140 I=1
150 FOR X=0 TO 319
160 POKE 780, X AND 255 : REM X-LO
170 POKE 781, X / 256   : REM X-HI
180 POKE 782, X * 0.625 : REM Y
190 POKE 783 ,I : REM SETZEN/LÖSCHEN
200 SYS 50272 : NEXT
210 GET A$ : IF A$="" THEN I=1-I : GOTO 150
220 SYS 50192 : REM GRAFIK AUS
```

Dieses Programm zieht nun eine diagonale Linie über den Bildschirm und löscht sie anschließend wieder. Ob unsere Routine Punkte setzt oder löscht können wir durch das Carryflag bestimmen, das zusammen mit den anderen Flags des Statusregisters in der Speicherstelle 783 übergeben wird. Da das Carryflag die Bitposition null innerhalb des Statusregisters hat, sind die entsprechenden Werte null und eins. Das Programm kann durch Drücken einer Taste unterbrochen werden. Der ursprüngliche Bildschirminhalt bleibt dabei erhalten. Ebenso bleibt der Grafikbildschirm erhalten. Wollen Sie später noch einmal auf Grafik umschalten, so lassen Sie einfach die Routine zum Löschen des Grafikbildschirms weg. Experimentieren wir jetzt doch einmal etwas mit der Routine für die Farbdarstellung.

```
100 SYS 50176 : REM GRAFIK EIN
110 SYS 50208 : REM GRAFIK LÖSCHEN
120 POKE 780,16
130 SYS 50240 : REM FARBE SETZEN
```

```
140 REM
150 FOR X=70 TO 150 : FOR Y=X TO 199
160 POKE 780, X : REM X-LO
170 POKE 781, 0 : REM X-HI
180 POKE 782, Y : REM Y
190 POKE 783 ,1 : REM SETZEN
200 SYS 50272 : NEXT : NEXT
210 FOR C=0 TO 255
220 FOR I=1 TO 500 : NEXT
230 POKE 780, C
240 SYS 50240 : REM FARBE
250 NEXT
260 SYS 50192 : REM GRAFIK AUS
```

Hier wird zunächst eine Figur gezeichnet und diese dann in allen 256 möglichen Farbkombinationen gezeichnet.

Was die Maschinenprogrammierung betrifft, so sollten Sie in diesem Kapitel einiges über die indizierte Adressierung gelernt haben. Wir haben uns mit den logischen Verknüpfungen beschäftigt, mit denen man Bits gezielt setzen und löschen kann. Auch den Stack haben wir zur Speicherung von Daten benutzt. Des weiteren haben wir Additionen und Verschiebungen über 16 Bits gesehen und was dabei zu beachten ist. Was uns jetzt noch an wichtigen Programmierkonzepten fehlt, ist die Unterprogrammtechnik, auf die wir im nächsten Abschnitt eingehen werden. Mit diesen Kentnissen und etwas Übung sollten Sie jetzt in der Lage sein, auch eine Hardcopy-Routine für die hochauflösende Grafik mittels eines Grafikdruckers zu schreiben. Hier sollten Sie sich zunächst mit dem Verfahren vertraut machen, wie Ihr Drucker Grafikdaten verarbeitet. Meist wird jeweils eine senkrechte Spalte von acht übereinander liegenden Punkten mit einem Byte übertragen, jedoch ist dies von Druckertyp zu Druckertyp unterschiedlich. Arbeitet Ihr Drucker nach dem oben beschriebenen Prinzip, so können Sie die Grafikdaten nicht einfach Byte für Byte aus dem Grafikram an den Drucker schicken, sondern Sie müssen aus jeweils 8 aufeinander folgenden Bytes die entsprechenden Bit mittels logischer Verknüpfungen isolieren

und zu dem Bitmuster zusammen setzen, das Sie dann an den Drucker schicken müssen. Wird das komplizierter, so nehmen Sie vielleicht besser ein Flußdiagramm zu Hilfe. Mit etwas Geduld werden Sie das sicher hinbekommen. Den logischen Ablauf Ihres Programms können Sie auch mit dem Einzelschritt-simulator in diesem Buch verfolgen.

8. BASIC-Erweiterungen und die Benutzung von Betriebssystem-Routinen

Im vorigen Abschnitt haben wir bei unseren Grafikroutinen die Parameter über POKE-Befehle in den Prozessor-Registern übergeben. Jetzt werden wir eine Möglichkeit kennen lernen, die dies eleganter ermöglicht und die zudem sehr einfach zu programmieren ist.

Wir werden Parameter nun genauso übergeben, wie dies der BASIC-Interpreter auch macht. Sehen wir uns dazu mal einen Befehl an, nehmen wir z.B. den POKE-Befehl.

```
POKE A, B
```

Sie wissen sicher, daß man anstelle von A und B auch beliebige Ausdrücke, Konstanten oder indizierte Variablen benutzen kann, z.B.

```
POKE A(1000)/750*INT(X%/9), EXP(ABS(SIN(3*A)))+2
```

Um nun solch einen Ausdruck auszuwerten, besitzt der BASIC-Interpreter eine universelle Routine, die dies übernimmt. Diese Routinen können wir uns nun zunutze machen und die Arbeit dem BASIC-Interpreter überlassen. Zusätzlich können durch verschiedene Einsprünge zusätzlich noch Bereichsüberprüfungen vorgenommen werden. Bei der POKE-Routine z.B. wird beim Einlesen der Parameter geprüft, ob der erste Parameter einen Wert zwischen 0 und 65535 hat (16-Bit-Wert). Ist dies nicht der Fall, wird die Fehlermeldung 'ILLEGAL QUANTITY' ausgegeben. Der zweite Parameter wird auf den Bereich von 0 bis 255, also den Byte-Bereich, geprüft und ansonsten auch die Fehlermeldung ausgegeben.

Wie können wir solche Routinen nun für unsere Zwecke nutzen?

Dazu beschäftigen wir uns erst mit einer Programmiertechnik, die wir bisher noch nicht benutzt haben: Die Unterprogrammtechnik.

Von dieser Möglichkeit haben Sie sicher in BASIC schon Gebrauch gemacht; die entsprechenden Befehle sind

GOSUB und RETURN

Der GOSUB-Befehl verzweigt zur angegebenen Zeilen. Im Unterschied zum GOTO-Befehl merkt der Interpreter sich jedoch die augenblickliche Stelle im Programm. Ist das Unterprogramm abgearbeitet und trifft das Programm auf den Befehle RETURN, so wird die vorher gespeicherte Rücksprungadresse zurückgeholt und wieder an die Stelle verzweigt, von der der Aufruf des Unterprogramms kam. Sie werden diese Möglichkeit sicher von BASIC her schon kennen.

In Maschinensprache stellt uns der 6510-Prozessor nun für den Unterprogrammaufruf zwei Befehle zur Verfügung, die exakt den BASIC-Befehlen GOSUB und RETURN entsprechen.

JSR und RTS

Dabei übernimmt JSR ('jump to subroutine') den Unterprogrammaufruf während der Befehl RTS ('return form subroutine') den Rücksprung zur aufrufenden Routine besorgt.

Wann benutzt man nun diese Befehle?

Die Unterprogrammtechnik kommt in Maschinensprache unter den gleichen Voraussetzungen zum Einsatz unter denen das auch in BASIC geschieht. Soll eine bestimmte Routine mehrmals genutzt werden, so formuliert man sie als Unterprogramm, das dann mehrmals aufgerufen wird. Dadurch kann man Speicherplatz sparen. Speziell in Maschinensprache sollten Sie noch folgendes beachten:

Um Ihre Unterprogamme möglichst universell zu gestalten, sollten Sie in den Unterprogrammen nur dann Konstanten verwenden, wenn diese bei alle Aufrufen gleich sein müssen. Ansonsten sollten Sie alle Möglichkeiten bei der Adressierung ausnutzen.

Was macht nun der Prozessor, wenn er auf den Befehl JSR stößt?

Bevor er zu der angegebenen Adresse verzweigt, muß er sich die augenblickliche Adresse (den Programmzähler) merken. Wo werden nun die beiden Bytes des Programmzählers abgelegt? Diesem Zweck dient der Stack, für den der Speicherbereich von $100 bis $1FF reserviert ist. Bei einem Unterprogrammaufruf werden nun die zwei Bytes der aktuellen Adresse im Stack abgelegt und beim Rücksprung mit RTS wird von dort die Rücksprungadresse zurückgeholt. Damit der Prozessor weiß, an welcher Stelle im Stack er die Adressen abspeichern und wieder zurückholen kann, hat er ein zusätzlichen Register, den Stackpointer (abgekürzt SP). Dieses Register dient als Zeiger auf die aktuelle Position im Stack. Sehen wir uns einmal an, was bei den Befehlen JSR und RTS auf dem Stack passiert.

```
C000  20 00 C1  JSR $C100
C003  ....

C100  60        RTS
```

Wenn wir das Programm an Adresse $C000 starten, wird in das Unterprogramm nach Adresse $C100 verzweigt. Dort steht in unserem Beispiel nun direkt der Befehls RTS und der Prozessor springt wieder auf den Befehl, der hinter dem Unterprogrammaufruf steht, in unserem Falle nach Adresse $C003. Sehen wir uns einmal an, was dabei auf dem Stack sowie mit den Stackpointer passiert.

Adresse	Befehl	Stackpointer	Stack
$C000	JSR $C100	$F9	$01F9 XX

Der Stackpointer soll z.B. den Wert $F9 haben. An der Stackadresse $01F9 stehen Daten aus vorherigen Operationen. Wenn der Prozessor nun auf den JSR-Befehl stößt, so nimmt er den Inhalt des Programmzeiger, erhöht ihn um 2 und zerlegt

ihn in Lo- und Hi-Byte. Jetzt nimmt er das Hi-Byte und speichert es an die Adresse ab, auf die der Stackpointer zeigt. Anschließend wird sofort der Stackpointer um eins erniedrigt. Es ergibt sich folgende Situation:

Adresse	Befehl	Stackpointer	Stack	
$C000	JSR $C100	$F8	$01F9	C0
			$01F8	XX

Jetzt wird das Lo-Byte der Adresse auf dem Stack gespeichert und der Stackpointer nochmal dekrementiert. Der Programmzähler wird dann auf die Startadresse des Unterprogramms gesetzt.

Adresse	Befehl	Stackpointer	Stack	
$C100	RTS	$F7	$01F9	C0
			$01F8	02
			$01F7	XX

Zusammenfassend können wir sagen, daß der Programmzähler auf dem Stack abgespeichert wurde und dabei der Stackpointer um zwei erniedrigt wurde. Der Stackpointer zeigt also immer auf den nächsten freien Platz im Stack. Sehen wir uns nun an, was beim RTS-Befehl passiert. Hier laufen die oben beschriebenen Befehle in umgekehrter Reihenfolge ab. Zuerst wird jetzt der Stackpointer erhöht und dann ein Wert vom Stack geholt.

Adresse	Befehl	Stackpointer	Stack	
$C100	RTS	$F8	$01F9	C0
			$01F8	02
			$01F7	XX

Dieser Wert $02 wird als Lo-Byte des Programmzählers betrachtet. Jetzt wird der Stackpointer nochmal erhöht und der nächste Wert vom Stack geholt.

Adresse	Befehl	Stackpointer	Stack	
$C100	RTS	$F9	$01F9	C0
			$01F8	02
			$01F7	XX

Es wurden also nacheinander $02 und $C0 vom Stack geholt. Dies wird als Programmzähler $C002 interpretiert. Abschließend wird der Programmzähler noch um eins erhöht und der nächste Befehl kann geholt werden.

Adresse	Befehl	Stackpointer	Stack	
$C003	...	$F9	$01F9	C0
			$01F8	02
			$01F7	XX

Beachten Sie, daß der Stackpointer nach der Rückkehr aus dem Unterprogramm wieder den gleichen Wert wie vorher hatte. Mit dieser Technik ist es auch möglich, Unterprogramme zu verschachteln. Wird von einem Unterprogramm aus eine weiteres Unterprogramm aufgerufen, so würde auch hier die Rücksprungadresse gespeichert und der Programmzeiger würde in unserem Beispiel auf $F5 stehen. Beim nächsten RTS-Befehl wird nun die letzte Rücksprungadresse geholt und der Stackpointer wieder auf $F7 erhöht. Bei einem RTS wird also immer an die Adresse zurückgesprungen, von der der letzte Unterprogrammaufruf aus geschah. Dadurch wird es möglich, Unterprogramme mehrfach zu schachteln.

Wenn Sie das Verfahren verstanden haben, so brauchen Sie sich bei der Anwendung der Unterprogrammtechnik nicht weiter darum zu kümmern, da der Prozessor dies beim Ausführen der Befehle automatisch für Sie erledigt.

Der 6510-Prozessor bietet auch die Möglichkeit, den Akkuinhalt sowie das Statusregister auf dem Stack zwischenzuspeichern und von dort wieder zurückzuholen. Dazu dienen die Befehle PHA und PLA für den Akku bzw. PHP und PLP für das

Statusregister. Auch bei diesen Befehlen wird der Stackpointer automatisch verwaltet: Bei Schreibbefehlen wird er erniedrigt und bei Lesebefehlen wird er erhöht. Mit diesen Befehlen kann man z.B. die Inhalte der Prozessorregister auf dem Stack zwischenspeichern und später wieder zurückholen.

```
PHA  ; Akku auf den Stack
TYA
PHA  ; und Y-Register
TXA
PHA  ; und X-Register
...
PLA
TAX  ; X-Register zurückholen
PLA
TAY  ; und Y-Register
PLA  ; und Akku
```

X- und Y-Register können wir nicht direkt auf dem Stack speichern; deshalb werden sie zuerst in den Akku übertragen und von dort auf den Stack gelegt. Beachten Sie dabei die umgekehrte Reihenfolge beim Zurückholen der Register! Dies entspricht dem Prinzip des Stacks. Was zuletzt auf dem Stack gespeichert wurde, wird zuerst wieder zurückgeholt, daher auch oft die Abkürzung LIFO ('last in, first out'). Das Abspeichern von Registerinhalten benutzt man oft vor dem Aufruf von Unterprogrammen, die die Registerinhalte verändern. Nach dem Aufruf kann man dann die ursprünglichen Registerinhalte zurückholen.

Sehr schön läßt sich die Funktion des Stacks und des Stackpointers auch mit unserem Einzelschrittsimulator demonstrieren, da Sie dort nach jedem Schritt die aktuellen Registerinhalte sehen können.

Kommen wir doch jetzt zu den Routinen des BASIC-Interpreters zurück, die wir für das Einlesen von Parametern verwenden können.

Da gibt es zunächst eine Routine, meist GETBYT genannt, die einen Ausdruck aus dem BASIC-Text liest, prüft ob der Wert zwischen 0 und 255 liegt und uns diesen Wert im X-Register zur Verfügung stellt. Diese Routine hat die Adresse $B79E. Eine weitere Routine erlaubt uns das Einlesen von 16-Bit Werten (0 bis 65535). Dazu sind zwei Routinen erforderlich. Die Routine mit dem Namen FRMNUM (Formel numerisch) holt einen beliebigen numerischen Ausdruck. Mit GETADR kann dann dieser Ausdruck auf den Wertebereich 0 bis 65535 überprüft werden. Liegt er in diesem Bereich, so wird in Adresse $14 das Lo-Byte und in $15 das Hi-Byte übergeben. Die Adressen dieser Routinen finden Sie in der Zusammenstellung weiter unten.

Wollen wir mehrere Parameter übergeben, so müssen wir diese Parameter voneinander trennen. Dazu benutzen wir, wie dies in BASIC auch geschieht, das Komma. Der BASIC-Interpreter stellt uns nun auch eine Routine zur Verfügung, die prüft, ob ein Komma folgt. Diese Routine hat den Namen CKHCOM. Oft wollen wir jedoch direkt ein Zeichen aus dem BASIC-Text lesen. Auch dafür stehen die geeigneten Programme schon zur Verfügung. Die Routine CHRGOT holt das Zeichen in den Akku auf das der Programmzeiger gerade zeigt, während mit CHRGET dieser Zeiger vor den Lesen erst um eins erhöht wird; also das nächste Zeichen geholt wird. Diese Routinen setzten auch bestimmte Flags, die weitere Informationen über das gelesene Zeichen geben. Ein gesetztes Zero-Flag bedeutet, daß entweder ein Nullbyte (in BASIC-Programmen am Zeilenende) oder ein Doppelpunkt ':' gelesen wurde, das Ende des Statement also erreicht wurde. Wurde eine Ziffer gelesen, so ist das Carry-Flag gelöscht. Hier zusammenfassend die Adressen:

```
GETBYT  $B79E
FRMNUM  $AD8A
GETADR  $B7F7
CHKCOM  $AEFD
CHRGOT  $0079
CHRGET  $0073
```

Wollen Sie nacheinander einen 16-Bit-Wert und einen 8-Bit-Wert lesen, wie dies z. B. die POKE-Funktion benötigt, so können Sie die Routine GETPAR benutzen, die auch auf das trennende Komma prüft. In GETPAR wird nacheinander FRMNUM, GETADR, CHKCOM und GETBYT aufgerufen.

```
GETPAR  $B7EB
```

Diese Routine könnten wir gut bei unseren Grafikroutinen gebrauchen, da der Wert für die X-Koordinate ein 16-Bit-Wert sein kann (0 bis 319) während für die Y-Koordinate (0 bis 199) ein 8-Bit-Wert ausreicht. Überschreiten die Werte 65535 bzw. 255, so wird vom BASIC-Interpreter 'ILLEGAL QUANTITY' ausgegeben. Wir können jedoch nachträglich unsere Grenzen noch überprüfen und bei Bedarf ebenfalls diese Fehlermeldung ausgeben. Dazu genügt ein Sprung nach Adresse $B248.

Wenn wir diese Routinen zur Parameterübergabe benutzen, könnte ein Aufruf z.B. so aussehen:

```
SYS 50240,X,Y
```

Mit diesem Verfahren ersparen wir uns zusätzliche POKEs und der Aufruf wird anschaulicher.

Entwickeln wir nun den Anfang unserer Punktsetzroutine neu, indem wir nach dem obigen Schema vorgehen:

```
100 JSR CHKCOM ; FOLGT KOMMA ?
110 JSR GETPAR ; KOORDINATEN HOLEN
120 STX YKOOR ; Y-KOORDINATE MERKEN
130 LDA $14
140 STA XL ; X-KOORDINATE LO
150 LDA $15
160 STA XH
```

Wir prüfen also zunächst auf ein Komma, das die SYS-Adresse von der X-Koordinaten trennt, und benutzen dann die Routine, die zwei Parameter holt. Den Wert des Byte-Parameters, unsere

Y-Koordinate steht im X-Register und wird an die Adresse YKOOR abgelegt; den 16-Bit-Wert mit der X-Koordinate holen wir aus $14/$15 und speichern ihn nach XL und XH. Jetzt wollen wir noch überprüfen, ob X- und Y-Koordinate auch im erlaubten Wertebereich liegen. Wir vergleichen dazu die Y-Koordinate mit 200. Ist der Wert kleiner, so ist er in Ordnung, ansonsten springen wir zur Anzeige von 'ILLEGAL QUANTITY'. Beim X-Wert müssen wir einen 16-Bit Vergleich machen.

```
170 CPX #200 ; Y >= 200 ?
180 BCC OK
190 FEHLER JMP ILLEGAL
200 OK LDA XH
210 CMP #>320
220 BCC OK1
230 BNE FEHLER
230 LDA XL
240 CMP #<320
250 BCS FEHLER
260 OK1 ...
```

Zuerst vergleichen wir den Inhalt des X-Registers, indem der Wert der Y-Koordinate steht, mit 200. Ist die Y-Koordinate kleiner, wird das Carry-Flag gelöscht und wir verzweigen zum Symbol OK. Ist sie dagegen größer oder gleich, so wird das Carry-Flag gesetzt und die Verzweigung findet nicht statt. Wir kommen dann auf die Anweisung mit der Marke FEHLER und springen zur Ausgabe der Fehlermeldung ($B248). War die Y-Koordinate in Ordnung, können wir jetzt die X-Koordinate überprpfen. Dazu laden wir den Akku mit dem Hi-Byte des Wertes und vergleichen ihn mit dem Hi-Byte von 320. Ist das Hi-Byte kleiner als das von 320, so ist der Wert auf jeden Fall kleiner als 320 und wir springen nach OK1. Ist es dagegen nicht gleich, so muß es größer sein und ist demnach auch sicher größer als 320. Wir springen also nach FEHLER. War das Hi-Byte jedoch gleich, so liegt der Wert zwischen 256 und 511 und wir müssen zur Entscheidung noch das Lo-Byte heranziehen. Dazu laden wir das Lo-Byte in den Akku und

vergleichen mit dem Lo-Byte von 320. Ist das Lo-Byte der X-Koordinate größer oder gleich dem von 320, so ist der Wert zu groß und wir springen zur Fehlerausgabe. Ansonsten gehts bei OK1 weiter und wir können die eigentliche Koordinatenberechnung durchführen.

Die Programmierung von Ein/Ausgabeoperationen

Wie Sie von Maschinensprache aus Peripheriegeräte ansprechen oder Zeichen über die Tastatur eingeben sowie auf dem Bildschirm ausgeben können, haben wir bis jetzt noch nicht gehört. Sehen wir uns dazu die entsprechenden Befehle in BASIC an.

OPEN
CMD
PRINT#
INPUT#
CLOSE

Bei der Programmierung in Maschinensprache gehen wir ganz analog vor. Das Betriebssystem hält für diese Fälle Routinen bereit, die den obigen BASIC-Befehlen entsprechen. Diese Routinen können Sie benutzen, wenn Sie Ein- oder Ausgabe-Operationen benötigen. Sehen wir uns diese Routinen mal im einzelnen an.

OPEN

Diese Routine benötigt drei Parameter: Logische Filenummer, Geräteadresse und Sekundäradresse sowie eventuell einen Filenamen. Diese Parameter werden durch die Routinen SETFLS und SETNAM gesetzt. Die OPEN-Routine selbst braucht keine Parameter, erfodert jedoch den vorherigen Aufruf der oben beschriebenen Routinen.

SETFLS

Laden Sie dazu den Akku mit der logischen Filenummer, das X-Register mit der Gerätenummer und das Y-Register mit der Sekundäradresse und rufen Sie dann diese Routine auf.

SETNAM

Damit können Sie einen Filenamen definieren. Die Länge des

Namens geben Sie in den Akku, (0 falls kein Filenamen benutzt wird) und die Startadresse des Namens (dort, wo Sie in im Speicher abgelegt haben) kommt ins X- (Lo-Byte) und ins Y-Register (Hi-Byte).

PRINT

Diese Routine ist sehr einfach zu benutzen: Laden Sie einfach das Zeichen, das Sie ausgeben wollen in den Akku und rufen Sie dann diese Routine auf. Normalerweise geht die Ausgabe auf den Bildschirm. Wollen Sie z.B. auf den Drucker ausgeben, so müssen Sie mit der OPEN-Routine erst ein File eröffnen und vor der Ausgabe die folgende Routine aufrufen:

CHKOUT

Diese Routine entspricht dem CMD-Befehl. Wollen Sie ein Zeichen in ein geöffnetes File ausgeben, so müssen Sie das X-Register mit der logischen Filenummer laden und die Routine CHKOUT aufrufen. Jetzt gehen mit PRINT sämtliche Ausgabe zu diesem Gerät, bis Sie mit der unten beschriebenen Routine wieder auf die Bildschirmausgabe zurückgehen.

CLRCH

Diese Routine hebt den CMD-Modus, den Sie durch CHKOUT erreicht haben, wieder auf. Diese Routine benötigt keine Parameter.

Für die Eingabe von Zeichen gibt es ebenfalls eine Routine.

INPUT

Diese Betriebssystemroutine holt ein Zeichen von der Tastatur und übergibt es im Akku. Wollen Sie Daten aus einer Datei holen, so muß diese vorher geöffnet werden und dann mit der folgenden Routine die Eingabe von dieser Datei aktiviert werden.

CHKIN

Diese Routine benötigt wieder die logische Filenummer im X-Register leitet die Eingabe auf eine logische Datei um. Nach dem Aufruf wird die Eingabe nicht mehr von der Tastatur geholt, sondern von der logischen Datei, deren Nummer im X-Register übergeben wurde. Aufheben können Sie diesen Modus wieder mit der Routine CLRCH.

CLOSE

Schließen können Sie eine Datei mit dieser Routine. Dazu muß die logische Filenummer im Akku sein.

Die Tabelle enthält die Aufrufadresse dieser Routinen.

Routine	Adresse
OPEN	$FFC0
SETFLS	$FFBA
SETNAM	$FFBD
PRINT	$FFD2
CHKOUT	$FFC9
CLRCH	$FFCC
INPUT	$FFCF
CHKIN	$FFC6
CLOSE	$FFC3

Dazu nun ein Beispiel: Wir wollen die BASIC-Befehle

```
OPEN 1,8,15
PRINT# 1,"I"
CLOSE 1
```

in Maschinensprache formulieren.

```
100 LDA #1 ; LOGISCHE FILENUMMER
110 LDX #8 ; GERÄTENUMMER
120 LDY #15 ; SEKUNDÄRADRESSE
130 JSR SETFLS
```

```
140 LDA #0
150 JSR SETNAM ; KEIN NAME
160 JSR OPEN ; FILE ÖFFNEN
170 LDX #1 ; LOGISCHE FILENUMMER
180 JSR CHKOUT ; AUSGABE AUF FILE
190 LDA #73 ; "I"
200 JSR PRINT
210 JSR CLRCH
220 LDA #1 ; LOGISCHE FILENUMMER
230 JSR CLOSE
240 RTS
```

In Zeile 100 bis 120 werden die Parameter für OPEN geladen und in Zeile 130 an die richtige Stelle abgespeichert. Zeile 140 legt die Länge des Filenamens mit null fest, es wird also kein Filename benutzt. Auch dieser Wert wird mit der zugehörigen Routine abgespeichert. Dann erfolgt in Zeile 160 das OPEN. Nun wird die Ausgabe auf die logische Datei 1 gelegt (Zeile 170 - 180) und der ASC-Wert von "I" mittels der PRINT-Routine zur Floppy übermittelt. Mit dem anschließenden CLRCH wird die nächste Ausgabe wieder auf dem Bildschirm geleitet. In Zeile 220 und 230 wird die Datei wieder geschlossen und mit RTS letztendlich wieder ins BASIC zurückgekehrt.

Im nächsten Beispiel wollen wir den Fehlerkanal der Floppy lesen und die Fehlermeldung auf dem Bildschirm anzeigen. Die Lösung in BASIC sieht so aus:

```
100 OPEN 1,8,15
110 INPUT#1, A,B$,C,D
120 PRINT A; B$; C; D
130 CLOSE 1
```

Da wir die Fehlermeldung direkt ausgeben wollen, brauchen wir dazu in Maschinensprache keine Variablen. Wir lesen einfach solange, bis die Statusvariable gleich 64 ist. Damit wird das Ende der Fehlermeldung signalisiert. Wir realisieren also folgende BASIC-Struktur:

```
100 OPEN 1,8,15
110 GET#1, A$ : PRINT A$;
120 IF ST <> 64 THEN 110
130 CLOSE 1
```

Dazu müssen wir noch wissen, daß die Statusvariable vom Betriebssystem in der Speicherstelle 144 gleich $90 gehalten wird. Probieren wir nun die Maschinenspracheversion.

```
 10 OPEN   = $FFCO
 20 SETFLS = $FFBA
 30 SETNAM = $FFBD
 40 PRINT  = $FFD2
 50 CLRCH  = $FFCC
 60 INPUT  = $FFCF
 70 CHKIN  = $FFC6
 80 CLOSE  = $FFC3
 90 STATUS = $90 ; STATUSVARIABLE
100 LDA #1 ; LOGISCHE FILENUMMER
110 LDX #8 ; GERÄTENUMMER
120 LDY #15 ; SEKUNDÄRADRESSE
130 JSR SETFLS
140 LDA #0
150 JSR SETNAM ; KEIN NAME
160 JSR OPEN ; FILE ÖFFNEN
170 LDX #1 ; LOGISCHE FILENUMMER
180 JSR CHKIN ; EINGABE VOM FEHLERKANAL
190 L1 JSR INPUT ; ZEICHEN HOLEN
200 JSR PRINT ; UND AUSGEBEN
210 BIT STATUS ; STATUS TESTEN
220 BVC L1
230 JSR CLRCH ; EINGABE AUF DEFAULT
240 LDA #1
250 JSR CLOSE
260 RTS
270 .EN
```

Die Sequenz zum Öffnen der Datei konnten wir aus der obigen Routine übernehmen. Jetzt soll jedoch die Eingabe von diesem

Kanal erfolgen. Das erreichen wir mit den Befehlen in Zeile 170 und 180. Das X-Register wird mit der logischen Filenummer 1 geladen und dann die Routine CHKIN aufgerufen. Ab nun wirken sämtliche Eingabebefehle auf die Floppy. In Zeile 190 holen wir uns nun ein Zeichen von der Floppy und geben es mit JSR PRINT in Zeile 200 auf den Bildschirm aus. Diese Ausgabe geht auf den Bildschirm, weil wir vorher kein CHKOUT benutzt haben. Als nächstes wird der Status getestet. Dies geschieht mit dem BIT-Befehl, den wir bis hierhin noch nicht angewendet haben. Ist das Ende der Datei erreicht, so wird der Status auf 64 gesetzt. 64 ist jedoch gleich 2_6, in Binärdarstellung %01000000. Bit 6 dieser Speicherzelle wird dann also gesetzt. Was macht nun der BIT-Befehl ? Er kopiert das Bit 6 der adressierten Speicherzelle ins V-Flag, während das Bit 7 ist N-Flag kommt. Wir brauchen also nach dem BIT-Befehl nur noch zu testen, ob das V-Flag gesetzt ist oder nicht. Der Befehl BVC verzweigt, wenn das V-Flag gelöscht ist. Dies ist dann der Fall, wenn der Status nicht 64 ist. In diesem Fall ist also das Ende noch nicht erreicht und wir springen zurück zur Eingabe von der Floppy. Ist das V-Flag aber gesetzt, so setzen wir mit JSR CLRCH den Eingabekanal zurück und schließen anschließend die Datei. Assemblieren Sie dieses Programm einmal und probieren Sie es dann aus. Bedenken Sie dabei jedoch, daß unser Assembler nur Symbolnamen bis maximal 5 Zeichen erlaubt.

```
6:    C000                .OPT P1,00
10:   FFC0        OPEN    =    $FFC0
20:   FFBA        SETFLS  =    $FFBA
30:   FFBD        SETNAM  =    $FFBD
40:   FFD2        PRINT   =    $FF02
50:   FFCC        CLRCH   =    $FFCC
60:   FFCF        INPUT   =    $FFCF
70:   FFC6        CHKIN   =    $FFC6
80:   FFC3        CLOSE   =    $FFC3
90:   0090        STATUS  =    $90
100:  C000 A9 01          LDA  #1      ; LOG. FILENUMMER
110:  C002 A2 08          LDX  #8      ; GERÄTENUMMER
```

```
120:   C004 A0 0F          LDY  #15     ; SEKUNDÄRADRESSE
130:   C006 20 BA FF       JSR  SETFLS
140:   C009 A9 00          LDA  #0      ; KEIN FILENAME
150:   C00B 20 BD FF       JSR  SETNAM
160:   C00E 20 C0 FF       JSR  OPEN    ; FILE OFFNEN
170:   C011 A2 01          LDX  #1      ; EINGABE
180:   C013 20 C6 FF       JSR  CHKIN   ; VOM FEHLERKANAL
190:   C016 20 CF FF L1    JSR  INPUT   ; ZEICHEN FLOPPY
200:   C019 20 D2 FF       JSR  PRINT   ; UND AUSGEBEN
210:   C01C 24 90          BIT  STATUS  ; STATUS TESTEN
220:   C01E 50 F6          BVC  L1
230:   C020 20 CC FF       JSR  CLRCH
240:   C023 A9 01          LDA  #1
250:   C025 20 C3 FF       JSR  CLOSE   ; FILE SCHLIESSEN
260:   C028 60             RTS
```

Wie Sie aus dem Listing entnehmen konnten, haben wir diesmal unser Programm mit einem anderen Assembler für den Commodore 64, PROFI-ASS 64 2.0, assembliert. Diesen Assembler finden Sie später in diesem Buch beschrieben. Die Anweisung an den Assembler in Zeile 6 bedeutet, daß das Listing in die vorher geöffnete logische Datei 1 gehen sollen, während der Objektkode (das erzeugte Maschinenprogramm) direkt in den Speicher geht. Probieren wir unsere Routine einmal durch Aufruf mit

```
SYS 49152
```

aus, so erscheint die Fehlermeldung der Floppy auf dem Bildschirm, z.B.

```
00, OK,00,00
```

9. Wie erzeugt man BASIC-Ladeprogramme ?

Wenn man Maschinenprogramme geschrieben hat und diese in Form von Listings weitergeben will, so steht man meist vor der Frage, in welcher Form man dies tun will. Hat der Anwender einen Assembler, so können Sie grundsätzlich das Assembler-Quellprogramm weitergeben. Oft verfügt der Anwender jedoch nicht über einen Assembler. In diesem Falle hat sich folgende Methode bewährt:

Man gibt das Maschinenprogramm als Folge von DATAs ein, die mit einem kleinen BASIC-Programm gelesen und mittels POKE im Speicher abgelegt werden. Man kann sich nun von einem BASIC-Programm mittels einer Schleife sämtliche Werte dezimal ausgeben lassen und diese als DATAs in ein Ladeprogramm einbauen. Wir haben für Sie zu diesem Zweck ein BASIC-Programm geschrieben, was dies vollautomatisch macht. Die Anwendung ist folgende: Laden Sie erst Ihr Maschinenprogramm und anschließend das unten abgedruckte BASIC-Program und starten es. Sie werden jetzt aufgefordert, Start- und Endadresse des Maschinenprogramms einzugeben. Alles weitere erledigt jetzt dieses Programm für Sie. Es erzeugt auf dem Drucker ein komplettes Ladeprogramm mit einer automatisch gebildeten Prüfsumme. Dazu werden einfach alle Werte des Maschinenprogramms addiert. Beim Laden werden ebenfalls die Werte summiert und die dabei erhaltene Prüfsumme mit der vom Programm erzeugten verglichen. Sind die beiden Werte nicht gleich, wird eine entsprechende Meldung ausgegeben. Dadurch wird der Anwender darauf aufmerksam gemacht, daß er beim Abtippen der DATAs einen Fehler gemacht hat. Dadurch lassen sich viele Eingabefehler feststellen und durch gezieltes Überprüfen korrigieren. Die Praxis hat nämlich gezeigt, daß gerade beim Abtippen von DATA-Statements sich recht häufig Fehler einschleichen können, da die DATAs für den Anwender ja keine sinnfällige Bedeutung haben wie z.B. Befehlsworte wie PRINT. In diesem Buch finden Sie übrigens ein derartiges Ladeprogramm in dem Kapitel über den Assembler.

```
100 OPEN 1,4 : Z=100
110 INPUT "STARTADRESSE ";A
120 INPUT "ENDADRESSE   ";E
130 CMD1 : PRINT Z "FOR I =" A "TO" E
140 I=A : Z=Z+10:PRINT Z "READ X : POKE I,X : S=S+X : NEXT
150 Z=Z+10 : N=0 : PRINT Z "DATA ";
160 X=PEEK(I) : S=S+X : PRINT RIGHT$("  "+STR$(X),3);:N=N+1
170 IF I=E THEN PRINT : GOTO200
180 I=I+1:IF N=12 THEN PRINT:GOTO150
190 PRINT",";:GOTO160
200 PRINT Z+10 "IF S <>" S "THEN PRINT"
    CHR$(34) "FEHLER IN DATAS !!" CHR$(34) : END
210 PRINT Z+20 "PRINT " CHR$(34) "OK" CHR$(34)
220 PRINT#1:CLOSE1
```

10. 6510-Disassembler für den Commodore 64

In diesem Kapitel stellen wir Ihnen einen Disassembler vor. Zweck eines solchen Programms ist es, Maschinenprogramme, die im Speicher Ihres Rechners stehen, in die symbolischen Bezeichnungen zurück zu übersetzen, die man auch bei der Eingabe von Maschinenprogrammen benutzt. Aus der Bytefolge $A9, $80 macht der Disassembler z.B. LDA #$80. Der Disassembler wird einfach mit RUN gestartet und fragt dann nach der Start- und der Endadresse des Bereiches, den er disassemblieren soll. Die Ausgabe erfolgt dann auf den Bildschirm, kann aber durch Einfügen eines entsprechenden OPEN-Befehls mit einer anschließenden CMD-Anweisung auch auf einen Drucker umgeleitet werden.

Noch kurz etwas zur Arbeitsweise des Disassemblers: Das Programm holt sich ein Byte aus den Speicher und interpretiert dies als Befehlskode. Dieser Befehlskode dient als Index in eine Tabelle, aus der das Befehlswort, z.B. LDA, sowie die Adressierungsart des Befehls geholt wird. Aus der Länge weiß der Disassembler, wie lang der Befehl ist und in welcher Form der Operand auszugeben ist. Dies geschieht über Unterprogramme, die im Anschluß an das Listing zusammen mit den wichtigsten Variablen beschrieben sind.

Der Disassembler kann sowohl für eigene Maschinenprogramme als auch zur Disassemblierung von Betriebssystem und BASIC-Interpreter eingesetzt werden. In diesen Programmen kann man oft Anregungen für die eigene Programmierung finden. Noch besser eignen sich dazu jedoch kommentierte Listings des Betriebssystems, wie sie z.B. in '64 intern' zu finden sind.

Hier nun das Listing des Disassemblers.

```
100 REM 6510 - DISASSEMBLER
110 DIMMN$(255),AD(255),H$(15)
120 FF=255:HI=256:UL=2↑16:SC=2↑15-1
130 PRINT"6510 - DISASSEMBLER"
140 FORI=0TO15:READH$(I):NEXT
150 FORI=0TO255:READMN$(I),AD(I):NEXT
160 PRINT"STARTADRESSE:-     $****";:INPUTA$
170 GOSUB540:S=A
180 PRINT"ENDADRESSE   :-     $****";:1NPUTA$:PRINT
190 GOSUB540:E=A
200 FORP=STOE
210 A=P:GOSUB450:REM ADRESSE
220 PRINT"  ";:A=PEEK(P):GOSUB480:PRINT"  ";:I=PEEK(P):OP=AD(I)
230 0N0PGOSUB510,520,520,510,530,520,520,530,530,520,520,520
    ,530
240 PRINT"  ";MN$(I)"  ";
250 ONOPGOSUB270,280,290,300,310,320,330,340,350,360,370,380
    ,400
260 NEXTP:GOTO160
270 PRINT:RETURN
280 PRINT"#";:GOSUB490:P=P+1:PRINT:RETURN
290 GOSUB490:P=P+1:PRINT:RETURN
300 PRINT" A":RETURN
310 OOSUB420:P=P+2:PRINT:RETURN
320 GOSUB490:P=P+1:PRINT",X":RETURN
330 OOSUB490:P=P+1:PRINT",Y":RETURN
340 GOSUB420:P=P+2:PRINT",X":RETURH
350 GOSUB420:P=P+2:PRINT",Y":RETURN
360 PRINT"(";:OOSUB490:P=P+1:PRINT"),Y":RETURN
370 PRINT"(";:GOSUB490:P=P+1:PRINT",X)":RETURN
380 T=PEEK(P+1):Q=T+HI*(T>127)+2+P
390 A=INT(Q/HI)*HI+((Q+(Q>SC)*UL)ANDFF):PRINT"$";:GOSUB450
    :P=P+1:PRINT:RETURN
400 PRINT"(";:G0SUB420
410 PRINT")":P=P+2:RETURN
420 PRINT"$";
430 A=PEEK(P+1)+HI*PEEK(P+2)
440 REM HEXADRESSE A
450 HB=INT(A/HI):A=A-HI*HB
```

```
460 PRINTH$(HB/16)H$(HBAND15);
470 REM HEXBYTE A
480 PRINTH$(A/16)H$(HAND15);:RETURN
490 PRINT"$";
500 A=PEEK(P+1):GOTO480
510 PRINT"      ";:RETURN
520 GOSUB500:PRINT"   ";:RETURN
530 OOSUB500:PRINT" ";:A=PEEK(P+2):GOTO480
540 IFASC(A$)=42THENEND
550 A=0:FORI=1T04:V=ASC(RIGHT$(A$,I))-48:V=V+(V>9)*7
    :A=A+V*(16↑(I-1)):NEXT:RETURN
1000 DATA0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
1010 DATA"BRK",1,"ORA",11,"???",1
1020 DATA"???",1,"???",1,"ORA",3
1030 DATA"ASL",3,"???",1,"PHP",1
1040 DATA"ORA",2,"ASL",4,"???",1
1050 DATA"???",1,"ORA",5,"ASL",5
1060 DATA"???",1,"BPL",12,"ORA",10
1070 DATA"???",1,"???",1,"???",1
1000 DATA"ORA",6,"ASL",6,"???",1
1090 DATA"CLC",1,"ORA",9,"???",1
1100 DATA"???",1,"???",1,"ORA",8
1110 DATA"ASL",8,"???",1,"JSR",5
1120 DATA"AND",11,"???",1,"???",1
1130 DATA"BIT",3,"AND",3,"ROL",3
1140 DATA"???",1,"PLP",1,"AND",2
1150 DATA"ROL",4,"???",1,"BIT",5
1160 DATA"AND",5,"ROL",5,"???",1
1170 DATA"BMI",12,"AND",10,"???",1
1180 DATA"???",1,"???",1,"AND",6
1190 DATA"ROL",6,"???",1,"SEC",1
1200 DATA"AND",9,"???",1,"???",1
1210 DATA"???",1,"AND",8,"ROL",8
1220 DATA"???",1,"RTI",1,"EOR",11
1230 DATA"???",1,"???",1,"???",1
1240 DATA"EOR",3,"LSR",3,"???",1
1250 DATA"PHA",1,"EOR",2,"LSR",4
1260 DATA"???",1,"JMP",5,"EOA",5
1270 DATA"LSR",5,"???",1,"BVC",12
```

```
1200 DATA"EOR",10,"???",1,"???",1
1290 DATA"???",1,"EOR",6,"LSR",6
1300 DATA"???",1,"CLI",1,"EOR",9
1310 DATA"???",1,"???",1,"???",1
1320 DATA"EOR",8,"LSR",8,"???",1
1330 DATA"RTS",1,"ADC",11,"???",1
1340 DATA"???",1,"???",1,"ADC",3
1350 DATA"ROR",3,"???",1,"PLA",1
1360 DATA"ADC",2,"ROR",4,"???",1
1370 DATA"JMP",13,"ADC",5,"ROR",5
1380 DATA"???",1,"BVS",12,"ADC",10
1390 DATA"???",1,"???",1,"???",1
1400 DATA"ADC",6,"ROR",6,"???",1
1410 DATA"SEI",1,"ADC",9,"???",1
1420 DATA"???",1,"???",1,"ADC",8
1430 DATA"ROR",8,"???",1,"???",1
1440 DATA"STA",11,"???",1,"???",1
1450 DATA"STY",3,"STA",3,"STX",3
1460 DATA"???",1,"DEY",1,"???",1
1470 DATA"TXA",1,"???",1,"STY",5
1480 DATA"STA",5,"STX",5,"???",1
1490 DATA"BCC",12,"STA",10,"???",1
1500 DATA"???",1,"STY",6,"STA",6
1510 DATA"STX",7,"???",1,"TYA",1
1520 DATA"STA",9,"TXS",1,"???",1
1530 DATA"???",1,"STA",8,"???",1
1540 DATA"???",1,"LDY",2,"LDA",I1
1550 DATA"LDX",2,"???",1,"LDY",3
1560 DATA"LDA",3,"LDX",3,"???",1
1570 DATA"TAY",1,"LDA",2,"TAX",1
1580 DATA"???",1,"LDY",5,"LDA",5
1590 DATA"LDX",5,"???",1,"BCS",12
1600 DATA"LDA",10,"???",1,"???",1
1610 DATA"LDY",6,"LDA",6,"LDX",7
1620 DATA"???",1,"CLY",1,"LDA",9
1630 DATA"TSX",1,"???",1,"LDY",8
1640 DATA"LDA",8,"LDX",9,"???",1
1650 DATA"CPY",2,"CMP",11,"???",1
1660 DATA"???",1,"CPY",3,"CMP",3
```

```
1670 DATA"DEC",3,"???",1,"INY",1
1680 DATA"CMP",2,"DEX",1,"???",1
1690 DATA"CPY",5,"CMP",5,"DEC",5
1700 DATA"???",1,"BNE",12,"CMP",10
1710 DATA"???",1,"???",1,"???",1
1720 DATA"CMP",6,"DEC",6,"???",1
1730 DATA"CLD",1,"CMP",9,"???",1
1740 DATA"???",1,"???",1,"CMP",8
1750 DATA"DEC",8,"???",1,"CPX",2
1760 DATA"SBC",11,"???",1,"???",1
1770 DATA"CPX",3,"SBC",3,"INC",3
1780 DATA"???",1,"INX",1,"SBC",2
1790 DATA"NOP",1,"???",1,"CPX",5
1800 DATA"SBC",5,"INC",5,"???",1
1810 DATA"BEQ",12,"SBC",10,"???",1
1820 DATA"???",1,"???",1,"SBC",6
1830 DATA"INC",6,"???",1,"SED",1
1840 DATA"SBC",9,"???",1,"???",1
1850 DATA"???",1,"SBC",8,"INC",8
1860 DATA"???",1
```

Programmbeschreibung

100 - 150 Initialisierung, Aufbau der Felder.

160 - 190 Eingabe von Start- und Endadresse zur Disassemblierung und Umwandlung in dezimale Form.

200 - 260 FOR-NEXT-Schleife zur Disassemblierung von Start- bis Endadresse. Befehle mit Operanden erhöhen den Programmzähler automatisch. In Zeile 220 wird der Befehlskode geholt sowie die laufende Adresse ausgegeben. In 230 werden abhängig von der Adressierungsart die Operandenbytes ausgegeben. Zeile 240 gibt das Befehlswort aus und in Zeile 250 wird der Operand entsprechend der Adressierungsart ausgegeben. Zeile 260 schließlich beendet die Schleife und springt zurück zur Eingabe.

280 - 410 Hier wird der Operand so ausgegeben, wie es der Adressierungart entspricht.

420 - 530 Diese Unterprogramme dienen zur hexadezimalen Ausgabe von Bytes und Adressen.

540 - 550 Umwandlung einer Hexzahl in die Dezimalform.

1000 - 1860 Tabellen mit Befehlsworten und Adressierungsarten.

Variablenbeschreibung

MN$(255) Tabelle der Befehlsworte

AD(255) Tabelle mit den zugehörigen Adressierungsarten.

H$(15) Feld mit den Hexziffern

FF Konstante 255

HI Konstante 256

UL Konstante 65536

SC Konstante 32767

A$ Stringvariable für Hexzahl

S Startadresse

E Endadresse

P Programmzähler

OP Adressierungsart

11. Professionelle Hilfsmittel zur Erstellung von Maschinenprogrammen

Im Verlaufe dieses Buches haben wir Ihnen einen Assembler vorgestellt, den wir zur Assemblierung unserer Maschinenprogramme eingesetzt haben. Dieser Assembler ist selbst in BASIC geschrieben und belegt ca. 11 KByte an Speicher. Dieser 2-Pass-Assembler erlaubte uns die symbolische Bezeichnung von Variablen und Adressen. Sofern wir nur relativ kleine Programme entwickeln wollen, können wir durchaus mit solch einem Assembler arbeiten. Werden unsere Programme jedoch mit wachsender Erfahrung größer und komplexer, so werden wir bald an die Grenzen eines solchen Assemblers stoßen. Besonders der Zeitfaktor macht sich bei längeren Quellprogrammen unangenehm bemerkbar. Haben wir Quellprogramme von mehreren hundert Programmzeilen, wie sie beim Assemblieren von längeren Routinen oder mehreren Routinen gemeinsam durchaus vorkommen, so wird unsere Geduld arg strapaziert. Besonders ärgerlich ist es da, wenn der Assembler noch einen Fehler entdeckt, den wir verbessern müssen. Dazu müssen wir erst das Quellprogramm neu laden, korrigieren, wieder abspeichern und dann den Assembler laden und die ganze Prozedur von neuen starten, um dann vielleicht festzustellen, daß in der übernächsten Programmzeile noch ein Fehler steckt.

Deshalb stellen wir Ihnen hier einen professionellen Assembler für den Commodore 64 vor, der selbst in Maschinensprache geschrieben ist und 8 KByte Speicher belegt. Die Eingabe des Quellprogramms erfolgt wie bei unserem Assembler analog einem BASIC-Programm, jedoch ist es nicht erforderlich, daß das Quellprogramm auf Diskette abgespeichert wird. Der Assembler wird einfach mit einem SYS-Befehl aufgerufen und assembliert dann in Sekunden das, wofür unser BASIC-Assembler Minuten braucht. Wir haben die wichtigsten Eigenschaften dieses Assemblers für Sie einmal zusammen gestellt.

PROFI-ASS 64 2.0 ist ein komfortabler 2-Pass 6510- bzw. 6502-MACRO-Assembler für den Commodore 64. Er ist selbst vollkommen in Maschinensprache geschrieben, belegt 8 KByte RAM-Speicher und wird von Diskette geladen. Er erlaubt eine formatfreie Eingabe über den residenten BASIC-Editor, komplette Assemblerlistings, ladbare Symboltabellen, verschiedene Möglichkeiten zur Speicherung des erzeugten Opcodes, redefinierbare Symbole und eine Anzahl verschiedener Pseudoopcodes (Assembleranweisungen). Die Syntax lehnt sich stark an das MOS-Standardformat an.

Assemblerquellprogrammewerdeneinfachwieeinbasic-Programm mit Zeilennummern eingegeben. Genauso wie in BASIC können Sie auch Zeilen ändern, löschen oder einfügen. Dadurch ist kein eigener Editor erforderlich und es steht Ihnen mehr Speicherplatz für Ihr Quellprogramm zur Verfügung; insgesamt 34 KByte. Mit PROFI-ASS 64 2.0 können Sie sogar mehrere Assemblerbefehle durch Doppelpunkt getrennt in eine Zeile schreiben, genau wie in BASIC.

Zeilen mit PROFI-ASS 64 2.0-Quellcode bestehen aus Symbolen, Befehlsmnemonics, den Operanden und Kommentaren. Zusätzlich gibt es noch mehrere sogenannte "Pseudo-Opcodes", die keine Maschinenbefehle darstellen, sondern PROFI-ASS 64 2.0 anweisen, spezielle Dinge zu tun.

Jeder Programmzeile mit einem Mnemonic oder einem Pseudop kann man ein Label (Marke, Symbol) geben. Soll eine Zeile ein Label erhalten, so schreibt man es einfach vor die Anweisung, gefolgt von einem oder mehreren Leerzeichen. Ein Label muß mit einem Buchstaben beginnen, danach können Buchstaben oder Ziffern oder Punkte folgen. Unterschiedliche Labels müssen sich in den ersten 8 Zeichen unterscheiden. Nicht alphanumerische Zeichen sind nicht erlaubt, obwohl nur ein Leerzeichen ein Label vom nachfolgenden Befehl trennt.

Befehlsmnemonics können nach einem Label oder am Anfang der Zeile eingegeben werden, falls kein Label da ist. Alle Mnemonics bestehen aus drei Buchstaben. Mnemonics sind

reservierte Worte und dürfen nicht als Label benutzt werden. Beginnt der Befehl mit einem Punkt ".", so handelt es sich um ein Pseudoop. Es gibt noch drei Pseudoops, die nicht mit einem Punkt, sondern speziellen Sonderzeichen beginnen. Alle Pseudoops müssen durch ein Leerzeichen von ihren Operanden getrennt werden, mit der Ausnahme von "=" und "*=". Pseudoops, die mit einem Punkt beginnen, werden nur in den ersten drei Buchstaben unterschieden, im Assemblerlisting jedoch vollständig ausgedruckt.

Eine Assemblerzeile kann an jeder Stelle durch ein Semikolon ";" abgeschlossen werden; alles was danach folgt, gilt als Kommentar. Kommentare werden beim Assemblerlisting mit ausgedruckt, ansonsten aber nicht beachtet. Ein Doppelpunkt innerhalb eines Kommentars beendet diesen und beginnt eine neue Anweisung, sofern er nicht innerhalb von Anführungsstrichen steht. Der Assembler erkennt auch Zeilen, die nur einen Kommentar enthalten, d.h. die mit einem Semikolon beginnen.

Das Operandenfeld enthält die Adressierungsart und einen Ausdruck für den Befehl oder den Pseudoop. Es darf wie gewöhnlich von einem Semikolon gefolgt werden.

Die Adressierungsarten mit den Ausdrücken haben folgende Syntax:

```
#ausdruck                 unmittelbare Adressierung
ausdruck                  absolute oder relative Adressierung
ausdruck,x  absolut,x     indiziert mit x
ausdruck,y  absolut,y     indiziert mit y
(ausdruck,x)              indiziert indirekte Adressierung
(ausdruck),y              indirekt indizierte Adressierung
(ausdruck)                indirekte Adressierung
```

PROFI-ASS 64 2.0 konvertiert die absolute Adressierung automatisch zur Zero-Page-Adressierung, wenn der Ausdruck

einen Wert von kleiner als 256 hat. Es ist also nicht wie bei unserem Assembler erforlich, dies durch einen vorangestellten Stern dem Assembler mitzuteilen. Will man dagegen absolute Adressierung erzwingen, kann man vor den Ausdruck ein Ausrufungszeichen setzen. LDA !5,X erzeugt den Code BD 05 00, die absolute Form von LDA; während LDA 5,X die Zero-Page-Adressierung B5 05 ergibt.

Ausdrücke

PROFI-ASS 64 2.0 erhält seine Vielseitigkeit durch die Fähigkeit, beliebig komplizierte Ausdrücke zu berechnen. Der Assembler hat eine rekursive Routine zur Berechnung von beliebig verschachtelten Ausdrücken, was Ihnen weit größere Möglichkeiten einräumt als der MOS Standard oder andere Assembler zulassen. Unser Assembler läßt ja nur Konstanten sowie Labels zu. Ein PROFI-ASS 64 2.0-Ausdruck kann überall dort stehen, wo das Wort "Ausdruck" in der obigen Aufzählung erscheint. Auch bei den Pseudoops ist solch ein Ausdruck erlaubt, sofern ein numerischer Ausdruck erwartet wird. Die Ausdrucksberechnung mit PROFI-ASS 64 2.0 ist deshalb so leistungsfähig, damit Sie Ihre Assemblerprogramme vollständig symbolisch bezeichnen können. Dadurch ist das Ändern und Verschieben von PROFI-ASS 64 2.0-Programmen besonders einfach und übersichtlich.

Die Syntax von Ausdrücken ist sehr einfach und hat den MOS-Standard als Untermenge. Ausdrücke werden so eingegeben, wie man sie in einen Taschenrechner eingibt, der keine bindende Operatoren, jedoch Klammern hat. Alle Operatoren werden strikt von links nach rechts abgearbeitet, jedoch sind sowohl runde als auch eckige Klammern erlaubt, um die Reihenfolge der Abarbeitung zu ändern.

Folgende Operatoren stehen zur Verfügung:

```
+   addiert Werte
-   subtrahiert rechten Wert von linkem Wert
*   multipliziert Werte
```

```
/   dividiert linken Wert durch rechten Wert
!   logisches OR von zwei Werten
&   logisches AND von zwei Werten
^   logisches XOR (exklusiv oder) von zwei Werten
>   verschiebt linkes Argument um soviel Bits nach rechts,
    wie das rechte Argument angibt
<   verschiebt linkes Argument um soviel Bits nach links, wie
    das rechte Argument angibt
```

Alle Operationen werden mit 16 Bit Werten durchgeführt, jedoch führen verschiedene Operationen zu Überläufen, z.B. Multiplikation mit Werten über 32767 oder Verschieben um mehr als 15 Bit nach links; es erscheint dann ein 'ILLEGAL QUANTITY ERROR'. Bei Addition und Subtraktion wird ein Ergebnis über 65535 jedoch als negative Zahl im Zweierkomplement betrachtet.

Die Operanden selbst können in verschiedenen Formen erscheinen. Im folgenden wird die Syntax anhand von Beispielen beschrieben.

Operanden-Typen

Typ	Beispiel
Hexadezimal	$1C3
Dezimal	127
Binär	%110011
PC	*
ASCII-Zeichen	"A"
Label	SYMB
Ausdruck	("Z"+6)

Jeder der obigen Terme kann mit den weiter oben beschriebenen Operatoren verknüpft werden. Diese können gruppenweise geklammert werden, um die Reihenfolge der Verarbeitung zu ändern. Vor jedem Operand, einschließlich einem geklammerten Ausdruck, kann ein Minuszeichen stehen, was bedeutet, daß das Zweierkomplement zu nehmen ist. Einem kompletten Ausdruck

kann noch ein modifizierendes Zeichen vorausgehen. Die Bedeutung des Ausrufungszeichen wurde bereits weiter oben erklärt. Zusätzlich ist noch das "größer" und "kleiner"-Zeichen erlaubt. ">" vor einem Ausdruck bedeutet, daß Sie nur das High-Byte wollen, während "<" das Low-Byte kennzeichnet. Dies ist bei der direkten Adressierung oder beim .BYTE Pseudoop erforderlich. Der High-Byte Operator (>) tut dasselbe wie

Ausdruck > 8

und der Low-Byte Operator ist gleichzusetzen mit

Ausdruck & $FF

Diese Operatoren kennt unser Assembler bei der unmittelbaren Adressierung auch.

Pseudo-Opcodes

.BYTE

Der .BYTE Pseudoop wird benutzt, um Ein-Byte-Werte an der Programmzählerposition einzufügen. Er entspricht damit unserer .BY-Anweisung. Als Operanden dienen gültige PROFI-ASS 64 2.0-Ausdrücke, die durch Komma getrent sind. Die Anzahl ist nur durch die Zeilenlänge und die Länge des PROFI-ASS 64 2.0-Puffers begrenzt. Es können beliebige Ausdrücke mit beliebigen Operatoren benutzt werden; das Ergebnis muß jedoch einen Ein-Byte-Wert ergeben, sonst gibt es einen 'ILLEGAL QUANTITY ERROR'. Zwei-Byte-Werte können also mit ">" und "<" modifiziert werden, um High- oder Low-Byte zu erhalten. Ein Ein-Byte-Wert liegt im Bereich 0 bis 255 oder $FF80 bis $FFFF. Die hohen Werte sind erlaubt, da sie normalerweise negative Zahlen von -128 bis -1 bedeuten. Deshalb ist auch die Zeile ".BYTE -1" erlaubt. .BYTE dient zum Definieren von Tabellen wie Sprungtabellen oder Zeigern. Man kann damit auch Befehle "einschmuggeln", wie z.B. den BIT-Befehl; ein bekannter Programmierkniff:

```
      .BYTE $2C  ; absoluter BIT-Befehl
LABEL LDA #-1 ; versteckter LDA-Befehl
```

.WORD

Der .WORD Pseudopcode wird benutzt, um Zwei-Byte Adressen im Standardformat Low, High in den Objektcode einzufügen. ".WORD ADRESSE" ist dasselbe wie ".BYTE <ADRESSE, >ADRESSE".

.FILE

Wenn man mehrere Quellprogramme verketten will, so dient dazu der .FILE Pseudoop. Die Syntax ist folgende:

.FILE Gerätenummer, "filename"

wobei 'Gerätenummer' die Nummer der Floppy (8) oder der Datasette (1) ist und 'filename' ist der Name des Assemblerprogramms, das nachgeladen werden soll. Wenn Sie also ein sehr langes Assemblerprogramm schreiben, so können Sie es in mehreren Teilen schreiben und diese beim Assemblieren mit .FILE miteinander verketten.

.IF

Das .IF wird für bedingte Assemblierung benutzt. Es hat einen Ausdruck als Argument und wird in Pass eins und zwei berechnet. Falls der Ausdruck nicht null ist, wird der Code in der selben Zeile nach .IF assembliert. Meist wird dort ein .GOTO zu einer weiteren Verzweigung stehen. Der zusätzlich Code in der Zeile muß durch ':' getrennt sein. Mit .IF, .GOTO und der Redefinition von Symbolen ist es möglich, Assemblerschleifen aufzubauen. Obwohl .IF nur auf ungleich Null testet, ist es mit einfachen Tricks möglich, andere Vergleiche anzustellen. Das Shiften um 15 Bits nach rechts gibt ein Ergebnis von 1, wenn der Ausdruck negativ war und 0 bei positiv. Zwei Zahlen können also verglichen werden, indem man sie voneinander abzieht und das Resultat auf positiv oder negativ überprüft.

.GOTO

Der .GOTO Pseudoop veranlaßt einen unbedingten Sprung zu der Zeilennummer, die als Argument angegeben ist. Diese Zeilennummer kann auch ein Ausdruck sein. Zusammen mit .IF und dem Dekrementieren eines Symbols läßt sich so eine Schleife aufbauen.

.GTB

Dieser Pseudop steht für Go To BASIC. Er hat kein Argument und übergibt die Kontrolle an BASIC. Die im Programm folgenden BASIC-Befehle werden ausgeführt. Die Rückkehr zum Assembler geschieht über einen speziellen Einsprungpunkt im Assembler.

.ASC "TEXT"

Dieser Pseudoop dient zum Einfügen von Text in das Assemblerprogramm. Damit könnten wir z. B. beim OPEN-Befehl den Filenamen in den Assemblertext ablegen.

.SYS

Hiermit kann ein eigenes Maschinenprogramm während der Assemblierung aufgerufen werden. Als Einsprungadresse kann ein beliebiger Ausdruck stehen. Der Befehl ist identisch mit dem SYS-Befehl im BASIC. Der Aufruf des Programms geschieht in Pass eins und zwei. Der .SYS-Befehl kann z.B. von Programmierern, die sich mit der internen Arbeitsweise von PROFI-ASS 64 2.0 gut auskennen, zum Generieren von eigenen Pseudoops benutzt werden.

.SST Gerätenummer, Sekundäradresse, "Filename"

Symboltabellen können auf IEC-Busgeräten wie z.B. Floppydisk abgespeichert und von dort wieder geladen werden. .SST wird nur in Pass eins durchgeführt und speichert die Symboltabelle so wie sie bis zu diesem Punkt generiert wurde. Das erste Argument ist die IEC-Gerätenummer, normalerweise 8 für die Floppydisk. Die Sekundäradresse kann dann zwischen 2 und 14 liegen. Der Dateinamen wird wie bei einem OPEN-Befehl angegeben, erfodert also ein ",S,W" ab Ende des Namens für sequentiell und write. Dieser Pseudoop wird auch benötigt, wenn man ein sortiertes Listing der Symbole und Labels erzeugen will. Diese Datei kann als Eingabe für das Programm zum Symbollisten dienen.

.LST Gerätenummer, Sekundäradresse, "Filename"

Dies ist das Gegenstück zu .SST. Hierbei kann man eine Symboltabelle laden, die von einen anderen Programm erzeugt wurde, z.B. eine Tabelle von Betriebssystemroutinen.

.END

Dieser Pseudoop beendet ein Quellprogramm und ist optional, also nicht erforderlich. .END veranlaßt ein .GTB am Ende von Pass zwei; weiterer BASIC-Text danach würde also abgearbeitet. Sie könnten also mit SYS Ihr gerade assembliertes Maschinenprogramm aufrufen.

.OPT

Der .OPT - Pseudoop steht für OPTion und gibt Ihnen die Entscheidung über das Assemblerlisting und den Objektcode. Die Syntax ist folgende:

.OPT option,option,option ...

Folgende Optionen stehen zur Verfügung:

P - Print. Damit können Sie bestimmen, ob ein Listing erzeugt werden soll und wo es hingehen soll. Geben Sie nur P ein, so geht das Listing auf den Bildschirm. Wollen Sie das Listing auf den Drucker schicken, so müssen Sie vorher mit OPEN von BASIC aus eine Datei auf den Drucker öffnen, z.B. OPEN 1,4. Dann geht mit .OPT P1 das Listing zu File 1, also zum Drucker.

O - Object bedeutet Objektcode-Ausgabe. Damit können Sie bestimmen, was mit dem erzeugten Machinenkode geschehen soll. Mit .OPT OO geht er direkt in den Speicher, so wie bei unserem Assembler. Mit .OPT O1 geht er jedoch auf die logische Datei 1. Wenn Sie z.B. vorher eine Programmdatei auf Diskette mit OPEN 1,8,1,"0:PROGRAMM" geöffnet haben, so wird ihr erzeugtes Maschinenprogramm direkt auf Diskette geschrieben und kann dann später mit LOAD geladen werden.

Das folgende Beispielprogramm schreibt den Inhalt der Zeropage ab Zeile LINE auf den Bildschirm. Es zeigt den generellen Aufruf und Umgang mit dem Assembler.

```
10 SYS 32768 : REM PROFI-ASS 64 2.0 AUFRUFEN
20 .OPT P,OO ; LISTING AUF BILDSCHIRM, OBJECTCODE IN SPEICHER
30 *= $C000 ; PROGRAMMSTARTADRESSE
40 LINE = 10 ; BILDSCHIRMZEILE
50 VIDEO = $400 ; ADRESSE DES BILDSCHIRMSPEICHERS
60 COLOR = $0800 ; ADRESSE DES FARBRAMS
70 FARBE = 1 ; WEISSE SCHRIFT
80 LDX #0
90 SCHLEIFE LDA 0,X ; BYTE AUS ZEROPAGE HOLEN
100 STA VIDEO+(40*LINE),X ; ZEICHEN AUSGEBEN
110 LDA #FARBE
120 STA COLOR+(40*LINE),X ; FARBE SETZEN
130 INX
140 BNE SCHLEIFE
150 RTS
160 .END
```

Im folgenden Beispiel wird der Objectcode direkt auf Diskette und das Listing zum Drucker geschickt. Das Quellprogramm besteht aus mehreren einzelnen Programmen.

```
10 OPEN 1,8,1, "0:OBJECTCODE"
20 OPEN 2,4 : REM DRUCKER
30 SYS 32768
40 .OPT 01,P2
50 ; ASSEMBLERBEFEHLE
...
1000 .FILE 8, "PROGRAMM 2"

Programm 2 enhält

10 ; WEITERE BEFEHLE
...
1000 .FILE 8, "PROGRAMM 3"

Programm 3 enhält

10 ; WEITERE BEFEHLE
...
1000 .END 8, "PROGRAMM 1"
```

wobei Programm 1 das Programm ist, welches das SYS 32768 enthält.

PROFI-ASS 64 2.0 hat eine Reihe von Fehlermeldungen, die im Klartext vor der fehlerhaften Zeile ausgegeben werden.

Auf den letzten Seiten haben wir einen professionellen Assembler für den Commodore 64 kennengelernt. PROFI-ASS 64 besteht als Paket PROFIMAT zusätzlich noch aus einem komfortablen Monitorprogramm PROFI-MON 64, das wir Ihnen hier vorstellen wollen.

Was ein Monitorprogramm ist, haben wir bei der Eingabe von Maschinenprogrammen bereits kurz erläutert. Es dient in erster Linie dazu, Speicher- und Registerinhalte anzuzeigen und am Bilschirm zu ändern. Weiterhin ist das Abspeichern sowie das Laden von Maschinenprogrammen möglich. Außerdem können Sie damit Maschinenprogramme ausführen. Schließen Sie diese mit einem BRK-Befehl ab, so wird anschließend wieder in das Monitorprogramm gesprungen und die Inhalte der Register werden angezeigt. Neben diesen Grundfunktionen, die jeder Monitor beinhaltet, bietet PROFI-MON 64 noch eine Reihe weiterer komfortabler Befehle, die das Arbeiten in Maschinensprache sowie das Austesten von Maschinenprogrammen sehr erleichtern können.

Wir erläutern Ihnen nun hier die Befehle, die Ihnen PROFI-MON 64 zur Verfügung stellt, und sagen Ihnen, wie man diese Befehle erfolgreich einsetzt.

PROFI-MON64wirdmitLOAD"PROFI-MON64",8,1geladenundmit SYS 49152 gestartet. Der Monitor meldet sich nun mit

```
C*
    PC   IRQ   SR AC XR YR SP  NV-BDIZC
>; 0003  EA31  32 34 02 A2 F8  00110010
```

Dabei bedeutet das 'C' einen Aufruf des Monitors durch 'Call', also den normalen Aufruf. Danach finden Sie nun die Anzeige der Registerinhalte, wie Sie sie auch in ähnlicher Form von unserem Einzelschrittsimulator kennen. Die Bezeichnungen haben die gleiche Bedeutung wie dort. Die Bezeichnung IRQ bedeutet Interruptvektor; das ist die Adresse, an die der Prozessor verzweigt, wenn ein laufendes Programm durch einen Impuls auf der Interruptleitung

unterbrochen wird. Sofern wir jedoch keine Interruptroutinen benutzen, brauchen wir uns um diesen Wert nicht zu kümmern.

Sie können nun die Registerinhalte einfach dadurch ändern, indem Sie mit dem Cursor über den alten Registerinhalt gehen, diesen mit dem neuen Wert überschreiben und durch Drücken der RETURN-Taste das ganze bestätigen. Wollen Sie die Flags ändern, so geschieht dies mittels des Statusregisters. Dadurch ändert sich dann automatisch auch die Anzeige der einzelnen Flags. Sie können die Registerinhalte jederzeit durch Eingabe von

```
>R
```

und Drücken der RETURN-Taste anzeigen.

Der nächste wichtige Befehl dient zur Anzeige und zum Ändern von Speicherinhalten. Dazu geben Sie 'M', gefolgt von der ersten und der letzten Adresse, die Sie ansehen wollen. Die Start- und die Endadresse müssen Sie als 4stellige Hexadezimalziffer eingeben, z.B.

```
>M A0A0 A0AF
>: A0A0  C4 46 4F D2 4E 45 58 D4 DFORNEXT
>: A0A8  44 41 54 C1 49 4E 50 55 DATAINPU
```

Das Zeichen '>' wird übrigens von PROFI-MON 64 selbst ausgegeben und sagt Ihnen, daß der Monitor zur Eingabe neuer Befehle bereit ist. Die Ausgabe dieses Befehls ist folgendermaßen zu interpretieren:

Nach dem Doppelpunkt wird eine Adresse ausgegeben. Diese Adresse ist die Adresse des ersten der nachfolgenden 8 Bytes. In unserem Beispiel enthält $A0A0 also den Wert $C4. Die Adresse $A0A1 enthält $46 usw. Insgesamt werden 8 Bytes dargestellt. Was nun hinter diesen 8 Bytes folgt, ist die ASCII-Darstellung dieser Bytes. Sie wird auf dem Bildschirm invertiert ausgegeben. Kann also ein Byte als druckbares Zeichen interpretiert werden, so finden Sie dies auf der

rechten Seiten der Ausgabe. Dies ist z.B. nützlich, wenn Sie Texte im Speicher finden wollen. Handelt es sich bei den Bytes jedoch um nicht druckbare Zeichen, so wird stattdessen ein Punkt ausgegeben.

Zum Ändern einzelner Bytes gehen Sie einfach mit dem Cursor auf das Byte, überschreiben mit dem neuen Wert und drücken RETURN. Die ASCII-Darstellung wird dabei automatisch mitgeändert. Die Sache funktioniert also analog wie bei den Registern.

Wollen Sie nun ein Programm starten, so dient dazu der Befehl G. Soll das Programm ab Adresse $CF20 gestartet werden, so geben Sie dazu

```
>G CF20
```

ein. Die Programmausführung wird dabei an dieser Adresse aufgenommen. Vorher werden die Prozessorregister mit den Werten versorgt, die über R angezeigt werden können. Als letzten Befehl in Ihrem Maschinenprogramm sollten Sie BRK benutzen, da das Programm dadurch wieder in den Monitor verzweigt. Dabei werden automatisch wieder die Registerinhalte angezeigt, z.B.

```
B*
    PC   IRQ   SR AC XR YR SP  NV-BDIZC
>; CF39  EA31  B3 8F 73 B0 F6  10110011
```

Dabei bedeutet das B, daß Ihr Programm auf einen BRK-Befehl gelaufen ist in den Monitor gesprungen ist. Der Programmzähler zeigt dabei auf ein Byte hinter den BRK-Befehl. Wenn Sie mehrere BRK-Befehle in Ihren Programmen haben, wissen Sie so genau, an welcher Stelle Ihr Programm unterbrochen wurde. Daraus kann man folgende Methode zur Austestung von Programmen entwickeln: Man setzt an allen strategisch wichtigen Stellen des Programms einen BRK-Befehl ein, so daß das Program an diesen Stellen unterbrochen wird. Nun kann man die Registerinhalte sowie eventuelle Daten im

Speicher kontrollieren. Ist bis dahin das Programm richtig gelaufen, so ersetzt man den BRK-Befehl wieder durch den ursprünglichen Befehl und setzt an die nächste kritische Stelle einen BRK-Befehl. Dadurch kann man das Programm schrittweise testen bis es zur Zufriedenheit läuft.

Zum Laden und Abspeichern von Maschinenprogrammen dienen die Befehle L und S. Dabei ist folgende Syntax zu beachten:

```
>L "NAME",XX
```

Dabei folgt zuerst der Name des Programms in Anführungsstrichen und danach durch Komma getrennt die Geräteadresse als 2stellige Hexzahl. Wollen Sie z.B. das Programm GRAFIK von Diskette laden, so sieht der Befehl dazu so aus:

```
>L "GRAFIK",08
```

Wollen Sie von Kassette laden, so benutzen Sie als Geräteadresse 01. PROFI-MON 64 bietet jedoch noch die Möglichkeit, an Programm an eine beliebige Adresse in den Speicher zu laden. Bei dem obigen Befehl wird das Programm an die Adresse geladen, von der es auch abgespeichert wurde. Geben Sie beim Laden jedoch zusätzlich eine Startadresse ein, so können wird die urprüngliche Adresse nicht benutzt, sondern das Programm wird ab der Adresse geladen, die Sie als Startadresse angeben. Wollen das obigen Programm z.B. ab Adresse $1000 laden, sieht der Befehl dann so aus:

```
>L "GRAFIK",08,1000
```

Der Save-Befehl funktioniert analog. Da der Rechner wissen muß, welchen Speicherbereich er abspeichern muß, ist hier die Angabe einer Start- und einer Endadresse erforderlich. Dabei muß die Endadresse eins größer sein als das letzte Byte, das wir abspeichern wollen. Der Befehl

```
>S "PROGRAMM",08,7000,8000
```

schreibt also den Speicherbereich von $7000 bis $7FFF einschließlich unter dem Namen auf Diskette ab. Auch hier können Sie wieder durch Angabe der Geräteadresse 01 auf die Datasette abspeichern.

Eine weitere Funktion die PROFI-MON 64 bietet, ist der eingebaute Disassembler. Durch Eingabe des Befehls D und eines Adressbereichs können Sie Maschinenprogramme in disassemblierter Form auf dem Bildschirm anzeigen. Dabei ist das Format analog unserem Assembler in BASIC. Geben Sie folgenden Befehl ein,

```
>D B824 B82C
```

so erhalten Sie folgende Ausgabe:

```
>, B824   20 EB B7   JSR $B7EB
>, B827   8A         TXA
>, B828   A0 00      LDY #$00
>, B82A   91 14      STA ($14),Y
>, B82C   60         RTS
```

Das ist übrigens der POKE-Befehl des BASIC-Interpreters.

Mit dem nächsten Befehl können Sie zwei Speicherbereiche vergleichen. Dabei werden Adressen, die nicht den gleichen Inhalt haben, am Bildschirm angezeigt. Der Befehl dazu heißt C. Dann folgen Start-und Endadresse des ersten Bereichs sowie anschließend die Startadresse des Bereichs, mit dem Sie vergleichen wollen, z.B.

```
>C 8000 8100 9000
```

Als Ausgabe erhalten Sie z.B.

```
9056
```

Es wurde der Speicherbereich von $8000 bis $8100 einschließlich mit dem Bereich $9000 bis $9100 verglichen. Dabei wurde festgestellt, daß der Inhalt der Adresse $9056 nicht mit dem der Adresse $8056 übereinstimmt.

Ein weiterer nützlicher Befehl von PROFI-ASS 64 erlaubt es, beliebige Speicherbereiche an eine anderen Stelle zu kopieren. Auch hier geben Sie wieder Start- und Endadresse des zu kopierenden Bereichs an sowie die Anfangsadresse des Zielbereichs. Ähnlich wie bei den Transferbefehlen des Prozessors bleibt der Inhalt des ursprünglichen Bereiches dabei erhalten. Mit

```
>T 6000 6FFF 3000
```

wird der Bereich von $6000 bis $6FFF einschließlich an die Adressen $3000 bis $3FFF kopiert.

Sehr interessant ist auch der Befehl zum Durchsuchen von Speicherbereichen. Damit ist es möglich, eine bestimmte Bytefolge innerhalb eines Speicherbereich zu suchen. Als Ergebnis werden die Adressen angezeigt, ab denen die Bytefolge gefunden wurde. Mit dem Befehl

```
>H E000 EFFF 20 D2 FF
```

wird z.B. der Speicherbereich von $E000 bis $FFFF nach der Bytefolge $20, $D2, $FF durchsucht, das ist der Aufruf der Ausgaberoutine JSR $FFD2. Als Ergebnis werden alle Adressen angezeigt, an denen dies gefunden wurde.

Diesen Befehl gibt es jedoch noch in einer alternativen Form. Dabei können Sie den Speicher direkt nach Texten durchsuchen. Anstelle einer Folge von Bytes geben Sie einfach in Anführungstrichen den Text an, nach dem Sie suchen wollen.

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

Als Ergebnis erhalten Sie z.B.

```
A378
```

Das Wort 'READY' steht also im Speicher ab Adresse $A378.

Der nächste Befehl dient zum Füllen von Speicherbereichen. Sie können dadurch einen Speicherbereich mit konstanten Werten füllen, um z.B. für ein Programm definierte Ausgangsbedingungen zu schaffen. Mit

```
>F 8000 8FFF 00
```

Wird z.B. der Bereich von $8000 bis $8FFF mit Nullbytes gefüllt.

Der letzte Befehle dient schließlich zum Verlassen des Monitorprogramms und bringt Sie wieder zum BASIC-Interpreter. Geben Sie dazu einfach

```
>X
```

ein, und der Interpreter meldet sich mit READY wieder. Wollen Sie später nocheinmal den Monitor benutzen, so verwenden Sie wieder den Befehl

```
SYS 49152
```

der Sie wieder in das Monitorprogramm zurück bringt.

Der Monitor des Commodore 128

Anwender des Commodore 128 haben gegenüber dem Commodore 64 den Vorteil, daß ihr Rechner bereits über einen eingebauten Maschinensprachemonitor verfügt. Dieser Monitor hat etwa den Befehlsumfang des PROFI-MON 64 und ist in der Anwendung nahezu identisch.

Es fehlen jedoch die Möglichkeiten des Einzelschrittverarbeitung. Um dem erweiterten Speicherbereich des Commodore 128 gerecht zu werden, können Adressen nicht nur als vier-, sondern auch als 5stellige Hexzahlen angegeben werden. Dabei gibt die oberste Hexziffer die sogenannte Speicherbank an. Von den möglichen 16 Werten sind drei Banks mit Speicher bestückt.

Bank 0 und 1 enthalten jeweils 64 K RAM, während das Betriebssystem im ROM als Bank 15 angesprochen wird. Der I/O-Bereich sowie der Charactergenerator befinden sich ebenfalls in dieser Bank.

Der Monitor wird von BASIC aus durch den Befehl MONITOR aufgerufen und läßt sich dann ähnlich wie PROFI-MON handhaben. Die Zahleneingabe kann hexadezimal (Defaultwert oder '$'), dezimal mit '+', oktal '&' (im Achtersystem, wenig gebräuchlich) sowie binär mit '%' erfolgen. Die Ausgabe erfolgt grundsätzlich hexadezimal, wobei Adressen immer fünfstellig erscheinen mit der Banknummer in der höchstwertigen Stelle.

Vom Monitor aus lassen sich auch Befehle an die Floppy schicken. Ebenso ist es möglich, den Fehlerkanal der Floppy zu lesen.

Gegenüber der 64er haben Sie den Vorteil, daß Ihnen der Monitor ständig zur Verfügung steht und daß er keinen RAM-Speicherplatz belegt.

12. Umrechnungs- und Befehlstabellen

Adressierungsarten und Befehlskodes

*	A	#	ZP	AB	ABX	ABY	ZPX	ZPY	,X)	),Y
ADC	-	69	65	6D	7D	79	75	-	61	71
AND	-	29	25	2D	3D	39	35	-	21	31
ASL	0A	-	06	0E	1E	-	16	-	-	-
BIT	-	-	24	2C	-	-	-	-	-	-
CMP	-	C9	C5	CD	DD	D9	D5	-	C1	D1
CPX	-	E0	E4	EC	-	-	-	-	-	-
CPY	-	C0	C4	CC	-	-	-	-	-	-
DEC	-	-	C6	CE	DE	-	D6	-	-	-
EOR	-	49	45	4D	5D	59	55	-	41	51
INC	-	-	E6	EE	FE	-	F6	-	-	-
LDA	-	A9	A5	AD	BD	B9	B5	-	A1	B1
LDX	-	A2	A6	AE	-	BE	-	B6	-	-
LDY	-	A0	A4	AC	BC	-	B4	-	-	-
LSR	4A	-	46	4E	5E	-	56	-	-	-
ORA	-	09	05	0D	1D	19	15	-	01	11
ROL	2A	-	26	2E	3E	-	36	-	-	-
ROR	6A	-	66	6E	7E	-	76	-	-	-
SBC	-	E9	E5	ED	FD	F9	F5	-	E1	F1
STA	-	-	85	8D	9D	99	95	-	81	91
STX	-	-	86	8E	-	-	-	96	-	-
STY	-	-	84	8C	-	-	94	-	-	-

Branch-	BPL	BMI	BVC	BVS	BCC	BCS	BNE	BEQ
Befehle	10	30	50	70	90	B0	D0	F0
Transfer-	TXA	TAX	TYA	TAY	TSX	TXS		
Befehle	8A	AA	98	A8	BA	9A		
Stack-	PHP	PLP	PHA	PLA				
Befehle	08	28	48	68				
Sprung-	BRK	JSR	RTI	RTS	JMP	JMP	NOP	
Befehle	00	20	40	60	4C	6C	EA	
Flag-	CLC	SEC	CLI	SEI	CLV	CLD	SED	
Befehle	18	38	58	78	B8	D8	F8	
Zähl-	DEY	INY	DEX	INX				
Befehle	88	C8	CA	E8				

Umrechnungstabelle Dezimal - Hexadezimal - Binär

dezimal	hex	binär	dezimal	hex	binär
0	00	00000000	26	1A	00011010
1	01	00000001	27	1B	00011011
2	02	00000010	28	1C	00011100
3	03	00000011	29	1D	00011101
4	04	00000100	30	1E	00011110
5	05	00000101	31	1F	00011111
6	06	00000110	32	20	00100000
7	07	00000111	33	21	00100001
8	08	00001000	34	22	00100010
9	09	00001001	35	23	00100011
10	0A	00001010	36	24	00100100
11	0B	00001011	37	25	00100101
12	0C	00001100	38	26	00100110
13	0D	00001101	39	27	00100111
14	0E	00001110	40	28	00101000
15	0F	00001111	41	29	00101001
16	10	00010000	42	2A	00101010
17	11	00010001	43	2B	00101011
18	12	00010010	44	2C	00101100
19	13	00010011	45	2D	00101101
20	14	00010100	46	2E	00101110
21	15	00010101	47	2F	00101111
22	16	00010110	48	30	00110000
23	17	00010111	49	31	00110001
24	18	00011000	50	32	00110010
25	19	00011001	51	33	00110011

dezimal	hex	binär	dezimal	hex	binär
52	34	00110100	78	4E	01001110
53	35	00110101	79	4F	01001111
54	36	00110110	80	50	01010000
55	37	00110111	81	51	01010001
56	38	00111000	82	52	01010010
57	39	00111001	83	53	01010011
58	3A	00111010	84	54	01010100
59	3B	00111011	85	55	01010101
60	3C	00111100	86	56	01010110
61	3D	00111101	87	57	01010111
62	3E	00111110	88	58	01011000
63	3F	00111111	89	59	01011001
64	40	01000000	90	5A	01011010
65	41	01000001	91	5B	01011011
66	42	01000010	92	5C	01011100
67	43	01000011	93	5D	01011101
68	44	01000100	94	5E	01011110
69	45	01000101	95	5F	01011111
70	46	01000110	96	60	01100000
71	47	01000111	97	61	01100001
72	48	01001000	98	62	01100010
73	49	01001001	99	63	01100011
74	4A	01001010	100	64	01100100
75	4B	01001011	101	65	01100101
76	4C	01001100	102	66	01100110
77	4D	01001101	103	67	01100111

dezimal	hex	binär	dezimal	hex	binär
104	68	01101000	130	82	10000010
105	69	01101001	131	83	10000011
106	6A	01101010	132	84	10000100
107	6B	01101011	133	85	10000101
108	6C	01101100	134	86	10000110
109	6D	01101101	135	87	10000111
110	6E	01101110	136	88	10001000
111	6F	01101111	137	89	10001001
112	70	01110000	138	8A	10001010
113	71	01110001	139	8B	10001011
114	72	01110010	140	8C	10001100
115	73	01110011	141	80	10001101
116	74	01110100	142	8E	10001110
117	75	01110101	143	8F	10001111
118	76	01110110	144	90	10010000
119	77	01110111	145	91	10010001
120	78	01111000	146	92	10010010
121	79	01111001	147	93	10010011
122	7A	01111010	148	94	10010100
123	7B	01111011	149	95	10010101
124	7C	01111100	150	96	10010110
125	7D	01111101	151	97	10010111
126	7E	01111110	152	98	10011000
127	7F	01111111	153	99	10011001
128	80	10000000	154	9A	10011010
129	81	10000001	155	9B	10011011

dezimal	hex	binär	dezimal	hex	binär
156	9C	10011100	182	B6	10110110
157	9D	10011101	183	B7	10110111
158	9E	10011110	184	B8	10111000
159	9F	10011111	185	B9	10111001
160	A0	10100000	186	BA	10111010
161	A1	10100001	187	BB	10111011
162	A2	10100010	188	BC	10111100
163	A3	10100011	189	BD	10111101
164	A4	10100100	190	BE	10111110
165	A5	10100101	191	BF	10111111
166	A6	10100110	192	C0	11000000
167	A7	10100111	193	C1	11000001
168	A8	10101000	194	C2	11000010
169	A9	10101001	195	C3	11000011
170	AA	10101010	196	C4	11000100
171	AB	10101011	197	C5	11000101
172	AC	10101100	198	C6	11000110
173	AD	10101101	199	C7	11000111
174	AE	10101110	200	C8	11001000
175	AF	10101111	201	C9	11001001
176	B0	10110000	202	CA	11001010
177	B1	10110001	203	CB	11001011
178	B2	10110010	204	CC	11001100
179	B3	10110011	205	CD	11001101
180	B4	10110100	206	CE	11001110
181	B5	10110101	207	CF	11001111

dezimal	hex	binär	dezimal	hex	binär
208	D0	11010000	234	EA	11101010
209	D1	11010001	235	EB	11101011
210	D2	11010010	236	EC	11101100
211	D3	11010011	237	ED	11101101
212	D4	11010100	238	EE	11101110
213	D5	11010101	239	EF	11101111
214	D6	11010110	240	F0	11110000
215	D7	11010111	241	F1	11110001
216	D8	11011000	242	F2	11110010
217	D9	11011001	243	F3	11110011
218	DA	11011010	244	F4	11110100
219	DB	11011011	245	F5	11110101
220	DC	11011100	246	F6	11110110
221	DD	11011101	247	F7	11110111
222	DE	11011110	248	F8	11111000
223	DF	11011111	249	F9	11111001
224	E0	11100000	250	FA	11111010
225	E1	11100001	251	FB	11111011
226	E2	11100010	252	FC	11111100
227	E3	11100011	253	FD	11111101
228	E4	11100100	254	FE	11111110
229	E5	11100101	255	FF	11111111
230	E6	11100110			
231	E7	11100111			
232	E8	11101000			
233	E9	11101001			

Bitmuster und Flag-Beeinflussung aller 6510-Befehle sowie Seitenverweise

*	Bitmuster	N	V	B	D	I	Z	C	Seite
ADC	011XXX01	X	X				X	X	28
AND	001XXX01	X					X		33
ASL	000XXX10	X					X	X	50
BCC	10010000								41
BCS	10110000								41
BEQ	11110000								41
BIT	0010X100	M	M				X		37
BMI	00110000								41
BNE	10010000								41
BPL	00010000								41
BRK	00000000			1		1			57
BVC	01010000								41
BVS	01110000								41
CLC	00011000							0	49
CLD	11011000				0				49
CLI	01011000					0			49
CLV	10111000		0						49
CMP	110XXX01	X					X	X	38
CPX	1110XX00	X					X	X	40
CPY	1100XX00	X					X	X	41
DEC	110XX110	X					X		48
DEX	11001010	X					X		47
DEY	10001000	X					X		48
EOR	010XXX01	X					X		36
INC	000XX110	X					X		47
INX	11101000	X					X		46
INY	11001000	X					X		47
JMP	01X01100								45
JSR	00100000								54
LDA	101XXX01	X					X		18
LDX	101XXX10	X					X		19
LDY	101XXX00	X					X		19
LSR	010XXX10	0					X	X	51
NOP	11101010								58

*	Bitmuster	N	V	B	D	I	Z	C	Seite
ORA	000XXX01	X					X		35
PHA	01001000								56
PHP	00001000								56
PLA	01101000	X					X		56
PLP	00101000	X	X	X	X	X	X	X	56
ROL	001XXX10	X					X	X	52
ROR	011XXX10	X					X	X	52
RTI	01000000	X	X	X	X	X	X	X	57
RTS	01100000								55
SBC	111XXX01	X	X				X	X	31
SEC	00111000							1	49
SED	11111000				1				49
SEI	01111000					1			49
STA	100XXX01								25
STX	100XX110								25
STY	100XX100								25
TAX	10101010	X					X		26
TAY	10101000	X					X		27
TSX	10111010	X					X		27
TXA	10001010	X					X		27
TXS	10011010								27
TYA	10011000	X					X		27

Stehen beim Bitmuster eines Befehls an einer oder mehreren Stellen ein 'X', so sind diese Bits von der Adressierungsart abhängig. Bei den Flags bedeutet ein X, daß dieses Flag von dem Befehl beeinflußt wird. 0 oder 1 heißt daß dieses Flag gelöscht oder gesetzt wird. Steht unter einem Flag kein Eintrag, so wird das Flag von diesem Befehl nicht beeinflußt.

Tabelle der 6510 Befehlskodes

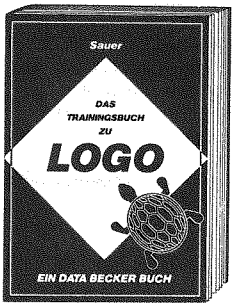
	0	1	2	4	5	6	8	9	A	C	D	E
0	BRK	ORA ,X)			ORA ZP	ASL ZP	PHP	ORA #	ASL A		ORA AB	ASL AB
1	BPL	ORA),Y			ORA ZPX	ASL ZPX	CLC	ORA ABY			ORA ABX	ASL ABX
2	JSR	AND ,X)		BIT ZP	AND ZP	ROL ZP	PLP	AND #	ROL A	BIT AB	AND AB	ROL AB
3	BMI	AND),Y			AND ZPX	ROL ZPX	SEC	AND ABY			AND ABX	ROL ABX
4	RTI	EOR ,X)			EOR ZP	LSR ZP	PHA	EOR #	LSR A	JMP AB	EOR AB	LSR AB
5	BVC	EOR),Y			EOR ZPX	LSR ZPX	CLI	EOR ABY			EOR ABX	LSR ABX
6	RTS	ADC ,X)			ADC ZP	ROR ZP	PLA	ADC #	ROR A	JMP IND	ADC AB	ROR AB
7	BVS	ADC),Y			ADC ZPX	ROR ZPX	SEI	ADC ABY			ADC ABX	ROR ABX
8		STA ,X)		STY ZP	STA ZP	STX ZP	DEY		TXA	STY AB	STA AB	STX AB
9	BCC	STA),Y		STY ZPX	STA ZPX	STX ZPY	TYA	STA ABY	TXS		STA ABX	
A	LDY #	LDA ,X)	LDX #	LDY ZP	LDA ZP	LDX ZP	TAY	LDA #	TAX	LDY AB	LDA AB	LDX AB
B	BCS	LDA),Y		LDY ZPX	LDA ZPX	LDX ZPY	CLV	LDA ABY	TSX	LDY ABX	LDA ABX	LDX ABY
C	CPY #	CMP ,X)		CPY ZP	CMP ZP	DEC ZP	INY	CMP #	DEX	CPY AB	CMP AB	DEC AB
D	BNE	CMP),Y			CMP ZPX	DEC ZPX	CLD	CMP ABY			CMP ABX	DEC ABX
E	CPX #	SBC ,X)		CPX ZP	SBC ZP	INC ZP	INX	SBC #	NOP	CPX AB	SBC AB	DEC AB
F	BEQ	SBC),Y			SBC ZPX	INC ZPX	SED	SBC ABY			SBC ABX	INC ABX

Die Befehlskodes lassen sich aus der Tabelle wie folgt ermitteln: Die Zeile ergibt das Hi-Nibble des Kodes während das Lo-Nibble aus der Spalte ermittelt wird. Beispiel: Befehlskode $49 entspricht EOR #.
Die Abkürzungen der Adressierungsarten haben dabei folgende Bedeutung:

A	Akkumulator
AB	absolut
ABX	absolut, X-indiziert
ABY	absolut, Y-indiziert
IND	indirekt
ZP	Zeropage
ZPX	Zeropage, X-indiziert
ZPY	Zeropage, Y-indiziert
#	unmittelbar
,X)	X-indiziert, indirekt
),Y	indirekt, Y-indiziert

Was liegt näher, als den Commodore 64 auch für schulische Zwecke einzusetzen? Hilfestellung in jeder Beziehung bietet dieses Schulbuch, dessen Stoff von erfahrenen Pädagogen didaktisch aufbereitet und strukturiert wurde. So lernt man nicht nur die Computeranwendung in den Fächern Mathematik, Physik, Chemie, Biologie, Fremdsprachen und Geographie, sondern es bleibt auch einiges Wissen über Elektronik und Informatik hängen.

Voß
Das Schulbuch zum Commodore 64
300 Seiten, DM 49,-
ISBN 3-89011-019-3



LOGO - eine bemerkenswerte Sprache nicht nur für Kinder sondern für viele Bereiche. Diese tiefgreifende Einführung bietet Ihnen sinnvolles Erlernen und Training der vielen Möglichkeiten, die LOGO bietet. Aus dem Inhalt: Grafikprogrammierung, Wörter- und Listenverarbeitung, Funktionsplotter, Maskengenerator, 3-D-Grafik, Prozeduren, Rekursion, Sprites und Musik, und vieles mehr.

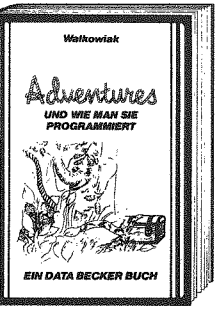
Sauer
Das Trainingsbuch zu LOGO
230 Selten, DM 39,-
ISBN 3-89011-044-4

lier kommt das Allround-alent des C64 voll zum Luge, mit pfiffigen Pro-rammen zum Nutzen nd Lernen: Gedichte om Computer, Einla-ung zur Party, Werbe-riefe, Autokostenbe-echnung, Rezeptkartei, Gesundheitsarchiv, Han-arbeitshilfen und noch nehr. Viele Anregungen, eichtverständlich und pannend geschrieben. ür jeden 64er-Anwender nbedingt empfehlens-vert!

Bartel
Das Ideenbuch zum Commodore 64
Rezepte für kreatives Computern
243 Seiten, DM 29,-
ISBN 3-89011-027-4

Mehr als 50 interessante Anwendungsprogramme aus allen Bereichen, zusammengestellt von 50 erfolgreichen Autoren. Vielfalt und Abwechslung sind garantiert! Spiele, Graphik & Sound, Mathematik, Hilfsprogramme und auch größere Anwendungsprogramme. Dazu wertvolle Tips & Tricks zum Seibermachen. Da ist mit Sicherheit für jeden etwas dabei!

DATA BECKERs große 64er Programmsammlung
252 Seiten, DM 49,-
ISBN 3-89011-014-2



Walkowiak
Adventures - und wie man sie programmiert
225 Seiten, DM 39,-
ISBN 3-89011-043-6

Ein faszinierender Führer in die phantastische Welt der Abenteuerspiele. Hier läßt sich ein arrivierter Autor in die Karten gucken: er zeigt, wie Adventures funktionieren, wie man sie erfolgreich spielt und wie man eigene Adventures programmiert. Der Clou des Buches ist neben fertigen Adventures zum Abtippen ein kompletter ADVENTURE-GENERATOR, mit dem das Selbsterstellen packender Adventures zum Kinderspiel wird. Achtung: dieses buch macht süchtig!

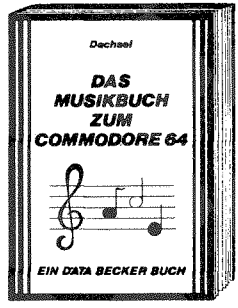
Sie wollten schcon immer mal ein Telespiel selbst programmieren? Hier ist für Sie das top-Buch, zugeschnitten auf den Commodore 64 und mit Berücksichtigung des Commodore 128! Schrittweise lernen Sie zu programmieren, wie man Pac Man durchs Labyrinth schleust oder wie Captain Future spannende Abenteuer in fremden Galaxien überlebt. Handfeste Anwendungen mit vielen Beispielen, Listings und Programmirtips. Auch mit wenig Programmir-Praxis stellen sich schnell überraschende Erfolge ein.

Linden
Superspiele - selbst gemacht
ca. 200 Seiten, DM 29,-
erscheint Mai 1985
ISBN 3-89011-087-8

Der Bestseller zur Grafikprogrammierung des C 64 vom Autor der berühmten Supergrafik. Für Einsteiger, Fortgeschrittene und Profis. Bringt alles über Sprites, High-Res-Grafik und Multicolor bis hin zu 3-D und CAD. Unzählige Superprogramme und Routinen zum Abtippen. Grafik ist zwar eine der großen Stärken des C 64, allerdings bleibt der Zugriff darauf mit BASIC gerade für den Anfänger ohne Anleitung wohl ein Wunschtraum. Mit diesem Buch nicht mehr!

Plenge
Das Grafikbuch zum Commodore 64
295 Seiten, DM 39,-
ISBN3-89011-011-8

Nutzen Sie die Klangmöglichkeiten des C 64! Neben einer kurzen Einführung in die Computermusik finden Sie hier Informationen zu Soundregistern, ADSR-Programmierung, Synchronisation und Ringmodulation. Zahlreiche Beispiele für Sound- und Songprogrammierung sowie wichtige Routinen runden den Inhalt ab. Nicht vergessen wurden auch Themen wie beispielsweise der Anschluß an eine Stereoanlage oder die Verarbeitung externer Tonsignale. Also, Komponisten, ans Werk!



Dachsel
Das Musikbuch zum Commodore 64
208 Seiten, DM 39,-
ISBN 3-89011-012-6

zczepanowski/Plenge
as Trainingsbuch zum
imons Basic
80 Seiten, DM 49,-
SBN 3-89011-009-6

Wer den großen Programmierkomfort, den SIMONs BASIC bietet, voll nutzen möchte, der muß mit den einzelnen Befehlen richtig umgehen können. Da aber das nicht gerade umfangreiche Handbuch sowie auch die Problempunkte, die dieses BASIC aufweist, dem Anwender einige Stolpersteine in den Weg legen, ist das Trainingsbuch ein "Muß" für jeden, der den optimalen Weg zu ausgesprochen leistungsfähigen Programmen gehen will.

Schäfer
Das Handbuch zur DFÜ Datenfernübertragung mit dem Commodore 64
173 Seiten, DM 39,-
ISBN 3-89011-058-4

Die Datenübertagungswelle rollt! Dieses Handbuch bietet eine praxisorientierte Einführung in die Grundlagen der Datenübertragung (Akustikkoppler, DATEX-P, Mailboxen, Datenbanken). Als Clou gibt es neben einem umfanagreichen Verzeichnis von Telefonbzw. DATEX-P-Nummern und Anschriften der Mailboxen und Datenbanken fix und fertige Mailbox- und Datenübertragungsprogramme, mit denen Sie z. B. Ihren C 64 in eine Mailbox umwandeln können.

Diese umfassende Einführung in die Textverarbeitung, speziell zugeschnitten auf das weitverbreitete Textverarbeitssystem TEXTOMAT für den C 64, behandelt grundsätzliche Probleme ebenso wie konkrete Anwendungsmöglichkeiten. Aus dem Inhalt: Vorteile einer Textverarbeitung - Laden und Installieren von TEXTOMAT - Druckeranpassungen für viele Drucker - Erstellen von Listen Tabellen, Formularen, Statistiken etc. - Kombination mit DATAMAT, Serienbriefe.

Froitzheim
Das Trainingsbuch zu TEXTOMAT
200 Seiten, DM 39,-
ISBN 3-89011-031-2

Schmidt
Das Trainingsbuch zu DATAMAT
317 Seiten, DM 39,-
ISBN 3-89011-035-5

Sicheres Beherrschen und sinnvoller Einsatz des Dateiverwaltungsprogramms DATAMAT ist Ziel dieses Trainingsbuches. So werden Themen wie z. B. Maske herstellen, Datei einrichten und pflegen, suchen, ändern, löschen, Etikettendruck und Datenübergabe an andere Programme genau beschrieben; im zweiten Teil des Buches sind darüber hinaus praktische Beispiele angeführt, zu denen Literatur- und Schallplattendatei ebenso gehören wie Firmenadressen oder Lagerverwaltung.

Angerhausen/Brückmann/Englisch/Gerits
64 Intern
Das große Buch zum Commodore 64 mit dokumentiertem Schaltplan

Die Herausforderung für jeden ernsthaften Anwender! Alles über Technik, Betriebssystem und fortgeschrittene Programmierung des Commodore 64. Mit ausführlichem ROM-Listing, sorgfältig dokumentierten Originalschaltplänen zum Ausklappen, zahlreichen Abbildungen, Schaltbildern, Blockdiagrammen und - natürlich - mit anspruchsvollen Programmen. Mit diesem unentbehrlichen Buch lernen Sie Ihren C 64 erst richtig kennen.

352 Seiten, 2 Schaltpläne, DM 69,-
ISBN 3-89011-000-2

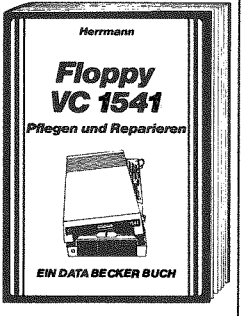
Brückmann/Gerits/Wiens
Das große Druckerbuch
369 Seiten, DM 49,-
ISBN 3-89011-020-7

Mit diesem Buch meistern Sie absolut jedes Drucker-Problem. Ob Sekundäradressen, Schnittstellen, Steuerzeichen, formatierte Datenausgabe oder Grafik-Hardcopy: alles hervorragend erklärt. Selbstverständlich wieder viele nützliche Programme zum Abtippen; außerdem wichtige Hilfen zur Druckeranpassung, ein Betriebssystemlisting des MPS 801 und ein eigenes Kapitel zum VC-1520. Jetzt holen Sie das Optimum aus Ihrem Drucker heraus!

Das Standardwerk zur Floppy VC 1541. Alles über Diskettenprogrammierung vom Einsteiger bis zum Profi. Neben grundlegenden Informationen zum DOS, zu den Systembefehlen und Fehlermeldungen stehen mehrere Kapitel zur praktischen Dateiverwaltung mit der Floppy. Umfangreiches, dokumentiertes DOS-Listing. Dazu eine Fundgrube verschiedenster Programme und Hilfsroutinen, die das Buch für jeden Floppy-Anwender einfach zur Pflichtlektüre machen.

Englisch/Szczepanowski
Das große Floppy-Buch
482 Seiten, DM 49,-
ISBN 3-89011-006-3

Selbsthilfe spart Zeit, Ärger und Geld - gerade Probleme wie Floppy-Justage oder Reparaturen der Platine sind mit oft einfachen Mitteln zu lösen. Anleitungen zur Behebung der meisten Störfälle, Ersatzteillisten und eine Einführung in Mechanik und Elektronik des Laufwerks. Natürlich gehören auch genaue Angaben zu Werkzeug und Arbeitsmaterial zum Buch, das in jeder Beziehung für "effektiv und preiswert" steht.



Hermann
VC-1541 Pflegen und Reparieren
ca. 200 Seiten, DM 49,-
ISBN 3-89011-179-7

Das Superbuch, das Ihnen zeigt, was alles in Ihrem Rekorder steckt. Informiert detailliert und leichtverständlich über Datasette und Cassettenspeicherung. Mit den Spitzenprogrammen Autostart, Catalog (sucht und lädt automatisch!), Backup von und auf Floppy, Save von Speicherbereichen und einem neuen Cassetten-Betriebssystem mit dem 10-20 mal schnelleren (!) Fasttape. Außerdem weitere nützliche Hinweise (Kopfjustage, Kontroll-Lautsprecher) und Programme.

Paulissen
Das Cassettenbuch zum Commodore 64 und VC-20
190 Seiten, DM 29,-
ISBN 3-89011-030-4

Brückmann
Der Commodore 64 und der Rest der Welt
229 Seiten, DM 49,-
ISBN 3-89011-015-0

Literatur speziell für den engagierten Hobbyelektroniker vom fähigen Techniker zusammengestellt. Schwerpunkt sind ausgesuchte Ideen zu verschiedenen Einsatzmöglichkeiten des C 64: Motorsteuerung, A/D-Wandler, Spannungs- und Temperaturmessung und Lichtorgel. Dazu eine Reihe hochinteressanter Schaltungen zum Nachbau: EPROM-Programmer, Sprachsynthesizer, Frequenzzähler und noch mehr.

Einführung in das System, Hardware- und Interfacebeschreibung, Erläuterung des VIC-Chips, des VDC (640 x 200 Grafik auf dem 80-Zeichen-Schirm, 28 Zeilen), SID, detaillierte Beschreibung der Memory-Management-Unit (MMU), ein sehr ausführlich dokumentiertes ROM-Listing. Mit sehr vielen hilfreichen Programmbeispielen. Ein Superbuch, wie alle Titel in der INTERN-Reihe!
128 INTERN, über 500 Seiten, DM 69,–

Falls Sie mit dem Commodore 128 in die CP/M-Welt einsteigen wollen, sind Sie hier richtig. Von grundsätzlichen Erklärungen zu Betriebssystem und Speicherung von Zahlen, Schreibschutz oder ASCII, Schnittstellen und Anwendung von CP/M-Hiltsprogrammen. Für Fortgeschrittene: CP/M und Commodore-Format, Erstellen von Submit-Dateien u.v.m. Nutzen Sie die Möglichkeiten von CP/M!

Das CP/M-Buch zum PC128,
ca. 250 Seiten, DM 49,–

Jetzt gibt es das große Floppybuch auch zur 1571 mit einer Einführung für Einsteiger. Arbeiten mit dem PC-128 und BASIC 7.0, sequentiellen und relativen Dateien. Für Fortgeschrittene: Nutzung der Direktzugriffsbefehle, Programme im DOS, wichtige DOS-Routinen und ihre Anwendung und natürlich ein ausführlich dokumentiertes DOS-Listing. Unentbehrlich zum effektiven Einsatz der 1571!

Das große Floppybuch 1571,
ca. 300 Seiten, DM 49,–

C COMPILER

C-COMPILER
DM 298,-*

"C" ist die kommende Programmiersprache! Bereits heute arbeiten große Softwarehäuser mit ihr - sogar das bekannte Betriebssystem "UNIX" wurde in "C" geschrieben. Jetzt kann auch der C-64-Anwender diese zukunftsweisende Computersprache nutzen, die bisher nur den "Großen" vorbehalten war. Und zwar in vollem Umfang, denn der C-Compiler von DATA BECKER ermöglicht nicht nur einen Einblick in dieses hochinteressante System, sondern bietet eine Möglichkeit zur professionellen Programmerstellung - eine echte Alternative zu anderen Sprachen. Das C-Paket enthält Editor, Compiler, Dienstprogramme sowie ein ausführlich dokumentiertes Handbuch.

Der C-COMPILER in Stichworten:
EDITOR: Full Screen Editor mit variabler Zeichenbreite, maximal 43 K Textspeicher, 2 Zeichensätzen und komfortablen Textverarbeitungsfunktionen.
COMPILER: Voller Sprachumfang nach Kernigan und Ritchie (außer Bitfeldern), erzeugt direkten Maschinencode, hat 50 KByte Speicher für den objektcode verfügbar, optimiert Ausdrücke und bietet 16(!)-stellige Fließkomma-Arithmetik.
LINKER: Bindet bis zu 7 getrennt compilierte Quellfiles zu einem lauffähigen Maschinenprogramm zusammen, das sowohl vom C-Hauptmenue als auch von BASIC aus gestartet werden kann.
COPY: Diskettendienstprogramm.
Mit dem DATA BECKER "C"-COMPILER steht dem C-64-Anwender jetzt ein mächtiges Softwarewerkzeug zur Verfügung. "C"-Programme lassen sich ohne großen Aufwand auf PC's oder selbst auf Großrechner übertragen. Der DATA BECKER "C"-COMPILER wurde komplett in Deutschland von deutschen Autoren entwickelt. Natürlich mit ausführlichem Handbuch. Programm wird auf Diskette (für Diskettenlaufwerk 1541) geliefert.

* unverbindliche Preisempfehlung. Alle Programme auf Diskette für VC-1541.

ADA-Trainingskurs
DM 198,-*

Diese Programmiersprache der Zukunft, wie seinerzeit COBOL vom Pentagon in Auftrag gegeben, kann jetzt mit dem DATA BECKER-Trainingskurs auch der C-64-Anwender erlernen. Der ADA-Trainingskurs enthält außerdem einen Compiler, der einen umfassenden SUBSET und die wesentlichen Elemente dieser Sprache bietet.

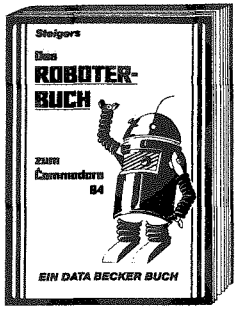
ADA-Trainingskurs in Stichworten:
blockstrukturierte Programme - modularer Aufbau der Programme - ermöglicht die Behandlung von Ausnahmezuständen - lexikalische, syntaktische und semantische Fehlerüberprüfung beim Übersetzen und zur Laufzeit - Nennung der fehlerhaften Zeilennummer führt zu problemloser Fehlersuche - ermöglicht das einfache Einbinden von Maschinenprogrammen - ausgesprochen leichtes Arbeiten mit Programmbibliotheken - Konstanten und Variablendefinitionen von verschiedenen Typen (Integer, Strings etc.) - Assembler erlaubt Kommentare, Benutzung von Labels, verschiedene Zahlenformate, Pseudo-Anweisungen (z.B. ".BYTE", ".MARKE", ".START", ".BLOCK", ".WORD", ".COUNT" etc.) und die Verwendung aller Mnemonics (MOS Standard) - Disassembler ermöglicht die Analyse von Maschinenprogrammen - 62 Schlüsselwörter, trotzdem genug Speicherplatz für eigene Programme vorhanden - Programmdiskette enthält Editor, Übersetzer, Assembler und Disassembler - umfangreiches deutsches Handbuch mit Übungen und Lösungsvorschlägen zu den Themen Textausgabe, Bildschirmsteuerung, Datenobjekte, Dateneinn- und -ausgabe, Wertzuweisung, Entscheidungen und Schleifen - ausführliche Darstellung und Erläuterung der Grammatikregeln - Grammatikindex.

* unverbindliche Preisempfehlung. Alle Programme auf Diskette für VC-1541.

Diese hochkarätige Einführung in die rechnerunterstützte Konstruktion liefert neben umfassenden Informationen reichlich Konstruktionsbeispiele mit etlichen Programmen. Konkret werden dreidimensionale Zeichnungen und deren Veränderung durch Zoomen, Duplizieren, Spiegeln etc. behandelt, Bausteinprinzip und Macros erklärt sowie darüber hinaus der Aufbau eines eigenen CAD-Systems erarbeitet. Ein brandaktuelles Buch der absoluten Spitzenklasse!

Helft
Einführung in CAD mit dem Commodore 64
302 Seiten, DM 49,-
ISBN 3-89011-067-3

Steigers
Das Roboterbuch zum Commodore 64
ca. 230 Seiten, DM 49,-
erscheint April 1985
ISBN 3-89011-86-X



STAR-TRECK im Wohnzimmer? Dieses packende Buch zeigt, wie man sich einen Roboter ohne großen finanziellen Aufwand selbst bauen kann und welche erstaunlichen Möglichkeiten der C 64 zur Programmierung und Steuerung bietet - anschaulich dargestellt mit vielen Abbildungen und etlichen Beispielen. Dazu ein spannender berblick über die historische Entwicklung des Roboters und eine umfassende Einführung in kybernetische Grundlagen. Unentbehrlich für jeden Roboterfan!

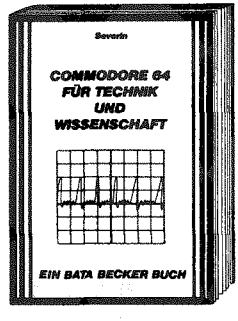
Voß
Einführung in die Künstliche Intelligenz Mit vielen Programmen für den C 64
395 Seiten, DM 49,-
ISBN 3-89011-081-9

Zentrales Thema aktueller Diskussionen: die Künstliche Intelligenz (KI). Eine ausführliche und interessante Einführung in deren Theorie und Einsatzmöglichkeiten, vom historischen Abriß über die "denkenden" und "lebenden" Maschinen bis zu Anwendungsbeispielen mit Programmen für den Commodore 64. Expertensystem, Such- und Auskunftsprogramm oder selbstlernende Programme werden ebenso dargestellt wie Computer-Kunst oder Simulationen.

Sasse
Compiler - verstehen, anwenden, entwickeln
336 Seiten, DM 49,-
ISBN 3-89011-061-4

Zu den wichtigsten Arbeitsmitteln des Programmierers überhaupt gehören Compiler, deren Grundlagen, Funktionen und Einsatzweise in diesem Buch systematisch erklärt werden. Auch die Entwicklung eines eigenen Compilers, lexikalische, syntaktische und semantische Analyse sowie Codegenerierung sind ausführlich beschrieben. Mit vielen nützlichen Programmen, speziell zugeschnitten auf den Commodore 64 - Pflichtlektüre für jeden ernsthaften Programmierer!

Konkurrenzlos! Dieses Buch enthält nicht nur eine umfangreiche Programmsammlung, sondern ist zuglich qualifizierte Standardwerk (inklusive Tips und Tricks!) für die anspruchsvolle wissenschaftliche Nutzung des C 64. Mit Sortier- und Mathematikprogramm, Statistik und weiteren interessanten Programmen für Chemie, Physik, Biologie und Elektronik wird der 64er zur wissenschaftlichen Hilfskraft. Ein breites Spektrum, gut und ausführlich dokumentiert.



Severin
Commodore 64 für Technik und Wissenschaft
296 Seiten, DM 49,-
ISBN 3-89011-021-5

Ein unentbehrliches Arbeitsinstrument für den Commodore-64-Anwender! Fachwissen von A-Z bei allen Fragen zur Computerei im allgemeinen und zum 64er im besonderen. Gleichzeitig ein Fachwörterbuch, natürlich mit deutscher Erklärung der englischen Fachbegriffe. Insgesamt eine unglaubliche Vielfalt an Informationen, die grundsätzliches Verständnis ebenso fördern wie fortgeschrittene Programmierung.

Jordan/Schellenberger
Das DATA BECKER Lexikon zum Commodore 64
354 Seiten, DM 49,-
ISBN 3-89011-0013-4