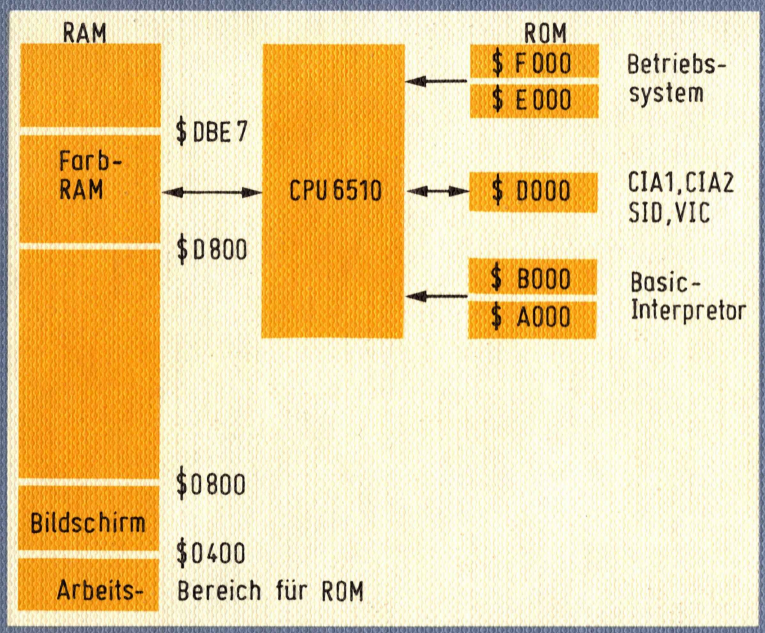
Franzis
Computer-Praxis

Wunderlich
Erfolgreicher mit dem VC 64 arbeiten

Eine verständliche Einführung für alle VC 64-Anwender in die Maschinensprache



Die Programmierung des 6502
Logische Zustände
Zahlensysteme
Das Hexadezimalsystem
Die "AND" (UND)-Funktion
Die "OR"(ODER)-Funktion
Die "EOR"(EXCLUSIV-ODER)-Funktion
Negative Zahlendarstellung
Die BCD-Zahlendarstellung
Der ASC II
Der Bildschirmcode
Detailliertes über Programme
Die CPU-Register
Die Indexregister X und Y
Der Stapelzeiger
Prozessor Status Register
Mnemonics – Assembler – Operationscode
Zeropage

Franzis'

Wunderlich
Erfolgreicher mit dem VC 64 arbeiten

In der Reihe
FRANZIS Computer-Praxis
sind erschienen:

Andersen/Zirpel, Die Programmierpraxis der technisch-naturwissenschaftlichen Taschenrechner
Busch, Basic für Aufsteiger
Busch, Basic für Einsteiger
Esders, Das Buch zum Apple II
Feichtinger, Mit Computern steuern
Haugg, Software-Engineering und ihre Qualitätssicherung
Klein/Klein, Z-80 Applikationsbuch
Klein, Basic-Interpreter
Klein, Mikrocomputer Hard- und Softwarepraxis
Klein, Mikrocomputer selbstgebaut und programmiert
Klein, Mikrocomputersysteme
Klein, Was ist Pascal?
Link, Messen, Steuern und Regeln mit Basic
Piotrowski, IEC-Bus
Plate, Betriebssystem CP/M
Plate/Wittstock, Pascal: Einführung – Programmentwicklung – Strukturen
Troitzsch, Mikrocomputer-Schaltungstechnik
Wunderlich, Erfolgreicher mit CBM arbeiten

Franzis Computer-Praxis

Dipl.-Ing. Franz Wunderlich

Erfolgreicher mit dem VC 64 arbeiten

Eine verständliche Einführung für alle VC 64-Anwender in die Maschinensprache

Mit 3 Abbildungen

Franzis'

CIP-Kurztitelaufnahme der Deutschen Bibliothek

Wunderlich, Franz:
Erfolgreicher mit dem VC 64 arbeiten : e. verständl. Einf. für alle VC 64-Anwender in d. Maschinensprache / Franz Wunderlich. – München : Franzis, 1985.
(Franzis-Computer-Praxis)
ISBN 3-7723-7781-5

Satz: SatzStudio Pfeifer, Germering
Druck: Offsetdruckerei Hablitzel, 8060 Dachau
Printed in Germany · Imprimè en Allemagne

ISBN 3-7723-7781-5

Vorwort

Dieses Buch wurde geschrieben, um den CBM-Anwendern, die bereits mit BASIC entsprechende Erfahrungen gesammelt haben, die tieferen Möglichkeiten des VC-64 näher zu bringen. Es handelt sich hierbei um eine Einführung in die Maschinensprache.

Das erste Kapitel über die grundsätzliche Maschinenprogrammierung des 6502 Prozessors habe ich dabei fast vollständig aus meinem Buch „Erfolgreicher mit CBM arbeiten" übernommen, da sich dieses Konzept bewährt hat und gerade für Anfänger der Maschinensprache einen leichten Einstieg bedeutet. Der VC-64 verfügt über den Prozessor 6510, der zwar zusätzliche Datenleitungen besitzt, aber sonst vollkommen softwarekompatibel zum 6502 ist. Am Schluß des 1. Kapitels befindet sich daher eine kurze Darstellung über den Unterschied zwischen 6502 und 6510. Das 2. Kapitel beschäftigt sich mit den gerätespezifischen Eigenschaften des VC-64. Dabei werden die Funktionen des Basic-Interpreters, der I/O-Bausteine und des Betriebssystems abgehandelt, wobei eine umfangreiche dokumentierte Adressenliste der ROM-Routinen den Abschluß dieses Kapitels bildet. Im 3. Kapitel befinden sich Programmbeispiele. Dort wird auch auf bekannte Programmiermethoden Rücksicht genommen, die sich der Maschinenprogrammierer merken sollte.

Den Abschluß dieses Buches bildet der Anhang mit verschiedenen Tabellen über den Befehlssatz der CPU, sowie eine hexadezimale und dezimale Umwandlungsliste.

Dieses Buch ist eine wertvolle Hilfe für alle VC-64 Anwender, die sich mit der Maschinensprache auseinandersetzen wollen und wird sicher lange Zeit als Nachschlagewerk dienen. Der Autor wünscht allen, die sich damit beschäftigen, für die spätere Programmierarbeit einen vollen und schnellen Erfolg.

Dipl.-Ing. (FH)
Franz Wunderlich, Kaufering

Wichtiger Hinweis

Die in diesem Buch wiedergegebenen Schaltungen und Verfahren werden ohne Rücksicht auf die Patentlage mitgeteilt. Sie sind ausschließlich für Amateur- und Lehrzwecke bestimmt und dürfen nicht gewerblich genutzt werden*).
Alle Schaltungen und technischen Angaben in diesem Buch wurden vom Autor mit größter Sorgfalt erarbeitet bzw. zusammengestellt und unter Einschaltung wirksamer Kontrollmaßnahmen reproduziert. Trotzdem sind Fehler nicht ganz auszuschließen. Der Verlag sieht sich deshalb gezwungen, darauf hinzuweisen, daß er weder eine Garantie noch die juristische Verantwortung oder irgendeine Haftung für Folgen, die auf fehlerhafte Angaben zurückgehen, übernehmen kann. Für die Mitteilung eventueller Fehler sind Autor und Verlag jederzeit dankbar.

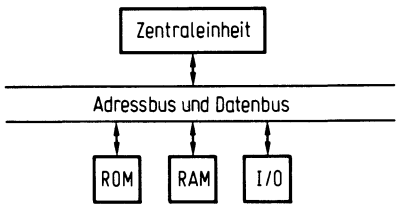
*) Bei gewerblicher Nutzung ist vorher die Genehmigung des möglichen Lizenzinhabers einzuholen.

Inhalt

1 Die Programmierung des 6502

1.1 Grundlegendes

Als erstes sehen Sie gleich ein allgemeines Mikrocomputerkonzept mit allerlei Fachausdrücken. Mit Akribie werden wir die einzelnen Teile daraus behandeln. Vorweg jedoch sollte man einige Begriffe erläutern, um auf den grundlegenden Informationen aufzubauen.



Da taucht zunächst einmal der Begriff *Software* auf, was auf deutsch übersetzt den komischen Begriff *Weichware* bedeutet. Gemeint ist jedoch im Zusammenhang mit der Computerei das alles, was man nicht direkt anfassen kann. Hierzu werden alle Programme, die in irgendwelchen Speichermedien verankert sind, und deren Dokumentationen (geistiges Gut) gezählt. Ich hoffe, daß der Leser so ungefähr weiß, was unter einem Programm zu verstehen ist. Wenn es Software gibt, dann gibt es auch *Hardware*. Die *Hardware* ist dann das, was man eben anfassen kann, also den Computer selbst, oder Tastatur, Bildschirm, Cassettenrecorder, Leitungen und so weiter.

Einer der wichtigsten Konzepte der Mikrocomputer ist der Speicher. Was ist ein Speicher? Nun darunter versteht man grob gesagt eine Schublade, in die man was hineinlegt und bei Bedarf wieder heraus genommen werden kann. Nur wird es beim Computer nicht mit einem Schubfach geregelt, sondern durch elektronische Schaltungen, die der Halbleitertechnik entstammen. In diese Halbleiterspeicher, die aus integrierten Schaltungsgliedern bestehen, werden nun irgendwelche Informationen bzw. Daten hineingeschrieben oder eben herausgelesen.

Im Zusammenhang mit den Speichern eines Mikrocomputers folgen zwei Begriffe, die in den technischen Beschreibungen eine wichtige Rolle spielen. Es handelt sich hierbei, wie auch aus der oben dargestellten Abbildung hervorgeht, um die Begriffe ROM und RAM. Man faßt auch diese beiden Elemente zusammen, und nennt das Ganze Hauptspeicher.

Zuerst wollen wir den RAM-Speicher kennenlernen. Den kann man nämlich mit der vorher erwähnten Schublade vergleichen. Darum wird er auch als Arbeitsspeicher bezeichnet. Es muß nämlich etwas getan werden, um etwas da hineinzulegen oder herauszuholen. Das Wort RAM stammt aus dem amerikanischen Fachenglisch und ist die Abkürzung für *R*andom *A*ccess *M*emory, was auf deutsch soviel wie „Speicher mit wahlfreiem Zugriff" bedeutet. Wie wir später noch sehen werden, trifft dieser Begriff nicht ganz zu. Wir mer-

ken uns nur, der RAM-Speicher ist ein Arbeitsspeicher. Man kann auch Schreib/-Lese-Speicher dazu sagen.

Etwas schwieriger zu erklären dürfte der Begriff des ROM-Speichers sein. ROM ist die Abkürzung für *R*ead *O*nly *M*emory und bedeutet übersetzt „Nur-Lese-Speicher". Man nennt ihn auch Programmspeicher, da er meist festgespeicherte Systemprogramme enthält. Da man auf den ROM ebenso wahlfrei zugreifen kann wie auf den RAM, stimmt eben die englische Bezeichnung für RAM nicht ganz. Um nun den Begriff ROM zu verstehen, bauen wir uns wieder einen Vergleich auf. Nehmen wir an, der ROM wäre ein Buch. Man kann nun aus dem Buch irgendwelche Daten herauslesen, aber keine hineinschreiben. Natürlich wird irgendwann einmal das Buch gedruckt. Mit dem ROM geschieht das Gleiche. Da werden nämlich auch irgendwann die entsprechenden Informationen (meist Programme) darin hinterlegt, die man dann später zu jeder Zeit herausnehmen bzw. herauslesen kann.

Fassen wir noch einmal zusammen:

RAM = Arbeitsspeicher oder auch Schreib/Lese-Speicher
ROM = Programmspeicher oder auch Nur/Lese-Speicher

Um nun bei dem Beispiel mit der Schublade zu bleiben. Der Speicher eines Mikrocomputers hat eine beträchtliche Ansammlung von Schubladen, die man als Speicherstellen oder Speicherzellen bezeichnet. Unser VC-64 hat da gleich 65536 Speicherstellen. Wie kommt man nur an eine dieser Zellen heran. Dazu haben sich die Computerhersteller etwas einfallen lassen, was allerdings nicht ganz neu ist.

Es wurden den 65536 Speicherzellen Nummern zugeordnet. Die erste Speicherstelle erhält die Nummer Null. An dieser Stelle sollten wir uns gleich merken, daß in der Computertechnik das Zählen nicht bei 1 losgeht, sondern immer bei 0. Die zweite Speicherstelle erhält die Nummer 1, die dritte Speicherstelle erhält die Nummer 2, und so weiter bis zur Nummer 65535. Die 65536. Speicherstelle hat also die Nummer 65535. Die Nummer einer Speicherstelle wird im Zusammenhang mit der Mikrocomputertechnik als Adresse bezeichnet, was eigentlich einleuchtend klingt. Und die Adressen von 0 bis 65535 ergeben eben den Adreßbereich oder auch Adressierbereich. Wie ich auf die Zahl 65536 komme, werde ich später erläutern.

Wenn ich nun dem Computer den Befehl gebe, hole den Inhalt aus der Speicherstelle mit der Adresse sowieso, dann muß der Computer diese Arbeit ausführen. Warum er das muß, werden wir noch lernen.

Jetzt frägt sich noch, was denn so eine Speicherstelle für einen Inhalt hat. Und da kommen wir wieder auf den ROM und RAM zurück. Unser Mikrocomputer hat zwei Speicherbereiche und somit auch zwei Adressenbereiche, die jeweils dem ROM und dem RAM zugeordnet sind. So verfügt unser VC-64 über 65536 Speicherstellen, die als RAM ausgebildet sind. Man kann nun in eine RAM-Speicherzelle Daten hineinlegen bzw. hineinschreiben und das heißt auf englisch Write. Oder man holt etwas heraus bzw. die Daten werden gelesen, was auf englisch Read heißt. In den RAM-Speicher mit den Adressen von 0 bis 65535 können also Daten eingeschrieben oder gelesen (Read/Write = R/W) werden. Beim ROM müßte nun eigentlich ein „Aha" erfolgen, weil ja ein ROM-Speicher ein Nur-Lese-Speicher ist. Somit kann man aus einer ROM-Speicherstelle deren Inhalt (meist Programmdaten) also nur lesen, aber keine Daten einschreiben.

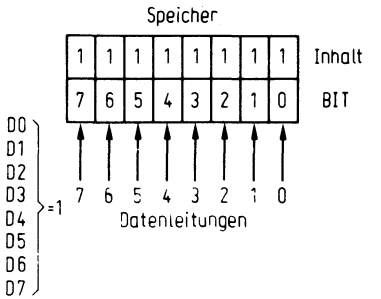
Wir werden nun untersuchen, welche Daten oder Informationen eine Speicherstelle enthält. Als Inhalt einer Speicherstelle können darin Buchstaben, Ziffern oder sonstige

Zeichen enthalten sein, die aber im Computer selbst in anderer Form (codiert) dargestellt werden. Wie nun der Computer diese Zeichenfolgen verwertet, kommt lediglich auf die Interpretation des Programmierers an.

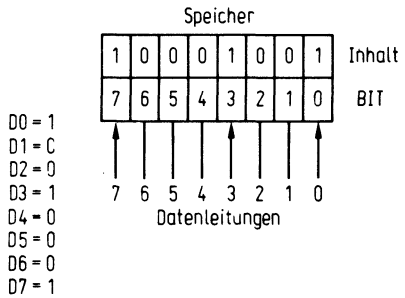
Wieviel steht nun in einer Speicherstelle darin und wie steht es darin?

1.2 Logische Zustände

Da kennt der Computer eigentlich nur zwei Zustände, nämlich 0 und 1. Professionell kann man auch niedriger Spannungspegel für 0 und hoher Spannungspegel für 1 definieren. Entweder es fließt ein Strom (= 1) oder es fließt kein Strom (= 0). Der Zustand selbst, der eben 0 oder 1 darstellt, wird mit dem in der Computerelektronik üblichen Begriff *Bit* bezeichnet. Das Wort Bit stammt aus dem Sprachgebrauch der Computertechnik und ist die englische Abkürzung für Binary Digit. Ein Bit ist damit der Zustand , der an einem bestimmten Platz in einer Speicherstelle vertreten ist. Somit besteht der Inhalt des Gesamtspeichers nur aus solchen Bits. Um nun Zeichen im Computer abzuspeichern, müssen in einer Speicherstelle mehrere Bits stehen. An eine Speicherstelle führen nun 8 Stromleitungen heran, die als Datenleitungen bezeichnet werden und parallel verlaufen. Jede einzelne Leitung kann nun ein Bit aufnehmen und an die Speicherstelle weiterleiten. Nennen wir nun diese Datenleitungen vorläufig D0, D1, D2 bis D7. Wenn wir nun Daten in den Speicher einschreiben und alle Leitungen führen Strom, dann stehen in der angesprochenen Speicherstelle lauter Einsen.



Werden in den Datenleitungen lauter 0-Bits zur Speicherstelle geschickt, dann stehen eben nach Ausführung lauter Nullen in der Speicherstelle. Es ist nun vorstellbar, daß z.B. die Datenleitungen D0, D3 und D7 Strom führen und die restlichen Leitungen sich in einem niedrigen Spannungszustand befinden. Nach erfolgtem Einschreiben in eine Speicherstelle steht eben dann diese Bit-Kombination in der Speicherstelle.



Werden nun 8 Bit parallel verarbeitet (d.h. 8 Bit parallele Daten zur gleichen Zeit), so werden sie zu einem Begriff zusammengefaßt, den man mit dem Wort Byte bezeichnet. Wenn also von einem Byte die Rede ist, werden immer 8 Bits damit gemeint. Somit wird in einer Speicherstelle eines Mikrocomputers immer 1 Byte abgespeichert. Um nun in einer Speicherstelle eine Zahl durch ein Byte darzustellen, könnte man folgende Festlegung treffen:

```
  0 = 00000000
  1 = 00000001
  2 = 00000010
  3 = 00000011
        ¦
        ¦
 15 = 00001111
 16 = 00010000
 17 = 00010001
        ¦
        ¦
254 = 11111110
255 = 11111111
```

Wie man nun an diesem Beispiel sieht, können aus 8 Bits 256 verschiedene Kombinationen angeordnet werden, mit denen man dann die Zahlen von 0 bis 255 oder aber 256 verschiedene Zeichen darstellen kann. Um nun zu verstehen, wie man die einzelnen Bits untereinander anordnet, um etwa Zahlen daraus zu interpretieren, machen wir einen Ausflug in die Mathematik.

1.3 Zahlensysteme

Unser normales Zahlensystem besteht aus den Ziffern 0 bis 9 und das sind insgesamt 10 Ziffern. Deshalb bezeichnet man dieses Zahlensystem als „Dezimalsystem". Jede Zahl ist aus einer oder mehreren Ziffern zusammengesetzt. Eine normale Dezimalzahl kann man sich folgendermaßen aufgebaut denken:

Beispiel: $2367 = 2*10^3 + 3*10^2 + 6*10^1 + 7*10^0$
(bekannt sollte sein, daß $10^0 = 1$)

Die Zahl 2367 besteht aus 2 Tausender, 3 Hunderter, 6 Zehner und 7 Einer und damit aus 4 Stellen. Jede Stelle besteht aus einer eigenständigen Zahl, die man in Exponentialform ganz allgemein so anschreiben kann:

$$\text{Eine Stelle einer Zahl} = M * B^E$$

M = Mantisse
B = Basis
E = Exponent

Die Mantisse ist eine Ziffer aus dem Ziffernvorrat des jeweiligen Zahlensystems (z.B. Dezimalsystem: 0, 1, 2, 4, 5, 6, 7, 8, 9). Die Basis ist, wie schon der Name aussagt, die Grundlage eines Zahlensystems. Sie stellt den Faktor dar, mit dem die Mantisse multipliziert wird. Im Dezimalsystem hat die Basis den Wert 10. Der Exponent ist ein Wert, der angibt, welche Stelle die Mantisse in der Gesamtzahl einnimmt. Man kann auch sa-

gen, der Exponent zeigt auf, wo das Komma steht. So wird die Dezimalzahl 34,12 in vier Faktoren zerlegt, wobei jeder Faktor in Exponentialform folgendermaßen dargestellt wird:

$3 * 10^1$ ——————— 2. Stelle vor dem Komma
$4 * 10^0$ ——————— 1. Stelle vor dem Komma
$1 * 10^{-1}$ ——————— 1. Stelle nach dem Komma
$2 * 10^{-2}$ ——————— 2. Stelle nach dem Komma

Prinzipiell unterscheidet sich das Dezimalsystem durch nichts von anderen Systemen. Hätten wir z.B. 12 Finger, so wäre uns das Dezimalsystem ein Greuel.

Aus dem Mittelalter kennen wir noch heute das Dutzend (B=12) und das Gros (144) – Reste des 12er-Systems.

1.3.1 Das Binärsystem

Das Binärsystem hat einen Ziffernvorrat von 2 Ziffern (0 und 1) und ist geradezu prädestiniert für die Computerverarbeitung mit Bits. Statt Binärsystem kann man auch Dualsystem sagen. Die allgemeine mathematische Schreibweise ist die gleiche, wie beim Dezimalsystem. Nur hat die Basis den Wert 2 und die Mantisse die Ziffern 0 und 1. Eine Dualzahl wird damit folgendermaßen aufgebaut:

Beispiel: $1101 = 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0$

Wenn man nun die Binärzahl 1101 aus dem Beispiel mit der Exponentialschreibweise regelrecht ausrechnet, dann ergibt das die Dezimalzahl 13.

$1 * 2^3 = 8$
$1 * 2^2 = 4$
$0 * 2^1 = 0$
$1 * 2^0 = 1$

8+4+0+1=13

Binär 1101 = Dezimal 13

Wir haben damit gleich die Umrechnung von binär nach dezimal. Die Umkehrung läuft von binär nach dezimal so ab:

$13/2^3 =$	1	Rest 5
$5/2^2 =$	1	Rest 1
$1/2^1 =$	0	Rest 1
$1/2^0 =$	1	Rest 0

→ 1101

Die Dezimalzahl wird durch den größtmöglichen Binärfaktor geteilt. Das heißt, der Exponent zur Basis 2 muß so groß sein, daß sich die Dezimalzahl gerade noch dividieren

läßt. Das Ergebnis ist die vorderste Binärziffer. Der Rest wird durch den nächst kleineren Faktor dividiert usw. Die jeweiligen Ergebnisse werden dann zur Binärzahl zusammengesetzt.

1.3.2 Das Hexadezimalsystem

Das Hexadezimalsystem wird für die abgekürzte Schreibweise von Binärzahlen gebraucht. Doch darauf wird später noch eingegangen. Das Hexadezimale Zahlensystem hat die Basis 16 und wird deshalb auch Sedezimalsystem genannt. Da wir nur 10 Ziffernsymbole (0–9) kennen, wurde für die 16 Ziffern des Hexadezimalsystem vereinbart, daß die Ziffern über 9 durch die großen Anfangsbuchstaben des Alphabets dargestellt werden. Die 16 Ziffern des Hexadezimalsystems kann man somit folgendermaßen bezeichnen:

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

Eine Zahl im Hexadezimalsystem ist dann folgendermaßen aufgebaut:

Beispiel: $3E6A = 3 \cdot 16^3 + E \cdot 16^2 + 6 \cdot 16^1 + A \cdot 16^0$
$= 3 \cdot 4096 + 14 \cdot 256 + 6 \cdot 16 + 10 \cdot 1$
= dezimal 15978

Auf diese Art haben wir gleich die Umrechnung von Hexadezimal nach dezimal. Die Umkehrung von dezimal nach hexadezimal läuft wie beim Binärsystem ab:

```
      15978/ 16^3  =  3
     -12288
     ------
Rest   3690/ 16^2  =  E   (dez.14)
     -  3584
     ------
Rest    106/ 16^1  =  6
     -   96
     ------
Rest     10/ 16^0  =  A   (dez.10)
                      |
                      +----------------> 3E6A
```

Wir haben damit 3 Zahlensysteme behandelt, die in unserem Assemblerkurs für Mikrocomputer Verwendung finden. Um nun die Zahlen in den jeweiligen Zahlensystemen voneinander unterscheiden zu können, wurde folgende Festlegung getroffen:

a) Vor eine Binärzahl wird das Prozentzeichen (%) gesetzt.
b) Vor eine Hexadez.-Zahl wird das Dollarzeichen ($) gesetzt.
c) Die Dezimalzahl bleibt ohne Zeichen.

Beispiele: dez.: 255
hex.: $FF
Bin.: %11111111

Wir können nun die ersten 16 Zahlen in allen drei Systemen wie folgt angeben:

Binär	Dezimal	Hexadezimal
%0000	0	$00
%0001	1	$01
%0010	2	$02
%0011	3	$03
%0100	4	$04
%0101	5	$05
%0110	6	$06
%0111	7	$07
%1000	8	$08
%1001	9	$09
%1010	10	$0A
%1011	11	$0B
%1100	12	$0C
%1101	13	$0D
%1110	14	$0E
%1111	15	$0F

Aus dieser Darstellung wird klar, daß sich eine Binärzahl aus vier Bits durch eine Hexadezimalziffer darstellen läßt. Somit kann man ein Byte mit zwei Hexziffern abkürzen. Darum ist die Schreibweise mit zwei Hex.-Ziffern am gängigsten, was auch im obigen Beispiel zum Ausdruck kommt. Um es gleich vorwegzunehmen:

Das Hexadezimalsystem dient nur dazu, Binärzahlen verkürzt darzustellen.
Es ist leicht einzusehen, daß z.B.

$AAAA

wesentlich besser zum Anschreiben ist als

%1010101010101010

1.4 Logische Funktionen

Sämtliche Operationen, die der Computer ausführt, kann man auf die grundsätzlichen Verknüpfungen von Bits zurückführen. Dabei werden die verschiedenen Bits addiert, gesetzt, gelöscht, getestet und invertiert. Diese Funktionen werden von den hochintegrierten Schaltgliedern in den jeweiligen Bausteinen vollzogen, wobei jeweils ein oder zwei Bitzustände in das Schaltglied eingehen, entsprechend verknüpft werden und als Ergebnis auf der anderen Seite wieder herauskommen. Als Assemblerprogrammierer brauchen wir natürlich nicht zu wissen, wie das elektronisch realisiert wird. Wir sollten nur die entsprechenden mathematischen Operationen kennen. Hier hilft uns die Schaltalgebra, wobei wir uns hier nur auf die Funktionen

AND (UND) NOT (NICHT)
OR (ODER) EOR (EXCLUSIV ODER)

konzentrieren brauchen.

1.4.1 Die „AND"-(Und)-Funktion

Die AND-Funktion hat das Symbol: ∧

Die mathematische Schreibweise ist: x ∧ y = z

Das bedeutet, daß an zwei Eingängen die Signale x und y anliegen und am Ausgang das Signal z. X und y werden durch „AND" verbunden und liefern z. Dabei gibt es folgende Möglichkeiten:

0 ∧ 0 = 0
0 ∧ 1 = 0
1 ∧ 0 = 0
1 ∧ 1 = 1

Das bedeutet, der Ausgang ist nur dann 1, wenn beide Eingänge ebenfalls 1 sind.

Eine der Aufgaben der „AND"-Funktion ist es, z.B. ein bestimmtes oder mehrere bestimmte Bits in einem Byte auf Null zu setzen. Das nennt man auch maskieren.

Beispiel:

1.Byte	%	101xxx11
„AND"-Verknüpfung mit		
2.Byte	%	11100011
Ergebnis:	%	10100011

die drei x im 1. Byte sollen ausdrücken, daß es sich um einen unbekannten Bit-Zustand handelt. In allen drei Fällen werden diese Bits durch die Und-Verknüpfung mit drei Nullen im 2. Byte auf Null gesetzt.

Eine andere Anwendung wäre die Prüfung, ob eine Zahl gerade oder ungerade ist. Dazu brauchen wir nur das niedrigste Bit mit 1 durch UND zu verknüpfen. Wenn das Ergebnis dieses Bits 1 ist, dann ist die Zahl ungerade, ansonsten gerade.

1.4.2 Die „OR"-(ODER)-Funktion

Die „OR"-Funktion hat folgendes Aussehen: V

Die mathematische Schreibweise ist:

x V y = z

Die ODER-Funktion hat folgende Möglichkeiten:

0 V 0 = 0
0 V 1 = 1
1 V 0 = 1
1 V 1 = 1

Das bedeutet, daß das Ergebnis am Ausgang nur dann 0 ist, wenn an beiden Eingängen 0 liegt.
Die hauptsächlichste Anwendung der ODER-Verknüpfung ist das Setzen von Bits.

Beispiel:
1.Byte % 000xxx10
ODER-Verknüpfung mit
2.Byte % 00011100

Ergebnis: % 00011110

1.4.3 Die „NOT"-(NICHT)-Funktion

Das NOT-Symbol hat folgendes Aussehen: ¬

Die mathematische Schreibweise lautet:

$x \neg = \overline{x}$

Aufgabe der NOT-Funktion ist es, einzelne Bits zu negieren (invertieren) bzw. zu komplimentieren. Also wird aus einer 0 eine 1 und aus einer 1 eine 0.

1.4.4 Die „EOR"-(EXCLUSIV-ODER)-Funktion

Die „EOR"-Funktion hat folgendes Aussehen: ⊻

Die mathematische Schreibweise lautet:

$x \veebar y = z$

Dabei gibt es folgende Möglichkeiten:

$0 \veebar 0 = 0$
$0 \veebar 1 = 1$
$1 \veebar 0 = 1$
$1 \veebar 1 = 0$

Das bedeutet, das Ergebnis am Ausgang ist nur dann 1, wenn beide Eingänge verschiedene Signale aufweisen.
Das Anwendungsgebiet der EOR-Funktion ist das Invertieren von Bits.

Beispiel:
1.Byte % 10101010
EOR-Verknüpfung mit
2.Byte % 11111111

Ergebnis: % 01010101

1.4.5 Die Addition

Im Grunde genommen kann der Computer nur eine mathematische Funktion, nämlich die der Addition von zwei Bits. Dabei gibt es vier Möglichkeiten:

$0 + 0 = 0$
$0 + 1 = 1$
$1 + 0 = 1$
$1 + 1 = (1)0$

Die ersten drei Additionen sind eigentlich ganz logisch. Bei der letzten Addition muß man jedoch aufpassen. Wenn zwei Bits mit jeweils Zustand 1 addiert werden, kommt als Ergebnis 0 heraus. Es entsteht jedoch ein Übertrag in die nächste Stelle mit dem Zustand 1. Addiert man mehrstellige Dualzahlen, dann muß ein Übertrag aus der vorangehenden Stelle mit addiert werden. Das geschieht in der Regel bei den ersten 8 Bits in Mikroprozessoren automatisch. Erst beim Übertrag im vordersten Bit eines Bytes muß der Programmierer Manipulationen vornehmen.

Beispiel einer Binäraddition mit 8 Stellen:

```
      % 10001100        (140)
+     % 10001111      + (143)
-------------------------------
= % (1) 00011011        (283)
```

Als Ergebnis kommt 283 heraus. Wenn wir jedoch nur das achtstellige Binärergebnis betrachten (% 00011011), dann hätten wir umgerechnet das Ergebnis als 27 zu interpretieren. Im Computer wird das auch so gehandhabt, da ja der Prozessor nur acht Bits verarbeitet und somit das 9. Bit vernachlässigt. Deshalb ist es die Aufgabe des Programmierers, in solchen Fällen entsprechende Korrekturen anzubringen. Um einen Übertrag aus 8 Bits feststellen zu können, besitzt der Prozessor ein Flagregister. Dieses hat eine Flag enthalten, die anzeigt, wann ein Übertrag aufgetreten ist. Mathematisch ist das der Fall, wenn sich nach der Addition eine Änderung des vordersten Bits ergeben hat. Dadurch wäre dann das Ergebnis nicht als 27 zu interpretieren, sondern als 256 + 27.

Man kann im Computer aber nicht nur addieren, sondern auch subtrahieren. Die Subtraktion wird jedoch über den Umweg der Addition durchgeführt.

Beispiel:
5 − 3 = 5 +(−3)

Um aber bei der Subtraktion addieren zu können, muß man Binärzahlen negativ darstellen. Dazu wurde in der Computertechnik festgelegt, daß eine Dualzahl negativ ist, wenn sie im vordersten Bit eine 1 enthält. Bei einer 0 wäre sie positiv. Man könnte nun die Zahl (−3) so darstellen:

```
  −3  =  %  10000011
```

Addieren wir nun

```
     5  =  %  00000101
+ (−3)  =  %  10000011
-------------------------
     2  =  %  10001000   =   (−8)
```

dann hätten wir das Ergebnis als (−8) zu interpretieren, was falsch ist. Um mit dieser Logik fertig zu werden, wurde eine andere Darstellungsart von negativen Dualzahlen festgelegt, das sogenannte

ZWEIERKOMPLIMENT

1.4.6 Die Subtraktion

1.4.6.1 Negative Zahlendarstellung

Um die Logik eines Prozessors nicht unnötig zu erweitern und damit kostspieliger zu machen, wurde auf die Einführung des Einerkompliments verzichtet. Man ist dazu übergegangen, das Zweierkompliment einzuführen.

Beim Zweierkompliment wird die positive Binärzahl in allen Stellen komplimentiert (invertiert). Dann addiert man in der hintersten Stelle den Bitzustand „1". Das Ergebnis stellt das Zweierkompliment dar.

Beispiel für die Zahl „–3":

positive Zahl	„+3"	/	%	00000011
Einerkompliment		/	%	11111100
Addition in Bit	0	/	%	00000001 (+)
Ergebniszahl	„–3"	/	%	11111101

Da eine Binärzahl beim Zweierkompliment im vordersten Bit ebenfalls eine „1" stehen hat, kann die negative Binärzahl leicht von einer positiven unterschieden werden, da diese im vordersten Bit eine „0" enthält. Deshalb geht eine Zahl im vorzeichengerechten Bereich nicht von –256 bis +255, sonder von

% 10000000 bis % 01111111
und das ist
–128 bis + 127

Alle Zahlen eines Bytes in vorzeichengerechter Form stellen sich wie folgt dar:

Dezimal	Binär	Hexadezimal
+127	01111111	7F
+126	01111110	7E
+125	01111101	7D
|	|	|
+3	00000011	03
+2	00000010	02
+1	00000001	01
0	00000000	00
–1	11111111	FF
–2	11111110	FE
–3	11111101	FD
|	|	|
–125	10000011	83
–126	10000010	82
–127	10000001	81
–128	10000000	80

Wir können nun die Subtraktion „5–3“ durch nachfolgende Addition durchführen.

```
  % 00000101   ---    5
+ % 11111101   --- (-3)
-------------------------
=(1) 00000010   ---    2
```

Läßt man den Übertrag außer acht, dann ist hier das Ergebnis richtig. In einem solchen Fall wird man auch keine Korrekturen mehr anbringen. Wenn man im vorzeichengerechten Bereich arbeitet, dann entsteht nicht im vordersten Bit ein Übertrag, sondern im siebten Bit, da ja nur sieben Stellen ohne Vorzeichen zur Verfügung stehen. Den Übertrag der 7. Stelle bezeichnet man nun nicht mit Übertrag, sondern mit Überlauf. Weil wir die Bezeichnungen „Übertrag“ und „Überlauf“ noch des öfteren verwenden und die englischen Begriffe dafür gebräuchlicher sind, wird in Zukunft folgende englische Bezeichnung geschrieben:

Übertrag = CARRY
Überlauf = OVERFLOW

Werden 2 Bits in der 7. Stelle addiert, und ergibt sich daraus ein Overflow, dann ändert sich natürlich der Zustand des Vorzeichenbit. Das passiert bei folgenden vier Fällen:

1.4.6.2 Overflow-Fälle

1. Wenn das Ergebnis einer Addition zweier positiver Zahlen den Wert „+127“ überschreitet.
2. Wenn das Ergebnis einer Addition zweier negativer Zahlen den Wert „–128“ unterschreitet.
3. Beim Subtrahieren einer großen positiven Zahl von einer großen negativen Zahl.
4. Beim Subtrahieren einer großen negativen Zahl von einer großen positiven Zahl.

1.4.6.3 Zusammenfassung

Wird ein Bytewert ohne Vorzeichen verwendet, dann hat dieses Byte einen Wertbereich von 0 bis 255. Werden also Zahlen ohne Vorzeichen verarbeitet, dann entsteht bei einer Bereichsüberschreitung ein Carry. In diesem Fall bleibt das Overflowbit unberücksichtigt. Im vorzeichengerechten Bereich hat ein Byte den Wertbereich von –128 bis +127. Bei Bereichsüberschreitung entsteht ein Wechsel des Zustandes im Overflowbit. Bei vorzeichengerechter Bedienung hat das Carrybit keinerlei Bedeutung bei Bereichsüberschreitung.

Wie wir noch sehen können, hat der Mikroprozessor zwei Bits in einem Register enthalten, die den Zustand eines Carry und Overflow in einem Speicherbyte feststellen. Diese zwei Bit werden als Carryflag und Overflowflag bezeichnet. Diese können dann auch entsprechend abgefragt und manipuliert werden.

1.4.7 Die Multiplikation

Im Dezimalsystem wird folgendermaßen multipliziert:

```
 33 • 18
 ------
  33
 264
 ---
 594
====
```

Der Multiplikand (1. Faktor) wird mit der vordersten Zahl des 2. Faktors (Multiplikator) multipliziert. Dann wird der 1. Faktor mit der nächsten Zahl des 2. Faktors multipliziert und das Ergebnis unter dem 1. Ergebnis angeschrieben, jedoch um eine Stelle weiter rechts. Bei mehrstelligen Multiplikationen setzt sich das so fort. Anschließend werden die jeweiligen Stellenergebnisse addiert und ergeben die gesamte Multiplikation.

Beim Binärsystem funktioniert das gleiche analog. Für die Multiplikation jeder Stelle gelten folgende Regeln, die wiederum logisch sind:

$0 \cdot 0 = 0$
$0 \cdot 1 = 0$
$1 \cdot 0 = 0$
$1 \cdot 1 = 1$

Multiplizieren wir z.B. einmal

$4 \cdot 3$

in binärer Darstellungsweise:

```
  0100 • 0011
  -----------
  0000
   0000
    0100
     0100
  ---------
% 0001100    = 12
================
```

1.4.8 Die BCD-Zahlendarstellung

Wir haben bisher die hexadezimale Schreibweise als Codierung für Binärzahlen benutzt. Es gibt aber noch eine andere Darstellungsart, nämlich die „*B*inär *C*odierte *D*ezimalziffer". Das Prinzip sagt aus, daß jede Stelle einer Dezimalzahl durch soviele Bit codiert wird, wie eben dazu benötigt werden. Mit drei Bits kann man nur 8 Ziffern verschlüsseln. Für 10 Ziffern braucht man daher vier Bits, ebenso wie für die hexadezimale Codierung. Allerdings werden von den 16 möglichen Bitmustern nur 10 für die BCD-Darstellung verwendet.

BCD	
dezimal	*binär*
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
X	1010
X	1011
X	1100
X	1101 wird nicht verwendet
X	1110
X	1111

Der Vorteil der BCD-Ziffern ist, daß man dezimal rechnen kann. Der Nachteil ist, daß man bei Berechnungen, die einen Übertrag aufweisen, besondere Algorithmen durchführen muß, um die binären Zahlen über 9 zu eliminieren.

Beispiele von Bits, die BCD-codiert sind:

33 = % 00110011
76 = % 01110110
99 = % 10011001

1.4.9 Die Zeichencodierung

Bisher haben wir nur Zahlen binär interpretiert. Der Computer arbeitet aber auch mit Textdaten. Texte bestehen aber nicht nur aus Ziffern, sondern auch aus Zeichen. Wie werden nun aber Zeichen binär dargestellt!

Nun ganz einfach, den Zeichen werden Binärzahlen zugeordnet, und zwar einfach in aufsteigender Reihenfolge der Binärzahlen. Das Ganze nennt man dann codieren oder verschlüsseln. Ein ganz bekannter Code, für die Verschlüsselung von Zeichen im Mikrocomputerbereich ist der ASCII-Code. ASCII ist die Abkürzung von „*a*merican *s*tandard *c*ode for *i*nformation *i*nterchange". Dies bedeutet übersetzt „amerikanischer Gebrauchscode für Informationsaustausch". Im ASCII werden 7 Bits mit 128 Zeichen codiert und damit lassen sich

numerische Zeichen,
alphanumerische Zeichen,
Steuerzeichen und
Sonderzeichen

darstellen.

1.4.9.1 Der ASCII

Nachfolgende Tabelle legt klar, wie der ASCII aufgebaut ist:

ASCII-Code Dezimal		0 – 31		32 – 64		65 – 95		96 – 127	
Hexa-dez.	HT*	0	1	2	3	4	5	6	7
NT*	Bits	000	001	010	011	100	101	110	111
0	0000	NUL	DLE	SPACE	0	@	P	–	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	,	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	l	—
D	1101	CR	GS	–	=	M	]	m	}
E	1110	SO	RS	.	>	N	∧	n	~
F	1111	SI	US	/	?	O	←	o	DEL

* NT = Niederwertiger Teil
HT = Höherwertiger Teil

In der Tabelle besteht die Bitreihe links aus 4 Bits pro Zeile, während die obere Bitreihe aus 3 Bits pro Spalte besteht. Die linke Reihe gibt die hinteren Bits an und die obere Reihe die vorderen Bits. Man kann nun ganz leicht ein Zeichen binär darstellen.

Beispiel des Großbuchstabens „A":

A = % 01000001

Die ersten 32 Zeichen sind Steuerzeichen. Diese sind einfach Angaben, die später im Programm festlegen, wie die Steuerung (z.B. der Zeilenvorschub) abzulaufen hat. Die Abkürzungen der Steuerzeichen haben folgende Bedeutung:

NUL = Null
SOH = Start of heading (Initialisierung)
STX = Start of Text (Textbeginn)
ETX = End of Text (Textende)
EOT = End of Transmission (Übertragungsende)
ENQ = Enquiry (Testanfrage)
ACK = Acknowledge (Quittierung)
BEL = Bell (Klingelzeichen)
BS = Backspace (Rückwärtsschritt)
HT = Horizontal tabulation

LF = Line feed (Zeilenvorschub)
VT = Vertical tabulation
FF = Form feed (Seitenvorschub bzw. Formatanpassung)
CR = Carriage return (Wagenrücklauf)
SO = Shift out
SI = Shift in
DLE = Data link escape (Datenverbindung umschalten)
DC = Device control (Gerätesteuerung)
NAK = Negative Acknowledge (Fehlerrückmeldung)
SYN = Synchronus (Synchronisierungszeichen)
ETB = End of transmission (Blockübertragungsende)
CAN = Cancel (Ungültigkeitsmeldung)
EM = End of medium (Datenträgerende)
SUB = Substitute (Austauschbefehl)
ESC = Escape (Hilfsfunktion bei editieren bzw. vorübergehende Umschaltung)
FS = File separator (File Trennung)
GS = Group separator (Einteilung in Gruppen)
RS = Record separator (Aufzeichnung einteilen)
US = Unit separator (Einheitentrennung)
SP = Space (Blank = Leerzeichen)
DEL = Delete (löschen)

1.4.9.2 Der Bildschirmcode

Da die Logik des Bildschirms anders verwaltet wird als z.B. ein Drucker, werden für die Ausgabe auf dem Bildschirm die Zeichen wiederum anders codiert. Da die verschiedenen Computer ihre Bildschirmausgaben auch verschieden codieren, wollen wir hier nicht näher auf den Bildschirmcode eingehen. Die Besprechung über den Bildschirmcode des VC-64 erfolgt später.

1.4.9.3 Die Interpretation der Zeichen

Man wird sich vielleicht fragen, wie der Computer feststellt, daß es sich um ein Zahlenbyte oder ein Zeichenbyte handelt?

Um es klipp und klar zu sagen, der Computer weiß das überhaupt nicht. Er hat nur ein entsprechendes Bitmuster im Speicher stehen. Der einzige, der für die Interpretation der Bitmuster verantwortlich ist, ist der Programmierer. Er hat dafür Sorge zu tragen, daß die Bytes in der richtigen Reihenfolge stehen und entsprechend manipuliert werden.

1.5 Zurück zum Speicher

Wir haben nun etwas über Zahlensysteme erfahren und ich hoffe auch verstanden. Falls nicht, sollten Sie dieses Kapitel noch einmal eingehend durchlesen oder besser die Beispiele mit Papier und Bleistift nachzuvollziehen.

Wie ich eingangs im Kapitel Speicher schon erwähnt habe, ist die Tatsache, daß im Speicher nur binäre Zustände (Bits) herrschen. Es liegt nun an der Struktur eines Mikro-

computersystems, daß immer 8 parallele Bits in einer Speicherstelle enthalten sind. Wie wir schon gehört haben, hat so ein Computersystem eine ganze Menge Speicherstellen. Und wenn wir 8-Bit-parallele Daten, die aus unterschiedlichen Bit-Kombinationen bestehen können, in eine Speicherstelle einschreiben oder darauslesen, sollte man wissen, welche Speicherstelle gemeint ist. Wir haben auch schon gehört, daß unser Computer 65536 Speicherstellen hat und jede Speicherstelle eine Adresse. So ist es unvermeidbar, wenn wir außer den Datenleitungen auch Adreßleitungen an die Speicherstellen heranführen. In diesen Adreßleitungen liegen dann die Informationen, welche Speicherstelle denn nun angesprochen ist. Vorläufig nehmen wir an, daß jede Speicherstelle 16 Adreßleitungen und 8 Datenleitungen beansprucht. Mit 16 Leitungen kann man nun 2 hoch 16 verschiedene Bit-Muster darstellen. Da 2 hoch 16 nach „Adam Riese" gleich 65536 ist, können somit 65536 Speicherstellen angesprochen werden. Wir haben schon gesagt, daß jede Speicherstelle eine Nummer bzw. eine Adresse hat. So können wir also mit den 16 Leitungen die 65536 Speicherstellen mit den Adressen von 0 bis 65535 ansprechen. Dieses Ansprechen oder besser Zugreifen auf eine Speicherstelle nennt man adressieren.

Ich habe vorher geschrieben, daß 16 Leitungen an eine Speicherstelle heranführen. Ganz so einfach ist es jedoch nicht. Der Speicher ist aus integrierten Schaltungen zusammengesetzt, die man auch Chips nennt. Es liegt nun am Aufbau eines solchen Bausteins, daß an die Speicherstellen die 16 Adreßleitungen nicht direkt festgelötet sind. Jeder Speicherbaustein hat eine interne Auswahlschaltung, die eben die entsprechenden Speicherstellen innerhalb des Bausteins ansprechen. Für den Gesamtspeicher werden nun mehrere Bausteine (Chips) verwendet. Nun existiert noch eine externe Auswahlschaltung, die ebenfalls auf einem Chip untergebracht ist, und die eben die Auswahl unter den Bausteinen übernimmt (Chip-Auswahl; englisch = chip select). Im Grunde genommen brauchen wir dieses Wissen für später eigentlich nicht mehr. Man sollte nur einmal etwas davon gehört haben.

Wir wissen jetzt, daß wir für den Zugriff auf den Speicher Adreßleitungen brauchen und für die Abspeicherung selbst Datenleitungen. Es gibt noch weitere Leitungen. Man muß nämlich den Speicherstellen mitteilen, daß Daten-Bits eingeschrieben werden sollen oder herausgelesen werden. Diese Leitungen nennt man Steuer-Leitungen, die angeben, in welcher Richtung die Daten fließen sollen.

Zusammenfassend werden die Leitungen mit Begriff BUS bezeichnet. Das Bussystem führt nun an alle Komponenten des Mikrocomputers heran. Man könnte lediglich den Bus noch in Daten-, Adreß- und Steuerbus einteilen.

1.6 Peripherie

Die Daten, die ein Mikrocomputer bearbeitet, müssen zuvor irgendwie eingegeben werden. Dazu hat unser Computer eine Tastatur. Nun nehmen wir an, wir haben ein Programm mit den dazugehörigen Werten eingegeben und geben ihm den Befehl zur Ausführung. So führt unser Computer diese Operationen durch. Die Ergebnisse stehen danach sehr wahrscheinlich in irgendwelchen Speicherstellen. Um nun die Ergebnisdaten für unser Auge sichtbar werden zu lassen, besitzt der Computer eine Anzeige. In unserem Fall ist es ein Bildschirm, oder sogenannte Terminal. Übrigens tut es bei manchen Mikrocomputersystemen auch ein altes Fernsehgerät.

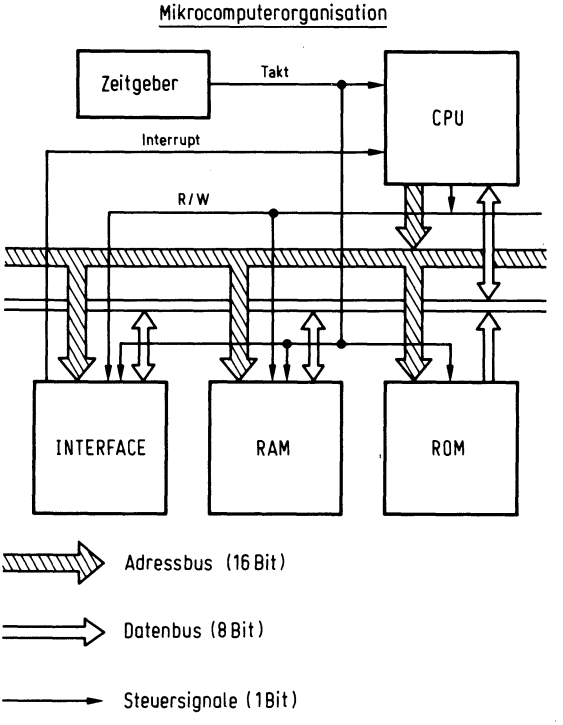
Zu diesen Systemeinheiten, wie Bildschirm und Tastatur, sagt man auch Peripherie-Einheiten. Die Amerikaner schreiben für diese Ein- und Ausgabebausteine so, wie sie es meinen, nämlich INPUT/OUTPUT (abgekürzt: I/O).

1.7 Der Verwalter

Wir haben bisher schon über einige Teile des Mikrocomputersystems Erfahrungen gemacht. Jetzt kommen wir zum wichtigsten Bauteil. Wie vielleicht schon viele ahnen, handelt es sich um den *Mikroprozessor*. Er ist das Herz des Systems. Er ist der Chef oder Verwalter des Ganzen. Auf amerikanisch wird er als CPU bezeichnet, als Abkürzung für *C*entral *P*rocessing *U*nit, was deutsch übersetzt Zentrale Prozessor Einheit bedeutet. Man kann schon sagen, daß in der CPU sämtliche Fäden zusammenlaufen. Wie nun die CPU das Gesamtsystem verwaltet, werden wir noch sehen.

1.8 Das Mikrocomputerkonzept

Auf nachfolgendem Schema sehen Sie die schematische Darstellung eines Mikrocomputerkonzepts aufgezeichnet. Hier sind alle grundlegenden Bausteine vorhanden,



die ein Mikrocomputer braucht. Das Konzept besteht aus den entsprechenden Bausteinen und deren Verbindungswegen. Wir haben folgende Elemente zur Verfügung:

A) Mikroprozessor
B) Zeitgeber
C) RAM
D) ROM
E) Interface
F) Datenbus
G) Adreßbus
H) Steuerleitungen

Beim VC-64 kämen noch die Tastatur und der Bildschirm dazu. Das ganze bezeichnet man dann als die

Hardware

eines Mikrocomputers. Wir wollen uns nun daraus die einzelnen Komponenten etwas vornehmen.

A) Der Mikroprozessor

Das Gehirn des Systems ist der Mikroprozessor. Er steuert alle Vorgänge im Gesamtkonzept. Man kann sagen, daß dort alle Fäden zusammenlaufen. Wir werden den Prozessor, und zwar speziell den „6502" noch genauer unter die Lupe nehmen.

B) Der Zeitgeber

Der Zeitgeber, meist auch als Taktgenerator bezeichnet, ist notwendig, um die einzelnen Bitsignale des Systems zu synchronisieren. Man kann sagen, daß der Taktgenerator das Herz eines Mikrocomputers ist. Der Taktgenerator ist bereits im Prozessor enthalten.

C) Der RAM-Speicher

Wie bereits festgelegt wurde, handelt es sich bei den RAM-Bausteinen um Arbeitsspeicherstellen. Diese Art kann gelesen, aber auch beschrieben werden. Man kann in den RAM-Speicher Bitzustände einschreiben, die dann solange erhalten bleiben, bis der Strom am Gerät abgeschaltet wird. Allerdings gehen dann die enthaltenen Daten verloren. Der RAM-Speicher ist ein „Schreib-Lese-Speicher", in dem Programme und Daten für den Anwender frei zur Benutzung stehen.

D) Der ROM-Speicher

Der ROM-Speicher ist ein „Nur-Lese-Speicher". Das heißt, man kann dort keine Bitzustände einschreiben, sondern eben nur herauslesen. Allerdings haben diese Bausteine gegenüber den RAM's den Vorteil, daß enthaltene Daten nach Abschalten des Stroms nicht verlorengehen. Sie stehen damit nach dem Einschalten des Stromes bzw. des Geräts sofort zur Verfügung. Im Falle des VC-64 sind im ROM-Speicher der Basic-Interpreter, Monitor usw. als Maschinenprogramme fest abgespeichert.

E) Interface

Mit Interface sind Bausteine gemeint, die die Peripheriegeräte, wie z.B. Tastatur oder Bildschirm, mit dem Prozessor verbinden. Da es vorkommt, daß die Signale eines Periphe-

riegerätes asynchron zu den Signalen des Prozessors verlaufen, ist es nötig, eine Anpassung vorzunehmen. Dies ist nun die Aufgabe der Interface-Bausteine.

F) Der Datenbus

Vorerst wird man sich fragen, was unter einem „BUS" ganz allgemein zu verstehen ist. Die Erklärung ist ganz einfach. Wir wissen, daß nicht einzelne Bits zeitlich nacheinander übertragen werden, sondern immer mehrere Bits gleichzeitig. Das Übertragungsmedium eines Bitzustandes ist eine entsprechende Stromleitung. Ein Bit wird damit gleichzeitig über 8 Leitungen geschickt. Mehrere Leitungen werden nun zu dem Sammelbegriff „BUS" vereint. Wie wir aus der Darstellung erkennen können, besitzt das System einen Datenbus von 8 Bit Breite. Das sind nun die acht Leitungen, die ein Byte mit dem entsprechenden Bitmuster vom Speicher oder zum Speicher übertragen. Wie wir aus der Darstellung erkennen können, werden die Daten in zwei Richtungen verschickt. Durch diese Möglichkeit nennt man den Datenbus auch bidirektionalen BUS.

G) Der Adreßbus

In der Darstellung ist der Adreßbus dicker gezeichnet. Dadurch soll gezeigt werden, daß für das Ansprechen von Speicheradressen noch mehr Leitungen erforderlich sind als beim Datenbus. Tatsächlich besteht im VC-64 der Adreßbus aus 16 Leitungen. Auf diesen Leitungen werden 2 Bits zur gleichen Zeit übertragen, und zwar immer nur in Richtung zum Speicher. Um aus einer Speicherstelle Daten entnehmen zu können, muß man wissen, um welche Speicherstelle es sich handelt. Dazu braucht jede Speicherstelle eine Nummer, eben die Adresse. Damit man eine bestimmte Speicherstelle identifizieren kann, wird eben auf dem Adreßbus das 16-Bit-Signal übertragen und die entsprechende Speicherstelle angesprochen.

H) Die Steuerleitungen

Manchmal werden die Steuerleitungen auch zu dem Begriff „Steuerbus" zusammengefaßt. Unter anderem gehört zum System die Schreib/-Lese-Steuerleitung. Diese teilt den entsprechenden Bausteinen mit, in welche Richtung die Daten fließen sollen. Um die Byteübertragungen synchron zu halten, werden auf einer Synchronisationsleitung die Taktsignale gesteuert.

Weitere spezielle Steuerleitungen wären die Interruptleitungen. Auf diesen wird dem Prozessor mitgeteilt, wann eine Bedienungsanforderung eines Peripheriegerätes erfolgt.

Wir wollen hier nicht weiter auf die Steuersignale eingehen, da sie momentan noch nicht richtig verstanden werden. Erst bei den entsprechenden Kapiteln wird das Verständnis leichter.

1.9 Detailliertes über Programme

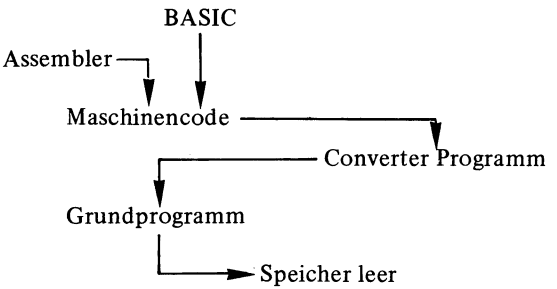
Die große Diskussion wurde entfacht, als die Hersteller die Mikroprozessoren in Massen auf den Markt schleuderten. Als „Jobkiller" wurden diese hochintegrierten Bausteine bezeichnet. Und bei manchen Personen blieb der Nachgeschmack zurück, als wäre ein Mikroprozessor ein intelligentes Gerät, das in wenigen Jahren den Menschen selbst zum

Befehlsempfänger macht. Daß das ganz und gar nicht der Fall ist, wird jedem einleuchten, der sich mit der Computertechnik intensiv befaßt. Im Grunde genommen kann unser Computer allein gar nichts. Er führt nur das aus, was wir ihm vorher eingeben bzw. eintrichtern. Nur ist es so, wenn wir ihm etwas eingegeben haben, dann führt er die entsprechenden Operationen mit ganz einfachen Schritten durch, aber dafür ungemein schnell. So wird z.B. eine Multiplikation intern nicht so ausgeführt wie wir es in der Schule gelernt haben, sondern es wird Bit für Bit addiert.

Was ist nun ein Programm?

Ein Programm ist eine Befehlsfolge, die irgendwann im Speicher hinterlegt wurde und die, wenn es erforderlich ist, einzeln der Reihenfolge nach vom Computer abgearbeitet wird. Die Amerikaner sagen für Befehlsfolge: Instructioncode. Was eigentlich keiner Übersetzung mehr bedarf.

1.9.1 Programmiersprachen



Wie man aus der obigen Abbildung ersieht, gibt es mehrere Programmsprachen, die in verschiedenen Ebenen angesiedelt sind. Wenn wir die unterste Ebene betrachten, so finden wir einen total leeren Speicher vor. Tatsächlich gibt es Computer- oder Prozessorsysteme, die nirgendwo ein Bit stehen haben. Hier muß man von Grund auf programmieren. Unser VC-64 hat nun schon so ein Grundprogramm. Dieses liegt in der CPU fest. Wenn wir nun ein bestimmtes Bit-Muster in die CPU eingeben, wird dieses Muster als Befehl interpretiert. Man sagt auch Befehlsdekodierung oder Befehlsentschlüsselung dazu. Der Mikroprozessor führt nun die richtigen Instruktionen aus. Das Grundprogramm bezeichnet man auch als Befehlssatz eines Mikroprozessors. Im Prozessor selber besteht der Befehlssatz jedoch nur aus Bit-Kombinationen, wobei wiederum ein Befehl durch ein Byte dargestellt ist. Da die Programmerstellung mit 8 Bit recht unvorteilhaft ist, wurde ein hexadezimaler Code eingeführt. Mit einer hexadezimalen Ziffer kann man, wie schon erläutert wurde, 4 Bit darstellen. Somit braucht man für einen Bytebefehl 2 Hexadezimale Ziffern. Das Programm, das nun mit jeweils 2 hexadezimalen Ziffern pro Befehl im Prozessor existiert, wird als Maschinencode bezeichnet. Es muß nun intern noch ein Converterprogramm geben, das die Umsetzung vom Maschinencode in den Binärcode vornimmt. Das Assemblerprogramm wollen wir vorerst einmal außer Betracht lassen.

In der obersten Ebene haben wir nun die Programmiersprache BASIC (Abkürzung für *B*eginners *a*ll *s*ymbolics *i*nstruction *c*ode).

BASIC ist eine höhere Programmiersprache und wurde eigentlich geschaffen, um Studenten und sonstigen Anfängern das Programmieren zu erleichtern. In unserem VC-64 ist BASIC fest verankert. Man kann nun mit BASIC z.B. einen Sinus berechnen lassen, indem ich dem Computer wörtlich mitteile: PRINT SIN (Winkel). Wollte man einen Sinus in Maschinensprache berechnen, so müßte man zuerst dafür ein Programm schreiben. Und genau so ein Programm, sagen wir einfach Unterprogramm dazu, ist in unserem Computer fest abgespeichert. Die einzelnen BASIC-Befehle sind eben lauter Unterprogramme des Maschinencodes. Dieses Maschinenprogramm wird als BASIC-Interpreter bezeichnet und ist fest im ROM-Speicher verankert. Es wurde schon erwähnt, daß aus dem ROM-Speicher als Festwertspeicher nur Daten ausgelesen werden können. Nun wäre es schlecht, wenn man in ein ROM etwas hineinschreiben könnte. Denn dadurch würde man z.B. bei unserem Computer den BASIC-Interpreter verändern, was wiederum katastrophale Konsequenzen zur Folge hätte. Zusätzlich zum BASIC-Interpreter ist im ROM noch ein Programm fest gespeichert, das uns die Dialogfähigkeit mit dem Computer ungemein erleichtert. Dieses Maschinenprogramm bezeichnet man als Betriebssystem. Doch darauf werden wir noch später zurückkommen.

An dieser Stelle ist noch die Erklärung fällig, warum man den ROM auch als Festwertspeicher bezeichnet? Wenn wir nämlich dem Computer den Strom entziehen, dann gehen sämtliche Informationen des RAM-Speichers verloren. Die Informationen im ROM jedoch bleiben erhalten und stehen nach dem Wiedereinschalten sofort zur Verfügung. Wie das elektronisch realisiert ist, brauchen wir nicht unbedingt zu wissen. Man sollte nur wissen, daß es ROMs gibt, die man mit einem Programmiergerät programmieren kann und mit ultraviolettem Licht wieder löschen kann. Diese ROM-Bausteine nennt man EPROM, als Abkürzung für *e*rasable *p*rogrammable ROM (löschbarer-programmierbarer-Nur-Lese-Speicher).

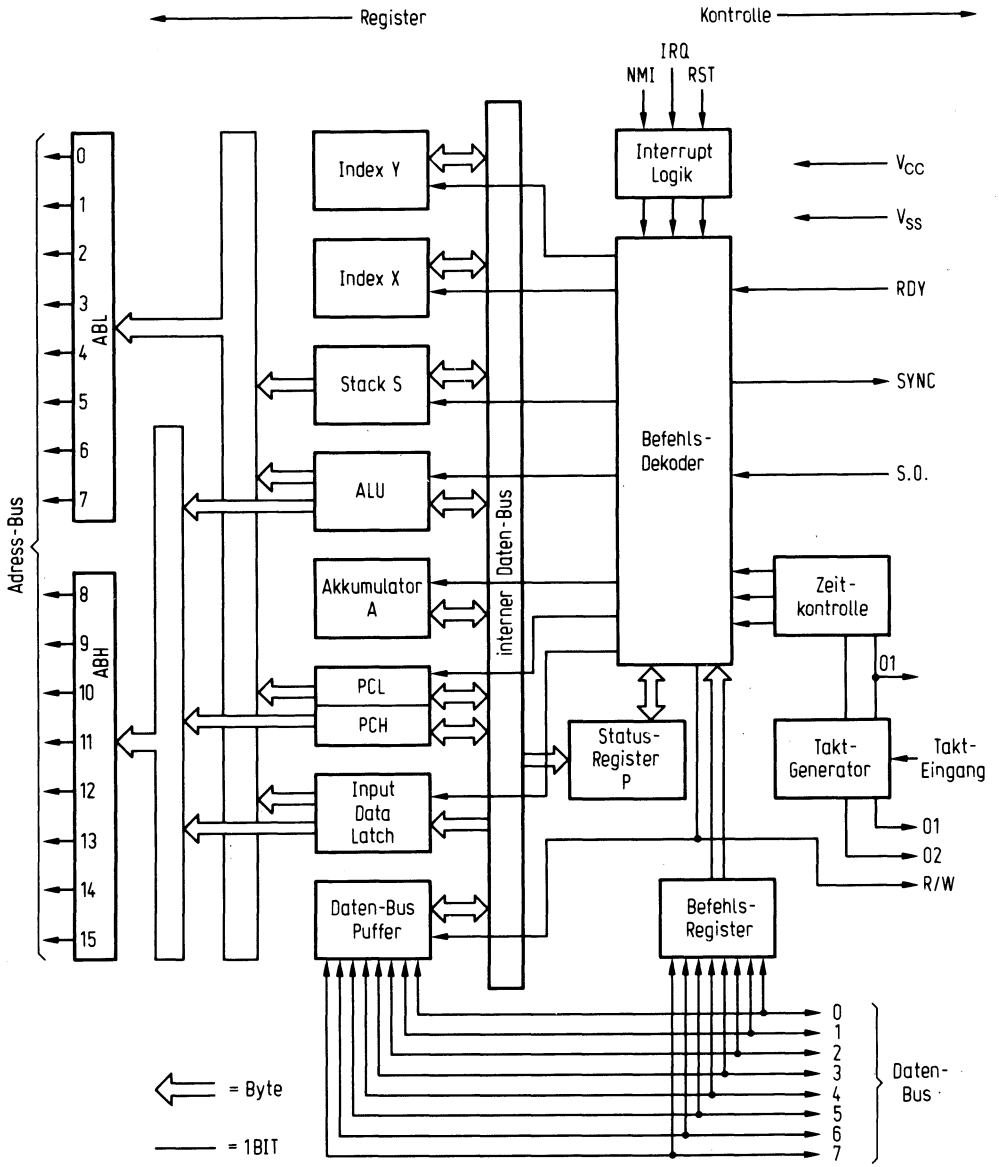
Wir haben uns nun einige Grundvoraussetzungen geschaffen und wenden uns nun der Zentraleinheit bzw. CPU zu.

1.10 Die CPU 6502

Wenn Sie das Blockschaltbild des Mikroprozessors 6502 betrachten, wird es zunächst verwirrend auf Sie einwirken. Es werden nun einige kurze Bemerkungen darüber erfolgen. Später wird dann ein vereinfachtes Schema dargestellt, das dann für die spätere Programmierarbeit erforderlich ist.

Die Innereien der CPU 6502 sind eigentlich auch so ähnlich aufgebaut wie das Computersystem selbst. So haben wir ebenso eine BUS-Struktur wie außerhalb. Die einzelnen Blöcke in der CPU sind im Grunde genommen auch wieder nur Speicherelemente. Nur bezeichnet man sie in der CPU als Register.

Der innere Datenbus wird über bidirektionale Treiber (Datenbus-Puffer), die vom Befehlsdecoder kontrolliert werden, an den außerhalb der CPU liegenden System-BUS angeschlossen. Der innere Adreßbus steuert über die Treiber ABL und ABH den System-Adreß-Bus. Das Befehlsregister ist direkt an den Systembus angekoppelt. In ihm sind die



Befehlsbytes abgespeichert. D.h. im Befehlsregister liegt das schon vorher erwähnte Grundprogramm bzw. der Befehlssatz des Prozessors. Gibt man nun einen Befehl ein, so sucht sich der Prozessor den entsprechenden Code aus den ca. 250 Befehlen, die im Befehlsregister liegen, aus und übergibt ihn an den Befehlsdekoder. Das Befehlsbyte wird nun im Befehlsdekoder entschlüsselt (decodiert) und somit werden durch das entsprechende Bit-Muster die jeweiligen Register und Busleitungen auf die erforderlichen Spannungspegel gesetzt und der Befehl wird ausgeführt.

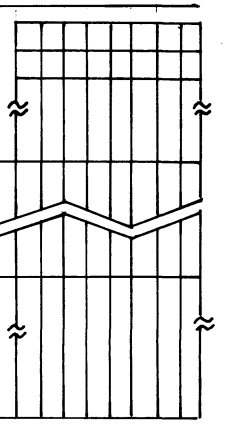
Die recht komplizierten inneren Abläufe müssen durch Zwei-Phasen-Taktsignale gesteuert werden. Auf dem Prozessor-Chip befinden sich daher ein Zweiphasentaktgenerator und eine Steuerlogik (Zeitkontrolle). Der 6502 benötigt also nur einen einfachen Einphasentaktgenerator als externes Steuerelement. Der Befehlsdecoder ist nicht nur ein einfacher Decoder, sondern er stellt die nötigen Ablaufsteuerungen der CPU dar. Die Interrupt-Logik beeinflußt die Ablaufsteuerung wesentlich. Die Vorgänge in den einzelnen Registern, die um den Datenbus gruppiert sind, werden von der Ablaufsteuerung geregelt.

An Registern sind im 6502 enthalten: Ein Akkumulator, zwei Indexregister, ein Programmzähler, eine arithmetische Logikeinheit (ALU), ein Stackregister und ein Zustandsregister (Status-Register). Bei Operationen, die bestimmte arithmetische oder logische Verknüpfungen berechnen, werden die Daten über die ALU (arithmetic logic unit) geleitet, abgeändert und dann im Akkumulator gespeichert. Mehr Informationen über die Hardware des 6502 sind nicht erforderlich, um seine Programmierung zu erlernen.

An dieser Stelle ist noch eine Erklärung darüber fällig, weshalb immer von 8 und 16 Bit die Rede war. Die Hersteller der Mikroprozessoren stellen zur Zeit eben nur Prozessoren her, die 8-Bit-parallele Daten verarbeiten können. Man bezeichnet daher unsere CPU als 8-Bit-Prozessor. Nun wird man sich fragen: „Wie werden dann die 16-Bit Adressen verarbeitet?“

Mit 8 Bits könnte man nur 2 hoch 8 verschiedene Speicherstellen ansprechen. Da 2 hoch 8 gleich 256 ist, kann man also mit einem Byte nur 256 verschiedene Speicherstellen adressieren, und das sind doch recht wenig Speichermöglichkeiten. So hat unsere CPU einen 16-Bit breiten Programmzähler, der eigentlich aus zwei 8-Bit Registern besteht, die jeweils nacheinander den Adreßbus steuern. Mit 16 Bit kann man nun 2 hoch 16(= 65536) Speicherstellen ansprechen. Das Konzept der Adressierung (Speicherzugriff) ist so festgelegt: Die 65536 Speicherstellen sind in Seiten und Nummern aufgeteilt. Auf einer Seite stehen 256 Nummern (pro Nummer 8 Bit oder 1 Byte). Und davon gibt es wiederum 256 Seiten. Wenn man nun 256 Seiten zu je 256 Nummern hat, dann ergibt der Gesamtplatz eben 256 mal 256 (= 65536) Nummern, Speicherstellen oder Adressen. Man sagt nun, eine 16-Bit Adresse besteht aus einem oberen Adreßteil mit 8 Bit und einem unteren Adreßteil mit ebenfalls 8 Bit. Somit verstehen wir auch, warum der Programmzähler aus zwei Registern besteht. In einem Register steht eben die Seite und im anderen Register steht die Nummer. Die Amerikaner sagen dazu auch LOW ORDER BYTE für niedriger Adreßteil und HIGH ORDER BYTE für hoher Adreßteil. Man könnte auch sagen, der Speicher besteht aus 256 Blöcken mit je 256 Adressen. Die Blöcke oder Seiten werden in der Computertechnik als PAGE bezeichnet. Die höherwertigen 8 Bits der Adresse (ADH) geben daher die PAGE an, auf der sich die Adresse befindet, und die niederwertigen 8 Bits (ADL) bezeichnen eine bestimmte Adresse auf dieser PAGE. Die erste PAGE im Speicher (ADH=00) wird als PAGE 0 oder als ZEROPAGE bezeichnet. Die nächsthöhere Page (ADH=1) ist dann die PAGE 1. Das geht nun so weiter bis PAGE 255.
Die nachfolgende Abbildung spricht für sich.

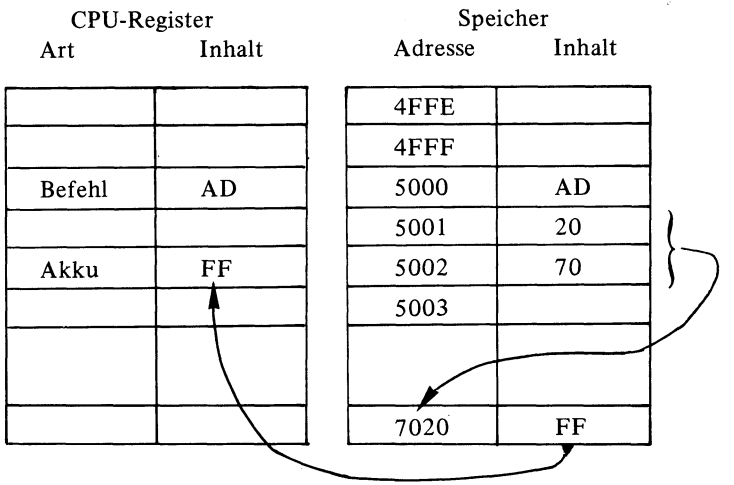
ADRESSE BINÄR HIGH ORDER 15 14 13 12	11 10 9 8	LOW ORDER 7 6 5 4	3 2 1 0	ADRESSE DEZIMAL BYTE NUMMER	ADRESSE HEXA-DEZIMAL PAGE/BYTE	ADRESS. SPEICHER-FELD (65536 BYTE)
0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0	00 – 00	
0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 1	1	00 – 01	
0 0 0 0	0 0 0 0	1 1 1 1	1 1 1 1	255	00 – FF	
0 0 0 0	0 0 0 1	0 0 0 0	0 0 0 0	256	01 – 00	
1 1 1 1	1 1 1 0	1 1 1 1	1 1 1 1	65279	FE – FF	
1 1 1 1	1 1 1 1	0 0 0 0	0 0 0 0	65280	FF – 00	
1 1 1 1	1 1 1 1	0 0 0 0	0 0 0 1	65281	FF – 01	
1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1	65535	FF – FF	



Im Folgenden wird nun die Erklärung gegeben, wie ein Mikrocomputer im Prinzip arbeitet. D.h. es wird ein vereinfachter Ablauf gegeben, welche Teile des Systems wann beeinflußt werden. Nehmen wir einmal an, wir befinden uns mitten in einer Programmausführung und sind gerade bei dem nächsten Befehl angelangt. Dieser Befehl steht jetzt irgendwo im RAM-Speicher bei einer bestimmten Adresse und zwar dort, wo eben unser Programm abgespeichert ist. Als Beispiel wollen wir den Befehl $AD (B 10101101) herausgreifen. Diese Instruktion ist ein ganz bestimmter Befehl aus dem Befehlssatz des Prozessors. Er bedeutet folgendes: Nimm den Inhalt der im Programm gespeicherten nachfolgend adressierten Speicherstelle und bringe diese Daten in der CPU in das Register, das mit Akkumulator bezeichnet wird.

Nehmen wir nun an, dieser Befehl steht z.B. an der Speicherstelle $5000, und die weiteren Programmdaten auf den folgenden Speicherstellen. Die nachfolgende Abbildung wird uns die Erklärung der Vorgänge im System erleichtern.

In der Speicherstelle mit der Adresse $5000 steht der Inhalt $AD. Dieser Inhalt gelangt nun in das Befehlsregister. Im Befehlsregister wird das Bit-Muster $AD mit sämtlichen vorhandenen Befehlsmustern verglichen und festgestellt, daß der Befehl $AD vorhanden ist und zur Dekodierung weitergeleitet werden muß. Das Befehlsmuster wird nun im Befehlsdekoder entschlüsselt. Während dieser Ausführungen wird zur gleichen Zeit auf die nächste Adresse ($5001) hochgezählt. Durch die Befehlsentschlüsselung weiß nun die CPU, daß noch mehr Informationen für Befehlsausführung notwendig sind. Somit werden nacheinander der untere Adreßteil und der obere Adreßteil der Speicherstelle $7020 in die ALU und den Akku geladen, wobei der Programmzähler nach jedem Byte hochgezählt wird. Die beiden Adreßteile werden im Zusammenwirken von ALU und Akku berechnet



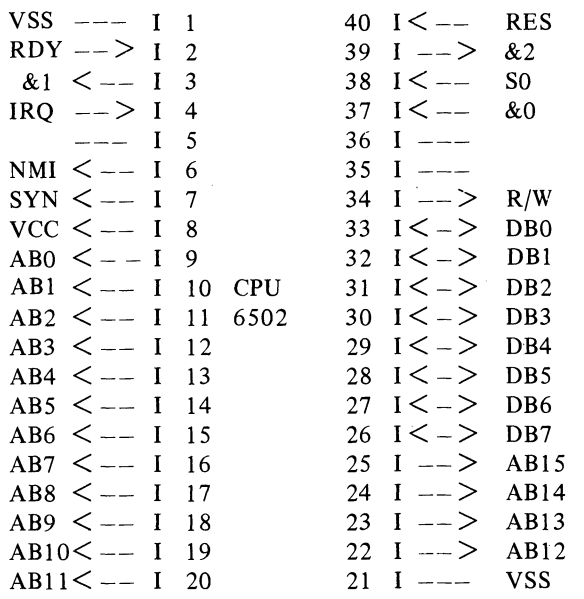
und so dann auf den Adreßbus gegeben. Dabei sind vom Befehlsdekoder die erforderlichen Komponenten bereits mit den entsprechenden Spannungspegeln belegt worden. Der Speicher weiß bereits durch ein Steuersignal, daß Daten aus einer Speicherstelle gelesen werden. Der Adreßbus signalisiert der Speicherstelle $7020, daß nur sie gemeint ist. Zur selben Zeit wird der Datenweg von der Speicherstelle über Datenbus zum Akkumulator durchgeschaltet und der Inhalt der Speicherstelle fließt in den Akku. Nach erfolgter Befehlsausführung würde in unserem Beispiel das Bit-Muster $FF stehen. Der gesamte Vorgang, der sich in dieser Darstellung langsam anhört, spielt sich in Bruchteilen einer tausendstel Sekunde ab.

Sie haben nun an dieser Ausführung gemerkt, welche internen Abläufe im System vor sich gehen. Genaugenommen braucht man jedoch nur zu wissen, daß der Befehl $AD die CPU veranlaßt, den Inhalt einer adressierten Speicherstelle in den Akkumulator zu übertragen.

An dieser Stelle sollte man sich gleich merken, daß die Adresse, die nach dem Befehl kommt, umgekehrt im Speicher stehen muß. Das heißt, der niederwertige Teil steht vor dem höherwertigen Teil, (ADH). Warum das so ist, liegt einfach an der Herstellung der CPU und deren Logik.

Zu erwähnen wären noch die Steuerleitungen. Diese werden vom Befehlsdecoder automatisch mit den entsprechenden Signalen belegt. (Ausnahme: die Interruptleitungen). So wird z.B. durch den Befehl „LDA“ über die R/W-Leitung der Speicherstelle mitgeteilt, daß das Datenbyte in Richtung CPU geht (lesen). Beim Lesen hat die R/W-Leitung das Signal 0, während beim Schreiben das Signal den Pegel 1 hat.

Die CPU 6502 ist mit 40 Anschlüssen zur Außenwelt verbunden, die wie folgt bezeichnet werden:



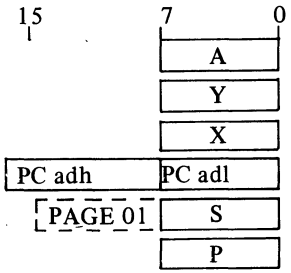
Anschlußbezeichnung	*Beschreibung*	*Art*
VCC, VSS	Stromversorgung und Masse	Stift B (VCC) = +5,0V DC +– 5%
RDY	Einzelzyklussteuerung	Eingang
&0	CPU-Takt	Eingang
&1, &2	System-Takte	Ausgang
IRQ	Maskierbarer Interrupt	Eingang
NMI	Unbedingter Interrupt	Eingang
RES (RESET)	Rücksetzsignal	Eingang
SYN	Synchronisation, Steuerung v. RDY	Ausgang (Holzykl.)
R/W	Lese/Schreibsteuerung	Ausgang
SO	Setzsignal für Überlaufflag	Eingang
AB0–AB15	Adreßbus	Ausgang
DB0–DB7	Datenbus	Bidirektional (Tristate)

Es wird nun hier nicht nicht näher auf die Steuerleitungen eingegangen, da sich dieses Buch nur mit softwaremäßigen Dingen beschäftigt. Wir werden noch ein paar Steuerfunktionen beim Befehlssatz besprechen.

1.10.1 Die CPU-Register

Um in Maschinensprache zu programmieren, ist es erforderlich, den Befehlssatz des Prozessors und die dazugehörigen Adressiermethoden zu kennen. Dazu braucht man ein ver-

einfachtes Prozessormodell, das nachfolgend abgebildet ist. Ich werde versuchen, die einzelnen Register daraus so einfach wie möglich zu erklären.



Akkumulator
Index Register Y
Index Register X
Programmzähler (PRG-Counter)
Stapelzeiger (Stack Pointer)
Prozessor Status Register (Zustandsregister)

1.10.1.1 Der Akkumulator
ist das Hauptregister in der CPU und besitzt als wichtigstes Datenregister eine Aufnahmefähigkeit von 8 Bits. In ihm werden sämtliche Ergebnisse aller logischen und arithmetischen Operationen abgespeichert. Der Akku liefert vor den Operatoren immer den ersten Wert bzw. Operand. Der zweite Operand wird, wenn erforderlich, aus irgendeiner Speicherstelle entnommen. Wenn zum Beispiel die CPU den Befehl, Addiere zum Inhalt des Akku den Inhalt einer Speicherstelle, erhält, dann wird eben dieser Befehl ausgeführt und das Ergebnis steht unmittelbar danach im Akku. Die erste Zahl, die im Akku gespeichert war, ist natürlich vom Ergebnis überschrieben worden und steht daher nicht mehr zur Verfügung. Die logischen Verknüpfungen und arithmetischen Operationen führt der Akku in enger Verbindung mit der ALU aus. Die ALU ist das Rechenwerk des Prozessors. Die ALU ist in unserem Modell nicht abgebildet, da sie automatisch vom Akku und den entsprechenden Befehlscodes beeinflußt wird.

1.10.1.2 Die Indexregister X und Y
werden als Speicher für Zwischenergebnisse benutzt. Man kann den Inhalt dieser Register bestimmten Befehlen inkrementieren (um 1 erhöhen) oder dekrementieren (um 1 erniedrigen). Es lassen sich mit ihnen auch ein paar arithmetische Operationen durchführen und das X-Register kann zusätzlich den Stapelzeiger laden bzw. lesen. Zudem haben diese beiden Register für die indizierte indirekte Adressierung eine wichtige Aufgabe. Doch darauf werden wir noch zurückkommen.

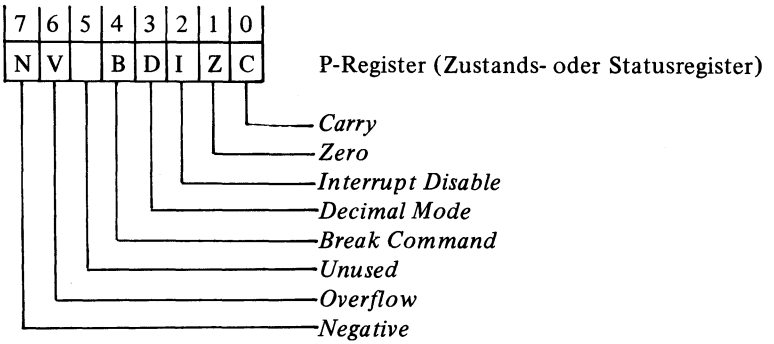
1.10.1.3 Dem Programmzähler (Program Counter)
ist als einziges 16-Bit-Register der CPU 6502 gestattet, die volle Adresse einer Speicherstelle (2 Byte) anzusprechen. In ihm ist die Adresse immer des folgenden Befehlsbytes abgespeichert. Im Grunde genommen wird der Zugriff zum Speicher vom Programmzähler beeinflußt. Wenn er nicht gerade durch einen Sprungbefehl beeinflußt wurde, wird er automatisch nach jedem Speicherzugriff inkrementiert (um 1 erhöht).

1.10.1.4 Der Stapelzeiger (Stack Pointer)
zeigt immer automatisch die aktuelle Adresse in seinem Bereich an. Was ist eigentlich unter einem Stack zu verstehen? Nun da der Stapelzeiger 8-Bit-Daten aufnehmen kann,

könnten 256 Datenbytes gespeichert werden. Wie der Name ausdrückt, basiert der Zugriff wie bei einem Stapel. Bei einem Kartenstapel lege ich z.B. 10 Karten der Reihenfolge nach einzeln aufeinander auf den Tisch. Dann liegt die erste Karte in diesem Stapel ganz unten und die letzte Karte ganz oben. Wenn ich nun die Karten vom Stapel wieder wegnehme, dann hebe ich die oberste Karte (letzte Karte) als erste und die unterste Karte (war 1. Karte) als letzte auf. So wie es beim Kartenbeispiel funktioniert, so wird es auch beim Stack Pointer geregelt. Nur werden im Stack die Daten nicht direkt gespeichert, sondern stehen in den Speicherstellen der Page 1 des Speichers. Im Stack selber sind nur die Adressen der Page 1 abgespeichert. Der Stack Pointer zeigt also nur auf die Page-1-Adressen, wo letztendlich dann der entsprechende Inhalt steht. Daher kommt die Bezeichnung Stapel*zeiger*. Nach einem Schreibbefehl wird sein Inhalt (01-Adresse) um 1 erniedrigt (dekrementiert) und bei einem Lesebefehl um 1 erhöht (inkrementiert). Somit ist die erste Adresse die im Stack steht, die Adresse $01FF. Daß gerade die Stackadressen die 1. Seite des Speichers benutzen, ist lediglich eine Festlegung der 6502 Hersteller.

Als letztes zu erwähnendes Register behandeln wir nun das
1.10.1.5 Prozessor-Status-Register

Dieses interessante Register zeigt uns immer den aktuellen Zustand des Prozessors an. Gesamt gesehen hat dieses Register zwar auch 8 Bit Fassungsvermögen, jedoch muß man hier die Bits einzeln betrachten. Folgende Darstellung soll die Erklärung der einzelnen Bits erleichtern:



Man kann nun mit Hilfe des P-Registers die Zustände der anderen Register feststellen. Wir wollen die einzelnen Bits einmal für sich allein betrachten. Für die einzelnen Bits kann man auch *Flag* (Kennzeichen, Marke) sagen.

a) Carry (Flag 0)
Die Carry Flag ist das Zustandsbit, das auf 1 gesetzt wird, wenn z.B. nach einer arithmetischen Operation ein Übertrag entstanden ist. Das Carry-Bit kann durch Befehl direkt beeinflußt werden.

b) Zero (Flag 1)
Die Zeroflag ist das Bit, das auf den Zustand 1 geht, wenn als Ergebnis z.B. im Akkumu-

lator eine 0 errechnet wurde. Die Zeroflag ist nicht direkt beeinflußbar, sondern wird automatisch nach den entsprechenden Operationen gesetzt.

c) Interrupt Disable (Interrupt außer Betrieb = Flag 2)
Wenn dieses Flag auf 1 gesetzt ist, dann werden vom Prozessor keine Befehle von außen (z.B. Tastatursignale) zugelassen. Das I-Flag wird direkt durch Befehl gesetzt oder zurückgesetzt (gelöscht).

d) Decimal Mode (Flag 3)
Mit diesem Flag ist die CPU 6502 den meisten anderen Prozessoren überlegen. Denn hier geht die CPU, wenn das Flag auf 1 gesetzt wird, auf die dezimale Betriebsart über. D.h. der Computer rechnet so, als wären es Dezimalzahlen. Natürlich wird intern schon binär gerechnet, wobei aber dem Bediener durch einen speziellen Binär-Code und die entsprechenden logischen Verknüpfungen dieser Code dezimal interpretiert wird. Auch das D-Flag wird vom Benutzer direkt durch Befehl beeinflußt.

e) Break Command (Flag 4)
Dieses Flag ist momentan etwas schwierig zu verstehen. Auf eine detailliertere Erläuterung werden wir noch zurückkommen. Hier sei vorläufig nur erwähnt, daß dieses Flag nur dazu benutzt wird, um in einem Interruptprogramm festzustellen, ob der Interrupt eben vom Break-Flag verursacht wurde oder von einem tatsächlichen Interrupt.

f) Unused (Flag 5)
Dieses Flag wird einfach nicht gebraucht und hat auch keine Auswirkungen. Das Unused-Flag wird wahrscheinlich in künftigen CPU-Modellen Bedeutung erlangen.

g) Overflow (Überlauf = Flag 6)
Dieses Flag wird auch als Überlauf-Status bezeichnet und wird automatisch gesetzt. Es ist also nicht direkt beeinflußbar. Dieses Bit erlangt nur bei vorzeichenbehafteter Arithmetik Bedeutung. Ansonsten kann es völlig ignoriert werden. Bei vorzeichenloser Arithmetik tritt z.B., wenn der Zahlenbereich von 0 – 255 überschritten wird, ein Carry auf. Wird aber mit Vorzeichen gerechnet, liegen die 8 Bit im Zahlenbereich von −128 bis +127. Da es aber bei der Interpretation der 8-Bit-Daten genau darauf ankommt, ist das V-Flag für vorzeichengerechte Operationen unbedingt erforderlich. Wir werden später noch sehen, wie das zu verstehen ist.

h) Negative (Flag 7)
Wenn als Ergebnis nach einer Operation eine negative Zahl in einem der Datenregister (A,X,Y) steht, wird das N-Flag auf 1 gesetzt. Wir werden noch kennenlernen, wie Bytezahlen mit einem Minuszeichen dargestellt werden.

Sie haben nun an diesen Ausführungen gesehen, daß das Status-Register eine wichtige Rolle innerhalb der CPU spielt.

Sie dürften nun einen groben Überblick über die einzelnen Register, die für das spätere Programmieren wichtig sind, besitzen. Dadurch ist gewährleistet, daß Sie den Befehlssatz der CPU leichter verstehen. Bevor wir auf die Adressierungsarten und dann zum Befehlssatz kommen, müssen wir uns noch ein paar Grundlagen beschaffen.

1.11 Mnemonics – Assembler – Operationscode

Wie wir bereits im Kapitel Programmiersprachen erfahren haben, besteht die Maschinensprache aus einer Hintereinanderordnung von zweiziffrigen Hexadezimalzahlen. Wobei jeweils zwei Hex.-Ziffern ein Byte darstellen. Bevor wir nun ein solches Programm in den Computer eingeben, wird man das Programm zuerst einmal schriftlich auf einen oder mehreren Blatt Papier entwickeln. Da aber ein Programm im Maschinen-Code doch recht unübersichtlich ist, hat man eine Form des Maschinenprogrammierens geschaffen, die aus sogenannten Mnemonics besteht und als Assembler-Code bezeichnet wird. Mnemonik ist eine Gedächtnisstütze durch leicht zu merkende alphabetische Abkürzungen. Für jeden Befehl des Befehlssatzes der CPU gibt es eine mnemonische Kürzel aus 3 Buchstaben. Der Haken dabei ist für uns Deutsche, daß die Mnemonics Abkürzungen aus der englischen Sprache sind. Wir werden später sehen, daß das jedoch auch nicht so schlimm ist. So bedeutet z.B. der Befehl LDA, load accumulator und das heißt übersetzt auf deutsch, lade den Akkumulator mit dem Inhalt eines Speicherplatzes. Sie sehen hier, daß LDA leichter zu merken und zu interpretieren ist als die Maschinensprachenfassung mit der hexadezimalen Zahl AD. Das gesamte Programm, das nun mit solchen Mnemonics geschrieben wird, bezeichnet man als Assemblerprogramm. Natürlich muß es intern wiederum ein Programm geben, das den Assemblercode (Mnemonics) umsetzt in den Maschinen-Code. Dieses Umsetzungsprogramm bezeichnet man einfach als Assembler. Wenn man nun umgekehrt eine Umwandlung vom Maschinen-Code in einen mnemonischen Code vornehmen lassen will, so braucht man dazu ein Programm, das als *Disassembler* bezeichnet wird. Um nun diese Begriffe für Sie zu erleichtern, habe ich anschließend ein kleines Programm mit den entsprechenden Interpretationen dargestellt.

	Assembler			*Maschinencode*			
1	0400	CLC		0400	18		
2	0401	CLD		0401	D8		
3	0402	LDA	#$02	0402	A9	02	
4	0404	STA	$7120	0404	8D	20	71
5	0407	LDA	#$04	0407	A9	04	
6	0409	ADC	$7120	0409	6D	20	71
7	0412	STA	$5000	0412	8D	00	50
	Speicher		Operanden	Speicher		Operanden	
		Mnemonic			Maschinen-code hexad.		

Das Zeichen # im Assemblercode bedeutet, daß eine Konstante unmittelbar in ein Register geladen wird, also kein Zugriff auf den Speicher stattfindet. Das Zeichen $ bedeutet, daß die nachfolgende Zahl eine hexadezimale Zahl ist.

Erklärung des obigen Programms
1) Lösche die Carry Flag im P-Register
2) Lösche Dezimal Flag im P-Register
3) Lade Akku unmittelbar mit Konstante $02
4) Speichere Inhalt des Akku in der Speicherstelle $7120
5) wie 3), Konstante ist hier jedoch $04
6) Addiere Akkuinhalt zum Inhalt des Speicherplatzes $7120 mit Carry
7) Speichere Akkuinhalt im Speicherplatz $7120

Sie können wahrscheinlich momentan nicht viel damit anfangen, aber die Assemblerversion ist sichtlich einfacher zu übersetzen als die nackten Maschinenzahlen.

1.11.1 Der Programmaufbau

Wenn der Prozessor ein Programm abarbeitet, muß er wissen, wo das Programm beginnt bzw. ab welcher Speicherstelle der 1. Befehl steht. Der Programmzähler muß deshalb die Adresse des 1. Befehls enthalten. Von BASIC her wissen wir, daß die Einstellung des Programmzählers auf eine bestimmte Adresse für ein Maschinenprogramm durch „SYS (DEZ. ADRESSE)" erfolgt. Es gibt noch andere Methoden, um den Programmzähler einzustellen. Wir wollen diese vorläufig noch zurückstellen. Wenn nun die Adressierung des Programmzählers geschehen ist, wird ein ab dieser Stelle enthaltenes Programm Byte für Byte abgearbeitet. Das geht solange, bis der Prozessor auf einen Unterbrechungsbefehl oder einen Rückkehrbefehl stößt. Bei der Abarbeitung des Programms wird dabei folgendermaßen vorgegangen:

Es wird der erste Befehlscode geholt und interpretiert. Durch die Interpretation des Codes stellt die CPU fest, daß zu diesem Befehl noch weitere Datenbytes erforderlich sind oder auch nicht.

Ist kein weiteres Byte mehr erforderlich, wird der nächste Befehlscode geholt und interpretiert. Sind durch den Befehlscode weitere Bytes erforderlich, so werden diese nacheinander in die CPU geladen und zusammen mit dem Befehlscode interpretiert. Danach wird der nächste Befehlscode geladen und weiterverarbeitet. Das Ganze setzt sich solange fort, bis eben das Programm beendet ist.

Ein Befehlscode ist ein ausgewählter Befehl aus dem Befehlssatz in codierter (Binär) Form. Man bezeichnet den Befehlscode meist mit Operationscode. Aus diesem Grunde werden wir auch sofort die Abkürzung „Op-Code" einführen. Hinter dem Op-Code können nun kein, ein oder zwei Datenbytes folgen. Ob nun kein oder weitere Datenbytes folgen, hängt mit der Adressierung zusammen. Es gibt nun vier Möglichkeiten, aus denen ein Programmbefehl bestehen kann. Im einfachsten Fall besteht ein Programmbefehl aus dem Op-Code und damit aus einem Programmschritt. Der längste Programmbefehl hat drei Schritte, die aus Op-Code und zwei weiteren Bytes zusammengesetzt sind.

Wir wollen uns die vier Möglichkeiten der Programmbefehlsarten anschauen. Dazu wollen wir annehmen, daß die jeweiligen Programmteile ab der Speicherstelle mit der Adresse $033A (=826) abgespeichert sind.

1. Der Programmbefehl besteht nur aus dem Op-Code

Speicherstelle *Adresse*	*Inhalt*
$0338	...
$0339	...
$033A	Op-Code
$033B	...
$033C	...

Bei solchen Programmbefehlen, die nur aus dem Op-Code bestehen, werden nur in der CPU Daten manipuliert. Es werden also keine Speicherstellen angesprochen. So wird z.B. bei Op-Code „TXA" der Inhalt vom X-Register in den Akkumulator übertragen.

2. Der Programmbefehl besteht aus dem Op-Code und einem Datenbyte

Speicherstelle

Adresse	*Inhalt*
$0338	...
$0339	...
$033A	Op-Code
$033B	Datenbyte
$033C	...
$033D	

In diesem Fall besteht das Folgebyte aus einem festen Wert, aus einer sog. Konstanten. Man bezeichnet das Folgebyte manchmal auch als Operand.

3. Der Programmbefehl besteht aus dem Op-Code und einem Adreßbyte

Speicherstelle

Adresse	*Inhalt*
$0338	...
$0339	...
$033A	Op-Code
$033B	ADL (ADH=$00)
$033C	...
$033D	...

Durch den entsprechenden Op-Code braucht die CPU noch den unteren Teil einer Adresse (ADL). Der obere Teil ist auf der Zeropage festgelegt.

4. Der Programmbefehl besteht aus dem Op-Code und 2 Byte einer Adresse

Speicherstelle

Adresse	*Inhalt*
$0338	...
$0339	...
$033A	Op-Code
$033B	ADL
$033C	ADH
$033D	...
$033E	...

Der entsprechende Op-Code benötigt zwei Folgebytes, die aus dem unteren und dem oberen Teil einer Adresse bestehen.

Ein typischer Ausschnitt aus einem Programm könnte dann z.B. folgendermaßen aussehen:

Speicherstelle

Adresse	*Inhalt*	
$033A	CLC	Op-Code
$033B	STA	Op-Code
$033C	$1E	ADL (ZP)
$033D	ADC	Op-Code
$033E	$2E	ADL
$033F	$71	ADH

Ob Bytes dem Op-Code folgen und wieviel, (bis 2 Bytes), hängt vom Op-Code ab. An sich gibt es etwa 56 verschiedene grundsätzliche Op-Codes. Wie jedoch schon erwähnt wurde, besitzt die CPU etwa 250 Op-Codes. Wie das zu verstehen ist, werden wir gleich sehen. Das hängt nämlich mit der Adressierung zusammen. Wenn wir uns einen Befehl in Assemblerschreibweise aus dem Befehlssatz herausgreifen, z.B. den Befehl „LDA", dann besagt dieser:

Lade den Akkumulator mit dem Inhalt einer adressierten Speicherstelle. Man kann nun eine Speicherstelle auf verschiedene Weise ansprechen. Der Befehl „LDA" hat z.B. 8 verschiedene Möglichkeiten, eine Speicherstelle zu spezifizieren. Wir haben zwar in Assemblerschreibweise 8 mal „LDA", in Wirklichkeit jedoch acht verschiedene Bitmuster und damit auch 8 verschiedene Codes. Da viele der Befehle aus dem Befehlssatz mehrere Adressierungsmethoden beherrschen und dadurch verschiedene Bit-Muster besitzen, besteht eben der gesamte Befehlssatz aus ca. 250 Op-Codes. Vielleicht ist das momentan etwas schwierig zu verstehen. Sie werden sich jedoch nach der Erklärung über die Adressierungsmethoden wesentlich leichter tun.

1.12 Die Adressierung

Wenn man den Inhalt einer Speicherstelle in die CPU übertragen will, so braucht man die Adresse dieser Speicherstelle. Es gibt nun mehrere Methoden, an eine Speicherstelle über deren Adresse zu gelangen. Den Zugriff auf einer Speicherstelle bezeichnet man als Adressierung. Wir werden uns nun die Adressierungsmethoden, die die CPU 6502 beherrscht, im einzelnen anschauen.

1.12.1 Implied (die implizierte Adressierung)

Genaugenommen wird hier gar nichts adressiert. Es finden nur Operationen innerhalb der CPU statt. So wird z.B. durch den Op-Code „SEC" (setze Carry Flag) nur das Bit 0 im Statusregister auf 1 gesetzt. Der Programmbefehl erfordert also keine Adresse, sondern nur den Op-Code.

1.12.2 Immediate (die unmittelbare Adressierung)

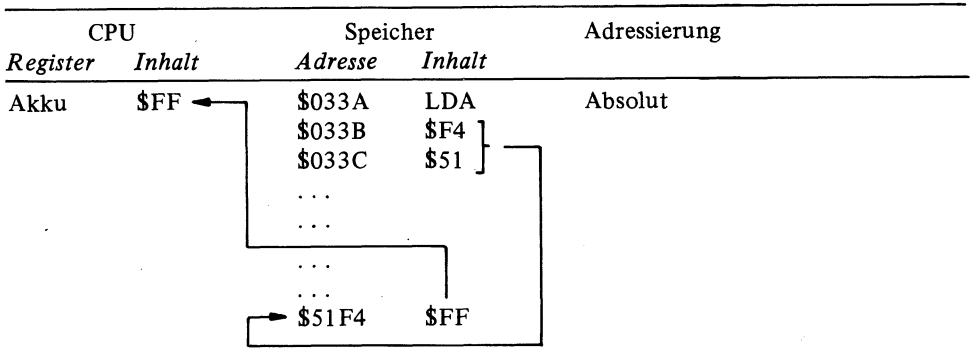
Auch hier wird nichts adressiert. Hinter dem Op-Code steht ein Folgebyte, das keine Adresse darstellt, sondern schon einen festen Wert. Dieses Datenbyte stellt damit eine Konstante dar.

CPU		Speicher		Adressierung
Register	*Inhalt*	*Adresse*	*Inhalt*	
Akku	$03	$033A	LDA	Immediate (#)
		$033B	# $03	

1.12.3 Absolute (Die absolute oder direkte Adressierung)

Diese Methode ist eine echte Adressierung. Hinter dem Op-Code stehen zwei Folgebytes, die aus ADL und ADH zusammengesetzt sind. ADL und ADH bilden zusammen die absolute Adresse im Hauptspeicher.

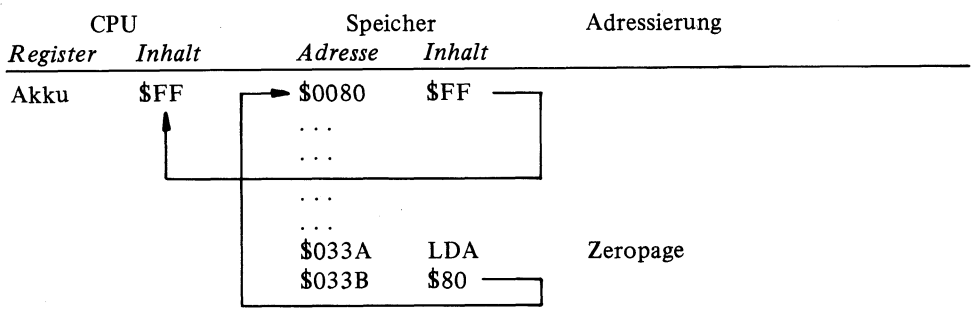
Beispiel: LDA $51F4



1.12.4 Zeropage (Die Nullseitenadressierung)

Der Programmbefehl besteht aus dem Op-Code und dem dahinterstehenden Folgebyte. Das Folgebyte besteht aus dem unteren Teil einer Adresse (ADL). Der obere Teil ist auf die Page 0 festgelegt. Die Seite Null wird im Englischen mit „Zeropage" bezeichnet. Der Vorteil dieser Adressierungsmethode ist dieser, daß Zeit und Platz gespart werden.

Beispiel: LDA $80



1.12.5 Relative (die relative Adressierung)

Der Programmbefehl besteht aus zwei Byte, dem Op-Code und einem Folgebyte. Dieses Folgebyte stellt einen Wert dar, der zum aktuellen Programmzählerinhalt hinzuaddiert wird. D.h., wird der Befehl ausgeführt, dann läuft das Programm bei einer neuen Programmadresse weiter. Es wird praktisch ein Sprung ausgeführt. Den Wert, der zum Pro-

grammzählerinhalt addiert wird, nennt man auch relativen Verschiebewert oder auch Offset-Wert. Normalerweise könnte man mit einem Byte 256 Speicherstellen weiterspringen. Der Offset-Wert wird jedoch von der CPU als Bytewert mit Vorzeichen interpretiert. Somit kann man auch Rückwärtssprünge ausführen. Maximal können darum 127 Stellen vorwärts oder 128 Stellen rückwärts gesprungen werden.

Beispiel für einen Vorwärtssprung: BEC $03

CPU			Speicher		Adressierung
Register	*Inhalt*		*Adresse*	*Inhalt*	
Akku	$FF		$033A	BEQ	Relativ
			$033B	$03	
		0	$033C	...	
		1	$033D	...	
		2	$033E	...	
		3	$033F	LDA	Unmittelbar
			$0340	#$FF	

Beispiel für einen Rückwärtssprung: BEG $FB (= –5)

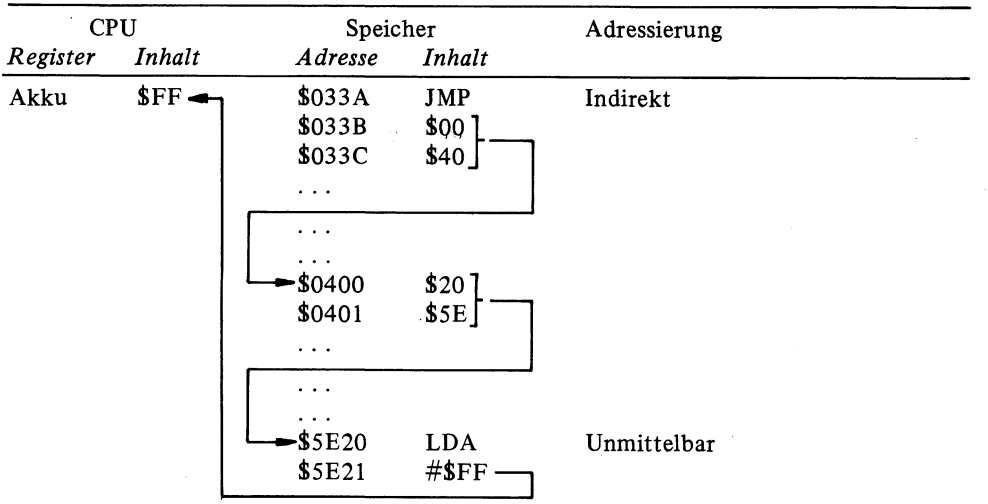
CPU			Speicher		Adressierung
Register	*Inhalt*		*Adresse*	*Inhalt*	
Akku	$FF	5	S0337	LDA	Unmittelbar
		4	S0338	#$FF	
		3	S0339	...	
		2	S033A	BEQ	Relativ
		1	S033B	$FB	
		0	S033C		

An den Beispielen sieht man, daß die Zählung mit dem Offset-Wert stets in der nächsten Speicherstelle nach dem Programmbefehl beginnt, und, wie in der Computertechnik üblich, mit 0.

1.12.6 Indirect (Die indirekte Adressierung)

Die reine indirekte Adressierung gibt es nur für einen Op-Code, den Sprungbefehl „JMP". Hinter dem Op-Code stehen zwei Folgebytes, die eine Adresse mit ADL und ADH darstellen. Der Inhalt dieser Adresse besteht nun nicht aus einem Datenbyte, sondern aus dem unteren Teil einer weiteren Adresse. Der obere Teil ist in der folgenden Speicherstelle enthalten. Erst in dieser weiteren Adresse ist dann letztlich der Verarbeitungswert abgespeichert.

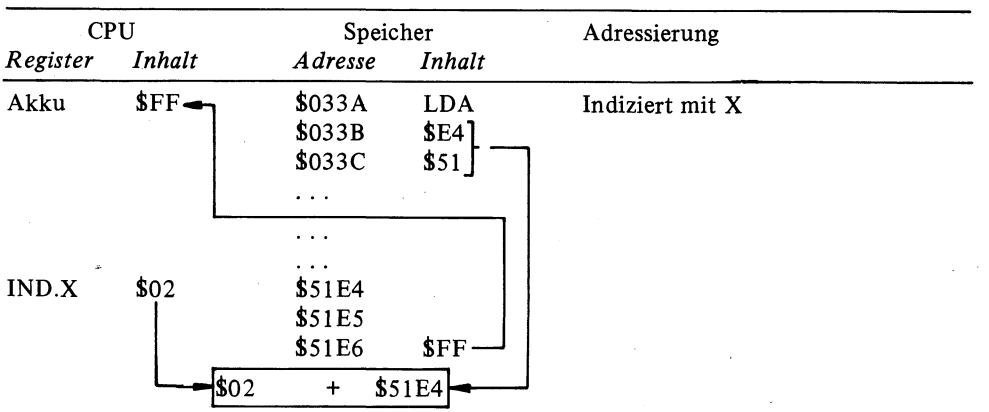
Beispiel: JMP ($0400)



1.12.7 Indexed (Die indizierte Adressierung)

Bei der indizierten Adressierung wird die aktuelle Adresse dadurch ermittelt, daß zu der angesprochenen Adresse ein Wert hinzuaddiert wird, der aus einem der Indexregister stammt.

Beispiel: LDA 51E4,X



Es gibt nun vier indizierte Adressierungsmethoden.

7a) Absolute indexed with X
(Absolut mit X indiziert)

Die aktuelle Adresse wird dadurch ermittelt, daß zur angesprochenen absoluten Adresse der Wert des Indexregisters X addiert wird.

7b) Absolute indexed with Y
(Absolut mit Y indiziert)
Hier geschieht das gleiche wie unter 7a, nur daß die Modifikation mit Y-Register erfolgt.

7c) Zeropage indexed with X
(Nullseitenadresse mit X indiziert)
Die aktuelle Adresse wird dadurch ermittelt, daß zur angesprochenen Zeropage-Adresse der Wert des X-Registers hinzuaddiert wird.

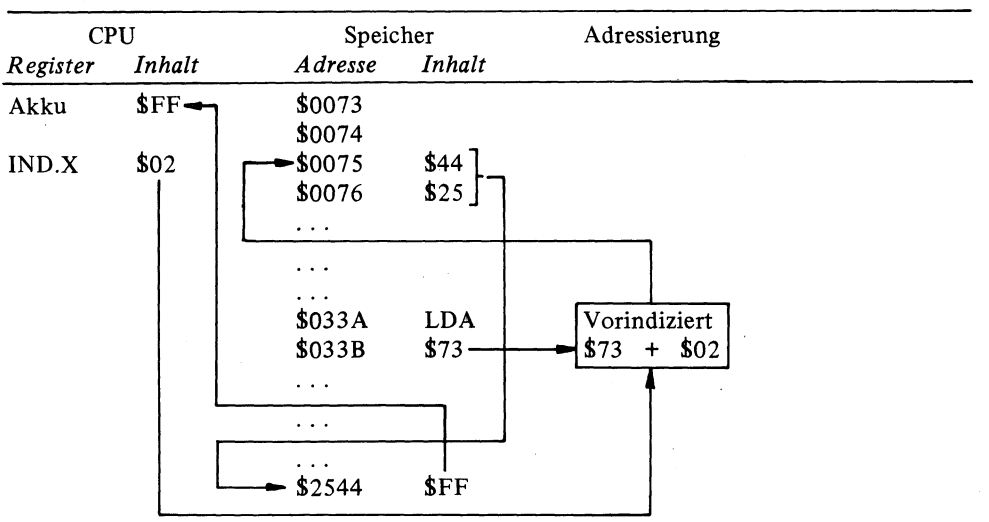
7d) Zeropage indexed with Y
(Nullseitenadresse mit Y indiziert)
Hier geschieht das gleiche wie unter 7c, nur daß die Modifikation mit dem Y-Register erfolgt.

Wir haben jetzt noch zwei Adressierungsmethoden, die aus einer Kombination von indirekter und indizierter Adressierung bestehen. Das sind die Vorindizierung und die Nachindizierung. Bei beiden besteht der Programmbefehl aus dem Op-Code und einem Folgebyte. Das Folgebyte stellt den unteren Teil einer Adresse in der Zeropage dar.

1.12.8 Indexed indirect with X
(indiziert, indirekt mit X → vorindiziert)

Vorindizieren kann man nur mit dem X-Register. Die angesprochene Zeropageadresse wird mit dem Wert des X-Registers modifiziert, (ADL+X). Das Ergebnis stellt nun den unteren Teil einer weiteren Adresse dar. In der nachfolgenden Speicherstelle steht dann der obere Teil dieser weiteren Adresse. Erst in dieser weiteren Adresse steht letztlich der Verarbeitungswert.

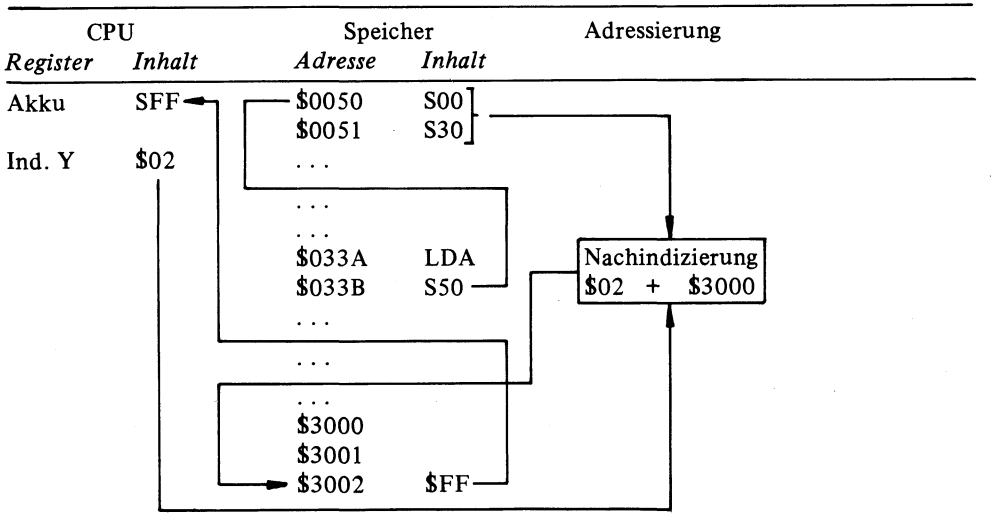
Beispiel: LDA ($73,)X



1.12.9 Indirect indexed with Y
(Indirekt indiziert mit Y = nachindiziert)

Bei der Nachindizierung findet nur das Y-Register Verwendung. Der Programmbefehl besteht wiederum aus zwei Bytes, dem Op-Code und einem Folgebyte. Das Folgebyte stellt den unteren Teil einer Zeropageadresse dar. In dieser Zeropageadresse ist wiederum der untere Teil einer weiteren Adresse enthalten. Der obere Teil dieser weiteren Adresse steht in der nachfolgenden Zeropageadresse. Zum ADL-Wert der weiteren Adresse wird nun der Inhalt des Y-Registers addiert. Das Ergebnis ist die aktuelle Adresse mit dem Verarbeitungswert.

Beispiel: LDA ($50),Y



1.12.10 Maschinen-, Assembler-, Dezimal-, Hexadezimal-, Binärform

Bevor wir uns endgültig mit dem Befehlssatz beschäftigen, wollen wir die Form festigen, wie der gesamte Programmbefehl mit Op-Code und Adressierungsart in verschiedenen Schreibweisen dargestellt wird.

Grundsätzlich wird ein Maschinenprogramm auf dem Papier entwickelt. Als Schreibweise wird die Assemblerform benutzt, wobei die Adressierung hierbei in hexadezimaler Sequenz dargestellt wird. Im VC-64 haben wir ein Monitorprogramm fest abgespeichert, mit dem man Änderungen, aber auch Eingaben von Maschinenprogrammen durchführen kann. Dieses Monitorprogramm zeigt jedoch nur die reine hexadezimale Schreibung auf dem Bildschirm an. Die tatsächliche Verarbeitung erfolgt aber mit Bits in Binärform. Man kann auch mit „Poke“ und „Peek“ von Basic aus auf die Speicherstellen zugreifen. Dann aber müssen die Zahlen und Adressen in dezimaler Form eingegeben werden.

Es werden nun im folgenden alle Sequenzen gegenübergestellt, wobei noch eine Unterteilung in Adressierungsmethoden dargestellt wird. Das gibt noch einmal eine kurze Übersicht über die Adressierung.

Adressierungs-methode	Assemblerform	Adresse		Speicher Inhalt (Beispiele)		
		Dez.	Hex.	Dezimal	Hexadezimal	Binär
Impliziert	CLC	826	$033A	24	$18	%00011000
Unmittelbar	LDA #$05	826	$033A	169	$A9	%10101001
		827	$033B	5	$05	%00000101
Absolut	LDA $8000	826	$033A	173	$AD	%10101101
		827	$033B	0	$00	%00000000
		828	$033C	128	$80	%10000000
Zeropage	LDA $40	826	$033A	165	$A5	%10101001
		827	$033B	64	$40	%01000000
Relative	BCS $04	826	$033A	176	$B0	%10110000
		827	$033B	4	$04	%00000100
Indirekt	IMP ($0400)	826	$033A	108	$6C	%01101100
		827	$033B	0	$00	%00000000
		828	$033C	4	$04	%00000100
Absolut, X	LDA $5E1A, X	826	$033A	189	$BD	%10111101
		827	$033B	26	$1A	%00011010
		828	$033C	94	$5E	%01011110
Absolut, Y	LDA $5E1A, Y	826	$033A	185	$B9	%10111001
		827	$033B	26	$1A	%00011010
		828	$033C	94	$5E	%01011110
Zeropage, X	LDA $02, X	826	$033A	181	$B5	%10110101
		827	$033B	2	$02	%00000010
Zeropage, Y	LDA $02, Y	826	$033A	182	$B6	%10110110
		827	$033B	2	$02	%00000010
Vorindiziert	LDA ($45), X	826	$033A	177	$B1	%10110001
		827	$033B	69	$45	%01000101
Nachindiziert	LDA ($45,),Y	826	$033A	161	$A1	%10100001
		827	$033B	69	$45	%01000101

An diesen Beispielen sehen Sie nun recht gut, wie die Adressierung im Assemblerformat aussieht, während der Inhalt der Speicherstellen in binärer, hexadezimaler und dezimaler Form dargestellt ist.

1.13 Der Befehlssatz der CPU 6502

Der Befehlssatz wird durch die einzelnen Adressierungsmethoden stark erweitert. Wir werden jetzt die einzelnen OP-Codes des Befehlssatzes im einzelnen betrachten. Dabei werden wir lernen, welche Auswirkungen die Befehle auf den Speicher und die CPU-Register haben. Da die einzelnen Befehle teilweise mit mehreren Adressierungsmethoden verbunden sind, werde ich im folgenden für die Adressierungsmethoden nur deren folgende Abkürzung verwenden:

IMP	=	impliziert
IM	=	immediate (unmittelbar)
ABS	=	absolut
R	=	relativ
ZP	=	zeropage (nullseitig)
IND	=	indirekt
ABX	=	absolut indiziert mit X
ABY	=	absolut indiziert mit Y
ZPX	=	zeropage indiziert mit X
ZPY	=	zeropage indiziert mit Y
(IND,)X	=	vorindiziert
(IND),Y	=	nachindiziert

In der nachfolgenden Erklärung der einzelnen Befehle wird die Erklärung selbst und folgendes Schema angegeben:

Abgekürzte Schreibweise: *)

Beeinflußte Flags im Statusregister:

Assemblerform / OP-Code $-Form / Adressierung / Byte / Zyklen

---

Zudem werden gleichartige Befehle in Gruppen zusammengefaßt.

1.13.1 Datenaustauschbefehle

Bei den Datenaustauschbefehlen gehen die Daten vom Speicher zur CPU oder von der CPU zum Speicher oder es erfolgt innerhalb der CPU zwischen den Registern der Datenaustausch.

LDA

Der Akkumulator wird mit dem Inhalt einer Speicherstelle geladen. Nach erfolgter Befehlsausführung steht als Inhalt im Akkumulator das gleiche Bitmuster wie in der spezifizierten Speicherstelle.

Abgekürzte Schreibweise: M → A

Beeinflußte Flags im Statusregister: N, Z

Assemblerform	OP-Code $-Form	Adressierung	Byte	Zyklen
LDA #Oprd	A9	IM	2	2
LDA Oprd	A5	ZP	2	3
LDA Oprd,X	B5	ZPX	2	4
LDA Oprd	AD	ABS	3	4
LDA Oprd,X	BD	ABX	3	4*
LDA Oprd,Y	B9	ABY	3	4*
LDA (Oprd,)X	A1	(IND,)X	2	6
LDA (Oprd),Y	B1	(IND),Y	2	5*

*Zum Zyklus ist 1 zu addieren, wenn die Pagegrenze überschritten wird

LDX

Das Indexregister X wird mit dem Inhalt einer Speicherstelle geladen.

Abgekürzte Schreibweise: M → X

Beeinflußte Flags im Statusregister: N, Z

Assemblerform	OP-Code $-Form	Adressierung	Byte	Zyklen
LDX #Oprd	A2	IM	2	2
LDX Oprd	A6	ZP	2	3
LDX Oprd,Y	B6	ZPY	2	4
LDX Oprd	AE	ABS	3	4
LDX Oprd,Y	BE	ABY	3	4*

*Zum Zyklus ist 1 zu addieren, wenn die Pagegrenze überschritten wird

LDY

Das Indexregister Y wird mit dem Inhalt einer Speicherstelle geladen.

Abgekürzte Schreibweise: M → Y

Beeinflußte Flags im Statusregister: N, Z

Assemblerform	OP-Code $-Form	Adressierung	Byte	Zyklen
LDY #Oprd	A0	IM	2	2
LDY Oprd	A4	ZP	2	3
LDY Oprd,X	B4	ZPX	2	4
LDY Oprd	AC	ABS	3	4
LDY Oprd,X	BC	ABX	3	4

*Zum Zyklus ist 1 zu addieren, wenn die Pagegrenze überschritten wird

*) Zu erwähnen wäre noch, daß Angaben in Klammern, der Inhalt von, bedeutet. Aber auch Zeiger (Vektoren) werden in Klammern geschrieben.

STA

Der Akkumulatorinhalt wird in einer Speicherstelle abgespeichert. Nach Befehlsausführung steht in einer Speicherstelle dasselbe Bitmuster wie im Akku. Ein etwaiger alter Wert in dieser Speicherzelle wird dadurch überschrieben.

Abgekürzte Schreibweise: A → M

Beeinflußte Flags im Statusregister: keine

Assemblerform		OP-Code $-Form	Adressierung	Byte	Zyklen
STA	Oprd	85	ZP	2	3
STA	Oprd,X	95	ZPX	2	4
STA	Oprd	8D	ABS	3	4
STA	Oprd,X	9D	ABX	3	5
STA	Oprd,Y	99	ABY	3	5
STA	(Oprd,)X	81	(IND,)X	2	6
STA	(Oprd),Y	91	(IND),Y	2	6

STX

Der Inhalt des Indexregisters X wird in einer Speicherstelle abgespeichert.

Abgekürzte Schreibweise: X → M

Beeinflußte Flags im Statusregister: keine

Assemblerform		OP-Code $-Form	Adressierung	Byte	Zyklen
STX	Oprd	86	ZP	2	3
STX	Oprd,Y	96	ZPY	2	4
STX	Oprd	8E	ABS	3	4

STY

Der Inhalt des Indexregisters Y wird in einer Speicherstelle abgespeichert.

Abgekürzte Schreibweise: Y → M

Beeinflußte Flags im Statusregister: keine

Assemblerform		OP-Code $-Form	Adressierung	Byte	Zyklen
STY	Oprd	84	ZP	2	3
STY	Oprd,X	94	ZPX	2	4
STY	Oprd	8c	ABS	3	4

TAX

Der Akkuinhalt wird in das Indexregister X transferiert.

Abgekürzte Schreibweise:	A → X			
Beeinflußte Flags im Statusregister:	N, Z			
Assemblerform	OP-Code $-Form	Adressierung	Byte	Zyklen
TAX	AA	IMP	1	2

TAY

Der Akkuinhalt wird in das Indexregister Y transferiert.

Abgekürzte Schreibweise:	A → Y			
Beeinflußte Flags im Statusregister:	N, Z			
Assemblerform	OP-Code $-Form	Adressierung	Byte	Zyklen
TAY	A8	IMP	1	2

TSX

Der Stackregisterinhalt wird in das Indexregister X transferiert.

Abgekürzte Schreibweise:	S → X			
Beeinflußte Flags im Statusregister:	N, Z			
Assemblerform	OP-Code $-Form	Adressierung	Byte	Zyklen
TSX	BA	IMP	1	2

TXA

Der Inhalt des Indexregisters X wird in den Akku transferiert.

Abgekürzte Schreibweise:	X → A			
Beeinflußte Flags im Statusregister:	N, Z			
Assemblerform	OP-Code $-Form	Adressierung	Byte	Zyklen
TXA	8A	IMP	1	2

TXS

Der Inhalt des Indexregisters X wird in das Stackregister transferiert.

Abgekürzte Schreibweise:	X → S			
Beeinflußte Flags im Statusregister:	keine			
Assemblerform	OP-Code $-Form	Adressierung	Byte	Zyklen
TXS	9A	IMP	1	2

TYA

Der Inhalt des Indexregisters Y wird in den Akku transferiert.

Abgekürzte Schreibweise:	Y → A			
Beeinflußte Flags im Statusregister:	N, Z			
Assemblerform	OP-Code $-Form	Adressierung	Byte	Zyklen
TYA	98	IMP	1	2

Wie aus den Darstellungen zu ersehen ist, wird das Statusregister nicht beeinflußt, wenn die Daten von der CPU zum Speicher übertragen werden. Ansonsten wird nur das Negativ-Flag und Zero-Flag folgendermaßen beeinflußt: Wenn das Bit 7 des Datenbyte 1 ist (z.B. negative Zahl), dann steht nach Befehlsausführung im N-Flag eine 1, sonst 0. Ist nun der Wert des Datenbytes Null (alle Bits sind 0), dann geht nach Befehlsausführung das Zero-flag auf 1, ansonsten bleibt es auf Null. Es werden somit die N- und Z-Flag automatisch nach den entsprechenden Transportbefehlen gesetzt oder zurückgesetzt.

1.13.2 Flag-Befehle

Um einzelne Flags des Statusregisters manipulieren zu können, beherrscht die CPU Befehle, um die Carry-, Decimal Mode-, Interrupt disable- und Overflow-Flag setzen oder löschen zu können. Die anderen Flags werden durch die entsprechenden Operationen automatisch behandelt.

CLC

Das Carry-Flag wird auf Null gesetzt (gelöscht).

Abgekürzte Schreibweise:	0 → C			
Beeinflußte Flags im Statusregister:	C			
Assemblerform	OP-Code $-Form	Adressierung	Byte	Zyklen
CLC	18	IMP	1	2

CLD

Das Decimal Mode-Flag wird auf Null gesetzt (gelöscht).

Abgekürzte Schreibweise:	0 → D			
Beeinflußte Flags im Statusregister:	D			
Assemblerform	OP-Code $-Form	Adressierung	Byte	Zyklen
CLD	D8	IMP	1	2

Damit wird auf die duale Betriebsart umgeschaltet.

CLI

Das Interrupt disable-Flag wird auf Null gesetzt (gelöscht).

Abgekürzte Schreibweise:	0 → I			
Beeinflußte Flags im Statusregister:	I			
Assemblerform	OP-Code $-Form	Adressierung	Byte	Zyklen
CLI	58	IMP	1	2

CLV

Das Overflow-Flag wird auf Null gesetzt (gelöscht).

Abgekürzte Schreibweise:	0 → V			
Beeinflußte Flags im Statusregister:	V			
Assemblerform	OP-Code $-Form	Adressierung	Byte	Zyklen
CLV	B8	IMP	1	2

SEC

Das Carry-Flag wird auf 1 gesetzt.

Abgekürzte Schreibweise:	1 → C			
Beeinflußte Flags im Statusregister:	C			
Assemblerform	OP-Code $-Form	Adressierung	Byte	Zyklen
SEC	38	IMP	1	2

SED

Das Decimal Mode-Flag wird auf 1 gesetzt.

Abgekürzte Schreibweise:		1 → D		
Beeinflußte Flags im Statusregister:		D		
Assemblerform /	OP-Code $-Form /	Adressierung /	Byte /	Zyklen
SED	F8	IMP	1	2

Damit wird auf die dezimale Betriebsart umgeschaltet. In dem Zusammenhang ist wichtig zu wissen, daß die CPU 6502 ein modifiziertes Addierwerk hat. Dadurch werden Zahlen zugelassen, die als zwei binär codierte Dezimalzahlen (BCD-Code) zu je vier Bit dargestellt sind.

SEI

Das Interrupt disable-Flag wird auf 1 gesetzt. D.h., die CPU kann keine Unterbrechungen außer dem NMI-Impuls annehmen.

Abgekürzte Schreibweise:		1 → I		
Beeinflußte Flags im Statusregister		I		
Assemblerform /	OP-Code $-Form /	Adressierung /	Byte /	Zyklen
SEI	78	IMP	1	2

1.13.3 Stack-Befehle

An sich gehören die 4 Stackbefehle zu den Transportbefehlen. Sie werden hier jedoch extra behandelt. Wie bereits bei der Erläuterung der CPU-Register erklärt wurde, werden die Daten nicht direkt im Stackregister gespeichert, sondern in den Speicherstellen der Page 1. Im Stackregister selbst steht nur die Adresse der Daten. Das Stackregister wird auch als Stapelzeiger bezeichnet, dient also nur als indirekter Zeiger.

PHA

Der Akkuinhalt wird auf dem Stapel (Page 1) gelegt.

Abgekürzte Schreibweise:		A ↓		
Beeinflußte Flags im Statusregister:		keine		
Assemblerform /	OP-Code $-Form /	Adressierung /	Byte /	Zyklen
PHA	48	IMP	1	3

PHP

Der Statusregisterinhalt wird auf dem Stapel (Page 1) gelegt.

Abgekürzte Schreibweise	P ↓			
Beeinflußte Flags im Statusregister:		keine		
Assemblerform /	OP-Code $-Form /	Adressierung /	Byte /	Zyklen
PHP	08	IMP	1	3

Die Schreibbefehle PHA und PHP dekrementieren den Stapelzeiger. Nach erfolgter Befehlsausführung stehen im Stapel an oberster Stelle ($01FF) die Daten, die übergeben wurden.

PLA

Das oberste Datenbyte wird aus dem Stapel geholt und in den Akku übertragen.

Abgekürzte Schreibweise	A ↑			
Beeinflußte Flags im Statusregister:		N, Z		
Assemblerform /	OP-Code $-Form /	Adressierung /	Byte /	Zyklen
PLA	68	IMP	1	4

PLP

Das oberste Datenbyte wird aus dem Stapel geholt und in das Statusregister übertragen.

Abgekürzte Schreibweise:	P ↑			
Beeinflußte Flags im Statusregister:		je nach Zustand des Datenbyte		
Assemblerform /	OP-Code $-Form /	Adressierung /	Byte /	Zyklen
PLP	28	IMP	1	4

Mit den beiden Befehlen PLA und PLP geschieht die Umkehrung der Befehle PHA und PHP. Es werden von der Page 1 die Daten in die entsprechenden CPU-Register übertragen, wobei der Stapelzeiger auf die oberste Adresse zeigt, von der aus dann die Übertragung stattfindet.

1.13.4 Logische und arithmetische Operationen

ADC

Zum Akkuinhalt wird der adressierte Speicherstelleninhalt und das Carry-Flag addiert.

Abgekürzte Schreibweise:	A + M + C → A, C
Beeinflußte Flags im Statusregister:	N, Z, C, V

Assemblerform	OP-Code $-Form	Adressierung	Byte	Zyklen
ADC #Oprd	69	IM	2	2
ADC Oprd	65	ZP	2	3
ADC Oprd,X	75	ZPX	2	4
ADC Oprd	6D	ABS	3	4
ADC Oprd,X	7D	ABX	3	4*
ADC Oprd,Y	79	ABY	3	4*
ADC (Oprd,)X	61	(IND,)X	2	6
ADC (Oprd),Y	71	(IND),Y	2	5*

* Zum Zyklus ist 1 zu addieren, wenn die Pagegrenze überschritten wird

Dieser Befehl addiert zum Akkuinhalt den Inhalt der angegebenen Speicherstelle. Dazu wird noch das Carry-Bit (Statusbit 0) der vorhergehenden Operation addiert. Wird z.B. mit dem ADC-Befehl eine Addition eingeleitet, muß dem ADC-Befehl der CLC-Befehl (C-Flag löschen) stehen, da sonst ein falsches Ergebnis entstehen kann. Im einfachsten Fall werden zwei 1-Byte Zahlen (Werte 0 – 255) addiert.

Zum Beispiel:

Binär	Dezimal	
0000 1010	10	1. Zahl
0111 1100	124	2. Zahl
+ 1	1	Carry
1000 0111	135	Ergebnis mit Carry

An diesem Beispiel sieht man, daß im Statusregister vor dem ADC-Befehl im Carry-Bit eine 1 gestanden hat, und somit nicht als Ergebnis 134, sondern 135 (Addition + Carry) entsteht. Damit ist natürlich das Ergebnis falsch. Es gibt nun zwei Möglichkeiten, um einen derartigen Fehler zu eliminieren. Die umständlichere Methode wäre, vor der Addition das Carry-Flag auf seinen Zustand abzufragen, um dann nach der Addition am Ergebnis eine eventuelle Korrektur anzubringen. Die elegantere Methode ist jedoch, vor der Addition das Carry-Flag mit CLC zu löschen (auf 0 setzen).

Das Carry-Flag wird bei Additionen in der CPU so behandelt, als wäre es ein 9. Ergebnisbit. D.h., wenn bei einer Addition das Ergebnis im Akku die dezimale Zahl 255 ($FF =

B 1111 1111) überschreitet, steht nach Ausführung im Carry-Flag eine 1. Bleibt die Ergebniszahl unter 255, dann steht nach Ausführung im Carry-Bit 0.
Beispiel 1:

				Akku	Carry
LDA	1. Zahl	34	B	0010 0010	X (beliebig)
CLC					0
ADC	2. Zahl	4	B	0000 0100	0
STA	Ergebnis	38	B	0010 0110	0

Beispiel 2:

LDA	1. Zahl	245	B	1111 0101	X (beliebig)
CLC					0
ADC	2. Zahl	25	B	0001 1001	0
STA	Ergebnis	14	B	0000 1110	1 (Bereichsüberschreitung)

Am Beispiel 2 sieht man, daß eine Überschreitung stattgefunden hat und das Ergebnis als 256 + 14 interpretiert werden muß. Im Akku steht also 14 und im Statusregister beim Carry-Flag eine 1. Diese 1 sagt somit aus, daß als Ergebnis der Zahlenbereich bis 255 überschritten wurde. Das Ergebnis ist also 256 + 14 = 270.

Der Programmierer sollte also nach Operationen, bei denen eine Bereichsüberschreitung möglich ist, immer im Programm nach solchen Operationen eine Korrekturroutine anbringen. Dieses Unterprogramm sollte die Carry-Flag abfragen und bei Überschreitung des Zahlenbereichs die entsprechende Korrektur durchführen.

Addition bei größeren Zahlen

Wenn man z.B. 16-stellige Binärzahlen addiert, wird folgendes Verfahren angewandt:

		1.Zahl	2.Zahl	Ergebnis
Niederwertiges Byte	=	L1	L2	EL
Höherwertiges Byte	=	H1	H2	EH

CLC	lösche Carry-Flag
LDA L1	lade Akku mit L1
ADC L2	addiere zum Akku L2 + Carry (= 0)
STA EL	speichere niederwertiges Ergebnisbyte
LDA H1	lade Akku mit H1
ADC H2	addiere zum Akku H2 + Carry (je nach Ergebnis aus EL)
STA EH	speichere höherwertiges Ergebnisbyte

Durch dieses Verfahren können Zahlen mit beliebiger Größe verarbeitet werden. Der Übertrag von einem niederwertigeren zu einem nächsthöheren Byte erfolgt durch das Carry-Flag automatisch.

Vorzeichengerechte Operationen

An dieser Stelle soll noch einmal auf die Darstellung negativer Zahlen eingegangen werden.

Es ist eine Festlegung, daß bei vorzeichengerechter Arithmetik in einem Byte das vorderste Bit das Vorzeichen darstellt. Dabei bedeutet eine 1 im Bit 7 das Minuszeichen und das Pluszeichen wird durch 0 ausgedrückt. Das vorderste Bit wird auch Vorzeichen-Bit genannt. Bei unseren 8-Bit Daten kommt somit nur Bit 7 in Frage. Das kommt schon damit zum Ausdruck, daß im Statusregister das Bit 7 die Negativ-Flag (Vorzeichen-Flag) ist. Wenn ein Datenbyte als vorzeichenbehaftetes Bitmuster interpretiert wird, erfassen die restlichen 7 Bits nur noch einen Zahlenbereich von 128 Bitkombinationen, deren Werte von 0 bis +127 oder von −1 bis −128 reichen, je nach Vorzeichen.

Negative Binärzahlen werden folgendermaßen dargestellt:

+15 = 0000 1111

Die positive Zahl wird komplementiert (Einerkomplement)

1111 0000

dann wird eine 1 addiert (Zweierkomplement)

−15 = 1111 0001

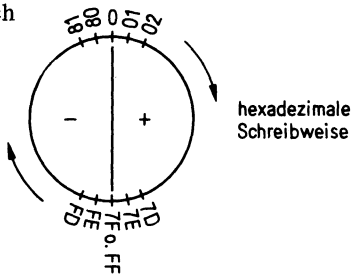
und das Ergebnis ist z.B. in der obigen Darstellung die negative Zahl −15 ($F1 = B 1111 0001).

Würde man nun vom Zahlenbereich der 128 Bitkombinationen das Zweierkomplement bilden, so haben wir den negativen Zahlenbereich mit den Werten von −1 bis −128. Insgesamt gesehen haben wir jedoch den Zahlenbereich von −128 bis +127 und somit 255 Werte.

Dezimal	Binär	Hexadezimal
127	0111 1111	7F
126	0111 1110	7E
.	.	.
.	.	.
.	.	.
2	0000 0010	02
1	0000 0001	01
0	0000 0000	00
−1	1111 1111	FF
−2	1111 1110	FE
.	.	.
.	.	.
.	.	.
−127	1000 0001	81
−128	1000 0000	80

Am besten läßt sich der Zahlenbereich mit Vorzeichen in einem Kreis darstellen.

Kreisschema mit vorzeichengerechtem Bereich



Da nun das Carry-Flag bei einem Übertrag an vorderster Stelle zu finden ist, könnte bei vorzeichengerechter Arithmetik ein falsches Ergebnis entstehen, wenn das Carry-Flag mit interpretiert wird. Deshalb hat das Carry-Bit bei vorzeichengerechten Operationen keine Bedeutung. An dessen Stelle tritt nun das Overflow-Flag (V-Flag). Dieses Flag zeigt nun an, wenn ein Übertrag von 7-Bit-Operationen aufgetreten ist. Deshalb wird das V-Flag im Statusregister auch auf Bit 6 geführt und davor auf Bit 7 die Negativ-Flag (N-Flag). Das V-Flag wird immer dann automatisch auf 1 gesetzt, wenn aufgrund einer arithmetischen Operation ein Vorzeichenwechsel stattgefunden hat. Das geschieht immer, auch wenn keine arithmetische Definition der Datenbytes vorliegt.

Beispiel einer vorzeichengerechten Addition, wobei eine Bereichsüberschreitung auftritt:

LDA	–128	1000 0000
ADC	– 2	1111 1110
Ergebnis	+126	0111 1110

Das Ergebnis würde normalerweise als +126 interpretiert werden. Durch den Vorzeichenwechsel wurde aber das V-Flag auf 1 gesetzt. Dadurch weiß der Programmierer, daß eine Bereichsüberschreitung vorliegt und somit ein Korrekturprogramm angebracht werden muß.

SBC

Der adressierte Speicherinhalt und das komplementierte Carry (Borrow) wird vom Akkuinhalt subtrahiert.

Abgekürzte Schreibweise: A – M – C → A

Beeinflußte Flags im Statusregister: N, Z, C, V

Assemblerform	OP-Code $-Form	Adressierung	Byte	Zyklen
SBC #Oprd	E9	IM	2	2
SBC Oprd	E5	ZP	2	3
SBC Oprd,X	F5	ZPX	2	4
SBC Oprd	ED	ABS	3	4
SBC Oprd,X	FD	ABX	3	4*
SBC Oprd,Y	F9	ABY	3	4*
SBC (Oprd,)X	E1	(IND,)X	2	6
SBC (Oprd),Y	F1	(IND),Y	2	5*

* Zum Zyklus ist 1 zu addieren, wenn die Pagegrenze überschritten wird

Dieser Befehl subtrahiert den Inhalt einer angegebenen Speicherstelle und das komplementierte Carry-Bit durch Zweierkomplement Addition vom Inhalt des Akkumulators.

Nach Ausführung steht im Akku die Differenz. Das komplementierte Carry-Bit ist einfach ein Einerkomplement des normalen Carry-Bit und wird auch als Borrow-Bit bezeichnet. Die Carry-Flag wird also auf 1 gesetzt, wenn kein Übertrag entstanden bzw. das Ergebnis kleiner Null (−) ist. Die Behandlung der Carry-Flag tritt beim SBC-Befehl genau umgekehrt auf, wie es beim ADC-Befehl war. Ansonsten werden die N-Flag und die V-Flag genauso beeinflußt wie bei der Addition.

AND

Logische UND-Verknüpfung zwischen dem Inhalt einer Speicherstelle und dem Akkuinhalt.

Abgekürzte Schreibweise:		A M → A		
Beeinflußte Flags im Statusregister:		N, Z		
Assemblerform /	OP-Code $-Form /	Adressierung /	Byte /	Zyklen
AND #Oprd	29	IM	2	2
AND Oprd	25	ZP	2	3
AND Oprd,X	35	ZPX	2	4
AND Oprd	2D	ABS	3	4
AND Oprd,X	3D	ABX	3	4*
AND Oprd,Y	39	ABY	3	4*
AND (Oprd,)X	21	(IND,)X	2	6
AND (Oprd),Y	31	(IND),Y	2	5

* Zum Zyklus ist 1 zu addieren, wenn die Pagegrenze überschritten wird

Hierbei werden jeweils die einzelnen Bits vom Speicher und Akku als logische UND-Funktion verknüpft. Das Ergebnis steht nach Ausführung im Akku. Sinn des AND-Befehls ist es z.B., irgendein Bit in einer adressierten Speicherstelle auf Null zu setzen.

Beispiel:

```
LDA   111i 0000   1. Operand
AND   1110 1111   2. Operand
-------------------------------
STA   1110 0000   Ergebnis
```

Das i im ersten Operanden demonstriert, daß es sich um irgendeinen (unbekannten) Bit-Zustand handelt. Auf jeden Fall wird dieses Bit durch eine 0 im 2. Operanden und der UND-Funktion auf Null gesetzt.

ORA

Logische ODER-Verknüpfung zwischen dem Inhalt einer Speicherstelle und dem Akkuinhalt.

Abgekürzte Schreibweise:	A v M → A
Beeinflußte Flags im Statusregister:	N, Z

Assemblerform	OP-Code $-Form	Adressierung	Byte	Zyklen
ORA #Oprd	09	IM	2	2
ORA Oprd	05	ZP	2	3
ORA Oprd,X	15	ZPX	2	4
ORA Oprd	0D	ABS	3	4
ORA Oprd,X	1D	ABX	3	4*
ORA Oprd,Y	19	ABY	3	4*
ORA (Oprd,)X	01	(IND,)X	2	6
ORA (Oprd),Y	11	(IND),Y	2	5

* Zum Zyklus ist 1 zu addieren, wenn die Pagegrenze überschritten wird

Wie beim AND-Befehl findet durch den ORA-Befehl eine logische Funktion statt. Dadurch kann z.B. ein beliebiges Bit einer Speicherstelle auf 1 gesetzt werden.
Beispiel:

```
LDA    000i 0000   1. Operand
ORA    0001 0000   2. Operand
-----------------
STA    0001 0000   Ergebnis
```

EOR

Logische EXKLUSIV-ODER-Verknüpfung zwischen dem Inhalt einer Speicherstelle und dem Akkuinhalt.

Abgekürzte Schreibweise:	A v M → A
Beeinflußte Flags im Statusregister:	N, Z

Assemblerform	OP-Code $-Form	Adressierung	Byte	Zyklen
EOR #Oprd	49	IM	2	2
EOR Oprd	45	ZP	2	3
EOR Oprd,X	55	ZPX	2	4
EOR Oprd	4D	ABS	3	4
EOR Oprd,X	5D	ABX	3	4*
EOR Oprd,Y	59	ABY	3	4*
EOR (Oprd,)X	41	(IND,)X	2	6
EOR (Oprd),Y	51	(IND),Y	2	5*

* Zum Zyklus ist 1 zu addieren, wenn die Pagegrenze überschritten wird

Mit der EOR-Funktion kann z.B. ein Byte negiert werden.
Beispiel:

LDA	0010 1011	1. Operand
EOR	1111 1111	2. Operand
STA	1101 0100	Ergebnis

CMP

Der Speicherinhalt wird mit dem Akkuinhalt verglichen.

Abgekürzte Schreibweise: A – M

Beeinflußte Flags im Statusregister: N, Z, C

Assemblerform	OP-Code $-Form	Adressierung	Byte	Zyklen
CMP #Oprd	C9	IM	2	2
CMP Oprd	C5	ZP	2	3
CMP Oprd,X	D5	ZPX	2	4
CMP Oprd	CD	ABS	3	4
CMP Oprd,X	DD	ABX	3	4*
CMP Oprd,Y	D9	ABY	3	4*
CMP (Oprd,)X	C1	(IND,)X	2	6
CMP (Oprd),Y	D1	(IND),Y	2	5*

* Zum Zyklus ist 1 zu addieren, wenn die Pagegrenze überschritten wird

Bei diesem Befehl wird ein Speicherbyte vom Akkuinhalt abgezogen. Nach Ausführung hat sich der Akkuinhalt jedoch nicht verändert. Der CMP-Befehl beeinflußt also nur die entsprechenden Flags im Statusregister.

CPX

Der Speicherinhalt wird mit dem Inhalt des Indexregisters X verglichen.

Abgekürzte Schreibweise: X – M

Beeinflußte Flags im Statusregister: N, Z, C

Assemblerform	OP-Code $-Form	Adressierung	Byte	Zyklen
CPX #Oprd	E0	IM	2	2
CPX Oprd	E4	ZP	2	3
CPX Oprd	EC	ABS	3	4

CPY

Der Speicherinhalt wird mit dem Inhalt des Indexregisters Y verglichen.

Abgekürzte Schreibweise:			Y − M		
Beeinflußte Flags im Statusregister:			N, Z, C		
Assemblerform	OP-Code	$-Form	Adressierung	Byte	Zyklen
CPY #Oprd	C0		IM	2	2
CPY Oprd	C4		ZP	2	3
CPY Oprd	CC		ABS	3	4

BIT

Prüfe Bits im Speicher mit Akkuinhalt.

Abgekürzte Schreibweise:			A ∧ M, $M_7 \rightarrow N$, $M_6 \rightarrow V$		
Beeinflußte Flags im Statusregister:			N(M_7), Z, V(M_6)		
Assemblerform	OP-Code	$-Form	Adressierung	Byte	Zyklen
BIT Oprd	24		ZP	2	3
BIT Oprd	2C		ABS	3	4

Der Bit-Befehl führt eine UND-Verknüpfung zwischen Speicherstelleninhalt und dem Akkuinhalt durch. Das Ergebnis beeinflußt jedoch den Akkuinhalt nicht. Es werden nur die Flags manipuliert.

Bit 6 und 7 werden ins Statusregister übertragen. Wenn das Ergebnis der UND-Funktion Null ist, wird die Zeroflag gesetzt, ansonsten auf 0 zurückgesetzt.

1.13.5 Inkrement- und Dekrement-Befehle

DEC

Der Inhalt einer adressierten Speicherstelle wird um 1 dekrementiert (erniedrigt).

Abgekürzte Schreibweise:			M − 1 → M		
Beeinflußte Flags im Statusregister:			N, Z		
Assemblerform	OP-Code	$-Form	Adressierung	Byte	Zyklen
DEC Oprd	C6		ZP	2	5
DEC Oprd,X	D6		ZPX	2	6
DEC Oprd	CE		ABS	3	6
DEC Oprd,X	DE		ABX	3	7

DEX

Der Inhalt des Indexregisters X wird um 1 dekrementiert (erniedrigt).

Abgekürzte Schreibweise:	X – 1 → X
Beeinflußte Flags im Statusregister:	N, Z

Assemblerform	OP-Code $-Form	Adressierung	Byte	Zyklen
DEX	CA	IMP	1	2

DEY

Der Inhalt des Indexregisters Y wird um 1 dekrementiert (erniedrigt).

Abgekürzte Schreibweise:	Y – 1 → Y
Beeinflußte Flags im Statusregister:	N, Z

Assemblerform	OP-Code $-Form	Adressierung	Byte	Zyklen
DEY	88	IMP	1	2

INC

Der Inhalt einer adressierten Speicherstelle wird um 1 erhöht (inkrementiert).

Abgekürzte Schreibweise:	M + 1 → M
Beeinflußte Flags im Statusregister:	N, Z

Assemblerform	OP-Code $-Form	Adressierung	Byte	Zyklen
INC Oprd	E6	ZP	2	5
INC Oprd,X	F6	ZPX	2	6
INC Oprd	EE	ABS	3	6
INC Oprd,X	FE	ABX	3	7

INX

Der Inhalt des Indexregisters X wird um 1 erhöht (inkrementiert).

Abgekürzte Schreibweise:	X + 1 → X
Beeinflußte Flags im Statusregister:	N, Z

Assemblerform	OP-Code $-Form	Adressierung	Byte	Zyklen
INX	E8	IMP	1	2

INY

Der Inhalt des Indexregisters Y wird um 1 erhöht (inkrementiert).

Abgekürzte Schreibweise:	Y + 1 → Y
Beeinflußte Flags im Statusregister:	N, Z

Assemblerform	OP-Code $-Form	Adressierung	Byte	Zyklen
INY	C8	IMP	1	2

1.13.6 Schiebe-Befehle

ASL

Arithmetische Linksverschiebung

Abgekürzte Schreibweise:	C ← [7][6][5][4][3][2][1][0] ← 0
Beeinflußte Flags im Statusregister:	N, Z, C

Assemblerform		OP-Code $-Form	Adressierung	Byte	Zyklen
ASL		0A	Akku	1	2
ASL	Oprd	06	ZP	2	5
ASL	Oprd,X	16	ZPX	2	6
ASL	Oprd	0E	ABS	3	6
ASL	Oprd,X	1E	ABX	3	7

Der ASL-Befehl verschiebt entweder den Akkuinhalt oder einen adressierten Speicherstelleninhalt um 1 Bit zyklisch nach links, wobei das Bit 7 in das Carryflag transferiert wird und im Bit 0 eine Null nachgeschoben wird.

LSR

Logische Rechtsverschiebung

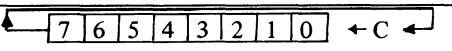
Abgekürzte Schreibweise:	0 → [7][6][5][4][3][2][1][0] → C
Beeinflußte Flags im Statusregister:	N(0), Z, C

Assemblerform		OP-Codes $-Form	Adressierung	Byte	Zyklen
LSR		4A	Akku	1	2
LSR	Oprd	46	ZP	2	5
LSR	Oprd,X	56	ZPX	2	6
LSR	Oprd	4E	ABS	3	6
LSR	Oprd,X	5E	ABX	3	7

Der LSR-Befehl verschiebt entweder den Akkuinhalt oder einen adressierten Speicherstellinhalt um 1 Bit nach rechts, wobei das Bit 0 in das Carryflag übertragen wird und in das Bit 7 eine Null nachgeschoben wird.

ROL

Rotation nach links

Abgekürzte Schreibweise: 

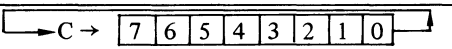
Beeinflußte Flags im Statusregister: N, Z, C

Assemblerform	OP-Code $-Form	Adressierung	Byte	Zyklen
ROL	2A	Akku	1	2
ROL Oprd	26	ZP	2	5
ROL Oprd,X	36	ZPX	2	6
ROL Oprd	2E	ABS	3	6
ROL Oprd,X	3E	ABX	3	7

Auch hier wird entweder der Akkuinhalt oder der adressierte Speicherinhalt zyklisch um 1 Bit nach links verschoben und Bit 7 wird ins Carryflag übertragen. Der vorhergehende alte Inhalt der Carryflag wird jedoch in das Bit Null nachgeschoben.

ROR

Rotation nach rechts.

Abgekürzte Schreibweise: 

Beeinflußte Flags im Statusregister: N, Z, C

Assemblerform	OP-Code $-Form	Adressierung	Byte	Zyklen
ROR	6A	Akku	1	2
ROR Oprd	66	ZP	2	5
ROR Oprd,X	76	ZPX	2	6
ROR Oprd	6E	ABS	3	6
ROR Oprd,X	7E	ABX	3	7

Durch den ROR-Befehl wird entweder der Akkuinhalt oder der adressierte Speicherinhalt zyklisch um 1 Bit nach rechts verschoben, wobei Bit 0 ins Carryflag übertragen wird, und der vorhergehende alte Inhalt der Carryflag in das Bit 7 nachgeschoben wird.

1.13.7 Sprünge

JMP

Springe zur nachfolgend angegebenen Adresse (bzw. setze den Inhalt des Programmzählers auf neue Adresse).

Abgekürzte Schreibweise:			(PC +1) → PCL u. (PC + 2) → PCH
Beeinflußte Flags im Statusregister:			keine
Assemblerform	OP-Code $-Form	Adressierung	Byte / Zyklen
JMP Oprd	4C	ABS	3 / 3
JMP (Oprd)	6C	IND	3 / 5

Der JMP-Befehl ist ein unbedingter Sprungbefehl, da er ohne irgendeine Abfragebedingung direkt zur angegebenen Programmadresse springt. Er ist auch der einzige Befehl, der die reine indirekte Adressierung versteht. D.h., der neue Programmzählerinhalt wird nicht direkt, sondern indirekt aus der nach dem JMP-Code angegebenen Adresse geholt (siehe auch Adressierungsmethode indirekt).
Außer dem Sprungbefehl JMP gibt es Verzweigungsbefehle, die ebenfalls Sprungbefehle sind. Im Gegensatz zu JMP sind sie jedoch an eine Bedingung geknüpft und werden außerdem relativ adressiert.

BCC

Verzweige bei gelöschter Carryflag

Abgekürzte Schreibweise:			branch on C = 0
Beeinflußte Flags im Statusregister:			keine
Assemblerform	OP-Code $-Form	Adressierung	Byte / Zyklen
BCC Oprd	90	R	2 / 2*

* Addiere 1, wenn die Verzweigung innerhalb der gleichen Page erfolgt.
* Addiere 2, wenn die Verzweigung zu einer anderen Page erfolgt.

Der BCC-Befehl frägt im Statusregister die Carryflag auf 0 oder 1 ab und führt den relativen Sprung aus, wenn in der Carryflag eine 0 steht.

BCS

Verzweige bei gesetztem Carry.

Abgekürzte Schreibweise:			branch on C = 1
Beeinflußte Flags im Statusregister:			keine
Assemblerform	OP-Code $-Form	Adressierung	Byte / Zyklen
BCS Oprd	B0	R	2 / 2*

* Addiere 1, wenn die Verzweigung innerhalb der gleichen Page erfolgt.
* Addiere 2, wenn die Verzweigung zu einer anderen Page erfolgt.

Der relative Sprung erfolgt nur, wenn im Carryflag eine 1 steht.

BEQ

Verzweige bei Ergebnis gleich Null.

Abgekürzte Schreibweise:	branch on Z = 1

Beeinflußte Flags im Statusregister:	keine

Assemblerform	OP-Code $-Form	Adressierung	Byte	Zyklen
BEQ Oprd	F0	R	2	2*

* Addiere 1, wenn die Verzweigung innerhalb der gleichen Page erfolgt.
* Addiere 2, wenn die Verzweigung zu einer anderen Page erfolgt.

Der relative Sprung erfolgt nur, wenn im Zeroflag eine 1 steht, bzw. das vorhergehende Ergebnis Null war.

BMI

Verzweige bei negativem Ergebnis

Abgekürzte Schreibweise:	branch on N = 1

Beeinflußte Flags im Statusregister:	keine

Assemblerform	OP-Code $-Form	Adressierung	Byte	Zyklen
BMI Oprd	30	R	2	2*

* Addiere 1, wenn die Verzweigung innerhalb der gleichen Page erfolgt.
* Addiere 2, wenn die Verzweigung zu einer anderen Page erfolgt.

Der relative Sprung erfolgt nur, wenn im Negativflag eine 1 steht, bzw. das vorhergehende Ergebnis als Minusergebnis interpretiert wurde.

BNE

Verzweige bei Ergebnis ungleich 0

Abgekürzte Schreibweise:	branch on Z = 0

Beeinflußte Flags im Statusregister:	keine

Assemblerform	OP-Code $-Form	Adressierung	Byte	Zyklen
BNE Oprd	D0	R	2	2*

* Addiere 1, wenn die Verzweigung innerhalb der gleichen Page erfolgt.
* Addiere 2, wenn die Verzweigung zu einer anderen Page erfolgt.

Der relative Sprung erfolgt nur, wenn im Zeroflag eine 0 steht, bzw. das vorhergehende Ergebnis ungleich 0 war.

BPL

Verzweige bei positivem Ergebnis

Abgekürzte Schreibweise:	branch on N = 0
Beeinflußte Flags im Statusregister:	keine

Assemblerform	/ OP-Code $-Form	/ Adressierung	/ Byte	/ Zyklen
BPL Oprd	10	R	2	2*

* Addiere 1, wenn die Verzweigung innerhalb der gleichen Page erfolgt.
* Addiere 2, wenn die Verzweigung zu einer anderen Page erfolgt.

Der relative Sprung erfolgt nur, wenn im Negativflag eine 0 steht, bzw. das vorhergehende Ergebnis positiv war.

BVC

Verzweige bei gelöschter Overflowflag

Abgekürzte Schreibweise:	branch on V = 0
Beeinflußte Flags im Statusregister:	keine

Assemblerform	/ OP-Code $-Form	/ Adressierung	/ Byte	/ Zyklen
BVC Oprd	50	R	2	2*

* Addiere 1, wenn die Verzweigung innerhalb der gleichen Page erfolgt.
* Addiere 2, wenn die Verzweigung zu einer anderen Page erfolgt.

Der relative Sprung erfolgt nur, wenn im Statusregister im V-Flag (Overflow) eine 0 steht.

BVS

Verzweige bei gesetzter Overflowflag

Abgekürzte Schreibweise:	branch on V = 1
Beeinflußte Flags im Statusregister:	keine

Assemblerform	/ OP-Code $-Form	/ Adressierung	/ Byte	/ Zyklen
BVS Oprd	70	R	2	2*

* Addiere 1, wenn die Verzweigung innerhalb der gleichen Page erfolgt.
* Addiere 2, wenn die Verzweigung zu einer anderen Page erfolgt.

Der relative Sprung erfolgt nur, wenn im V-Flag eine 1 steht.

Die relativen Sprungbefehle fragen den jeweiligen Zustand der entsprechenden Flags ab. Ist die Bedingung nicht erfüllt, geht das Programm bei der nächsten Programmadresse weiter. Ist die Bedingung erfüllt, wird zum Programmzähler der relative Verschiebungswert hinzuaddiert und das Programm fährt bei der neuen Programmadresse fort.

1.13.8 Unterprogramme

Ein Unterprogramm ist ein eigenständiges Programm, das irgendwo im Speicher ab einer bestimmten Adresse steht. Es kann nun von einem normalen Programm von verschiedenen Programmstellen aus zu jeder Zeit aufgerufen werden.

JSR

Sprung zum Unterprogramm

Abgekürzte Schreibweise:	PC + 2 ↓ , (PC + 1) → PCL u. (PC + 2) → PCH

Beeinflußte Flags im Statusregister:	keine

Assemblerform	/ OP-Code $-Form	/ Adressierung	/ Byte	/ Zyklen
JSR Oprd	20	ABS	3	6

Der JSR-Befehl legt die Programmadresse, die nach dem OP-Code und der dazugehörigen Sprungadresse folgt, im Stapel ab und springt an die angegebene Unterprogrammadresse (bzw. setzt den Programmzähler auf diese Adresse).

RTS

Rückkehr vom Unterprogramm

Abgekürzte Schreibweise:	PC ↑, PC + 1 → PC

Beeinflußte Flags im Statusregister:	keine

Assemblerform	/ OP-Code $-Form	/ Adressierung	/ Byte	/ Zyklen
RTS	60	IMP	1	6

Am Ende jeden Unterprogramms muß der Befehl RTS stehen. Dadurch holt sich die CPU die durch den JSR-Befehl im Stapel abgespeicherte Programmadresse des normalen Programms zurück und lädt diese in den Programmzähler. Damit fährt nun des Programm an der Stelle fort, wo es vor Behandlung des Unterprogramms aufgehört hat.

1.13.9 Interrupts und sonstige Befehle

Um das Konzept über die Interrupts zu entwickeln, müssen wir wissen, was ein „Vektor" ist. Ein Vektor besteht meist aus zwei Speicherstellen. Der Wert in diesen Speicherstellen ist wiederum aus dem unteren und oberen Teil einer Adresse zusammengesetzt. Diese Adresse zeigt nun auf eine Speicherstelle, von der ab Programmdaten stehen. Die ursprünglichen zwei Speicherstellen stellen daher einen Zeiger dar. Das Fremdwort für Zeiger heißt eben Vektor. Somit ist ein Vektor eine indirekte Adresse. Wie wir noch sehen werden, benützt das Betriebssystem und der Basic-Interpreter beim VC-64 in der Zeropage

viele Speicherstellen als Vektoren, weil auf die Zeropage vor- oder nachindiziert zugegriffen werden kann.

Was versteht man nun unter einem Interrupt? „Interrupt" bedeutet direkt übersetzt „Unterbrechung". Wenn wir das Blockschaltbild der CPU ansehen, erkennen wir über dem Befehlsdecoder ein Register, das mit Interruptlogik bezeichnet ist und drei Eingänge hat.

N M I = Non maskable Interrupt (Unterbrechung nicht abschaltbar)
I R Q = Interrupt request (normale Unterbrechung)
R S T = Reset (Initialisieren des Systems)

Tritt nun während einer Programmadresse an einem dieser Eingänge ein Signal auf, dann passiert folgendes:

Das laufende Programm wird unterbrochen, wobei der zuletzt behandelte Befehl jedoch noch ganz abgeschlossen wird.

Die Adresse im Programmzähler, die die Programmadresse des nächsten Befehls im Hauptprogramm darstellt, wird zusammen mit dem Zustand (Inhalt) des Statusregisters automatisch auf den Stapel abgelegt. Anschließend holt sich der Programmzähler aus einer Vektoradresse eine Unterprogrammadresse. Das Unterprogramm, als Interruptprogramm bezeichnet, wird nun abgearbeitet, bis es auf einen Rückkehrbefehl trifft. Nun läuft alles umgekehrt ab. Das Statusregister holt sich den alten Zustand vom Stapel zurück. Ebenso wird die auf dem Stapel abgespeicherte Hauptprogrammadresse vom Stapel zurück in den Programmzähler geladen und das laufende Hauptprogramm setzt sich fort.

Man wird sich nun fragen, das gleiche wird doch auch mit einem Unterprogrammsprung erreicht = (JSR)! Dazu müßte man jedoch den „JSR"-Befehl im Programm einbringen. Die Interrupts wurden nun dazu geschaffen, den aktuellen Zustand im Computer durch externe Signale zu unterbrechen. Die externen Signale können von verschiedenen Peripheriegeräten stammen. Die hauptsächlichste Interruptverarbeitung dürfte durch die Tastatur entstehen. Ohne Interrupts müßte z.B. ein Unterprogramm enthalten sein, das zyklisch ca. alle 100 Mikrosekunden abgerufen wird, um dann die Tastatur abzufragen, ob eine Taste gedrückt wurde oder nicht. Solch ein Unterprogramm stellt einen immensen Aufwand dar und verlangsamt das Computerkonzept erheblich. Das war auch der Grund, warum Interrupts in die Computerlogik eingeführt wurden.

Im VC-64 ist zwar auch ein Unterprogramm enthalten, das die Tastatur abfragt. Diese Tastaturabfrage ist aber ein Interruptprogramm. D.h., die Tastatur wird nur dann abgefragt, wenn eine Taste gedrückt und der Interrupt zugelassen wird. Ansonsten verliert die CPU keine Zeit bei der Programmausführung.

Jetzt muß nur noch geklärt werden, wo die Interruptprogramme abgespeichert sind und wie das Hauptprogramm dorthin verzweigt.

Die Interruptlogik in der CPU enthält drei Adressen, die als Vektoren zu den entsprechenden Interruptprogrammen dienen. Sie sind in der 6502 Interruptlogik so fest-

gelegt, daß sie die letzten 6 Speicherstellen des Gesamtspeichers bezeichnen. Die Vektorenadressen werden folgendermaßen dargestellt:

		Vektoradresse	
Dezimal	Hexadez.	Adreßteil	Interruptbezeichnung
65530	$FFFA	ADL	NMI
65531	$FFFB	ADH	
65532	$FFFC	ADL	RST
65533	$FFFD	ADH	
65534	$FFFE	ADL	IRQ
65535	$FFFF	ADH	

Je nachdem, welches Signal an der Interruptlogik anliegt, wird über den Befehlsdecoder der Programmzähler mit den entsprechenden Adreßteilen aus einem der Interruptvektoren versorgt. Dadurch befindet sich die CPU am Anfang des entsprechenden Interruptprogrammes. Vorher wurde noch der Inhalt des Statusregisters und der alte PC-Inhalt auf den Stapel gerettet. Normalerweise müßte man selber die Adressen $FFFA bis $FFFF mit Adreßwerten belegen. Beim VC-64 brauchen wir uns darum nicht zu kümmern, da diese bereits in den ROM-Speicherstellen fest enthalten sind und damit zum Betriebssystem zählen.

RTI

Rückkehr vom Interruptprogramm

Abgekürzte Schreibweise:	P ↑ PC ↑
Beeinflußte Flags im Statusregister:	vom Stapel

Assemblerform	OP-Code $-Form	Adressierung	Byte	Zyklen
RTI	40	IMP	1	6

„RTI" muß am Ende eines jeden Interruptprogramms stehen, damit der Anfangszustand, wie er vor der Interruptverarbeitung war, wieder hergestellt wird.

Meist ist es nötig, daß auch die alten Akku-, X- und Y-Registerinhalte nach der Interruptbearbeitung wieder erforderlich sind. Durch den Interrupt werden aber nur der Inhalt des Statusregisters und der Inhalt des Programmzählers auf den Stapel geschrieben. Im Interruptprogramm selber werden die Arbeitsregister (Akku, X und Y) mit anderen Werten versorgt und daher der alte Inhalt zerstört. Deshalb wird man in das eigentliche Interruptprogramm am Anfang Rettungsbefehle für die Arbeitsregister einbauen. Vor dem Rückkehrbefehl „RTI" werden dann noch die Befehle zur Wiederherstellung der alten Inhalte der Arbeitsregister veranlaßt. Das sieht nun folgendermaßen aus:

Interruptprogramm

```
PHA   Rette Akku
TXA   X -- A
PHA   Rette X
TYA   Y -- A
PHA   Rette Y
.
.
Eigentliches Interruptprogramm
.
.
PLA   Hole Y
TAY   Hole Y
PLA   A -- Y
TAX   Hole X
PLA   A -- X
RTI   Rückkehr aus dem Interruptprogramm
```

Wir haben in der CPU drei Interruptleitungen. Es wird nun gezeigt, welche Interruptart welche Auswirkung hat.

R S T (RESET)

Wenn wir ein Computersystem einschalten, dann befindet es sich in einem undefinierbaren Zustand. Um ein unkontrolliertes Loslaufen zu verhindern, wird die Reset-Leitung aktiviert. D.h., das Signal wird durch das Einschalten auf den Reset-Eingang (RST) an der Interruptlogik gelegt.

Damit holt sich der Programmzähler die Adresse aus dem RST-Vektor ($FFFC, FFFD) und verzweigt zu einem Interruptprogramm. Das Interruptprogramm muß nun sämtliche Befehle enthalten, die das System auf einen bekannten Anfangszustand setzt. Man nennt das auch „initialisieren". Der VC-64 enthält im ROM-Bereich bereits so ein Interruptprogramm, das eigentlich aus zwei Teilen besteht – Einschaltreset I und II. Wenn die Initialisierung am Bildschirm erfolgt ist, können wir beim VC-64 folgenden Schriftzug lesen:

**** COMMODORE 64 BASIC V2 ****
64K RAM SYSTEM 38911 BASIC BYTES FREE

I R Q (Interrupt Request = Unterbrechungsanforderung)

In unserem Computersystem sind die Peripheriegeräte (Tastatur, Bildschirm, Floppy und Drucker) mit der CPU über Schnittstellenbausteine verbunden. Diese Bausteine werden zu dem Sammelbegriff „Interface" zusammengefaßt. Die Interfacebausteine sind nun mit der CPU u.a. auch über die IRQ-Leitung verbunden. Wird nun durch ein Signal, z.B. Tastendruck, auf den IRQ-Eingang gelegt, dann fordert die Interruptlogik Bearbeitung an. Das Statusregister in der CPU hat ein Bit, das für die IRQ-Anforderung zuständig ist. Ist dieses Interruptflag auf „1" gesetzt, dann wird die Interruptanforderung vernachlässigt bzw. zurückgehalten. Erst wenn das I-Flag gelöscht wird, kann die Ausführung des angeforderten Interrupts in der beschriebenen Weise erfolgen. Nachdem ein Interrupt

von der CPU anerkannt wurde, wird die I-Flag automatisch auf „1" gesetzt. Der Zweck ist, daß während eines Interruptprogramms kein weiterer Interrupt zugelassen wird. Sollte jedoch im Interruptprogramm ein weiterer Interrupt zugelassen werden, so muß man an der entsprechenden Programmstelle dafür sorgen, daß ein „CLI"-Befehl eingebaut ist.

Nach Rückkehr der IRQ-Bearbeitung wird das I-Flag automatisch wieder auf „0" gesetzt, um die CPU für die nächste IRQ-Anforderung bereit zu machen.

N M I (Non maskable Interrupt = nicht abschaltbar)

Ein Signal an diesem Eingang hat eine höhere Priorität als der IRQ. Nach einem NMI-Signal verzweigt das Programm zum NMI-Vektor, auch wenn das I-Flag im Statusregister gesetzt sein sollte. Es erfolgt also auf jeden Fall eine Interruptverzweigung. Deshalb auch der Name „nicht abschaltbar". Ansonsten geschieht das gleiche wie beim IRQ. Statusregister- und Programmzählerinhalt werden auf den Stapel gelegt. Das I-Flag wird auf „1" gesetzt und die Vektoradresse des NMI gelangt in den Programmzähler. Das NMI-Programm wird ausgeführt.

BRK

Softwareunterbrechung (Break)

Abgekürzte Schreibweise:	PC + 2 ↓ P ↓
Beeinflußte Flags im Statusregister:	I (1)

Assemblerform	OP-Code $-Form	Adressierung	Byte	Zyklen
BRK	00	IMP	1	7

Die Interrupts werden durch Signale erzeugt; sind also hardwaremäßig realisiert. Der „BRK"-Befehl kommt jedoch aus dem Befehlssatz und erzeugt dadurch einen Software-Interrupt. Dabei benützt der „BRK"-Befehl den IRQ-Vektor. Zur Unterscheidung, ob es sich um einen IRQ oder einen „BRK" handelt, besitzt das Statusregister die Break-Flag in Bit 4. Diese B-Flag geht auf „1", wenn die Unterbrechung durch „BRK" erzeugt wurde. Tritt ein Interrupt auf und enthält die B-Flag eine „0", dann handelt es sich um einen echten IRQ-Interrupt.

Erreicht nun ein aktuelles Programm den Befehl „BRK", dann wird folgendes ausgeführt:

Im Programmzähler wird 2 addiert und der neue Inhalt (PC + 2) wird zusammen mit dem Inhalt des Statusregisters auf den Stapel gelegt. Dann verzweigt das aktuelle Programm zu einem Interruptprogramm über den IRQ-Vektor. Am Anfang des Interruptprogramms muß nun eine Abfrage stehen, die das B-Flag auf „0" oder „1" überprüft. Da das B-Flag durch „BRK" auf eins gesetzt ist, verzweigt das Interruptprogramm zu einem weiteren Unterprogramm, dem „BRK"-Programm. Nach dem Rückkehrbefehl wird der Statusregisterinhalt und der Inhalt des Programmzählers wieder vom Stapel geholt und das „BRK"-Programm springt in das aktuelle Hauptprogramm zurück.

Da nach einem „BRK"-Befehl im Interruptprogramm noch zu einem „BRK"-Programm verzweigt wird und somit eine zweite Analyse bzw. Korrektur stattfindet, wird der Inhalt des Programmzählers plus dem Wert 2 am Stapel abgespeichert. Falls gewünscht wird, daß nach einem „BRK"-Befehl das laufende Hauptprogramm bei der unmittelbar darauffolgenden Programmspeicherstelle fortgesetzt werden soll, müssen im „BRK"-Programm Speicherverminderungsbefehle im Stapel den Programmzählerinhalt zurücksetzen.

Der „BRK"-Befehl wird dazu benutzt, um laufende Programme zu unterbrechen und dabei die entsprechenden Kontrollen oder Korrekturen anzubringen. Man kann damit hervorragend Einzelschrittbetrieb ausführen. Die hauptsächlichste Anwendung des Befehls „BRK" dürfte jedoch das Austesten bei der Programmentwicklung sein. So setzt man z.B. statt eines normalen Programmbefehls „BRK" ein und ersetzt erst später in der Endphase der Entwicklung den nötigen Befehl.

NOP

No Operation (2 Zyklen)
(Keine Operation)

Es werden keine Flags beeinflußt. Der Befehl ist impliziert.

NOP IMP $EA

Dieser Befehl inkrementiert nur den Programmzähler und hat ansonsten keine Funktion. Er verlängert die Laufzeit eines Programms um zwei Mikrosekunden und dient höchstens als Lückenfüller zur Zeitverzögerung.

Wir sind damit am Ende des Befehlssatzes angelangt. Sie werden jetzt wahrscheinlich noch nicht gleich in Assembler programmieren können. Dazu muß man ein paar Programmbeispiele durchexerzieren, die jedoch erst in den nächsten Kapiteln gebracht werden. Zum Abschluß dieses Kapitels müssen wir noch etwas über die Befehlszeiten erfahren.

1.13.10 Befehlszeiten

Die CPU 6502 arbeitet mit einer Taktfrequenz von 1 MHz. Da der Taktgenerator mit zwei Phasen das System steuert, benötigen die Befehle auch mindestens zwei Zeitzyklen, wobei ein Zyklus durch eine Mikrosekunde festgelegt ist. Jeder Takt ist ein Speicherzyklus, der mit einer Phase den Adreßbus und mit der anderen Phase den Datenbus steuert. Damit wird klar, daß ein Befehl eben mindestens 2 Zyklen erfordert.

Nämlich einen für den Zugriff zum OP-Code und einen für die Dekodierung des OP-Codes. Wenn man nun weiß, wieviel Zyklen jeder Befehl hat, dann kann man die Zeit berechnen, die ein Programm zur Ausführung braucht. Die Anzahl der Befehlszyklen ist dann gleich der Anzahl der Mikrosekunden.

1.14 Die CPU 6510 (Der Unterschied zum 6502)

VCC	Stromversorgung	+5V
&1	Systemtakteingang	Eingang
&2	Systemtaktausgang	Ausgang
AEC	Adressensteuerung	Eingang
RDY	Einzelzyklussteuerung	Eingang
R/W	Lese/Schreibsteuerung	Ausgang
IRQ	Maskierbarer Interrupt	Eingang
NMI	Unbedingter Interrupt	Eingang
RES	Rücksetz Interrupt	Eingang
P0 – P5	Port 0 – 5	bidirektional
AB0 – AB15	Adreßbus	Ausgang
D0 – D7	Datenbus	bidirektional

Hier sehen Sie gleich die Liste der Anschlußbelegungen. Im Gegensatz zum 6502 fallen hier die Anschlüsse P0 – P5 auf, die es beim 6502 eben nicht gibt. Diese zusätzlichen Leitungen werden hardwaremäßig mit den Speicherstellen 0 und 1 in der Zeropage des RAM-Speichers so verbunden, daß sie als Zusatzregister des 6510 wirken. Allerdings werden jeweils nur die unteren 6 Bits genützt, da es ja nur 6 Prozessorports gibt. Beim VC-64 werden diese Register verwendet, um die Speicherkonfiguration festzulegen. Darüber wird aber noch genaueres gesagt. Grundsätzlich dient Register 0 als Datenrichtungsregister. Das heißt, je nach gelöschtem oder gesetztem Bit wird das entsprechende Datenbit im Register 1 auf Eingang oder Ausgang gelegt. Das Register 1 ist somit das Datenregister. Das Bitmuster darin wird also von oder zur Peripherie übermittelt.

Ansonsten kann man ruhig behaupten, daß der 6510 softwarekompatibel zum 6502 ist. Er hat den gleichen Befehlssatz wie der 6502, so daß man das vorher beschriebene Kapitel ohne Einschränkung übernehmen kann.

Wir sind damit am Ende des 1. Kapitels angelangt. Wenn Sie den bisherigen Stoff verstanden haben, besitzen Sie die Grundvoraussetzung für das Programmieren in Maschinensprache des VC-64. Um nicht die Übersicht zu verlieren, befinden sich im Anhang einige Tabellen über den Befehlssatz zum Nachschlagen.

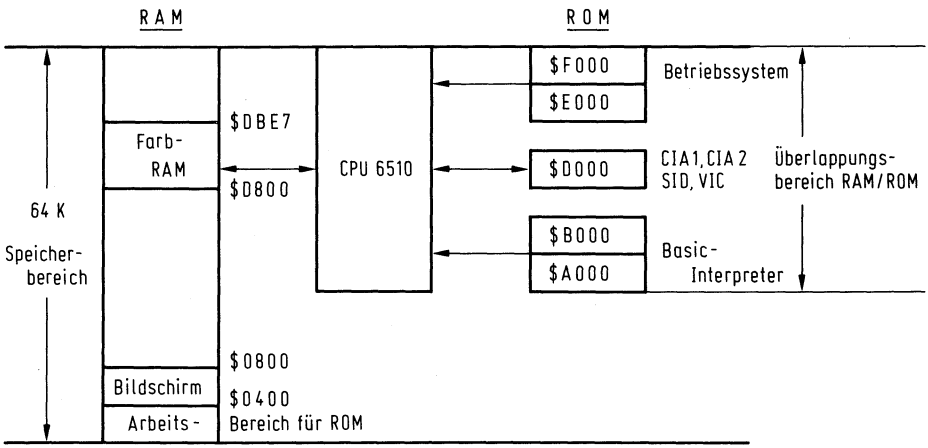
Von dem nächsten Kapitel an beschäftigen wir uns nur noch mit VC-64-spezifischen Dingen.

2 VC-Spezifisches

Im Kapitel 1 wurden die Grundvoraussetzungen geschaffen, um überhaupt in Maschinensprache beginnen zu können. Das heißt aber nicht, daß man nun schon voll programmieren kann. Dazu fehlen einfach noch verschiedene gerätespezifische Informationen. Wir werden deshalb in diesem Kapitel uns die entsprechenden Grundlagen der VC-Serie verschaffen. Dabei gehen wir auf einzelne Aspekte der verschiedenen VC-Typen ein. Ab und zu werden im Text auch einzelne Basic-Programme eingestreut. Deshalb wird vorausgesetzt, daß der Leser das Commodore-Basic einigermaßen beherrscht.

Beginnen wir also und schauen uns gleich einmal an, wie der gesamte Speicherplatz im VC aufgeteilt ist. Das dürfte wohl auch schon die wichtigste Information sein. Wie wir wissen, kann die CPU 65536 Speicherstellen adressieren, die man von 0 bis 65535 durchnumeriert. Für den Maschinenprogrammierer eröffnen sich vielfältige Möglichkeiten, wenn er über den Aufbau des gesamten Speicherbereichs Bescheid weiß.

2.1 VC Speicherübersicht



Die vorliegende Darstellung könnte man schon fast als physikalische Aufteilung bezeichnen. Da das noch nicht reicht, werden wir noch eine grundlegende Übersicht erarbeiten und erst später ins Detail gehen.

2.1.1 Der Adreßraum des VC-64

dezimal	hexadezimal	Art der Belegung	Bausteinart
0– 255	0000–00FF	Zeropage, Arbeits- u. Zeigerbereich	RAM
256– 511	0100–01FF	Page 1, Stackbereich	RAM
512– 767	0200–02FF	Page 2, Eingabepuffer, Flags, RS232–Zeiger und Spritepuffer	RAM
768– 1023	0300–03FF	Page 3, Sprunglisten (Vektoren in RAM-Version erleichtern Systemumstellung)	RAM
1024– 2047	0400–07FF	Bildschirmspeicher für Zeichen-Code	RAM
2048–40959	0800–9FFF	Freier Anwenderspeicher für Basic	RAM
–65535	–FFFF	Zusätzlicher Anwenderspeicher nur für Maschinenprogrammierung geeignet	RAM
40960–49151	A000–BFFF	Basic-Interpreter	ROM
53248–53294	D000–D02E	Video-Controller VIC	I/O
54272–54300	D400–D41C	Musik-Synthesizer SID	I/O
55296–56295	D800–DBE7	Bildschirmspeicher für Farben-Code	RAM
56320–56335	DC00–DC0F	Interface Adapter CIA 1	I/O
56576–56591	DD00–DD0F	Interface Adapter CIA 2	I/O
57344–65535	E000–FFFF	Betriebssystem	ROM

Wenn man sich die Bereiche etwas näher ansieht, kann man folgendes vorerst spezifizieren.

PAGE 0 (Zeropage)

Wie man in Kapitel 1 bei den Adressierungsmethoden sehen kann, wird durch die Zeropage-Adressierung Zeit und Platz gespart. Deshalb benutzt das Betriebssystem vorwiegend diesen Speicherbereich für die Zwischenspeicherung von allen möglichen Zeigern, Adressen und Flags. Zur Abspeicherung von Programmen sollte die Zeropage nicht verwendet werden, sie ist aber durchaus möglich.

PAGE 1 (Stapelbereich)

Die Speicherstellen 256 – 511 werden permanent vom Stackpointer (Stapelzeiger) verwendet. Das bedeutet, daß in diesem Bereich Adressen zwischengespeichert werden, die ein BASIC-Programm benötigt. Deshalb kann man auch nicht unbegrenzt Unterprogramme verschachteln und Klammerebenen bilden. Das Betriebssystem benutzt den Stackbereich unter anderem für die Umwandlung von numerischen Codes in ASCII-Codes und als Zwischenspeicher für Bandkorrekturen.

PAGE 2 und 3

Diese Seiten stellen dem Betriebssystem mehrere Pufferbereiche zur Verfügung. Da ist zunächst der Basic-Eingabepuffer mit 80 Zeichen Länge und einem 0-Byte, das einen eingegebenen String begrenzt. Danach folgen Filetabellen und dahinter der Tastaturpuffer; anschließend liegen einige Hilfsspeicherstellen als Flag, Zähler, Zeiger usw. Wichtig da-

bei ist, daß hier Vektoren abgespeichert sind, die vom Betriebssystem indirekt angesprungen werden. Das hat den Vorteil, daß man ohne großen Aufwand Änderungen anbringen und so eigene Maschinenroutinen einbinden kann. Die Speicherstellen 828 – 1019 sind als Bandpuffer reserviert. Die Page 2 und 3 können auch als Spritepuffer 11–15 fungieren.

PAGE 4 – 7 (Bildschirmspeicher für Zeichen)

Das ist der RAM-Bereich für die Ausgabe eines Zeichens auf dem Bildschirm. Es wird allerdings außer dem Bildschirmcode auch noch der Farbcode benötigt. Dieser befindet sich jedoch in einem anderen Speicherbereich.

PAGE 8 bis 159 (Anwenderbereich Basic)

Ab der Speicherstelle 2048 beginnt der Bereich, in dem die Basic-Programme abgespeichert sind. Zusammen mit dem Programm selbst werden unmittelbar hinter dem Programm die einzelnen Variablen und deren Inhalte abgelegt; doch darauf werden wir noch genauer eingehen. Der Anwenderbereich kann aber auch für größere Maschinenprogramme genutzt werden. Sinnvollerweise schiebt man dann diese meist in den obersten RAM-Bereich und schützt sie dort vor Basic.

PAGE 160 – 255 RAM (Anwenderbereich Maschinenebene)

Der VC-64 verfügt über 64K-RAM. In der Normalkonfiguration nach dem Einschalten sind Teilbereiche davon von ROM's überlagert. Es ist daher leicht einzusehen, daß diese RAM-Bereiche dann nicht von BASIC aus ohne weiteres benutzt werden können. Für Maschinenprogrammierer eröffnen sich jedoch die vielfältigsten Möglichkeiten. Man kann nämlich in diesen Bereich Maschinenprogramme unterbringen und braucht sie nicht einmal vor Basic zu schützen. Allerdings erfordert der Aufruf wieder besondere Verfahren. Darauf gehen wir aber noch genauer ein.

PAGE 160 – 191 ROM (BASIC-INTERPRETER)

Dieser Teil des Rechners ermöglicht überhaupt eine Programmierung in Basic. Er enthält unter anderem Adreßtabellen der Basic-Routinen, Basic-Schlüsselwörter, Fehlermeldungen, Stackbehandlung, Speicherverwaltung, Formelauswertung, Funktions- und Variablenverwaltung und eben die Basic-Routinen.

PAGE 224 – 255 ROM (BETRIEBSSYSTEM)

Am Anfang dieses Speicherbereichs ist noch der Rest des Interpreters enthalten. Anschließend folgt das eigentliche Betriebssystem mit den Editorroutinen. Desweiteren befinden sich hier Interruptroutinen sowie die Routinen zur Bedienung der Peripheriegeräte.

Wir haben nun damit einen gewissen Gesamtüberblick über alle Adressen bzw. Speicherstellen erhalten. Sie können momentan wahrscheinlich damit noch nicht viel anfangen. Erst im Verlaufe der weiteren Ausführungen werden dann langsam die Zusammenhänge klar.

Um überhaupt zum erstenmal in Maschinensprache zu experimentieren, werden wir einmal eine genaue Übersicht über die ersten 1024 Speicherzellen aufstellen. Denn aus diesen Adressen benötigen wir einige Bytes zur Erklärung der Basic-Interpretation.

2.1.2 Speicherbelegung Page 0 – 4

Adresse dez	Adresse hex	Erklärung des Inhaltes
0	0000	Daten Richtungsregister (Hilfsspeicher für MCS 6510)
1	0001	Ausgabe Register (Hilfsregister für MCS 6510)
2	0002	Wird vom Betriebssystem nicht verwendet
3	0003	ADL Zeiger auf Routine "Umwandlung FAC nach Integer"
4	0004	ADH
5	0005	ADL Zeiger auf Routine "Integer nach FAC"
6	0006	ADH
7	0007	Zwischenspeicher für Hexziffer oder Suchzeichen
8	0008	Flag f. Gänsefußmodus o. Offset zum nächst. Trennzeich.
9	0009	Zähler für TAB-Werte
10	000A	Flag für LOAD (=0) und VERIFY (=1)
11	000B	Hilfszähler für Basic-Eingabepuffer
12	000C	Flag für DIM-Fehler, Feldnameninitialisierung und Flag für AND und OR
13	000D	Flag für Variablentyp (0 = Zahl, 255 = Zeichen)
14	000E	Flag für numerische Variable (0=Fließkomma,128=Integer)
15	000F	Flag für ", REM, DATA, LIST, Garbage Collect u. Memory
16	0010	Flag für Indexabfrage (? Index erlaubt) und FN DEF
17	0011	Flag für INPUT (=0), GET (=40) und READ (=98)
18	0012	Flag für ATN Vorzeichen und Vergleichsauswertung
19	0013	Aktiver Ein- und Ausgabekanal
20	0014	ADL Integeradresse zur Berechnung für GOTO, SYS und
21	0015	ADH GOSUB
22	0016	Indexzeiger für den nächsten Deskriptor (Tabelle der Variablenadressen)
23	0017	ADL Zeiger auf den zuletzt benutzten String
24	0018	ADH
25	0019	Beginn der Tabelle für Deskriptoren für Variable (je
..	....	zwei Byte)
34	0022	ADL Indirekter Index #1 / und Stringverschiebung
35	0023	ADH
36	0024	ADL Indirekter Index #2 / und Zahlenverschiebung
37	0025	ADH
38	0026	Pseudoregister für Operanden von Funktionen
..	....	
43	002B	ADL Zeiger auf Beginn der Basic-Programme: normal 2048
44	002C	ADH
45	002D	ADL Zeiger auf Beginn der Variablentabelle bzw. Zeiger
46	002E	ADH auf Ende des Basic-Programms
47	002F	ADL Zeiger auf Beginn der Tabelle der Felder bzw. Zei-
48	0030	ADH ger auf Ende der Variablentabelle
49	0031	ADL Zeiger auf Ende der Felder bzw. Beginn des freien
50	0032	ADH RAM-Platzes
51	0033	ADL Zeiger auf Beginn der Strings (Anlage verläuft
52	0034	ADH rückwärts)
53	0035	ADL Normaler Zeiger auf Stringende für die Reser-
54	0036	ADH vierung neuer Strings
55	0037	ADL Zeiger auf höchste RAM-Adresse (normalerweise
56	0038	ADH $A000 bzw. 40960)

Adresse dez	Adresse hex	Erklärung des Inhaltes
57	0039	ADL Laufende Zeilennummer (255 = Direktmodus)
58	003A	ADH
59	003B	ADL Vorhergehende Zeilennummer
60	003C	ADH
61	003D	ADL Zeiger auf nächsten Befehl für CONT-Ausführung)
62	003E	ADH
63	003F	ADL Aktuelle Zeilennummer der laufenden DATA-Zeile
64	0040	ADH
65	0041	ADL Zeiger auf aktuelle DATA-Werte
66	0042	ADH
67	0043	ADL Zeiger für INPUT, READ und GET zum Speichern bei
68	0044	ADH CHRGET-Routine
69	0045	Name der laufenden Variable (1.Zeichen)
70	0046	" " " " (2.Zeichen)
71	0047	ADL Zeiger auf laufende Variable im RAM (Eintritts-
72	0048	ADH punkt nach dem Namen)
73	0049	ADL Zeiger auf laufende FOR....NEXT-Variable / oder
74	004A	ADH Zwischenspeicher für WAIT-Parameter
75	004B	ADL Zeiger auf laufenden Operator in ROM-Tabelle
76	004C	ADH
77	004D	Operatormaske für Formelauswertung; Bits 0,1,2 bedeuten <, = und >
78	004E	ADL Zeiger auf DEF FN-Funktion oder auf
79	004F	ADH Garbage Collect
80	0050	ADL Zeiger auf String-Deskriptor
81	0051	ADH
82	0052	Länge des Strings
83	0053	Konstante bzw. Schrittweite für den Gebrauch von Garbage Collect
84	0054	Code für JMP (76=$4C) zum Funktionsaufruf
85	0055	ADL Zeiger für Funktionsaufruf zur Formelauswertung
86	0056	ADH
87	0057	Beginn Fließkomma Akkumulator #3 (bis 92)
88	0058	ADL Zeiger für Blockverschiebung oder Teil von FAK #3
89	0059	ADH (neues Blockende+1) oder Teil von FAK #3
90	005A	ADL Zeiger für Blockverschiebung oder Teil von FAK #3
91	005B	ADH (altes Blockende+1) oder Teil von FAK #3
92	005C	gehört noch zu FAK #3
93	005D	Zähler für Stringumwandlung in Fließkomma
94	005E	Zähler für Stringumwandlung in Fließkomma
95	005F	ADL Zeiger für Blockverschiebung (alter Blockanfang)
96	0060	ADH
97	0061	Fließkomma Akkumulator #1 -> Exponent + 128
98	0062	-> Bruchteil MSB Fließkomma
99	0063	-> Bruchteil
100	0064	-> Bruchteil MSB Integer
101	0065	-> Bruchteil LSB
102	0066	-> Bruchteilzeichen
103	0067	Flag für Vorzeichen oder Zähler für Polynomauswertung
104	0068	Shiftzähler zur Normalisierung von FAK #1
105	0069	Fließkomma Akkumulator #2 (bis 110)
...	....	

Adresse dez	Adresse hex	Erklärung des Inhaltes
111	006F	Überlaufbyte für Vorzeichenvergleich aus FAK #1 und #2
112	0070	Byte zur Rundung für FAK #1
113	0071	ADL Zwischenspeicher für PRG-Zeiger L (VAL)
114	0072	ADH " " " H "
115	0073	Beginn der CHRGET-Routine (bis 138)
...	....	
139	008B	Nach dem entsprechenden Aufruf steht in den Zellen
...	....	139 bis 143 die nächste Zufallszahl
144	0090	Statusbyte für Berechnung von ST
145	0091	Inhalt von CIA 1 Port B zum Test der StOP- u.RVS-Taste
146	0092	Rekorderbedienung (Konstante für das Timing)
147	0093	I/O-Flag (0=LOAD, 1=VERIFY)
148	0094	Serieller Bus: Ausgabe Flag
149	0095	Bytepuffer für Seriellen Bus zur Ausgabe eines Zeichens
150	0096	Rekorderbedienung (Eingabe Flag für Bandende)
151	0097	Zwischenspeicher für X-Register (wird benötigt für GET# vom Band)
152	0098	Anzahl der geöffneten Files (maximal 10)
153	0099	Ersatzparameter für Eingabegerät (normal = 0 = Tastat.)
154	009A	Ersatzparameter für Ausgabegerät (normal = 3 = Schirm)
155	009B	Paritätsbyte für Kassettenausgabe
156	009C	Flag für gültige Bytes (Rekorderbedienung)
157	009D	Ausgabekontrolle bei Filenamensuche
158	009E	Offset für Filenamen im Header (Bandfehlertest)
159	009F	Zähler für Filenamenlänge (Bandfehlertest)
160	00A0	Uhrzeit 1. Stelle
161	00A1	Uhrzeit 2. Stelle
162	00A2	Uhrzeit 3. Stelle
163	00A3	Serieller Bit-Zähler für Rekorderbedienung
164	00A4	Zyklenzähler (zählt jedes Byte, das vom Band kommt)
165	00A5	Zähler für das Synchronisationssignal beim Bandschreib.
166	00A6	Indexzeiger in den Kassettenpuffer (0-192)
167	00A7	Fehler Flag beim Bandlesen
168	00A8	Flag für Bandstart / Fehler Flag beim Bandlesen
169	00A9	Flag für Lesefehler im Zyklus 1
170	00AA	Flag für Lesefehler im Zyklus 2
171	00AB	Flag für Bandlesen: 0 = Abtasten, 1-15 = zählen, 40 = LOAD, 80 = Kennzeichen für Bandende
172	00AC	ADL Startadresse für LOAD
173	00AD	ADH
174	00AE	ADL Endadresse für LOAD
175	00AF	ADH
176	00B0	Konstante zur Synchronisierung des Kassettenbetriebs
177	00B1	wie 176
178	00B2	ADL Zeiger auf Startadresse des Kassettenpuffers
179	00B3	ADH
180	00B4	Bandzähler bitweise (1 = gesetzt)
181	00B5	Band EOT oder aktuelles Ausgabe-Bit für RS232
182	00B6	Zwischenspeicher für auszugebendes RS232-Byte
183	00B7	Länge des Filenamens (0 = kein Name)
184	00B8	Logische Nummer des laufenden Files

Adresse dez	Adresse hex	Erklärung des Inhaltes
185	00B9	Sekundär Nummer des laufenden Files
186	00BA	Gerätenumer des laufenden Files
187	00BB	ADL Zeiger auf laufenden Filenamen
188	00BC	ADH
189	00BD	Hilfsspeicher für serielle Ein/Ausgabe
190	00BE	Anzahl der Blöcke bei Band Ein/Ausgabe
191	00BF	Serieller Wortpuffer
192	00C0	Flag für Rekorder zur Motorsteuerung (Tastenabfrage)
193	00C1	ADL Startadresse für Ein/Ausgabe
194	00C2	ADH
195	00C3	ADL Zeiger auf Tabelle der Interrupt- und I/O-Routinen
196	00C4	ADH
197	00C5	Matrixkoordinate der zuletzt gedrückten Taste
198	00C6	Tastaturpuffer-Index (Anzahl der gedrückten Zeichen)
199	00C7	Flag für Reversmodus am Bildschirm (0=normal/12=revers)
200	00C8	Zähler für Zeichenlänge einer Bildschirmzeile bei der Eingabe
201	00C9	Cursorzeile
202	00CA	Cursorspalte
203	00CB	aktueller Zeiger-Offset in die Tastaturtabelle
204	00CC	Flag für Cursor EIN/AUS (0=EIN, sonst AUS)
205	00CD	Zähler für Blinkdauer des Cursors
206	00CE	Zeichencode von aktueller Cursorposition
207	00CF	Flag für Cursor während der Blinkphase (1=EIN,0=AUS)
208	00D0	Flag für die Eingabe vom Bildschirm (=3) oder für die Eingabe von der Tastatur (=0)
209	00D1	ADL Adresse der aktuellen Cursorzeile (Anfangsadresse
210	00D2	ADH erste Spalte dieser Zeile)
211	00D3	Aktuelle Cursorspalte (0 bis 39)
212	00D4	Flag für Gänsefußmodus (0=aus-, sonst eingeschaltet)
213	00D5	enhält den Wert 39 (Länge einer Bildschirmzeile -1)
214	00D6	Nummer der aktuellen Cursorzeile (0 bis 24)
215	00D7	ASCII-Code der letzt gedrückten Taste oder sonstige Ausgabefunktionen
216	00D8	Zähler fär Anzahl gedrückter Tasten (Insert-Modus)
217	00D9	bis 242

		Tabelle der ADH-Werte der Bildschirmzeilen (jeweils Anfang der Bildschirmzeile mit Spalte 0) Die ADL-Werte sind im ROM-Bereich abgespeichert.
243	00F3	ADL Zeiger in Farbspeicher (RAM-Bereich)
244	00F4	ADH
245	00F5	ADL Zeiger auf Tastaturtabelle (Dort stehen die ASCII-
246	00F6	ADH Werte)
247	00F7	ADL Zeiger auf RS232 Eingabepuffer
248	00F8	ADH
249	00F9	ADL Zeiger auf RS232 Ausgabepuffer
250	00FA	ADH
251	00FB	bis 255 wird vom Betriebssystem nicht verwendet

Adresse dez	Adresse hex	Erklärung des Inhaltes
256	0100	bis 511 Stapelbereich ----------
		Der gesamte Stapel dient verschiedenen Funktionen. Einmal wird er als Zwischenspeicher für die Umwandlung von Zahlen in ASCII-Codes verwendet. Dann wiederum wird er als Zwischenspeicher für die Korrektur beim Bandlesen benützt. Hauptsächlich Hat er seine Berechtigung bei der Anwendung der Basic-Interpretation.
512	0200	Beginn des Basic-Eingabepuffers (bis 600)
601	0259	Beginn der Tabelle der 10 logischen Filenummern
611	0263	Beginn der Tabelle der 10 Gerätenummern
621	026D	Beginn der Tabelle der 10 Sekundäradressen
631	0277	Beginn der Tastaturpuffers (normalerweise bis 640)
641	0281	ADL Startadresse des Basic-RAM (normal 2048)
642	0282	ADH
643	0283	ADL Endadresse des Basic-RAM (normal 40960)
644	0284	ADH
645	0285	Flag für Zeitüberschreitung auf seriellen Bus
646	0286	Aktuelle Zeichenfarbe (Code 0-15)
647	0287	Aktuelle Zeichenfarbe an Cursorposition
648	0288	ADH-Wert des Bildschirmspeichers (normal 4 ergibt 1024)
649	0289	Länge des Tastaturpuffers (normal 10 Zeichen)
650	028A	Repeatfunktion für alle Tasten (Bit 7 = 1 dann Repeat, d.h. POKE 650,X / X-Werte > 128)
651	028B	Zähler für Wiederholungstakt
652	028C	Zähler für Repeatverzögerung
653	028D	Flag für Tastendruck mit SHIFT, C= und CONTROL (Bit 0, 1 und 2)
654	028E	Letzter Zustand der Shifttaste
655	028F	ADL Zeiger auf ROM-Routine, die die Tastatur auf SHIFT
656	0290	ADH COMMODORE- und CONTROL-Taste prüft.
657	0291	Flag für SHIFT und C=-gesperrt
658	0292	Flag für Automatisches Scrollen
659	0293	RS232 Controllregister
660	0294	RS232 Befehlsregister
661	0295	Bit-Übertragungszeit (nicht Standard)
662	0295	wie 661
663	0297	RS232 Statusregister
664	0298	Anzahl der zu sendenden Bits
665	0299	Baud Rate
666	029A	wie 665
667	029B	RS232 Zeiger auf Empfangsbyte
668	029C	RS232 Zeiger auf Eingabebyte
669	029D	RS232 Zeiger auf Übertragungsbyte
670	029E	RS232 Zeiger auf Ausgabebyte
671	029F	ADL Zwischenspeicher für IRQ-Vektor während des Kas-
672	02A0	ADH settenbetriebes
673	02A1	NMI-Flag --> CIA 2
674	02A2	TIMER A --> CIA 1

Adresse dez	Adresse hex	Erklärung des Inhaltes
675	02A3	Interruptflag --> CIA 1
676	02A4	Flag für TIMER A --> CIA 1
677	02A5	Zwischenspeicher für Bildschirmzeilenberechnungen
678	02A6	Flag für Farbsystem bei Bildscirmen (1=PAL/0=NTSC)
679	02A7	bis 703 ungenutzt
704	02C0	Spritepuffer
768	0300	ADL 58251 = $E38B Vektor zeigt auf READY-Routine
769	0301	ADH
770	0302	ADL 42115 = $A483 Vektor zeigt auf Eingabewarteschleife
771	0303	ADH
772	0304	ADL 42364 = $A57C Vektor zeigt auf Routine zur Umwand-
773	0305	ADH lung einer Zeile in komprimierten Basictext
774	0306	ADL 42778 = $A71A Vektor zeigt auf Routine zur Umwand-
775	0307	ADH lung in Basictext für LIST
776	0308	ADL 42980 = $A7E4 Vektor zeigt auf Routine die für den
777	0309	ADH Aufruf der Basic-Routinen verantwortlich ist
778	030A	ADL 44678 = $AE86 Vektor zeigt auf Routine zur Formel-
779	030B	ADH auswertung (FRMEVL)
780	030C	Zwischenspeicher für SYS-Aufruf 6510-Akkuwert
781	030D	Zwischenspeicher für SYS-Aufruf 6510-X-Registerwert
782	030E	Zwischenspeicher für SYS-Aufruf 6510-Y-Registerwert
783	030F	Zwischenspeicher für SYS-Aufruf 6510-Statusregisterwert
784	0310	76 = $4C -> OP-Code für Sprungbefehl JMP (für USR)
785	0311	ADL 54640 = $B248 Vektor zeigt innerhalb der Routine
786	0312	ADH "dimensionierte Variable holen" auf Fehlermeldung Routine ILLEGAL QUANTITY Falls keine USR-Adresse an die Speicherstellen 785 und 786 übergeben wird, erscheint nach Aufruf von USR die Fehlermeldung ILLEGAL QUANTITY
787	0313	Wird vom Betriebssystem nicht verwendet
788	0314	ADL 59953 = $EA31 Vektor zeigt auf Routine IRQ
789	0315	ADH
790	0316	ADL 65126 = $FE66 Vektor zeigt auf Routine BRK
791	0317	ADH
792	0318	ADL 65095 = $FE47 Vektor zeigt auf Routine NMI
793	0319	ADH
794	031A	ADL 62282 = $F34A Vektor zeigt auf Routine OPEN
795	031B	ADH
796	031C	ADL 62097 = $F291 Vektor zeigt auf Routine CLOSE
797	031D	ADH
798	031E	ADL 61966 = $F20E Vektor zeigt auf Routine CHKIN
799	031F	ADH
800	0320	ADL 62032 = $F250 Vektor zeigt auf Routine CKOUT
801	0321	ADH
802	0322	ADL 62259 = $F333 Vektor zeigt auf Routine CLRCH
803	0323	ADH
804	0324	ADL 61783 = $F157 Vektor zeigt auf Routine BASIN
805	0325	ADH
806	0326	ADL 61898 = $F1CA Vektor zeigt auf Routine BSOUT
807	0327	ADH
808	0328	ADL 63213 = $F6ED Vektor zeigt auf Routine STOP

Adresse dez	Adresse hex	Erklärung des Inhaltes
809	0329	ADH
810	032A	ADL 61758 = $F13E Vektor zeigt auf Routine GETIN
811	032B	ADH
812	032C	ADL 62255 = $F32F Vektor zeigt auf Routine CLALL
813	032D	ADH
814	032E	ADL 65126 = $FE66 Vektor zeigt auf Routine INIT
815	032F	ADH
816	0330	ADL 62629 = $F4A5 Vektor zeigt auf Routine LOAD
817	0331	ADH
818	0332	ADL 62957 = $F5ED Vektor zeigt auf Routine SAVE
819	0333	ADH
820	0334	Beginn des Kassettenpuffers
...	....	
832	0340	Beginn Spritepuffer 13 (832 = 13*64)
...	....	
896	03BE	Beginn Spritepuffer 14 (896 = 14*64)
...	....	
960	03C0	Beginn spritepuffer 15 (960 = 15*64)
...	....	
1024	0400	bis 2048 Beginn der Bildschirmspeichers

Wie Sie an dieser Übersicht der ersten 2048 Speicherstellen erkennen können, benötigt das Betriebssystem diese Hilfsspeicherstellen, um die verschiedensten Aufgaben durchzuführen. Der Grund dafür, warum diese Speicherstellen im RAM-Bereich liegen, ist der, daß ja im ROM-Bereich keine Veränderungen stattfinden können.

Sie werden vorerst mit diesen Speicherstellen kaum etwas anfangen können. Erst nach einiger Programmierübung am Gerät selbst werden wir dann die Möglichkeiten dieser Zellen ausnützen.

Dieser Speicherbereich steht also dem reinen Basicprogrammierer nicht zur Verfügung. Deshalb kann man in Basic nicht 40960 Bytes verwenden, sondern nur 38911 Bytes (2048 bis 40959). Das kommt auch noch dadurch zum Ausdruck, daß nach dem Einschalten der Geräte folgender Text am Bildschirm erscheint:

```
**** COMMODORE 64 BASIC V2****
64K RAM SYSTEM  38911 BASIC BYTES FREE
READY
```

Bevor wir weiter ins Detail des CBM gehen, müssen wir uns ein paar kleinere Programmbeispiele ansehen, um so grundsätzliche Programmierkenntnisse zu erlangen.

2.2 Ein erstes Maschinenprogramm

Eine wichtige Erkenntnis ist, daß der Bildschirm so behandelt wird, als wäre er ein Speicher. Tatsächlich haben wir auch einen Bildschirmspeicher, der ein RAM-Baustein ist.

Dieser ist nun über die geeignete Hardware mit dem Videoteil verbunden. Das hat zur Folge, daß die 1000 Bildschirmzellen so angesprochen werden, als handle es sich um Adressen im Gesamtspeicher. Wenn wir ein Zeichen auf den Bildschirm bringen wollen, dann benötigt man nur den Bildschirmcode des Zeichens und schreibt ihn in den Bildschirmbereich ein. Als Fallbeispiel wollen wir einmal das Wort „COMMODORE" auf den Bildschirm bringen. Den Bildschirmcode entnehmen wir dabei aus dem ROM-Bereich. Dort muß er auf jeden Fall enthalten sein, da ja beim Einschalten des Gerätes der Text „****COMMODORE 64 BASIC****" am Bildschirm erscheint. Desweiteren soll das Wort in der rechten unteren Bildschirmecke stehen. Das Programm sieht nun folgendermaßen aus:

	hex	dez
1.	JSR $E544	JSR 58692
2.	LDX #$00	LDX #0
3.	LDA $E47E,X	LDA 58494,X
4.	STA $07DF,X	STA 2015,X
5.	INX	INX
6.	CPX #$0A	CPX #10
7.	BNE $F5	BNE −11
8.	RTS	RTS

Wollen wir nun das Programm genau untersuchen. Das ganze Programm besteht aus 8 Befehlen. Der erste Befehl ruft ein Unterprogramm im Betriebssystem auf, das den Bildschirm löscht. Das wird auch in Basic durch

PRINT CHR$ (147)

oder durch die Steuerfunktion mit den Tasten „SHIFT + CLR/HOME" erreicht. Der 2. Befehl (LDX) lädt in das X-Register den Wert Null. Das Indexregister wird im Programm als Buchstabenzähler verwendet. Im 3. Befehl wird auf das Wort „COMMODORE", das im ROM-Bereich liegt, indiziert zugegriffen. Das heißt, zur Adresse 58494 wird der Wert des X-Registers addiert und das ist beim ersten Zugriff Null. Deshalb wird der Code aus der Adresse 58494 geholt. Von der Adresse 58494 stehen folgende Codes:

Inhalt dezimal	67	79	77	77	79	68	79	82	69
Inhalt ASCII	C	O	M	M	O	D	O	R	E

Durch den 4. Befehl wird der jeweilige Bildschirmcode auf den Bildschirm gebracht, wobei der X-Wert wiederum addiert wird. Der 5. Befehl (INX) erhöht den Wert im X-Register um eins. Das bedeutet, daß beim zweiten Zugriff zur Adresse 58494 der Wert 1 hinzuaddiert wird und damit eben auf die Adresse 58494 eingegangen wird. „CPX #10" im 6. Befehl vergleicht den Wert im X-Register mit dem Wert „10". Wenn dieser Wert nicht gleich ist (BNE im 7. Befehl), wird zum 3. Befehl zurückgesprungen und der nächste Zugriff beginnt. Ist der X-Wert „10", dann fährt das Programm im 8. Befehl fort und das bedeutet mit „RTS" den Rücksprung aus diesem Programm.

Wie bringt man nun ein solches Programm in den Speicher? Natürlich steht im Speicher nicht „LDA" oder „STA", sondern der entsprechende OP-Code. Der OP-Code für die Befehle kann aus Kapitel 1 oder dem Anhang entnommen werden. Für den VC-64 gibt es lediglich eine Möglichkeit die Programmcodes in den Speicher zu übertragen.

Die Basic-Methode

Die Methode um dieses Maschinenprogramm im RAM abzuspeichern, bietet Basic mit dem Statement „POKE". Mit „POKE" werden entsprechende Daten im Speicher abgelegt. Allerdings benötigt „POKE" seine Parameter in dezimaler Form. Zu diesem Zweck müssen die hexadezimalen Werte in dezimale umgewandelt werden. Für die Umwandlung steht im Anhang eine Tabelle zur Verfügung. Die dezimale Sequenz des Programms sieht folgendermaßen aus:

32,68,229,162,0,189,126,228,157,223,7,232,224,10,208,245,96

Man könnte nun mit

```
POKE 634,32
POKE 635,68
POKE 636,229
:
:
usw.
```

das Maschinenprogramm Byte für Byte in die entsprechenden Speicherstellen bringen. Dies wäre jedoch nicht sehr effizient. Es gibt da eine elegantere Methode, die durch ein Basicprogramm realisiert wird. Für unser Beispiel würde das Basicprogramm etwa so aussehen:

```
10 DATA 32,68,229,162,0,189,126,228,157
20 DATA 223,7,232,224,10,208,245,96
30 FOR I = 828 TO 844:READ X:POKE I,X:NEXT
```

Bisher haben wir uns als Platz für das Maschinenprogramm die Speicherstellen ab 828 ausgesucht. Das ist der Beginn des Kassettenpuffers. Solange man nicht mit dem Rekorder arbeitet, spielt das auch keine Rolle. Ansonsten müssen Sie sich einen anderen Bereich suchen. Die dezimalen Codes werden in DATA-Statements der Reihenfolge nach abgelegt. Dann wird eine FOR. . NEXT-Schleife aufgestellt, deren Schleifenvariable „I" die Adressen der Speicherstellen darstellt. Bei jedem Schleifendurchgang werden die dezimalen Codes in die Variable „X" eingelesen und mit „POKE" auf die entsprechende Speicherstelle gesetzt.

Mit „RUN" wird dann das gesamte Maschinenprogramm in den Speicherbereich übertragen. Danach kann dieses Maschinenprogramm sofort zur Ausführung gebracht werden. Man muß nur den Programmzähler auf die Adresse 828 setzen. Das geschieht mit „SYS". Wird nun im Direktmodus „SYS 828" gegeben, sollte am rechten unteren Bildeck das Wort „COMMODORE" erscheinen. Tut es das nicht, so muß noch der Farbspeicher umgestellt werden. Das Programm hat zwar die Codes in den Bildschirm geschrieben, aber mit der gleichen dunkelblauen Farbe des Hintergrundes. Daher ist es zweckmäßig, die Hintergrundfarbe mit „POKE 53281,1" umzustellen. Löschen Sie nun noch einmal den Bildschirm und geben „SYS 828" ein. Jetzt erscheint unmittelbar rechts unten „COMMODORE", aber nicht Buchstabe für Buchstabe, sondern blitzschnell als gesamtes Wort.

Man wird sich nun fragen, mit

```
10 PRINT CHR$(147)
20 FOR I = 1 TO 23 : PRINT : NEXT
30 PRINT TAB(31) „COMMODORE“
40 PRINT CHR$(19)
```

erreicht man doch ohne viel Umstände das Gleiche? Nun, dieses Maschinenprogramm wurde nur zu Demonstrationszwecken geschaffen und soll das Verständnis für die Maschinenprogrammierung wecken. Außerdem zeigt es ein wenig auf, wie schnell Maschinenprogramme sind. Das kommt zwar noch nicht so richtig zur Geltung, aber Sie werden später noch sehen wie schnell solche Programme sind.

2.3 Assemblerprogramme

Bisher haben wir die Umwandlung der mnemonischen Befehle und deren Operanden in die hexadezimale Sequenz (Maschinencode) manuell durchgeführt. Das Umwandeln selbst bezeichnet man mit

Assemblieren.

Das Assemblieren per Hand ist ansich nur bei sehr kleinen Routinen, wie zum Beispiel im vorhergehenden Programm, sinnvoll. Sollen größere Programmierungen vollzogen werden, wird man ein eigenes Programm benützen, das die Umwandlungen automatisch vornimmt. Solch ein Programm nennt man dann:

Assembler.

Es gibt auf dem Markt eine Reihe vonAssemblerprogrammen, wobei Versionen in Basic und Maschinensprache angeboten werden. Dazu muß man auch noch unterscheiden zwischen Programmierhilfen in TOOLKITS und direkten Assemblern (z.B. MAE), die dann mit symbolischen Namen für die Adressen arbeiten. Wir wollen hier nun nicht näher darauf eingehen, sondern uns erst später damit befassen.

Wenn wir nun irgendein Maschinenprogramm abgespeichert, bzw. von Diskette oder Kassette geladen haben, dann kann man mit der Ansammlung der Maschinencodes nicht viel anfangen. Wir müssen die einzelnen Codes erst in ihre mnemonische Form bringen bzw. in Assemblerschreibweise zurückverwandeln. Das könnten wir manuell machen oder eben wiederum mit einem Programm. Das Zurückverwandeln nennt man **„Disassemblieren“** und das Programm das dies durchführt **„Disassembler“**.

Wir haben nun einen ersten Einblick in die Maschinenprogrammierung gewonnen und werden uns nun anschauen, wie der Basic-Interpreter im Betriebssystem arbeitet. Dazu müssen wir wissen, welche Datenarten unser VC-64 verarbeitet.

2.4 Die Datenarten des VC-64

Es können folgende Datenarten vorkommen:

Bytedaten
Adressen
Integerzahlen
Gleitkommazahlen (Fließkomma)
Zeichen (Strings – > ASCII oder Bildschirmcode)

2.4.1 Bytedaten

Die grundsätzliche Form von Daten, die der Computer enthält, besteht pro Zeichen aus einem Bitmuster mit 8 Bits, die man zusammengefaßt als Byte bezeichnet. Durch die Anordnung der Bits können „2 hoch 8 = 256" verschiedene Bytes dargestellt werden. Die dezimale Bezeichnung der Bytes geht von 0 bis 255. Die hexadezimale Sequenz hat den Bereich von $00 bis $FF. Binär kann man ein Byte von %00000000 bis %11111111 verschieden anordnen.

2.4.2 Adressen

Da der Gesamtspeicher zur CPU 6502 aus 65536 Speicherstellen bestehen kann, benötigt man für die Adressen der Speicherstellen zu deren Darstellung 16 Bits (2 hoch 16 = 65536). Die CPU ist ein 8-Bit-Prozessor. Die 16 Bits können deshalb nicht als eine Einheit betrachtet werden, sondern als zwei Teile mit je 8 Bits bzw. 2 Bytes. Das ist auch der Grund, warum man den Adreßbereich in Seiten (Page) und Nummern einteilt, und den Page-Wert als oberen Adreßteil (ADH) und die Nummer in der Page als unteren Adreßteil (ADL) spezifiziert.

Die Adressen sind vorzeichenlose ganze Zahlen. Die nachfolgende Abbildung stellt die hexadezimale Sequenz der dezimalen gegenüber und zeigt außerdem deren Verbindung untereinander auf.

Adressenbeispiele

Hexadezimale Sequenz			Dezimale Sequenz			
Adresse	ADL	ADH (PAGE)	ADH	ADL	Verbindung ADH*256+ADL	= Adresse
$0000	$00	$00	0	0	0*256+0	0
$0001	$01	$00	0	1	0*256+1	1
$0100	$00	$01	1	0	1*256+0	256
$0101	$01	$01	1	1	1*256+1	257
$80FF	$FF	$80	128	255	128*256+255	33023
$FFFE	$FE	$FF	255	254	255*256+254	65534
$FFFF	$FF	$FF	255	255	255*256+255	65535

Die dezimale Darstellung der Adressen sowie deren Berechnung sind unumgänglich für die Bearbeitung in Basic mit „PEEK" und „POKE".

2.4.3 Integerzahlen

Integerzahlen sind ganze Zahlen. Sie werden im VC mit 2 Bytes verwaltet. Von diesen 16 Bits wird das vorderste Bit als Vorzeichen verwendet. Was wiederum heißt, daß negative Zahlen im Zweierkomplement dargestellt werden. Mit den restlichen 15 Bits kann man nun „2 hoch 15 = 32768" Zahlen anordnen. Daher haben die Integerzahlen folgenden Bereich:

$8000	=	–32768
$8001	=	–32767
:		:
$FFFE	=	–2
$FFFF	=	–1
$0000	=	0
$0001	=	1
$0002	=	2
:		:
$7FFE	=	32767
$7FFF	=	32768

Der Bereich der Integerzahlen geht somit von $8000 (–32768) bis $7FFF (+32767).

2.4.4 Gleitkommazahlen (Fließkomma)

FK-Zahlen sind reelle Zahlendaten. Diese werden nun nicht wie bei den Integerzahlen so abgespeichert, je mehr Bits desto größer die Zahl, sondern auf eine andere Art. Dabei werden im VC zwei Formate benutzt:

Das Speicherformat
Das Registerformat

Der Unterschied zwischen beiden ist der, daß bei der Abspeicherung von FK-Zahlen der Interpreter das Speicherformat benützt. Werden mit FK-Zahlen aber arithmetische Operationen durchgeführt, so verwendet der Rechner das Registerformat.

2.4.4.1 Das Speicherformat

Um das Grundprinzip dieses Formats zu verstehen, müssen wir noch einmal auf die Darstellung von Dezimalzahlen zurückgreifen. Nehmen wir uns z.B. die Zahl „0,00364" vor. Diese Zahl hat nach dem Komma zwei Nullen stehen. Die Nullen haben nun keine andere Funktion, als die Stelle des Kommas anzugeben. Wir können nun die Zahl aber auch noch anders darstellen, nämlich in Exponentialform:

$0{,}364 * 10^{-2}$

Bei dieser Form fallen die 2 Nullen in der Mantisse weg. „–2" stellt den Exponenten dar. Das Überführen einer Zahl in seine Exponentialform wird durch Streichen bedeutungsloser Nullen links von ihr und anschließendes Anpassen des Exponenten gewonnen. Betrachten wir noch eine andere Zahl:

Normalform:	79,51
Exponentialform:	$0{,}7951 * 10^{2}$

Man könnte nun die beiden Beispielzahlen allerdings in Exponentialdarstellung auch so anschreiben:

$0{,}00364 = 0{,}00364 * 10^0$
$79{,}51 = 79{,}51 * 10^0$

Dies ist aber nicht sehr sinnvoll, da ja „10 hoch 0 = 1" ist. Bei der Normalform der Exponentialdarstellung steht also das Komma unmittelbar links vor der Zahl, während der Exponent angibt, um wieviel Stellen das Komma gerückt werden muß. Bei einem negativen Exponenten wird das Komma nach links verschoben. Ist der Exponent positiv, so erfolgt eine Schiebung nach rechts. Dabei wird jeweils das Komma um soviele Stellen gerückt, wie die Zahl des Exponenten angibt.

Beispiele:

$0{,}125 * 10^3 = 125{,}0$
$0{,}1 * 10^{-1} = 0{,}01$

Da üblicherweise eine Zahl an die Normalform der Exponentialdarstellung angepaßt wird, nennt man diese Überführung:

Normalisierung

Deswegen wird die Mantisse, deren Komma direkt links vor der Zahl steht, als normalisierte Mantisse bezeichnet. Ganz allgemein kann man für eine normalisierte Zahl schreiben:

$Z \text{ normalisiert} = M * 10^E$

Die normalisierte Mantisse wird durch eine Zahl zwischen 1 und 0,1 charakterisiert, sofern es sich nicht direkt um den Wert 0 handelt. Somit kann man den Wert der Mantisse auch durch folgende Ungleichung mathematisch ausdrücken:

$0{,}1 =< M < 1 \text{ oder } 10 =< M < 10$

Sie werden sich nun fragen, für was man denn dies braucht? Nun, für die binäre Sequenz gilt das gleiche:

$2^{-1} =< M < 2^0 \text{ oder } 0{,}5 =< M < 1$

Hier wird dabei die Mantisse ohne Vorzeichen betrachtet. Schauen wir uns einmal eine Binärzahl an:

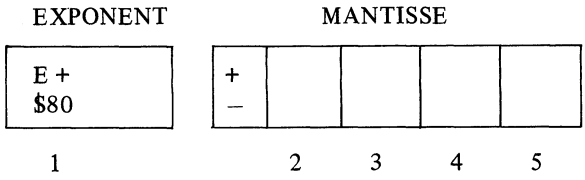
% 110,1 = 6,5
Normalisiert: $0{,}1101 * 2^3 = 0{,}65 * 10^1$

Im Computer können die Null und das Komma wegfallen. Es werden also nur die Mantissenzahl und die Exponentialzahl benötigt. Für dieses Beispiel sieht das dann so aus:

Mantisse = 1101
Exponent = 3

Da wir nun das Prinzip der Normalisierung einer Zahl kennengelernt haben, können wir dazu übergehen, das Speicherformat einer Fließkommazahl im VC-64 festzulegen.

Im Speicherformat des VC sind die Zahlen in 5 hintereinander liegenden Bytes abgelegt. Das erste Byte stellt den Exponenten dar. Die restlichen 4 Bytes werden von der Mantisse belegt.



Da der VC-Interpreter einen negativen Exponent oder eine negative Mantisse nicht im Zweierkomplement darstellt, werden die FK-Zahlen auf andere Weise normalisiert.

Zum Exponent wird $80 addiert. Dadurch vermeidet man, daß der Exponent einen negativen Wert erhält. Damit geht der Wertebereich des Exponent von 0 bis 255.

Beispiele:

TATSÄCHLICHER WERT hex	dez	SPEICHERFORMAT WERT hex	dez	DEZIMAL E-WERTE	
$7F	127	$FF	255	2 hoch	127 = 1,7 * 10 hoch 38
$1C	28	$9C	156	2 hoch	28 = 2,7 * 10 hoch 8
$01	1	$81	129	2 hoch	1 = 2
$00	0	$80	128	2 hoch	0 = 1
$FF	–1	$7F	127	2 hoch	–1 = 0,5
$E4	–28	$64	100	2 hoch	–28 = 3,7 * 10 hoch –9
$80	–128	$00	0	2 hoch	–128 = 2,9 * 10 hoch –39

Diese Tabelle zeigt auch sehr schön auf, in welchem Zahlenbereich mit Gleitkommazahlen gearbeitet werden kann. Wenn der Wert des Exponent größer als $80 ist, wird das Komma nach rechts verschoben; ist der Wert kleiner, wird das Komma nach links geschoben. So geht z.B. bei $83 das Komma um 3 Stellen nach rechts und bei $7F um 2 Stellen nach links.

Wenn die Fließkommazahl direkt den Wert Null hat, steht in allen 5 Bytes „0".

0 = $00 00 00 00 00

Da jede Zahl größer Null einen Zahlenwert besitzt, sich also mindestens ein Bit mit dem Zustand „1" in den Mantissenbytes befindet, ist das grundlegende normalisierte Format einer FK-Binärzahl:

% 0,1 * 2^0 = % 1,0 * 2^{-1} = 1/2 = 0,5

Normalerweise würde nun die FK-Zahl „0,5" im Speicher auf folgende Weise abgebildet sein:

$80 80 00 00 00

Durch diese Darstellung könnte man aber nicht das Vorzeichen darstellen. Deshalb wird beim Abspeichern des Formats das vorderste Bit weggelassen und durch ein Vorzeichenbit ersetzt, wobei eine „0" einen Pluswert und eine „1" einen Minuswert anzeigt. Wir können nun im Endeffekt die Zahlen „0,5" und „–0,5" im Speicherformat so darstellen:

0,5 = $80 00 00 00 00
–0,5 = $80 80 00 00 00

Sie werden sich nun fragen, wenn wir Berechnungen mit FK-Zahlen durchführen, werden die Ergebnisse durch dieses Format nicht verfälscht? Die Erklärung dafür ist recht einleuchtend.

Das Speicherformat ist nur zur Abspeicherung der Fließkommazahlen zuständig. Solange die Zahlen im Speicher stehen, wird ja nicht damit manipuliert. Dadurch genügen die 5 Bytes zur Darstellung. Das ist nun bei der Durchführung von Zahlenoperationen anders. Dazu wird das Speicherformat in das Registerformat umgewandelt.

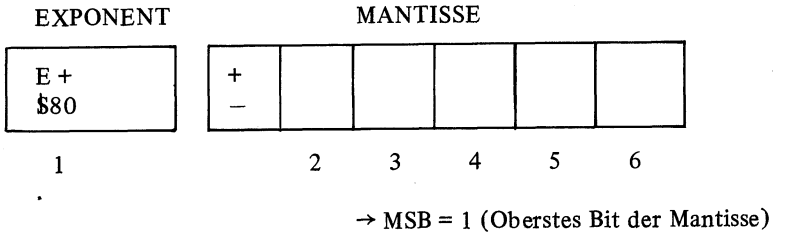
2.4.4.2 Das Registerformat

Wenn wir eine Berechnung ausführen, z.B.:

3,65 * 0,2

dann brauchen wir die Fließkommazahlen im Registerformat.

Das Registerformat wird durch 6 Bytes abgebildet:



Der Exponent (1 Byte) wird genauso dargestellt wie im Speicherformat, also mit dem Additionswert $80. Die Bytes 2 bis 5 enthalten die Mantisse wie beim Speicherformat. Das Byte 6 stellt das Vorzeichen der Mantisse dar.

Da mit dem Registerformat gerechnet wird, muß das Grundformat einer binären FK-Zahl erhalten bleiben. So ist das oberste Bit der Mantisse in Byte 2 des Registerformats immer auf „1" gesetzt. Dadurch kann natürlich dort kein Vorzeichenbit stehen. Deshalb wird das Vorzeichen im Byte 6 abgespeichert. Wie wird nun eine Fließkommazahl vom Speicherformat in das Registerformat umgewandelt und umgekehrt? Für diese Operationen nehmen wir uns einmal als Beispiel die FK-Zahlen „0,5" und „–0,5" vor.

1. Beispiel:

Speicherformat: 0,5 = $80 00 00 00 00

1	2	3	4	5	6
% 10000000	00000000	00000000	00000000	00000000	
	└──────────────────────────────────────→				00000000
% 10000000	00000000	00000000	00000000	00000000	00000000
	10000000 (ODER-Verknüpfung)				
ergibt					
% 10000000	10000000	00000000	00000000	00000000	00000000

= Registerformat: 0,5 = $80 80 00 00 00 00

2. Beispiel:

Speicherformat: –0,5 = $80 80 00 00 00

1	2	3	4	5	6
% 10000000	10000000	00000000	00000000	00000000	
					→ 10000000
% 100000000	10000000	00000000	00000000	00000000	10000000
	10000000 (ODER-Verknüpfung)				
ergibt					
% 10000000	10000000	00000000	00000000	00000000	10000000

= Registerformat: 0,5 = $80 80 00 00 00 80

Ganz allgemein kann man sagen:

Zuerst wird das Speicherformat in die ersten 5 Bytes des Registerformats übertragen. Dann wird das Byte 2 nach Byte 6 dupliziert und anschließend wird Byte 2 mit „$80" durch eine logische ODER-Verknüpfung behandelt. Das Ergebnis ist dann das Registerformat.

Die Umwandlung vom Registerformat ins Speicherformat ist noch wesentlich einfacher abzuwickeln. Das Byte 6 wird in Byte 2 eingesetzt. Danach braucht man nur noch Byte 6 zu löschen und schon hat man das Speicherformat.

Schauen wir uns einmal in der folgenden Tabelle ein paar Fließkommazahlen in beiden Formaten an.

Dezimal	Speicherformat	Registerformat
4	83 00 00 00 00	83 80 00 00 00 00
3	82 40 00 00 00	82 C0 00 00 00 40
2	82 00 00 00 00	82 80 00 00 00 00
1,5	81 40 00 00 00	81 C0 00 00 00 40
1	81 00 00 00 00	81 80 00 00 00 00
0,5	80 00 00 00 00	80 80 00 00 00 00
0,25	7F 00 00 00 00	7F 80 00 00 00 00
0	00 00 00 00 00	00 00 00 00 00 00
–0,25	7F 80 00 00 00	7F 80 00 00 00 80
–0,5	80 80 00 00 00	80 80 00 00 00 80
–1	81 80 00 00 00	81 80 00 00 00 80
–1,5	81 C0 00 00 00	81 C0 00 00 00 C0
–2	82 80 00 00 00	82 80 00 00 00 80
–3	82 C0 00 00 00	82 C0 00 00 00 C0
–4	83 80 00 00 00	83 80 00 00 00 80

Wie schon erwähnt: das Speicherformat dient zur Abspeicherung der Fließkommazahlen im RAM-Bereich. Wo werden nun die Zahlen im Registerformat abgespeichert? Dazu dienen mehrere Speicherstellen in der Zeropage. Wenn Sie sich in der Übersicht der ersten 1023 Speicherstellen umsehen, finden Sie dort 3 Pseudoregister.

87 – 92 : FAK #3
97 – 102 : FAK #1
105 – 110 : FAK #2

FAK ist die Abkürzung für Fließkommaakkumulator. Diese Pseudoregister sind Zwischenspeicher für die Fließkommazahlen im Registerformat. Sämtliche Berechnungen des Interpreters werden mit diesen Registern durchgeführt.

2.4.5 Zeichen

Wie wir von Basic her wissen sollten, können Daten als Zahlen oder als Zeichen interpretiert werden. Wenn nur Zeichen verarbeitet werden, dann spricht man von Stringverarbeitung. Ein String kann aus einem oder mehreren Zeichen (Zeichenfolge) bestehen. Es muß noch festgelegt werden, welches Bitmuster (Byte) welchem Zeichen entspricht und welche Zeichen überhaupt verwendet werden. Das Zuordnen eines entsprechenden Zeichens zu einem Byte nennt man Codierung. Einer der bekanntesten Codes für Mikrocomputer ist der ASCII-Code, der bereits in Kapitel 1 aufgeführt ist. Leider ist es nun so, daß der ASCII-Code mit dem Code im VC-64 nicht völlig identisch ist. Wir werden deshalb im folgenden den Code des VC-64 als ASC64-Code bezeichnen.

2.4.6 Der ASC64-Code

Der Zeichencode des VC-64 ist bis $5F (= 95) identisch mit dem ASCII-Code. Erst ab $60 besteht ein wesentlicher Unterschied. Einer der Hauptgründe dafür ist der, daß der ASCII-Code nur einen Zeichenvorrat von 128 Zeichen hat. 8 Bit werden hier nämlich als Paritätsmerkmal verwendet. Der ASC64-Code kann dagegen auf 256 Zeichen zurückgreifen. Zudem kann der VC-64 in 2 Modis umgeschaltet werden. Einmal gibt es da den Graphikmodus, der bei den meisten Geräten der VC-Serie automatisch beim Einschalten eingestellt ist. Zum anderen hat man den Textmodus zur Verfügung. Natürlich kann man bei allen Geräten in den jeweilig anderen Modus umschalten. Damit enthält der VC zusätzlich noch einige Graphikzeichen, die beim ASCII-Code nicht vorhanden sind; und wir dürfen dabei auch nicht die Farbencodes vergessen.

2.4.7 Der Bildschirmcode

Der Bildschirmcode ist nun leider auch wieder nicht gleich zum ASC64-Code, da die Matrixlogik des Videoteils völlig andere Wege geht als z.B. die Logik der Tastatur. Der BS-Code besitzt nun auch einen Zeichenvorrat von 256 Bytes. Die Zeichen von 0 – 127 sind völlig gleich mit den Zeichen von 128 – 255. Der Unterschied besteht darin, daß die zweite Zeichenreihe ab 128 in reverser (schwarz auf hellem Grund) Darstellung erfolgt.

2.4.8 Der Zusammenhang zwischen BS-, ASC64- und ASCII-Code

Vielfach ist es so, daß am VC-64 noch ein Drucker dranhängt, der nun mit dem echten ASCII-Code arbeitet. Wenn nun ein Maschinenprogramm geschrieben wird, das mit Bildschirm, Tastatur und Drucker arbeitet, müssen natürlich die einzelnen Codes untereinander umgewandelt werden. Wenn man sich die einzelnen Bits der binären Darstellung der Codes betrachtet, so wird man feststellen, daß nur Manipulationen mit den vordersten 3 Bits gemacht werden müssen, um die entsprechenden Umwandlungen zu vollziehen.

Im Handbuch ist nun der gesamte ASC64-Code und der Bildschirmcode abgebildet. Sie sollten sich einmal die Mühe machen, die einzelnen Bitmuster der verschiedenen Codes einander gegenüberzustellen. Sie werden dann sehen, welche Bits sich unterscheiden. Zudem ist das eine sehr gute Übung zur Zahlenumwandlung.

Im nächsten Abschnitt werden wir uns dann um die Verwaltung der Daten und Zeichen kümmern, das bei Basic der Interpreter des Betriebssystems übernimmt.

2.5 Der Basic-Interpreter

Das Herzstück unseres VC-64 ist der Basic-Interpreter. Damit kann der Anwender, der nur Basic beherrscht, mit dem Computer in Kontakt treten. Wir wollen nun einmal sehen, wie sich der Interpreter aus der Sicht des Maschinenprogrammierers verhält. Dieses Wissen wäre auch für den „Nur-Basic-Programmierer" von Bedeutung, da man dadurch mit Basic einige Tricks vornehmen kann.

Wenn wir z.B. die Basic-Zeile

```
10 PRINT,,COMMODORE"
```

eingeben, dann legt der Interpreter die Zeichenfolge „COMMODORE" als ASC64-Code in Hilfsspeicherstellen ab. Die Hilfsspeicherstellen befinden sich im RAM-Bereich von der Adresse 512 ($0200) bis 592 ($0250), also 80 Speicherstellen. Diesen Bereich bezeichnet man als Basic-Eingabepuffer (oder BASIC INPUT PUFFER). Hier erfolgt praktisch die Eingabe der Basic-Instruktionen. Nun wird vielleicht auch klar, warum eine Basic-Zeile nicht mehr als 80 Zeichen enthalten kann.

Solange nicht „RETURN" gedrückt wird, passiert nichts. In dem Moment, in dem die Tastaturabfrage registriert, daß die Returntaste gedrückt wurde, wird der Aufruf zu verschiedenen ROM-Routinen veranlaßt. Wir wollen nun an dieser Stelle nicht jede einzelne Routine auseinandernehmen, sondern nur das grundlegende Verhalten betrachten.

Nach dem Drücken von „RETURN" wandelt eine Routine den Text im Eingabepuffer in eine komprimierte Zeichenfolge um. Dabei wird noch durch eine andere Routine abgefragt, ob dieser Text überhaupt eine Zeilennummer hat. Wenn nicht, dann wird direkt zur Interpreterroutine „PRINT" gesprungen und damit der Befehl im Direktmodus ausgeführt. Falls die Instruktionszeile eine Zeilennummer enthält, wird der komprimierte Text im RAM-Speicher abgelegt und zwar dort, wo eben die Basicprogramme abgespeichert sind. Normalerweise ist das die Speicherstelle 2049 (= $ 801). Eigentlich beginnt der Basicbereich bei 2048. Dort steht immer eine „0", um eben den Beginn zu kennzeichnen. Der Basic-Code ist auf folgende Art im Speicher abgelegt, wobei hier das vorige Beispiel zugrunde gelegt wird.

Adresse dez	Adresse hex	Inhalt dez	Inhalt hex	Erklärung
2048	0800	0	00	ist immer der Fall (Erklärung folgt)
2049	0801	18	12	ADL Zeiger auf neue Basiczeile
2050	0802	8	08	ADH Zeiger auf neue Basiczeile
2051	0803	10	0A	Zeilennummer (niederwertiges Byte)
2052	0804	0	00	Zeilennummer (höherwertiges Byte)
2053	0805	153	99	Interpreter-Code für PRINT
2054	0806	34	22	ASC64-Code für PRINT
2055	0807	67	43	ASC64-Code für C (Graphikmodus)
2056	0808	79	4F	ASC64-Code für O "
2057	0809	77	4D	ASC64-Code für M "
2058	080A	77	4D	ASC64-Code für M "
2059	080B	79	4F	ASC64-Code für O "
2060	080C	68	44	ASC64-Code für D "
2061	080D	79	4F	ASC64-Code für O "
2062	080E	82	52	ASC64-Code für R "
2063	080F	69	45	ASC64-Code für E "
2064	0810	34	22	ASC64-Code für E
2065	0811	0	00	Ende der Basiczeile
2066	0812	??	??	Eventuell Beginn einer neuen Zeile

In der Tabelle sieht man sehr gut, wie der komprimierte Basictext im Speicher steht. Die Speicherstellen 2049 und 2050 enthalten als ADL- und ADH-Wert die Adresse der nächsten Basiczeile. Diese zwei Adreßwerte werden oft auch als „LINK-Adressen" bezeichnet. „LINK" bedeutet „verknüpfen, verketten". Die Linkadressen stellen daher die Verbindung der Zeilen eines Programms dar.

Die nächsten beiden Speicherstellen enthalten die Zeilennummer einer Basiczeile. Der vordere Wert ist das niederwertige Byte. Man kann sich die Zeilennummer als Integerzahl ohne Vorzeichen vorstellen. Allerdings läßt der Interpreter nur Zahlen bis 63999 zu.

Der nächste Speicherplatz (im Beispiel 2053) enthält einen Wert, der die codierte Form des Basic-Befehls „PRINT" darstellt. Um nicht unnötig wertvollen Speicherplatz zu verschwenden, besitzt der VC einen Interpretercode, der die Basicbefehle in Bytewerte verschlüsselt. Das ist auch der Grund, warum die meisten Statements abgekürzt (1. Zeichen normal / 2. Zeichen geshiftet) eingegeben werden können. In der folgenden Tabelle ist nun der Interpreter-Code ausgelistet. Das sind praktisch die ungeshifteten Codes ab 128.

Interpreter-Code:

dez	hex	Befehl	dez	hex	Befehl	dez	hex	Befehl	dez	hex	Befehl
128	$80	END	151	$97	POKE	174	$AE		197	$C5	VAL
129	$81	FOR	152	$98	PRINT#	175	$AF	AND	198	$C6	ASC
130	$82	NEXT	153	$99	PRINT	176	$B0	OR	199	$C7	CHR$
131	$83	DATA	154	$9A	CONT	177	$B1	<	200	$C8	LEFT$
132	$84	INPUT#	155	$9B	LIST	178	$B2	=	201	$C9	RIGHT$
133	$85	INPUT	156	$9C	CLR	179	$B3	>	202	$CA	MID$
134	$86	DIM	157	$9D	CMD	180	$B4	SGN	203	$CB	GO
135	$87	READ	158	$9E	SYS	181	$B5	INT			
136	$88	LET	159	$9F	OPEN	182	$B6	ABS			
137	$89	GOTO	160	$A0	CLOSE	183	$B7	USR			
138	$8A	RUN	161	$A1	GET	184	$B8	FRE			
139	$8B	IF	162	$A2	NEW	185	$B9	POS			
140	$8C	RESTORE	163	$A3	TAB(	186	$BA	SQR			
141	$8D	GOSUB	164	$A4	TO	187	$BB	RND			
142	$8E	RETURN	165	$A5	FN	188	$BC	LOG			
143	$8F	REM	166	$A6	SPC(	189	$BD	EXP			
144	$90	STOP	167	$A7	THEN	190	$BE	COS			
145	$91	ON	168	$A8	NOT	191	$BF	SIN			
146	$92	WAIT	169	$A9	STEP	192	$C0	TAN			
147	$93	LOAD	170	$AA	+	193	$C1	ATN			
148	$94	SAVE	171	$AB	–	194	$C2	PEEK			
149	$95	VERIFY	172	$AC	*	195	$C3	LEN			
150	$96	DEF	173	$AD	/	196	$C4	STR$	255	$FF	PI-Zeich.

Für das Beispiel (10 PRINT,,COMMODORE") bedeutet das, daß in der Speicherstelle 2053 der Interpreter-Code 153 (=$99) steht. Nach dem Interpreter-Code folgt in diesem Beispiel ein Stringausdruck. Die Zeichen des Ausdrucks werden durch den ASC64-Code definiert.

Wie unterscheidet eigentlich der Interpreter zwischen normalen Basic-Zeichen und Befehlen? Dazu wollen wir uns einmal ein paar Codes in Binärform anschauen.

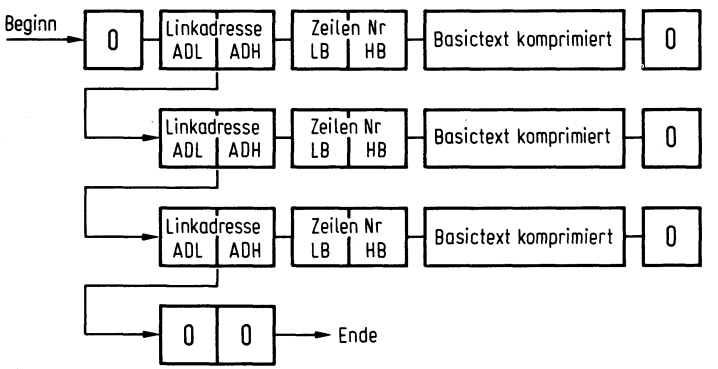
Zeichen	Statement	Code		
		dez	hex	binär
#	–	35	$23	%00100011
4	–	52	$34	%00110100
M	–	77	$4D	%01001101
–	END	128	$80	%10000000
–	GOSUB	141	$8D	%10001101
–	MID$	202	$CA	%11001010

Bei diesem Beispiel sieht man, daß bei den Zeichen das höchstwertigste Bit den Zustand „0" hat. Bei den Befehlswörtern ist dieses Bit auf „1" gesetzt und daher erkennt eben der Interpreter den Unterschied zwischen normalen Zeichen und Basic-Befehlen.

Am Ende jeder Basic-Zeile, das wäre in unserem Beispiel die Speicherstelle 2065, steht seine Null und charakterisiert dadurch das Ende einer Zeile. Erst wenn der Interpreter beim Ablegen der Basic-Zeile in den Programmbereich auf die Null trifft, wird die Linkadresse gesetzt. Das bedeutet: Durch die Null weiß der Interpreter, wo die Zeile aufhört bzw. wo die nächste Zeile beginnt. Das ist auch der Grund, warum in 2048 ($0800) immer Null steht.

Das Ende des gesamten Basicprogramms wird dadurch gekennzeichnet, daß hinter der Null der letzten Zeile zwei weitere Nullen angehängt werden. Das Ende eines Programms wird somit durch drei Nullen gekennzeichnet. Die letzten zwei Nullen wären normalerweise der Linkzeiger auf eine weitere Basic-Zeile. Da dadurch aber auf Null gezeigt wird, stellt es eben das Ende dar.

Die nachfolgende Darstellung zeigt nun nochmals auf, wie ein Basic-Programm insgesamt abgespeichert ist.



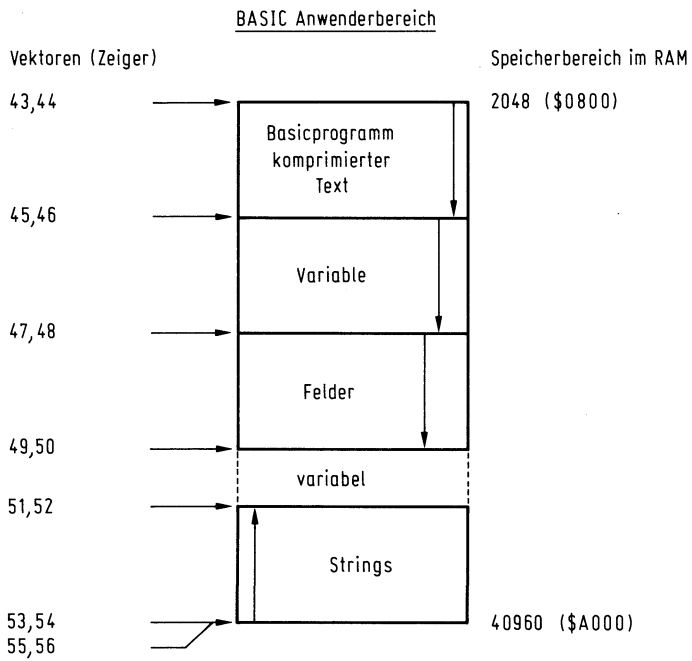
Woher wissen wir nun, wo und wie ein Basic-Programm im Speicher steht? Da die Programme meist aus der Verarbeitung von Variablen bestehen, taucht zunächst die Frage auf, wo die Variablen und deren Werte abgespeichert sind. Im komprimierten Basic-Text können sie nicht sein, da sich ja die Werte laufend verändern können.

Die Verwaltung der Variablen wird durch Vektorisierung erreicht. Das heißt: das Ganze wird durch Zeigerverwaltung geregelt. Dazu benutzt der Interpreter mehrere Hilfsspeicherstellen in der Zeropage. Diese Hilfsspeicherstellen enthalten Adressen (ADL, ADH), die auf den Beginn, aber auch auf das Ende der Variablenbereiche zeigen. Es genügt aber nicht nur festzulegen, wo die Variablen stehen, sondern es muß auch definiert werden, wo Felder (dimensionierte Variable) und Strings stehen. So sind z.B. die Strings einer Stringvariablen nicht direkt beigeordnet, sondern liegen in einem eigens reservierten Speicherbereich.

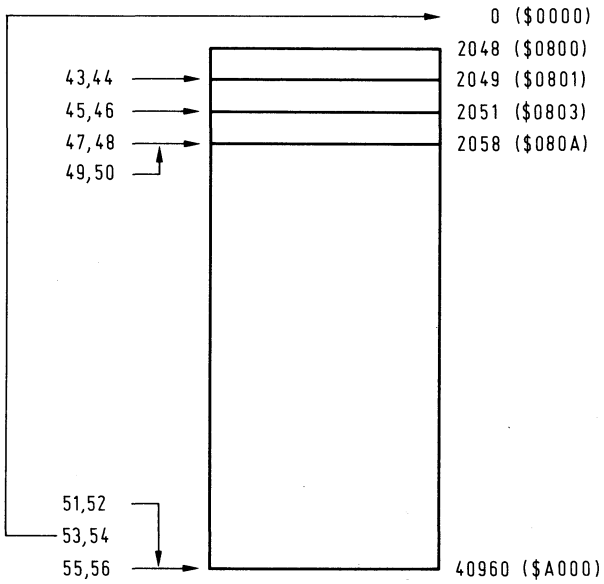
Die ganze Verwaltung der Variablen und deren Werte werden über 7 Zeigerpaare (Vektoren) geregelt. Diese 7 Vektoren liegen in der Zeropage auf folgenden Adressen und haben die gezeigte Funktion:

43–44 $002B– $002C (ADL–ADH)	Zeiger auf Beginn des Basic-Programms
45–46 $002D– $002E (ADL–ADH)	Zeiger auf Beginn der Variablentabelle
47–48 $002F– $0030 (ADL–ADH)	Zeiger auf Beginn der Felder
49–50 $0031– $0032 (ADL–ADH)	Zeiger auf Ende der Felder
51–52 $0033– $0034 (ADL–ADH)	Zeiger auf Beginn der Strings (rückwärts)
53–54 $0035– $0036 (ADL–ADH)	Zeiger auf Ende der Strings
55–56 $0037– $0038 (ADL–ADH)	Zeiger auf höchste RAM-Adresse

Das nachfolgende Schema zeigt, wie ein Programm und deren Veränderliche mit ihren Werten im Speicher liegen. Die Variablen stehen hinter dem komprimierten Programmtext, anschließend folgen die Felder. Dabei wird jeweils die Reihenfolge eingehalten, in der die Variablen oder Felder im Programm auftreten. Die Strings werden umgekehrt abgespeichert. Das heißt, sie werden der Reihenfolge nach, wie sie im Programm vorkommen, vom obersten RAM-Bereich nach unten herab abgelegt. Warum das so ist, wird noch erläutert.



Wenn der VC-64 eingeschaltet wird, werden auch die Zeigerpaare vom Betriebssystem initialisiert. Danach hat das vorhergehende Schema folgendes Aussehen:



Die Zeiger werden beim Initialisieren so gesetzt, daß die Vektoren 43–50 auf den unteren RAM-Bereich deuten, während die Zeiger 51–56 auf die obere RAM-Grenze hinweisen.

Es fällt noch in beiden schematischen Darstellungen auf, daß die Zeiger 53,54 und 55,56 auf denselben Adreßwert eingestellt sind. Normalerweise braucht man den Vektor 55,56 überhaupt nicht, er bringt aber einen entscheidenden Vorteil mit sich. Denn dadurch sind wir in der Lage, größere Maschinenprogramme gleichzeitig mit einem Basicprogramm im Speicher zu halten. Wir können nämlich den Zeiger in 55,56 auf einen kleineren Wert verändern: Die obere RAM-Adresse wird nämlich künstlich herabgesetzt und zwischen dem simulierten obersten RAM-Bereich bis zur tatsächlichen RAM-Grenze können wir Maschinenprogramme unterbringen.

Wollen wir z.B. die obere RAM-Grenze auf den Wert 25000 (= $61A8) heruntersetzen, dann braucht nur der ADL- und ADH-Wert des Zeigers 55,56 entsprechend geändert werden.

25000 = 97* 256 + 168
ADL = 168
ADH = 97

Im Direktmodus: POKE 52,168 : POKE 53,97 : CLR

Durch das „CLR“ wird erreicht, daß die anderen Zeiger vom Interpreter ebenfalls richtig gesetzt werden. Anschließend stehen dem Programmierer die Speicherstellen 25000 – 40960 zur Benutzung von Maschinenprogrammen frei zur Verfügung.

Wie wir noch sehen werden, können wir beim VC-64 die gesamten 64 K für Maschinenprogramme nützen. Wie das geschieht, werden wir noch sehen.

Wir haben uns nun einen Gesamtüberblick verschafft, wo die einzelnen Variablenbereiche abgespeichert sind. Schauen wir uns einmal im Detail das jeweilige Format der verschiedenen Variablentypen an.

2.5.1 Variablenname und Typ

Der Interpreter stellt 3 Variablentypen zur Verfügung:

Gleitkommavariable
Integervariable
Stringvariable

Unabhängig vom Typ belegt jede Variable 7 Bytes im Speicher. Die ersten 2 Bytes enthalten den Namen und den Typ der Variablen. In den restlichen 5 Bytes stehen die Angaben über deren Werte.

Der Variablenname und Typ wird also durch die ersten beiden Bytes definiert. Das erste Zeichen enthält einen Buchstaben im ASCBM-Code. Das zweite Zeichen besteht ebenfalls aus einem ASC64-Zeichen, kann hier aber durch einen Buchstaben oder einer Ziffer dargestellt werden.

Wenn bei einer Variablen der Name nur aus einem Buchstaben besteht, dann enthält das 2. Byte automatisch den Wert 0. Das ist allerdings nicht zu verwechseln mit der Ziffer „0". Diese hat nämlich den ASC64-Code „48". Tritt keine Veränderung des Variablentyps ein, dann handelt es sich bei diesem Namen um eine Gleitkommavariable.

Die Namen von Integervariablen haben das gleiche Format. Nur wird zum ersten und zum zweiten Byte jeweils der Wert 128 ($80) addiert. Das heißt, in beiden Bytes ist jeweils das höchstwertigste Bit auf 1 gesetzt.

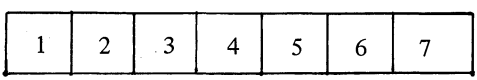
Bei Stringvariablen ist nur das höchstwertigste Bit im 2. Byte auf 1 gesetzt.

Schauen wir uns als Demonstration einmal ein paar Variablennamen im Beispiel an:

Variablenart	Variablennamen	1. und 2. Byte dez		hex		binär	
Gleitkomma	A	65	0	41	00	01000001	00000000
	M8	77	56	4D	38	01001101	00111000
	XX	88	88	58	58	01011000	01011000
Integer	A%	193	128	C1	80	11000001	10000000
	M8%	205	184	CD	B8	11001101	10111000
	XX%	216	216	D8	D8	11011000	11011000
String	A$	65	128	41	80	01000001	10000000
	M8$	77	184	4D	B8	01001101	10111000
	XX$	88	216	58	D8	01011000	11011000

2.5.1.1 Gleitkommavariable

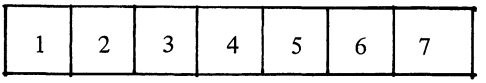
Die gesamte Gleitkommavariable steht folgendermaßen im Speicher:



1–2 = Variablenname
3 = Exponent (+$80)
4–7 = Mantisse

Die ersten beiden Bytes enthalten den Variablennamen entsprechend der Regeln des Typs. Dahinter folgt in 5 Bytes der aktuelle Variablenwert der Fließkommazahl im Speicherformat.

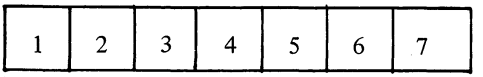
2.5.1.2 Integervariable



1–2 = Variablenname
3 = MSB-Wert der Integerzahl (Höherwertiges Byte)
4 = LSB-Wert der Integerzahl (Niederwertiges Byte)
5–7 = enthalten je 0

Die Integerzahlen enthalten in den ersten 2 Bytes den Variablennamen, entsprechend des Typs. Byte 3 beinhaltet den höherwertigen Teil und Byte 4 den niederwertigen Teil der Integerzahl. In den restlichen drei Bytes (5–7) steht jeweils ein Nullwert. Dadurch werden bei diesem Typ pro Variable 3 Bytes verschenkt.

2.5.1.3 Stringvariable



1–2 = Variablenname
3 = Anzahl der Stringzeichen (= Stringlänge)
4 = ADL Zeiger auf Beginn des Strings
5 = ADH Zeiger auf Beginn des Strings
6–7 = enthalten je 0

Die ersten beiden Bytes enthalten wiederum entsprechend ihres Typs den Variablennamen. In Byte 3 ist die Anzahl der Zeichen des Strings abgespeichert. Dieser Wert wird z.B. von der Basic-Funktion „LEN" verarbeitet. Da dieser Wert ein Bytewert ist, können deshalb nur Strings mit einer Maximallänge von 256 Zeichen auftreten. Byte 4 und 5 stellen einen Zeiger dar, der die Adresse enthält, wo der String im Speicher beginnt. Byte 6 und 7 werden nicht benötigt und daher auf Null gesetzt. Das heißt, es werden 2 Byte wieder verschenkt.

2.5.1.4 Stringabspeicherung

Die verschiedenen Strings werden der Reihenfolge nach ihres Auftretens im Basicprogramm vom oberen RAM-Bereich nach unten abgelegt. Die Zeichenfolge im String selbst ist jedoch in der richtigen Reihenfolge abgespeichert, also in aufsteigender Art.

Nehmen wir einmal an, wir hätten 3 Stringvariable mit „INPUT" auf folgende Weise definiert:

10 INPUT A$ ---> Eingabe ist VC-64
20 INPUT B$ ---> Eingabe ist BAM
30 INPUT C$ ---> Eingabe ist 07324

Dann liegen diese folgendermaßen im Speicher:

Inhalt	Speicher	
	ASC64-Code dezimal	Adresse
0	48	40947
7	55	40948
3	51	40949
2	50	40950
4	52	40951
B	66	40952
A	65	40953
M	77	40954
V	86	40955
C	67	40956
–	45	40957
6	54	40958
4	52	40959

Die Abspeicherung für dieses Beispiel geht dabei so vor sich:
Nach dem ersten „INPUT" wird der Stringanfangszeiger (51,52) auf 40955 (Adresse wo „v" steht) und der Stringendezeiger (53,54) auf 40960 (Adresse wo „M" steht plus 1) gesetzt. Beim nächsten „INPUT" erfolgt die Zeigersetzung auf 40952 (Beginn des Strings „BAM") und auf 40955 (Beginn des vorletzten Strings). Nach der Eingabe von „C$" im nächsten „INPUT", hat der Zeiger 51,52 den Wert 40947 und der Zeiger 53,54 den Wert 40952. Die Berechnung innerhalb der für diese Manipulationen verantwortlichen Routinen wird einfach durch Addition oder Subtraktion der Stringlänge erreicht. Das ganze „Spielchen" setzt sich nun solange fort, bis die Stringeingaben erfolgen oder die Adresse in 51,52 den gleichen Wert erreicht hat, wie die Adresse im Vektor 49,50. Ganz allgemein kann man feststellen, daß alle Basic-Zeiger voneinander verschieden sein sollen. Ist das nicht der Fall, dann hat ein Bereich den anderen erreicht und die Fehlermeldung „OVERFLOW ERROR" erscheint.

2.5.1.5 GARBAGE COLLECT

Bei der Stringabspeicherung haben wir noch eins zu beachten. Wenn nämlich innerhalb des Basicprogrammlaufs ein String durch einen anderen ersetzt wurde, oder die gleiche Stringvariable erhält einen anderen Wert, dann wird der alte String gelöscht und der untere Stringbereich um diesen freien Platz nach oben geschoben. Der neue String der gleichen Variablen wird anschließend am untersten Stringbereich angehängt. Diese Stringmanipulation wird von einer Systemroutine übernommen, die sogenannte „GARBAGE COLLECTION". Diese Routine wird auch dann aufgerufen, wenn sich der Zeiger 51,52 mit anderen Zeigern überschneidet. Dann durchsucht nämlich diese Routine den gesamten Stringbereich, sondert nicht mehr gebrauchte Strings aus und schiebt jeweils den Rest nach oben. So wird wieder Platz für neue Strings geschaffen. Es erfolgt praktisch eine

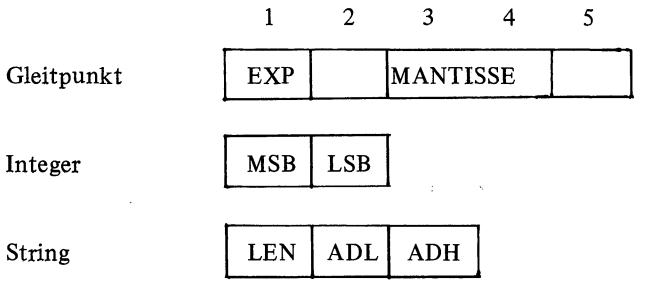
„Ausmistung" des Stringbereichs. Daher kommt auch der Name „GARBAGE COLLECTION", den man fast wörtlich mit „Stringmüllbeseitigung" übersetzen kann. Der Nachteil dieser Routine liegt darin, daß bei extremen Stringmanipulationen im Programmlauf die Stringausmistung einige Zeit dauern kann. Und so mancher Anwender, der von dieser Routine nichts wußte, hat den Rechner abgeschaltet, weil sich im Basicprogramm angeblich nichts mehr tat.

Bei Stringvariablen nennt man die drei Byte's nach dem Namen auch Stringbeschreiber; da hört sich die englische Bezeichnung mit „STRINGDESCRIPTOR" schon besser an. Der Vektor am Stringende zeigt also auf den Beginn des Stringdescriptors. In Basic 4 legt nun die Garbage Collect-Routine bei der Neuanlage von Stringinhalten den Stringdescriptor auch nicht mehr neu an, sondern tauscht nur die Werte aus. Welche Schnelligkeit dadurch erreicht wurde, erkennt man daran, daß Garbage Collect im ungünstigsten Fall bei Basic 4 höchstens eine Sekunde dauert, während in Basic 2 fast eine halbe Stunde vergeht!

2.5.1.6 Bereich der Felder

Bei Integer- oder Stringvariablen werden die letzten 2 oder 3 Bytes umsonst mitgeschleppt und sie verbrauchen dadurch bei vielen Variablen erheblich Speicherplatz. Wenn man im Programm große Datenmengen hat, wird man für deren Verwaltung die Variablen dimensionieren. Der Vorteil aus der Sicht in der Maschinenebene ist der, daß die dimensionierten Variablen optimaler genützt werden, als normale Variable. Allerdings sind sie deshalb auch etwas komplexer zu handhaben. Die Abspeicherung der Felder beginnt im RAM-Bereich nach dem normalen Variablenbereich.

Jede indizierte Variable besteht aus einem Kopf (Header) und den dazu gehörigen Indizies. Dabei ist der Wert von jedem Index so abgespeichert, wie es der Typ im Variablennamen angibt. Hierbei bestehen Fließkommazahlen aus 5 Bytes, Integerzahlen aus 2 Bytes und Stringvariable aus 3 Bytes.



Die Nullen hinter Integer- oder Stringvariablen sind bei dimensionierten Variablen nicht vorhanden.

Der „Header" ist praktisch der Kopfeintrag, der die Tabellen verwaltet. Er besteht bei eindimensionalen Feldern aus 7 Bytes. Für jede weitere Dimension werden an den Kopf zwei weitere Bytes angehängt.

1	2	3	4	5	6	7	8	9	10	11

1 – 2 = Variablenname entsprechend des Typs
3 = LSB Anzahl der belegten Bytes
4 = MSB Anzahl der belegten Bytes
5 = Anzahl der Dimensionen
6 = MSB Anzahl der Elemente (letzte Dimension)
7 = LSB Anzahl der Elemente (letzte Dimension)
8 = MSB Anzahl der Elemente (weitere Dimension)
9 = LSB Anzahl der Elemente (weitere Dimension)
10 = MSB Anzahl der Elemente (erste Dimension)
11 = LSB Anzahl der Elemente (erste Dimension)

Byte 1 und 2 bestehen wie üblich aus dem Variablennamen. Byte 3 und 4 enthalten als Wert die Anzahl der Speicherstellen, die durch die dimensionierte Variable belegt werden. In Byte 5 ist die Anzahl der Dimensionen abgespeichert. So hätte z.B. bei

A%(2,3,3)

Byte 5 den Wert „3". Byte 6 und 7 enthalten als Wert die Anzahl der Elemente pro Index. Die Zählung der Anzahl beginnt hier allerdings bei Null. Haben wir z.B.

DIM A(15)

stehen, dann liegt in Byte 6 der Wert „16" und drückt aus, daß es sich um 16 Elemente handelt, nämlich von 0 bis 15. Byte 8 bis 11 enthalten nun jeweils die Anzahl der Elemente weiterer Dimensionen. Diese werden allerdings nur dann an den Header angehängt, wenn vorher die Routine „DIM" durchlaufen wurde. Wird nicht dimensioniert, aber eine dimensionierte Variable verwendet, dann wird automatisch ein Feld mit 11 Elementen gebildet.

Auf eins sollte man noch bei mehrdimensionalen Variablen achten: Der Wert der Elementenzahl in Byte 6 und 7 bezieht sich immer auf die letzte Dimension, während die Elementenanzahl in den letzten Bytes des Kopfes auf die erste Dimension deuten.

Beispiel:

DIM A(20,13,4)

Wert in Byte 6 und 7 = 5 ---> (4+1)
Wert in Byte 8 und 9 = 14 ---> (13+1)
Wert in Byte 10 und 11 = 21 ---> (20+1)

Schauen wir uns einmal zwei Beispiele an, wie dimensionierte Variable im Speicher stehen. Es wird dabei angenommen, daß der Felderbereich bei der Adresse $2000 beginnt.

1. Beispiel:

```
.......
210 DIM A(1)
220 A(0) = 100 : A(1) = 200
```

Nr.	Adresse	Inhalt		Erklärung
1	$2000	65	$41	1. Zeichen Variablenname „A“
2	$2001	0	$00	2. Zeichen Variablenname (Typ Fließkomma)
3	$2002	17	$11	LSB Belegte Speicherstellen = 17
4	$2003	0	$00	MSB ($2000 bis $2010)
5	$2004	1	$01	Anzahl der Dimensionen = 1
6	$2005	0	$00	MSB Anzahl der Elemente
7	$2006	2	$02	LSB Anzahl der Elemente = 2
8	$2007	135	$87	Exponent der Fließkommavariablen A(0)
9	$2008	72	$48	Mantisse (Byte 2)
10	$2009	0	$00	Mantisse (Byte 3)
11	$2010	0	$00	Mantisse (Byte 4)
12	$2011	0	$00	Mantisse (Byte 5)
13	$2012	136	$88	Exponent der Fließkommavariablen A(1)
14	$2013	72	$48	Mantisse (Byte 2)
15	$2014	0	$00	Mantisse (Byte 3)
16	$2015	0	$00	Mantisse (Byte 4)
17	$2016	0	$00	Mantisse (Byte 5)

2. Beispiel:

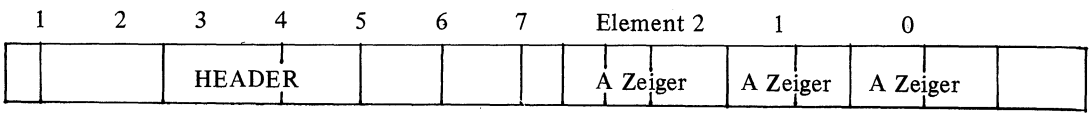
```
. . . . . .
210 DIM A% (1,2)
220 A (0,0) = 100 : A (0,1) = 110 : A (0,2) = 120
230 A (1,0) = 200 : A (1,1) = 210 : A (1,2) = 220
```

Nr.	Adresse	Inhalt		Erklärung
1	$2000	193	$C1	1. Zeichen Variablennamen „A“
2	$2001	128	$80	2. Zeichen Variablennamen „%“
3	$2002	21	$15	LSB Belegte Speicherzellen = 21
4	$2003	0	$00	MSB ($2000 – $2014)
5	$2004	2	$02	Anzahl der Dimensionen = 2
6	$2005	0	$00	MSB Anzahl der Elemente der 2. Dimension
7	$2006	3	$03	LSB Anzahl der Elemente = 3
8	$2007	0	$00	MSB Anzahl der Elemente der 1. Dimension
9	$2008	2	$02	LSB Anzahl der Elemente = 2
10	$2009	100	$64	LSB Integerzahl A% (0,0)
11	$2010	0	$00	MSB Integerzahl A% (0,0)
12	$2011	200	$C8	LSB Integerzahl A% (1,0)
13	$2012	0	$00	MSB Integerzahl A% (1,0)
14	$2013	110	$6E	LSB Integerzahl A% (0,1)
15	$2014	0	$00	MSB Integerzahl A% (0,1)
16	$2015	210	$D2	LSB Integerzahl A% (1,1)
17	$2016	0	$00	MSB Integerzahl A% (1,1)
18	$2017	120	$78	LSB Integerzahl A% (0,2)
19	$2018	0	$00	MSB Integerzahl A% (0,2)
20	$2019	220	$DC	LSB Integerzahl A% (1,2)
21	$2020	0	$00	MSB Integerzahl A% (1,2)

In diesem Beispiel sieht man, wie die Werte der Elemente der Reihenfolge nach abgespeichert sind.

Die Werte der Elemente bei dimensionierten Stringvariablen sind genau umgekehrt abgespeichert.

Beispiel: DIM A$(2)



Das heißt, die Zeiger in den Elementen passen sich der allgemeinen Stringabspeicherung an. Jedes Element besteht aus 3 Bytes auf diesem String: Die Anzahl der Stringzeichen und die Zeigerwerte ADL und ADH.

2.5.1.7 Konstante

Wenn wir im Basicprogramm Konstante haben, die im Programm z.B. auf folgende Weise einer Variablen zugeordnet werden,

```
      :
      :
      :
210 A=3.65
220 A%=100
230 A$= „CBM“
      :
      :
      :
```

dann sind die entsprechenden Zeiger beim aktuellen Zustand nicht auf ihren Bereich gesetzt, sondern zeigen direkt auf die entsprechende Adresse im Programmtext. Das gleiche gilt auch für die Zeiger, die auf die laufenden DATA-Konstanten zeigen.

(DATA-Zeiger = 62,63 (Zeropage)

Nach jedem Zugriff durch „READ“ wird dieser Zeiger um eins erniedrigt.

Wir haben nun gesehen, wie der Basic-Interpreter die Bytes manipuliert. Zum Abschluß dieses Abschnitts werde ich noch eine Übersicht über die einzelnen Basic-Befehle geben. Auf eine detaillierte Erklärung wurde verzichtet, da es einerseits sehr viel Literatur über die Programmiersprache Basic gibt und andererseits im Handbuch der CBM-Rechner darauf eingegangen wird. Eine Abhandlung darüber würde den Rahmen dieses Buches sprengen.

2.5.2 Die Basic-Statements

ABS(Wert)
Aus einem beliebigen Zahlenwert wird der Absolutwert gebildet.

AND
Aus Argumenten werden neue Werte durch eine logische „UND"-Verknüpfung ermittelt.

ASC(String)
Ein Stringzeichen wird in den ASC64-Code umgewandelt und in dezimaler Form ausgegeben.

ATN(Wert)
Ist die arithmetische Umkehrfunktion des Tangens (Arcustangens).

CHR$(Wert)
Als Umkehrung des ASC-Befehls wird der zugehörige Bytewert in ein Stringzeichen umgewandelt.

CLOSE FILE-NUMMER
Eine unter der File-Nummer geöffnete Datei wird wieder geschlossen.

CLR
Es werden sämtliche Variable gelöscht.

CMD
Leitet alle Bildschirmausgaben auf gewähltes Peripheriegerät um.

CONT
Ein laufendes Programm wird nach END oder STOP fortgesetzt.

COS(Wert)
Diese arithmetische Funktion berechnet den Cosinuswert vom Bogenmaß.

DATA
Direkte Konstantenabspeicherung im Basic-Programm.

DEF FN
Eine bestimmte arithmetische Funktion wird definiert und kann später im Programm durch FN mit verschiedenen Werten aufgerufen werden.

DIM
Damit werden Variable dimensioniert, das heißt, die Indexanzahl einer indizierten Variablen wird im voraus festgelegt.

END
Beendet Programmlauf.

FOR Laufvariable = Anfangswert TO Endwert STEP Schrittweite.
Damit wird eine Schleifenfunktion sooft ausgeführt, wie der Anfangswert, der Endwert und die Schrittweite angegeben sind.

FRE(x)
Damit wird festgestellt, wieviel Speicherplatz noch zur Verfügung steht. Der X-wert in der Klammer kann aus irgendeinem Zeichen bestehen, um die Syntaxprüfung zu befrie-
SPC

GET
GET ist ein Eingabebefehl, der sich das nächste Zeichen vom Tastaturpuffer holt.

GET#
Holt das nächste Zeichen vom beliebigen Peripheriegerät.

GOSUB
Springt in ein Basic-Unterprogramm.

GOTO
Es erfolgt ein Sprung zur angegebenen Basic-Zeile.

IF Abfrage THEN Ausführung
Wird eine Bedingung in der Abfrage erfüllt, dann erfolgt die Ausführung. Ansonsten wird in der nächsten Basic-Zeile fortgefahren.

INPUT
Übernimmt Eingabe vom Bildschirm in eine Variable.

INPUT#
Übernimmt die Eingabe von einem beliebigen Peripheriegerät.

INT(Wert)
Wandelt Fließkommazahlen in Integerzahlen um.

LEFT$(
Ein angegebener linker Teil eines Strings wird abgezweigt.

LEN(String)
Berechnet die Länge eines Strings (Anzahl der Zeichen).

LET
Damit wird einer Variablen ein Wert zugewiesen. Ist beim VC-64 nicht unbedingt erforderlich, sondern wird durch das Gleichheitszeichen ersetzt.

LIST
Damit wird das gesamte Basic-Programm am Bildschirm Zeile für Zeile ausgegeben.

LOAD
Übernimmt ein Programm vom angegebenen Peripheriegerät in den Rechnerspeicher.

LOG
Es wird der natürliche Logarithmus ermittelt.

MID$
Von einem Zeichen an wird eine angegebene Anzahl von Zeichen aus einem String abgezweigt.

NEXT
Der NEXT-Befehl bildet die Anweisung für FOR . . TO, die Schleifenvariable zu erhöhen und bildet zugleich den Abschluß einer Schleife.

NEW
Löscht ein Basic-Programm aus dem Rechnerspeicher.

NOT
Durch diesen Befehl wird ein Ausdruck logisch negiert (bitweise).

ON
ON steht entweder vor GOTO oder GOSUB. Dadurch ist dem Programm die Möglichkeit gegeben, abhängig von Bedingungen, und zwar je nach Bedingung, an mehrere Stellen des Programms zu springen.

OPEN
Eine Datei mit Filenummer wird geöffnet.

OR
Es erfolgt zwischen zwei Ausdrücken eine logische Funktion durch eine ODER-Verknüpfung.

PEEK
Der Inhalt einer Speicherstelle wird als Dezimalwert berechnet.

POKE
Ein angegebener Bytewert (dezimal) wird in eine vorgegebene Speicherstelle abgelegt.

POS(x)
Die Spaltenposition des Cursors wird übernommen.

PRINT
Bringt hinter PRINT angegebene Variable oder Strings auf den Bildschirm.

PRINT#
Variable oder Strings werden auf Peripheriegerät ausgegeben.

READ
DATA-Konstante werden in Variable übernommen.

REM
Hinter REM liegende Zeichen werden im Interpreter überlesen und dienen meist nur zur Kommentierung des Basic-Programms.

RESTORE
Setzt den READ-Zeiger wieder auf Beginn.

RETURN
Rückkehrbefehl aus einem Unterprogramm.

RIGHT$
Ein angegebener rechter Teil eines Strings wird abgezweigt.

RND (Wert)
Dieser Befehl bildet Zufallszahlen zwischen 0 und 1.

RUN
Damit wird ein Basic-Programm gestartet.

SAVE
Ein Programm wird auf ein Peripheriegerät abgespeichert.

SGN
Bildet einen Wert, der angibt, welches Vorzeichen eine Zahl hat.

SIN
Arithmetische Funktion berechnet Sinuswert im Bogenmaß.

SPE
Bewirkt hinter PRINT eine Ausgabe einer angegebenen Anzahl von Leerzeichen.

SQR
Berechnet die Quadratwurzel eines angegebenen Wertes.

STOP
Unterbricht einen Programmlauf. Allerdings erscheint eine Meldung, wo das Programm unterbrochen wurde.

STR$
Wandelt eine Zahl in einen String um.

SYS
Aufruf eines Maschinenprogramms als Unterprogramm.

TAB
Wird hinter PRINT angewandt und tabuliert die entsprechend angegebenen Cursor-nach-rechts-Zeichen vom linken Bildschirmrand.

TAN
Arithmetische Funktion berechnet den Tangenswert im Bogenmaß.

USR
Aufruf eines Maschinenunterprogramms mit Übernahme einer Variablen.

VAL
Wandelt einen String in eine Zahl um.

VERIFY
Vergleicht ein Programm im Rechner mit demselben aber bereits abgespeicherten Programm auf Disk oder Rekorder.

WAIT
Hält solange ein Programm an, bis die logischen Verknüpfungen der hinter WAIT angegebenen Parameter den Wert ungleich Null erreicht haben.

Wir haben nun fast alle Basic-Anweisungen kennengelernt. Es fehlen jetzt nur noch ein paar spezielle Funktionen des Interpreters. Dazu zählen die Operatoren, die ja eigentlich auch Anweisungen sind, und die fest verwendeten Variablen.

2.5.3 Die Operatoren

Die Operatoren kann man einteilen in mathematische Operatoren, Vergleichsoperatoren und logische Operatoren. Die logischen Operatoren AND, NOT und OR, wurden bereits aufgeführt. Bei den mathematischen Operatoren haben wir aber noch folgende Größen:

↑ = potenzieren
* = multiplizieren
/ = dividieren
+ = addieren (oder Stringverknüpfung)
– = subtrahieren

Folgende Vergleichsoperatoren stehen zur Verfügung:

„=“ = Gleichheitszeichen
<> o. >< = Ungleich
> = Größer
< = Kleiner
>= = Größer gleich
<= = Kleiner gleich

Beim Gleichheitszeichen ist noch zu erwähnen, daß es praktisch als Zuweisung behandelt wird. So wäre z.B. die Gleichung X = X + 1 in der Mathematik falsch. Der Interpreter im Computer dagegen bringt keine Fehlermeldung, sondern führt diese Gleichung aus. Das liegt daran, daß der Rechner die Variable X links vom Gleichheitszeichen als Variable mit einem neuen Wert betrachtet, während er rechts vom Gleichheitszeichen den Wert der Variablen X als alten Wert interpretiert. So könnte man die Gleichung auch folgendermaßen hinschreiben:

Xneu = Xalt + 1

Der Interpreter muß auch in der Rangfolge der verschiedenen Operatoren unterscheiden können, was vor wem Vorrang hat. Dabei gilt beim VC-64 folgende Rangfolge, angefangen mit der höchsten Priorität:

↑
* /
+ –
=
< >
>
<
>=
<=
NOT
AND
OR

Die Reihenfolge kann durch Setzen von Klammern verändert werden.

2.5.4 Die festverwendeten Variablen

Der Interpreter verwendet auch ein paar Variable, die fest belegt sind und denen der Anwender keinen x-beliebigen Wert zuweisen kann. Dazu zählen folgende Variable:

ST = Status
TI = Zeitwert in 1/60 Sekunden
TI$ = Tagesuhrzeit als String

Die reservierte Variable ST ist zuständig bei Kommunikation mit bestimmten Peripheriegeräten. Doch darauf werden wir noch an geeigneter Stelle zurückkommen.

Die reservierten Variablen TI und TI$

Damit wird eine Zeitfunktion aufgerufen, mit der man den VC-64 als Uhr betreiben kann. Die Genauigkeitsjustierung besteht hier aus einer Taktfolge von 1/60 Sekunde. Mit der Variablen TI$ läßt sich die Uhrzeit eines Tages einstellen. Der String darf nicht länger als 6 Zeichen sein und hat folgendes Format:

TI$ = „hhmmss“

(h = Stunden / m = Minuten / s = Sekunden)

Die Zahlenvariable TI enthält den Wert der Stringvariablen TI$. TI kann kein Wert direkt zugewiesen werden. Die Einstellung erfolgt über TI$. Wenn als TI$ der Wert „0“ zugewiesen wird, dann hat TI ebenfalls den Wert „0“. Die Variable TI enthält normalerweise den Zeitwert seit dem Einschalten des Gerätes.

Wir sind nun damit am Ende des Abschnitts über den Basic-Interpreter angelangt. Wir werden uns nun mehr und mehr in die Innereien des CBM wagen. Als nächstes werden wir uns mit der Ein- und Ausgabe beschäftigen.

2.6 INTERFACETECHNIK BEIM VC-64

In diesem Abschnitt befassen wir uns nicht nur mit der Schnittstellenbehandlung, was Interfacetechnik bedeutet, sondern auch mit der Programmierung der Peripheriebausteine, die ja erst die universellen Möglichkeiten des VC-64 ausmachen. Der VC-64 hat nämlich eine Reihe von neuentwickelten Chips aus der 65xx-Baureihe, die frühere umständliche Programmierungen, z.B. im Bereich der Graphik, wesentlich erleichtern. Desweiteren ist der Rechner sehr flexibel, was die Speicherverwaltung von RAM und ROM angeht. Daher werden wir in diesem Abschnitt auch auf die Speicherkonfiguration und deren Manipulation eingehen. So besteht zum Beispiel die Möglichkeit, einen eigenen Zeichensatz zu kreieren und den hardwaremäßigen Zeichengenerator auszublenden.

2.6.1 Der Interfacebaustein CIA (MCS 6526)

Meist stammen Peripheriegeräte wie Tastatur oder Bildschirm von fremden Herstellern. Diese kann man nun nicht direkt mit der CPU verbinden, da es sonst Anpassungsschwierigkeiten geben würde. Aus diesem Grund müssen Bausteine dazwischengeschaltet werden, die eben diese Anpassung vornehmen und als Schnittstelle zwischen CPU und Peripherie dienen. Wir wollen uns nun einmal mit dem CIA (Complex Interface Adapter), der im VC-64 zweimal vorhanden ist, auseinandersetzen.

Die Interfacebausteine haben Leitungen, die mit dem CHIP-SELECT-Signal (Bausteinauswahl) angesprochen werden. Das heißt, über diese Leitung wird eine Basisadresse im Gesamtspeicher des Rechners festgelegt. Das bezeichnet man oft auch mit „I/O MEMORY MAPPED". Dadurch können die Interfacechips durch normale Speicherbefehle von der CPU aus gesteuert werden. Es wird zum Beispiel ein entsprechendes Bitmuster durch

LDA #Wert

geladen und anschließend mit

STA CIA-Adresse

in den Baustein abgesetzt. Der CIA hat intern ebenso wie die CPU mehrere Register und die dazugehörigen Verbindungsleitungen (interner Bus). 16 Register können nun von der CPU aus durch Speicherbefehle manipuliert werden. Die Auswahl der einzelnen Register innerhalb des CIA erfolgt durch „REGISTER-SELECT-SIGNALE" RS0 – RS3. Diese 4 Leitungen können 4 Bits darstellen. Da somit 16 verschiedene Bitmuster erzeugt werden, ist auch klar, daß 16 Register angesprochen werden können. Die eigentliche Adresse eines Registers errechnet sich dadurch, daß zur Basisadresse der Offsetwert des Bitmusters hinzugezählt wird.

Beispiel:

Basisadresse des CIA ist 56320 im Gesamtspeicher
Offsetwert Register %1111 ist 15
Adresse des Registers 15 im Gesamtspeicher ist 56320+15 = 56335

Der Zugriff auf den CIA erfolgt also über 16 Adressen, die folgende Bedeutung haben:

BASISADRESSE + binär	hex	dez	Erklärung der Zugriffsart
0000	00	0	Ausgaberegister für Tor A (PRA = Port Register A)
0001	01	1	Ausgaberegister für Tor B (PRB = Port Register B)
0010	02	2	Datenrichtungsregister A (DDRA = Data Direction Register A)
0011	03	3	Datenrichtungsregister B (DDRB = Data Direction Register B)
0100	04	4	Timer A low (niederwertiger Teil des Zählers A)
0101	05	5	Timer A high (höherwertiger Teil des Zählers A)
0110	06	6	Timer B low (niederwertiger Teil des Zählers B)
0111	07	7	Timer B high (höherwertiger Teil des Zählers B)
1000	08	8	TOD 10ths (Time On Day = Tageszeit in 1/10 Sekunden)
1001	09	9	TOD sec (Time On Day = Tageszeit in Sekunden)
1010	0A	10	TOD min (Time On Day = Tageszeit in Minuten)
1011	0B	11	TOD hr (Time On Day = Tageszeit in Stunden)
1100	0C	12	Schieberegister (Serial Data Register SDR)
1101	0D	13	Interrupt Control Register ICR
1110	0E	14	Control Register A (CRA)
1111	0F	15	Control Register B (CRB)

Vermutlich können Sie damit noch nicht viel anfangen. Daher müssen wir uns mit den einzelnen Registern etwas näher beschäftigen.

2.6.1.1 Die Ausgaberegister PRA und PRB

Diese Register sind nun tatsächlich jeweils über 8 Leitungen mit der Peripherie verbunden. Die entsprechenden Datensignale können sowohl von der Peripherie empfangen, als auch zur Peripherie gesendet werden. Die Assemblerbearbeitung für die Datenübermittlung ist denkbar einfach.

```
LDA  PRA  -> empfangen
STA  PRA  -> senden
```

Man kann die einzelnen Bits dieser Register auf Eingang oder Ausgang schalten, so daß zur gleichen Zeit Signale empfangen werden, während Signale zur Peripherie gehen. In welche Richtung die Daten laufen, hängt vom Datenrichtungsregister ab.

2.6.1.2 Das Datenrichtungsregister DDRA oder DDRB

Das Datenrichtungsregister legt fest, in welche Richtung die Übertragung stattfindet. Wenn alle 8 Bits den Zustand 1 haben, dann gehen die Daten von der CPU zur Peripherie. Enthalten alle Bits den Zustand 0, dann findet ein Datentransfer zur CPU statt. Man kann

nun die einzelnen Bits im Datenrichtungsregister einzeln auf Ein- oder Ausgang stellen, je nach Einschreiben eines entsprechenden Wertes. So legt z.B.

```
LDA   #$AA        ($AA = %10101010)
STA   DDRA
```

fest, daß die Datensignale Bit 0,2,4 und 6 in die CPU fließen, während Bit 1,3,5 und 7 über PRA an die Peripherie gelangen. Um aufzuzeigen, wie man mit Datenrichtungsregister und Ausgaberegister umgeht, dient folgendes Beispiel:

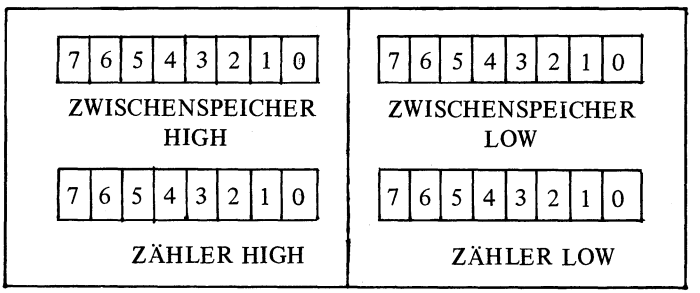
Deklaration:

Der Buchstabe „A" im ASC64-Code soll über PRB zur Peripherie gesendet werden.

```
LDA   #$FF     ; $FF=%11111111 Alle Bits auf Ausgang
STA   DDRB     ; im Datenrichtungsregister setzen
LDA   #$41     ; $41 = „A" ASC64-Code holen
STA   PRB      ; über Port B ausgeben
```

2.6.1.3 Der TIMER A oder B

Man kann sich dieses Register so vorstellen, als bestehe es aus vier Speicherstellen, die aus zwei Adressen mit je einem Low- und einem High-Byte bestehen.



Wenn man sich die Registerauswahl ansieht, dann sind für den Timer A 2 Adressen zuständig, und zwar für den niederwertigen und den höherwertigen Teil. Je nachdem, ob mit LDA ein Lesezugriff oder mit STA ein Schreibzugriff vollzogen wird, hat jeder Teil eine andere Funktion.

Egal, um welchen Teil (Low oder High) es sich beim Timer handelt, wird auf alle Fälle bei einem STA-Zugriff in den Zwischenspeicher eingeschrieben. Bei einem Lesezugriff wird der Wert aus dem Zähler geholt, der gerade den aktuellen Inhalt übergibt. Es gibt beim Timer mehrere Arten der Funktion. So gibt es z.B. den monostabilen Betrieb, in dem der Zähler nur auf Null gezählt wird und dann einen Interrupt auslöst, oder der Zähler wird zyklisch immer wieder auf Null heruntergezählt und schiebt dabei ein Bit nach dem anderen über Bit 6 oder 7 des Ausgaberegisters aus. Das nennt man dann Impulsbetrieb durch Zählung. Welchen Betriebszustand der Timer einnimmt, hängt vom jeweiligen Steuerregister (CRA oder CRB) ab. Dort ist auch ein Bit für den Start oder Stop des Timers verantwortlich.

2.6.1.4 Die Echtzeituhr TOD (Time On Day)

Der CIA enthält in 4 Registern (8–11) eine Echtzeituhr, die praktisch die Tageszeit bis zu 1/10 Sekunden angibt. Betrieben wird sie durch Systeminterrupt, der 60 mal in der Sekunde die Tastatur abfragt. Die 4 Register haben folgende Bedeutung:

TOD 10THS: Dieses Register ist zuständig für das Betreiben der Uhr in 1/10 Sekunden. Die oberen 4 Bits (4–7) sollten immer auf 0 gesetzt sein. Die unteren 4 Bits (0–3) stellen die 10tel Sekunden in binär codierter Dezimal-Form (BCD-Form) zur Verfügung.

TOD SEC: Dieses Register ist zuständig für das Betreiben der Uhr in Sekunden. Bit 0–3 stellen die Einersekunden im BCD-Code zur Verfügung, während die Bits 4–6 die Zehnersekunden im BCD-Format anzeigen. Bit 7 sollte immer auf 0 gesetzt sein.

TOD MIN: Dieses Register ist zuständig für das Betreiben der Uhr in Minuten. Wie beim Sekundenregister (TOD SEC) sind die Bits 0–3 für Einerminuten und Bit 4–6 für die Zehnerminuten im BCD-Format zuständig. Bit 7 sollte auf 0 stehen.

TOD HOUR: Dieses Register ist zuständig für das Betreiben der Uhr in Stunden. Hier sind im BCD-Code die Bits 0–3 für die Einerstunden verantwortlich. Bit 4 legt die 11. oder 12. Stunde fest. Bit 5–6 sollten auf 0 stehen. Bit 7 zeigt an, ob es sich um die Uhrzeit 0.00 bis 12.00 (Bit 7 = 0) oder um 12.00 bis 24.00 Uhr (Bit 7 = 1) handelt.

Es ist nun wieder der Unterschied von einem Lesevorgang gegenüber dem Schreibzugriff zu beachten. Werden die TOD-Register gelesen, so erfolgt ein Übertrag der gesamten Uhrzeit in einen Zwischenspeicher, sobald ein Lesezugriff auf TOD HOUR (Stundenregister) passiert. Das ist deshalb notwendig, weil ja die Uhrzeit ständig hochgezählt wird und vermutlich die 1/10-Sekunden während des Lesens bereits eine Änderung erfahren könnten. Um danach den Zwischenspeicher wieder benutzen zu können, müssen unbedingt auch die 1/10 Sekunden gelesen werden. Um eine Zeit in die TOD-Register einzuschreiben, muß das Hochzählen der Register angehalten werden. Das geschieht dadurch, daß ein Wert in das Stundenregister (TOD HOUR) eingeschrieben wird. Die Uhr hält nun so lange an, bis ein 1/10-Sekunden-Wert in das Register 8 (TOD 10THS) eingegeben wird.

Ob sich die Uhr im Alarm- oder Uhrzeit-Modus befindet, hängt vom Zustand des Control Register B (CRB) im CIA ab. Ist Bit 7 im CRB auf 0 gesetzt, dann handelt es sich um den normalen Uhrzeitbetrieb. Eine 1 in Bit 7 des CRB legt den Alarm-Modus fest. Im Alarmzustand wird das sogenannte Alarmregister eingeschaltet, auf das nur im Schreibverfahren zugegriffen werden kann und das dieselben Register belegt wie TOD.

2.6.1.5 Das Schieberegister SDR (Serial Data Register)

Dieses Register hat eine Leitung, über das ein Datenbyte bitweise ein- oder ausgeschoben werden kann. Man bezeichnet dieses Pin mit SP, was die Abkürzung für serieller Port ist. Gesteuert wird dieses Register vom Bit 3 des Interrupt Control Registers. Ist dieses Bit gesetzt, dann erfolgt der Betrieb der seriellen Datenübertragung. Ansonsten ist das SDR ausgeschaltet. In welche Richtung die Bits geschoben werden, hängt vom Bit 6 des Control Registers A ab. Wenn CRA(6) = 0 ist, dann wird über SP aus SDR geschoben. Bei CRA(6) = 1 wird eingeschoben.

2.6.1.6 Das Interrupt Control Register ICR

Dieses Register wird für die verschiedensten Interruptsteuerungen der CIA-Register benötigt. Dabei sind Schreib- und Lesezugriff wiederum zu unterscheiden. Es muß auch beachtet werden, daß jedes einzelne Bit dieses Registers eine andere Bedeutung hat.

ICR READ/	Bit 0 = 1 →	Zeigt Unterlauf des Timers A an (Zähler < 0)
	Bit 1 = 1 →	Zeigt Unterlauf des Timers B an (Zähler < 0)
	Bit 2 = 1 →	Zeigt an, daß die Zeit in TOD mit gewählter Zeit übereinstimmt
	Bit 3 = 1 →	Schiebungsbetrieb im SDR ist eingeschaltet
	Bit 4 = 1 →	Zeigt an, daß ein Signal am Flageingang aufgetreten ist
	Bit 5	Sollte immer 0 sein
	Bit 6	Sollte immer 0 sein
	Bit 7	IRQ-Ausgang ist 0, wenn ein Ereignis eintritt (z.B. Unterlauf eines Timers)
ICR WRITE/	Bit 0 = 6	Gleiche Bedeutung wie beim Lesezugriff
	Bit 7 = 0 →	Jedes Bit, das im ICR auf 0 gesetzt ist, löscht auch das entsprechende Bit, für das es zuständig ist.
	Bit 7 = 1 →	Jedes Bit, das im ICR auf 1 gesetzt ist, setzt auch das entsprechende Bit, für das es zuständig ist.

2.6.1.7 Die Control Register A und B (CRA und CRB)

Die Control Register sind für die Steuerung von Timer, SDR und TOD verantwortlich, wobei auch noch teilweise eine Leitung der Ausgaberegister mit beeinflußt wird. Das sind die Leitungen PB6 für Timer A und PB7 für Timer B. Die einzelnen Bits haben die gleiche Bedeutung bei Schreib- und Lesezugriff.

CRA	Bit 0 = 0	Timer A wird angehalten
	Bit 0 = 1	Timer A wird gestartet
	Bit 1 = 0	kein Unterlauf von Timer A
	Bit 1 = 1	Unterlaufsignal vom Timer A wird an PB6 übergeben
	Bit 2 = 0	High-Impuls an PB6 durch Unterlauf des Timers A
	Bit 2 = 1	Unterlaufsignal invertiert PB6-Signal
	Bit 3 = 0	ständiger Timerdurchlauf von jeweiligem Ausgangswert
	Bit 3 = 1	Einmaliger Timerdurchlauf
	Bit 4 = 0	Timer benötigt keinen neuen Startwert
	Bit 4 = 1	Unbedingtes Setzen eines neuen Startwertes
	Bit 5 = 0	Timer zählt hoch bei jedem Systemtakt
	Bit 5 = 1	Timer zählt hoch bei Impuls über CNT-Leitung
	Bit 6 = 0	serieller Port am Schieberegister ist Ausgang
	Bit 6 = 1	serieller Port am Schieberegister ist Eingang
	Bit 7 = 0	Triggerung von TOD bei 60 Hz
	Bit 7 = 1	Triggerung von TOD bei 50 Hz

CRB Bit 0 = 0 Timer B wird angehalten
Bit 0 = 1 Timer B wird gestartet
Bit 1 = 0 kein Unterlauf von Timer B
Bit 1 = 1 Unterlaufsignal vom Timer B wird an PB7 übergeben
Bit 2 = 0 High-Impuls an PB7 durch Unterlauf des Timers B
Bit 2 = 1 Unterlaufsignal invertiert PB7-Signal
Bit 3 = 0 ständiger Timerdurchlauf von jeweiligem Ausgangswert
Bit 3 = 1 Einmaliger Timerdurchlauf
Bit 4 = 0 Timer benötigt keinen neuen Startwert
Bit 4 = 1 Unbedingtes Setzen eines neuen Startwertes
Bit 5,6 -> 00 = Timer B Zählung durch Systemtakt
01 = Timer B Zählung durch Unterläufe von Timer A
10 = Timer B Zählung durch CNT-Impuls
11 = Timer B Zählung durch Unterläufe von Timer A, wenn CNT-Leitung auf 1 steht
Bit 7 = 0 TOD in Uhrzeitmodus versetzen
Bit 7 = 1 TOD auf Alarm-Modus stellen

2.6.1.8 Zusammenfassendes über den CIA

Wenn Sie die Beschreibung der einzelnen Register gelesen haben, können Sie bestimmt ermessen, welche Leistungsfähigkeit dieser Baustein hat. Hobby-Elektroniker könnten mit diesem Baustein allerhand anfangen, sofern sie den CIA-Chip zusätzlich kaufen. VC-64-Anwender können nur bedingt die CIA's verwenden, da das Betriebssystem auch darauf zurückgreift. Doch darauf kommen wir noch zurück.

2.6.2 Der Video-Controller VIC (MCS 6569)

Der VIC ist ein neuer Chip aus der 65xx-Baureihe, der einiges kann, sofern man weiß, was seine Register können. Dieser Baustein hat gleich 47 Register. Erschrecken Sie aber bitte nicht, wenn Sie vorher die Beschreibung des CIA gelesen haben. Einige der Register haben nämlich gleichartige Funktionen für verschiedene bewegbare Bildmuster, den sogenannten Sprites. Durch den VIC kann beim VC-64 auch hochauflösende Graphik (320 * 200 Punkte) erzeugt werden und es besteht die Möglichkeit, einen eigenen Zeichensatz zu entwerfen. Auf diese Möglichkeiten werden wir aber noch genauer eingehen. Neben der Aufrechterhaltung des Fernsehbildes im PAL-Format wird im VIC auch der Systemtakt erzeugt. Daher kann VIC geschickt die Systemtaktlücken des Prozessors ausnutzen. Das ist dann der Fall, wenn er z.B. auf den Farb-RAM zugreift.

Genau wie beim CIA werden die einzelnen Register per Adresse angesprochen. Das heißt, mit LDA können die Register gelesen und mit STA beschrieben werden. Man muß nur wissen, welche Basisadresse der VIC besitzt. Im folgenden wollen wir uns nun mit den Registern im einzelnen beschäftigen.

2.6.2.1 Die XY-Register 0–15 (Spriteregister)

Der Bildschirm beim VC-64 wird in ein X und Y Koordinatensystem aufgeteilt, wobei die X-Koordinate den Abstand vom linken und die Y-Koordinate den Abstand vom oberen Bildschirmrand bedeutet. Werden Sprites erzeugt, so geben die XY-Register Auskunft darüber, wo sich das Sprite befindet. Dabei ist der entsprechende Koordinatenpunkt genau der linke obere Eckpunkt der Spritematrix.

Es stehen für die Spriteposition 16 Register zur Verfügung für acht Sprites, und zwar Register 0,2,4,6,8,10,12,14 für die X-Koordinate und Register 1,3,5,7.9,11,13,15 für die Y-Koordinate.

Der Bildschirm ohne Rahmen besteht aus einem Raster von 320 Punkten in X-Richtung und von 200 Punkten in Y-Richtung. Da jedes Register nur 8 Bits aufnehmen kann, können natürlich nur Werte von 0 bis 255 eingeschrieben werden. Wie bringt man dann ein Sprite in den X-Bereich zwischen 256 und 320? Das wurde so geregelt, daß für X-Werte des Sprites über 255 das Register 16 zuständig ist. Dort braucht man nur ein Bit zu setzen und schon befindet sich das Sprite im X-Bereich ab 256. Wenn man es genau betrachtet, geht der X-Bereich eines Sprites von 0 bis 255 und dann weiter von 0 bis 87 bzw. 256 bis 343. In die Y-Register können wir auch Werte zwischen 0 und 255 abspeichern. Der Y-Bereich hat aber nur 200 Punkte zur Verfügung. Das Ganze hat nun den Vorteil, daß man die Sprites in jeder Richtung im Rahmen verschwinden lassen kann. Das sieht so aus, als würde das Sprite hinter dem Rahmen eintauchen.

2.6.2.2 Das X-High-Register (Spriteregister 16)

Dieses Register ist verantwortlich für die Spriteposition im X-Bereich über 255. Jedes Bit (0–7) dieses Registers ist zuständig für eines der 8 Sprites (0–7). Wird ein Bit auf 1 gesetzt, dann kann sich das Sprite in X-Richtung zwischen der Position 256 bis 320 bewegen. Sollte zum Beispiel Sprite 1,3,4,6 und 7 im Bereich über 255 manipuliert werden, während die Sprites 0,2 und 5 zwischen 0 und 255 positioniert sind, dann müßte folgender Wert in Register 16 geschrieben werden:

Register 16 für Sprite	7	6	5	4	3	2	1	0
Inhalt	1	1	0	1	1	0	1	0

%11011010 = $DA = 218

Ist nun ein bestimmtes Bit gesetzt, so beginnt der Wert im entsprechenden X-Register wieder mit 0. Das heißt, die X-Koordinate bewegt sich im Bereich zwischen 0 und 87. Beim Wert 87 ist gerade noch der linke Rand der Spritematrix sichtbar.

2.6.2.3 Control Register A (Register 17)

Wie auch bei den Steuerregistern im CIA, haben die einzelnen Bits verschiedene Funktionen. Mit den Bits 0 bis 2 könnte man z.B. eine kleine Bildschirmverschiebung in Y-Richtung erreichen. Allerdings werden wir das kaum benötigen. Bit 3 legt die Anzahl der Bildschirmzeilen fest (Bit 1 –> 0 = 24 Zeilen / –> 1 = 25 Zeilen). Mit Bit 4 kann man den Zugriff des VIC auf den Bildschirm ein- und ausschalten. Wird Bit 5 auf 1 gesetzt, dann befindet sich der Rechner im hochauflösenden Graphik-Modus. Das heißt,

daß ein Rasterpunkt der 320*200-Matrix einem Farb-RAM-Bit entspricht. Schaltet man das Bit 6 auf 1, dann wird der sogenannte EXTENDED COLOUR Modus bedient, indem hier noch die Hintergrundfarbe eines einzelnen Zeichens gewählt werden kann. Bit 7 dient als Flag für den Übertrag von Register 18.

2.6.2.4 Interruptregister Zeilensignal (Register 18)

Der VIC erzeugt das Fernsehsignal und rast dabei zeilenweise über sämtliche Bildschirmpunkte. Das ganze geschieht so schnell, daß das gesamte Fernsehbild 20 mal in der Sekunde aufgebaut wird. Im Register 18 steht nun immer die aktuelle Zeilennummer einer Fernsehzeile. Aus hardwaremäßigen Geschwindigkeitsgründen läßt sich nun nicht eine bestimmte Zeile direkt per LDA zu einem bestimmten Zeitpunkt ermitteln. Mit dem Register kann man aber einen IRQ auslösen, indem man in das Register einen entsprechenden Zeilenwert einschreibt.

Wie sie dem normalen Bild des VC-64 entnehmen können, muß auch das Signal für den Rahmen erzeugt werden. Insgesamt können bis zu 280 Fernsehzeilen aufgebaut werden. Auf dem Bildschirm erscheinen allerdings nur 250. Dadurch läßt sich nämlich der Bildschirm nach oben oder unten verschieben. Allerdings wird, da sich Registerwerte über 256 ergeben können, ein Übertrag nach Bit 7 im Register 17 fällig.

2.6.2.5 Die Lightpen XY-Register (Register 19 u. 20)

Mit Hilfe eines Lichtgriffels kann man von außen eben einen Lichtpunkt abfragen und mit Hilfe der Register 19 und 20 die entsprechenden Koordinaten ermitteln. Register 19 dient dabei als X-Register. Allerdings werden in X-Richtung jeweils immer nur 2 Punkte gleichzeitig abgefragt und somit pro Zeile 160 Punkte bewertet. Das bedeutet, wenn wir mit Hilfe des Lichtgriffels einen Punkt setzen wollen, so muß softwaremäßig eine Umrechnung auf den Grafikmodus erfolgen. Das Register 20 dient in der gleichen Weise aber für die Y-Richtung. Hierbei wird keine Umrechnung benötigt, da ja nur mit 200 Zeilen gearbeitet wird.

2.6.2.6 Das Sprite-Start Register (Register 21)

In diesem Register ist jedem Bit ein Sprite zugeordnet. Bit 0 schaltet Sprite 0, Bit 1 schaltet Sprite 1 und so weiter bis 7. Wird ein Bit gesetzt, so wird ein Sprite ein-, ansonsten ausgeschaltet.

2.6.2.7 Control Register B (Register 22)

Hier muß man wiederum die einzelnen Bits für sich betrachten. Die Bits 0 – 2 können für eine kleine Verschiebung in X-Richtung hergenommen werden. Mit Bit 3 legt man die Anzahl der Spalten fest, wobei Bit gesetzt 40 und Bit gelöscht 38 Spalten bedeutet. Mit Bit 4 schaltet man, wenn es gesetzt wird, in den sogenannten MULTI COLOUR MODUS um, indem die Zeichen in einer 4*8 Matrix dargestellt werden. Ein Matrixpunkt wird dann aus Farbpunkten gebildet, so daß umfangreiche Farbschattierungen zustande kommen. Bit 5 bis 7 des Control Registers B werden überhaupt nicht ausgenutzt; sie liegen praktisch brach.

2.6.2.8 Register 23 für die Y-Erweiterung eines Sprites

Die Bits 0 bis 7 sind wiederum für die Sprites 0 bis 7 zuständig. Wird ein Bit gesetzt, so dehnt sich das Sprite in die Y-Richtung, also nach unten um das doppelte aus.

2.6.2.9 Adreßregister für Bildschirmspeicher und Zeichengenerator (Register 24)

Bit 0 wird nicht genützt. Bis 1 bis 3 sind für die Adressierung des Zeichengenerators zuständig. Sie werden dabei als Adreßbits 11 bis 13 verwendet. Die Bits 4 bis 7 legen als Adreßbits 10 bis 13 die Adresse des Video-RAM fest. Dabei wäre noch zu bemerken, daß der VIC nur die ersten 14 Bits zur Basisadressierung der verschiedenen Speicherbereiche verwendet und daher normalerweise nur die ersten 16 K des Speichers anspricht. Beim VC-64 wurde das umgangen, indem die fehlenden Bits (14 und 15) vom CIA dazugeführt werden. Darauf werden wir später aber noch genauer eingehen.

2.6.2.10 IRQ-Register IRR (Interrupt Request / Reg.25)

Wird dieses Register gelesen, so zeigt ein gesetztes Bit an, weshalb ein Interrupt aufgetreten ist. Dabei gibt es folgende Möglichkeiten:

Bit 0 = 1 → Interrupt tritt auf, wenn man in Register 18 einen entsprechenden Zeilenwert einschreibt.
Bit 1 = 1 → Trifft ein Sprite auf ein Hintergrundzeichen, so wird dieses Bit auf 1 gesetzt (siehe auch Register 31).
Bit 2 = 1 → Treffen zwei Sprites aufeinander, so wird dieses Bit auf 1 gesetzt (siehe auch Register 30).
Bit 3 = 1 → Dieses Bit geht auf 1, sofern ein Signal bei der Lightpen-Funktion auftritt.
Bit 4 – 6 → Diese Bits werden nicht genützt und liegen daher brach.
Bit 7 = 1 → Dieses Bit wird gesetzt, sobald eins der Bits 0 bis 3 auf 1 geht.

Zu beachten ist noch, daß Register 25 nach einer Bedienung sofort gelöscht werden sollte, da es eventuell zu einem ständigen Auslösen eines Interrupts kommen kann.

2.6.2.11 Das Maskierbare IRQ-Register (Register 26)

Mit diesem Register kann der Anwender bestimmen, ob ein Interrupt ausgelöst werden soll oder nicht. Die Belegung der Bits entspricht dabei vollständig der des Register 25. Ist in beiden Registern das gleiche Bit gesetzt, so tritt am IRQ-Ausgang des VIC ein Interruptsignal auf. Wird das Register 26 vollkommen gelöscht, so wird ein eventuell auftretender Interrupt gesperrt, auch wenn sämtliche Bits im Register 25 auf 1 stehen.

2.6.2.12 Prioritätsregister für Sprites (Register 27)

Die einzelnen Bits der 8 Bits dieses Registers sind wiederum den einzelnen Sprites zugeordnet. Ist ein Bit auf 1 gesetzt, so hat der Hintergrund Vorrang vor dem Sprite. Bei 0 hat, genau umgekehrt, das Sprite Vorrang.

2.6.2.13 Mehrfarbenspriteregister (Register 28)

Auch hier bestimmen die einzelnen Bits den Zustand eines entsprechenden Sprites. Wird ein Bit auf 1 gesetzt, so befindet sich das jeweilige Sprite im Multi Color Modus.

2.6.2.14 Register 29 für die X-Erweiterung eines Sprites

Die Bits 0 bis 7 sind wiederum für die Sprites 0 bis 7 zuständig. Wird ein Bit gesetzt, so dehnt sich das Sprite in X-Richtung, also nach rechts, um das Doppelte aus.

2.6.2.15 Kollisionsregister Sprite-Sprite (Reg. 30)

Jedes Bit ist wiederum einem Sprite zugeordnet. Berühren sich nun z.B. die Sprites 2 und 7, so werden im Kollisionsregister die Bits 2 und 7 auf 1 gesetzt. Zugleich geht Bit 2 im Register 25 auf 1 und gibt damit zu verstehen, daß jetzt ein Interrupt ausgelöst werden könnte.

2.6.2.16 Kollisionsregister Sprite-Hintergrund (Reg. 31)

Dieses Register reagiert wie Register 30, nur, daß hier bei Berührung mit einem Zeichen des Hintergrundes das entsprechende Bit auf 1 geht und zugleich wird Bit 1 im Register 25 auf 1 gesetzt.

2.6.2.17 Rahmenfarbenregister (Register 32)

Wie schon der Name sagt, legt dieses Register die Rahmenfarbe des Bildschirms fest. Da bis zu 16 Farben auf dem Schirm erscheinen können, legen die unteren 4 Bits eine Farbe fest, was mit Werten von 0–15 zum Ausdruck kommt.

2.6.2.18 Die Hintergrundfarbenregister 0–3 (Register 33–36)

Im normalen Zustand ist nur Register 33 für die Farbinformation des Hintergrundes verantwortlich. Die Register 34 bis 36 werden dann für Farbinformationen im Multi Color und erweiterten Multi Color Modus benötigt.

2.6.2.19 Die Mehrfarbenspriteregister 0 und 1 (Reg. 37 u. 38)

Jedes Bit dieser Register entspricht wiederum einem Sprite; sie sind zuständig für die Darstellung der Sprites als Multi Color Sprites.

2.6.20 Spritefarbenregister 0–7 (Register 39–46)

Mit den unteren 4 Bits dieser Register legt man die jeweilige Farbe des entsprechenden Sprites im Normalmodus fest. Das heißt, es können bis zu 16 Farben (Werte von 0–15) für ein Sprite gewählt werden.

Wie wir gesehen haben, besitzt der VIC eine Menge von Registern, wobei hauptsächlich die einzelnen Bits dieser Register manipuliert werden müssen. Man kann zwar von Basic aus mit POKE und PEEK darauf zugreifen, was aber mindestens genauso umständlich ist, wie der Zugriff mit LDA und STA von Maschine aus. Zudem läßt sich die Umwandlung von binären Ziffern ins Hexadezimalsystem einfacher gestalten, als die Umwandlungen von binär nach dezimal und umgekehrt.

2.6.3 Der Musikbaustein SID (Sound Interface Device MCS 6581)

Dieser Baustein stammt ebenfalls aus der 65xx-Baureihe und bietet mit der Erzeugung von mehrstimmigen Signalen am Audio-Ausgang hervorragende Möglichkeiten zur Ton- oder Musikausgabe. Ebenso wie Chips CIA und VIC erfolgt die Programmierung seiner Register mit den normalen Speicherzugriffen LDA und STA. Dabei ergibt sich die Adresse eines Registers wiederum aus der Addition des Registeroffsets plus Basisadresse des SID. Die einzelnen Register haben folgende Bedeutung:

2.6.3.1 Die Frequenzregister 0,1,7,8,14 und 15

Der SID wurde so konstruiert, daß er 65536 verschiedene Frequenzen anbieten kann. Da ein Register immer nur 8 Bits aufnehmen kann, wird eine Stimme in einen nieder- und höherwertigen Teil zerlegt, so daß für eine Stimme 2 Frequenzregister zuständig sind. Außerdem können 3 Stimmen einzeln programmiert werden, die dafür 6 Register benötigen. Die Register 0, 7 und 14 sind somit für den niederwertigen und die Register 1, 8 und 15 für den höherwertigen Tonhöhenbereich der Stimmen 1, 2 und 3 zuständig. Betreibt man die Frequenzregister, die auch als Oszillatoren bezeichnet werden, mit dem Systemtakt von 1 MHz, so bewegen sich die Tonhöhen in einem Basisfrequenzbereich von 0 bis etwa 80 000 Hz.

2.6.3.2 Die Tastverhältnisregister 2,3,9,10,16 und 17

Diese Register werden ebenfalls in nieder- und höherwertige Register für die Stimmen 1, 2 und 3 eingeteilt. Allerdings nutzt man beim höherwertigen Teil nur die Bits 0 bis 3, so daß insgesamt Werte zwischen 0 und 4095 entstehen können. Die Register bestimmen nun das Tastverhältnis bei Rechteckimpulsen zwischen Impuls und Pause.

2.6.3.3 Die Steuerregister 4,11 und 18

Diese Steuerregister sind zuständig für die Steuerung des An- und Abschwellens und Halten eines Tones, sowie für die Wellenform (Klangfarbe: Rauschen, Rechteck, Sägezahn, Dreieck). Das Setzen der einzelnen Bits dieses Registers hat folgende Bedeutungen:

Bit 0 = 1 → Steuert den Hüllkurvengenerator (Reg. 5,6,12,13,19 u. 20), der das An- und Abschwellen eines Tones besorgt. Der Ton schwingt dabei in einer bestimmten Zeit und bleibt auf einem bestimmten Pegel, bis Bit 0 wieder gelöscht wird.

Bit 1 = 1 → Register 4: Die Frequenzen der Stimmen 1 und 3 werden zueinander synchronisiert.
Register 11: Die Frequenzen der Stimmen 1 und 2 werden zueinander synchronisiert.
Register 18: Die Frequenzen der Stimmen 2 und 3 werden zueinander synchronisiert.

Bit 2 = 1 → Register 4: Die Ringmodulation bewirkt, daß die Wellenform Dreieck der Frequenz Stimme 1 durch die Frequenzmischung der Stimmen 1 und 3 ersetzt wird.
Register 11: Die Ringmodulation bewirkt, daß die Wellenform Dreieck der Frequenz Stimme 2 durch die Frequenzmischung der Stimmen 1 und 2 ersetzt wird.
Register 18: Die Ringmodulation bewirkt, daß die Wellenform Dreieck der Frequenz Stimme 3 durch die Frequenzmischung der Stimmen 2 und 3 ersetzt wird.

Bit 3 Wurde zusammen mit dem Rauschen eine weitere Wellenform in derselben Stimme ausgewählt, so kann es passieren, daß das Rauschen abgeschaltet wird. Das Abschalten kann man mit diesem Bit verhindern.
Bit 4 = 1 → Auswahl der Wellenform Dreieck
Bit 5 = 1 → Auswahl der Wellenform Sägezahn
Bit 6 = 1 → Auswahl der Wellenform Rechteck
Bit 7 = 1 → Auswahl der Wellenform Rauschen

2.6.3.4 Die Register des Hüllkurvengenerators

Der Hüllkurvengenerator bestimmt die Zeit für das Anklingen, Halten, Abschwellen und Ausklingen eines Tones. In dem Zusammenhang wird man auf den Begriff ADSR stoßen. ADSR sind praktisch die Parameter ATTACK (Zeitmaß für Anschwingen), DECAY (Zeitmaß für Abschwingen), SUSTAIN (Lautstärkewert in Haltephase) und RELEASE (Zeitmaß für Ausklingen).

Register 5, 12 und 19 für die Stimmen 1, 2 und 3

Bit 0 – 3: Diese 4 Bits legen das Zeitmaß fest, in dem der Ton vom maximalen Lautstärkewert auf den Haltewert abfällt.
Bit 4 – 7: Diese 4 Bits legen das Zeitmaß fest, in dem der Ton von 0 bis zum Maximalwert anschwillt.

Register 6, 13 und 20 für die Stimmen 1, 2 und 3

Bit 0 – 3: Diese 4 Bits legen das Zeitmaß fest, in dem der Ton vom Haltewert ausklingt.
Bit 4 – 7: Diese 4 Bits enthalten den Lautstärkewert in der Haltephase.

2.6.3.5 Die Filterfrequenzregister (Register 21 und 22)

Durch das Einschalten eines Filters werden, wie schon der Name sagt, verschiedene Frequenzen bei der Tonausgabe herausgefiltert. Welche Frequenzen zu filtern sind, wird durch die Frequenzregister festgelegt. Register 21 ist dabei für den nieder- und Register 22 für den höherwertigen Frequenzwert zuständig.

2.6.3.6 Das Filterstimmenauswahlregister (Reg. 23)

Die Bits 0 – 2 dieses Registers steuern die gefilterte Ausgabe der Stimmen 1 – 3. Wird Bit 3 auf 1 gesetzt, wird der Filter für eine externe Stimmenquelle eingeschaltet. Die Bits 4 bis 7 bestimmen die Resonanzfrequenz des Filters, in der der Filter praktisch selber als Oszillator schwingt.

2.6.3.7 Das Filter Modus und Lautstärkeregister (Reg. 24)

Mit den Bits 0 – 3 wird die Lautstärke der gesamten Tonausgabe geregelt. Die 4 Bits schalten dabei die Lautstärkewerte zwischen 0 und 15. Die Bits 4 – 6 als Paßfilter lassen nur bestimmte Frequenzen durch den Filter.

Bit 4 dient als Tiefpaßschalter und filtert hohe Frequenzen aus. Bit 5 dient als Bandpaßschalter, der nur einen bestimmten Frequenzbereich durch den Filter läßt.

Bit 6 dient als Hochpaßschalter und filtert niedere Frequenzen aus.

Bit 7 = 1 schaltet den Oszillator 3 (Stimme 3) stumm, um eventuell die Werte aus den Registern für Stimme 3 als Zwischenspeicher zu benützen.

2.6.3.8 Register 25 und 26 zur Analog/Digitalwandlung

Diese Register haben überhaupt nichts mehr mit der Tonausgabe zu tun. Die Register 25 oder 26 dienen lediglich dazu, um Analogsignale, wie sie zum Beispiel von einem Drehpotentiometer kommen, in digitale Signale umzuwandeln, wie sie im Computer verwendet werden. Ein Beispiel für die Anwendung dieser Register ist beim VC-64 der Anschluß von Paddles.

2.6.3.9 Register 27 als Zwischenspeicher des Oszillators Stimme 3

Dieses Register enthält eine Zufallszahl, die dem gegenwärtigen Zustand des Klangfarbengenerators der Stimme 3 in der Rauschphase entspricht.

2.6.3.10 Register 28 als Zwischenspeicher des Hüllkurvengenerators der Stimme 3

Dieses Register enthält den gegenwärtigen Lautstärkewert der ADSR-Manipulation im Hüllkurvengenerator der Stimme 3. Abhängig von der Lautstärke könnte man somit Frequenzänderungen durchführen.

Mit Hilfe der Register des SID lassen sich effektvolle Klangergebnisse erzielen. Das Spektrum reicht dabei von allen möglichen Musikmelodien über phantastische Geräusche bis hin zur Sprachausgabe.

2.6.4 Der Adreßraum-Controller

Wenn man sich die gesamte Speicheraufteilung betrachtet, so stellt man fest, daß der VC-64 über 64 K RAM, 16 K Betriebssystem und Basic-Interpreter, 1 K Farb-RAM, 4 K Zeichen-ROM und I/O-Register verfügt. Der Prozessor selbst kann lediglich zur gleichen Zeit auf 64 K zurückgreifen und die wären bereits nur durch das RAM voll belegt. Es gibt nun zwei Methoden, um auf einen Speicher größer 64 K zuzugreifen. Die eine Methode wurde zum Beispiel beim CBM 8096 realisiert und beruht auf dem Prinzip der Speicherbereichsumschaltung (BANK-SELECT). Die andere Methode wird beim VC-64 durch Multiplexen erreicht. Das heißt, während sich der Prozessor zwischen zwei Speicherzyklen befindet, greift ein anderer Baustein auf den Adreßbus zu. Beim VC-64 wäre das der VIC. Nun ist es aber so, daß der Adreßraum nie größer als 64 K werden kann, da der Adreßbus nur über 16 Leitungen verfügt. Somit können insgesamt immer nur bestimmte Komponenten zur gleichen Zeit am Adreßbus liegen. Hier wird die Tatsache ausgenützt, daß man alle Systemelemente einfach nicht zur gleichen Zeit braucht. Die Komplexität dieser Logik

liegt aber darin, daß man die entsprechenden Bausteine (RAM, ROM, I/O, VIC usw.) so zusammenschaltet, daß sie im richtigen Moment als ein Block fungieren. Und genau diese Aufgabe wird durch einen Baustein geregelt, dem sogenannten Adreßraum-Controller.

Der Adreßraum-Controller ist ein Chip, der eine Unmenge von logischen Verknüpfungen zuläßt, und beim VC-64 auf der einen Seite mit 16 Eingangs- und auf der anderen Seite mit 8 Ausgangsleitungen verbunden ist. Je nachdem, welche Eingangssignale anliegen, konfiguriert der Adreßraum-Controller entsprechend über die Ausgangsleitung die Speicherkomponenten. Es dürfte klar sein, daß bei 256 möglichen Ausgangskombinationen, von denen im Normalfall auch nur ein paar benötigt werden, aus 65536 Eingangskombinationen ebenfalls nur ein paar davon gebracht werden können.

Was uns in diesem Zusammenhang interessiert, sind ansich nur 3 Eingangssignale am Adreßraum-Controller. Diese sind nämlich mit den Prozessorports 0 bis 2 verbunden und lassen sich daher softwaretechnisch behandeln. Es sind die Signale:

LORAM: Prozessorport Bit 0
Dieses Bit hat mit der Umschaltung von ROM auf RAM im Bereich $8000 bis $BFFF zu tun.

HIRAM: Prozessorport Bit 1
Dieses Bit hat mit der Umschaltung von ROM auf RAM im Bereich $E000 bis $FFF zu tun.

CHAREN: Prozessorport Bit 2
Dieses Bit ist für die Umschaltung von Zeichensatz-ROM auf RAM im Bereich $D000 bis $DFFF verantwortlich.

Mehr wollen wir hier nicht über den Adreßraum-Controller spezifizieren, da es ohnehin schon für den Anfänger ziemlich schwierig ist, das Bisherige zu erfassen. Wir haben nun praktisch sämtliche I/O Bausteine behandelt und können uns jetzt damit beschäftigen, wie diese im VC-64 behandelt werden.

2.6.5 Die Manipulation der Speicherbereiche

Nach dem Einschalten und der dadurch bedingten Systeminitialisierung liegt folgende Speicherkonfiguration fest:

$0000 – $7FFF	RAM Basic-Anwenderbereich
$8000 – $0FFF	RAM Basic-Anwenderbereich evt. Modul-ROM
$A000 – $BFFF	ROM Basic-Interpreter
$C000 – $CFFF	RAM Freier Anwenderbereich
$D000 – $DFFF	ROM Zeichengenerator (wird vom VIC gelesen)
$D000 – $D02E	VIC Register
$D400 – $D41C	SID Register
$D800 – $DBFF	RAM Farbspeicherung (Farbenspeicher)
$DC00 – $DC0F	CIA 1 Register
$DD00 – $DD0F	CIA 2 Register
$DE00 – $DEFF	RAM I/O-Bereich für Cartridge (CP/M-Option)
$DF00 – $DFFF	RAM I/O-Bereich zur freien Anwendung
$E000 – $FFFF	ROM Kernal-Bereich ist Betriebssystem

In dieser Systemzusammenstellung wird der Zeichengenerator nur vom VIC ausgelesen, während alle anderen angegebenen Speicherbereiche durch die CPU gelesen werden können und somit auch von Basic aus durch PEEK erreichbar sind.

An dieser Stelle sei noch erwähnt, daß es einen Unterschied bedeutet, ob ein Lese- oder ein Schreibzugriff auf den Speicher erfolgt. In der Einschaltkonfiguration wird nämlich beim Schreiben nicht in das ROM gespeichert, da das sowieso nicht geht, sondern in den darunter liegenden RAM-Bereich. Beim Lesen jedoch wird auf das ROM zurückgegriffen. Wir werden uns jetzt anschauen, wie man den Gesamtspeicher manipulieren kann, um z.B. auch an den RAM-Speicher zu gelangen, der im gleichen Adreßbereich sitzt wie das ROM.

Man schaltet um, indem man die Bits 0–2 im Datenregister der CPU setzt oder löscht. Das Datenregister selbst ist nicht in der CPU enthalten, sondern steht als RAM-Speicherstelle 1 in der Zeropage der CPU zur Verfügung. Im Normalzustand sind die Bits 0,1 und 2 auf 1 gesetzt. Betrachten wir einmal die 6 Fälle dieser Bits:

Bit 2	1	0	Erklärung
1	1	1	Normalkonfiguration
X	1	0	RAM von $0000 – $CFFF: Interpreter-ROM ist abgeschaltet
X	0	1	RAM von $0000 – $CFFF und $E000 – $FFFF: Interpreter- und Betriebssystem-ROM ist abgeschaltet
X	0	0	RAM von $0000 – $FFFF: Alles ist abgeschaltet. Die I/O-Bausteine funktionieren aber noch.
1	X	X	ROM von $D000 – $DFFF: VIC greift auf Zeichengenerator zu
0	X	X	ROM von $D000 – $DFFF: CPU kann jetzt zusätzlich auf den Zeichengenerator zugreifen.

Die Umschaltung darf an sich nur von Maschine aus vorgenommen werden. Das dürfte einleuchtend sein, da plötzlich der leere RAM eingeschaltet ist und das Gerät somit zu der Adresse springt, wo im ROM die Interruptroutine steht, im RAM aber nichts mehr vorfindet. Wurden alle 3 Bits gelöscht, zum Beispiel durch

```
POKE 1,0
```

so helfen nicht einmal mehr die Tasten STOP/RESTORE. Der Netzschalter muß betätigt werden. Übrigens werden Maschinensprache-Anfänger ebenfalls oft darauf zurückgreifen müssen.

Werden die Speicherbereiche aber von Maschine aus umgeschaltet, z.B. durch

```
LDA  #$00
STA  $01
```

so muß nach Beendigung einer Maschinenroutine vor RTS die Sequenz

```
LDA  #$07
STA  $01
```

angebracht werden, die die Normalkonfiguration wieder herstellt.

2.6.6 Die Belegung der Register im VC-64

Bisher haben wir nur ganz allgemein die Funktionen der I/O-Bausteine des VC-64 betrachtet. Will man aber die Möglichkeiten des Gerätes voll ausschöpfen, so muß man natürlich wissen, wo etwas, mit wem und wie verbunden ist. Dazu dient nun der nachfolgende Abschnitt.

Schauen wir uns zuerst die CIA-Bausteine an. Der VC-64 besitzt zwei Stück dieser Interface Adapter, die unter anderem zur Tastatur- und Joystickabfrage, für den seriellen Bus, den RS232-Bus und als Hilfsregister für den VIC Verwendung finden.

2.6.6.1 CIA 1

Der CIA 1 mit der Basisadresse $DC00 (=56320) hat seine Bidirektionalleitungen PA0-7 und PB0-7 mit der Tastaturmatrix verbunden. Parallel dazugeschaltet liegen PA0-4 und PB0-4 an den Ports 1 und 2, an denen Joysticks angeschlossen werden können. Will man die Joysticks abfragen, so müssen die jeweiligen Portleitungen (PA0-4, PB0-4) auf Eingang geschaltet werden, indem im Datenrichtungsregister (56322,56323) die Bits 0 – 4 gelöscht werden. Je nachdem, welches Bit von 0 – 4 im Ausgaberegister (56320,56321) auf 1 geht, zeigt es genau auf, was beim Joystick bewegt wird. So sind folgende Bedienungsmöglichkeiten festgelegt.

Bit 0 = 1 → Hebelbewegung nach oben
Bit 1 = 1 → Hebelbewegung nach unten
Bit 2 = 1 → Hebelbewegung nach links
Bit 3 = 1 → Hebelbewegung nach rechts
Bit 4 = 1 → Tastenknopf gedrückt

Mit den Bits 6 und 7 des Ausgaberegisters Port A wird zusätzlich noch auf Paddle-Betrieb umgeschaltet. Diese können dann vom Register 25 und 26 des SID bedient werden.

Der TIME ON DAY Eingang der CIA's wird mit dem Systemtakt von 50 Hz versorgt und innerhalb der CIA's auf 10 Hz heruntergeteilt. Das Register TOD 10THS wird dadurch im 1/10-Sekunden-Takt beeinflußt.

Der IRQ-Anschluß der CIA's ist mit dem IRQ-Eingang des Prozessors verbunden und kann daher auch vom CIA aus gesteuert werden. Zu erwähnen ist noch, daß die CIA's je über 8 Datenleitungen verfügen, die am Datenbus angeschlossen sind und über die die Registerdaten laufen.

Die CIA haben einen Ausgang SP, der als serieller Port vom Eingang oder Ausgang des Schieberegisters gesteuert wird. Beim CIA 1 ist dieser am USERPORT-Stecker Pin 5 angeschlossen und wird von einer eventuell betriebenen RS232-Schnittstelle gehandhabt.

2.6.6.2 CIA 2

Der VIC kann selbst nur auf eine 14-Bit Speicheradresse zugreifen. Das würde bedeuten, daß er auf alle Informationen nur im Adreßbereich von 0 bis 16383 Zugriff hat. Dem wurde abgeholfen, so daß der VIC die restlichen Adreßbits 14 und 15 aus einer anderen Quelle benutzt. Dafür stehen nun die Bits 0 und 1 des Ausgaberegisters A im CIA 2 zur Verfügung.

Das Portbit PA2 liegt am Anschluß M des USERPORT-Steckers an und erfüllt sonst keinerlei Funktion.

Die Bits 3 – 7 des Ausgaberegisters A (PA3–7) sind für den seriellen Bus zuständig und sind wie folgt belegt:

PA3 → ATN-Ausgang (zugleich auch Ausgangssignal am Userport Pin 9)
PA4 → CLK-Ausgang
PA5 → DATA-Ausgang
PA6 → CLK-Eingang
PA7 → DATA-Eingang

Die Bits 0 – 7 des Ports B (PB0–7) liegen normalerweise voll am USERPORT. Steht jedoch am Cartridge-Eingang eine RS2332-Schnittstelle, so werden die Ports wie folgt mit RS232-Signalen belegt:

PA0 → RXD
PA1 → RTS
PA2 → DTR
PA3 → RI
PA4 → DCD
PA6 → CTS
PA7 → DSR

Der IRQ-Ausgang des CIA 2 wurde im Gegensatz zum CIA 1 am NMI-Eingang des Prozessors angeschlossen und kann dadurch per Steuerung einen unbedingten Interrupt auslösen.

Zum Schluß der CIA-Belegung sei noch erwähnt, daß Bit 4 im Register des CIA 2 für das RXD-Signal der RS232-Schnittstelle zuständig ist.

2.6.6.3 VIC

Beim VIC brauchen wir eigentlich nichts über seine Belegung zu wissen, außer vielleicht, daß er Signale über den Antennenausgang oder Videoausgang schickt. Alles andere wird intern geregelt, wie zum Beispiel die Wiederauffrischung der dynamischen RAMs. Erwähnenswert ist vielleicht noch, daß er nur über 14 Adreßleitungen (A0 – A13) verfügt. Die Programmierung des VIC kann nur über seine Register geschehen, die dann automatisch die entsprechenden Ein- und Ausgänge beschalten.

2.6.6.4 SID

Ebenso wie beim VIC wird hier intern alles erledigt. Lediglich die Eingänge A1 und A2 sind zu betrachten, da damit über die Register 25 und 26 Drehpotentiometer (z.B. Paddles) gesteuert werden. Daher liegen die Eingänge an den Control Ports an, und die Umschaltung dazu wird vom CIA 1 geregelt.

2.6.6.5 Zusammenfassung

Auf eine genaue Darstellung aller Belegungen und deren Pinbeschreibung wurde im Rahmen dieser Ausgabe verzichtet. Zum einen ist diese Materie für den Maschinenanfänger ohnehin ein „Buch mit sieben Siegeln". Zum anderen gibt es auf dem Markt hervorragende Literatur über dieses Thema, die keine Fragen offen läßt. Außerdem werden wir uns im nächsten Abschnitt mit den Interface-Schnittstellen auseinandersetzen und dabei auch einige Belegungen kennenlernen.

2.6.7 Interface am VC-64

Wie schon erwähnt, stellen die Interfacebausteine die Verbindung zu den Peripheriegeräten her. Beim VC-64 bestehen folgende Verbindungen:

1) TV-Antennenausgang
2) Audio & Video Ausgang
3) Tastatur
4) Rekorder Anschluß
5) USERPORT
6) Serieller Bus
7) Cartridge Anschluß
8) Control Port 1 und 2

Als Basic-Programmierer braucht man sich nicht um die Entwicklung von Interfaceprogrammen kümmern. Außer der USERPORT- und CONTROL PORT-Steuerung nimmt einem das Betriebssystem diese Arbeit ab. Der Maschinenprogrammierer jedoch sollte die Funktionen der Interfacebausteine kennen, zumindest die grundlegenden Methoden.

Im folgenden werden wir auf die Control Ports, den User Port, den seriellen Bus und auf den Cartridge-Eingang Bezug nehmen.

2.6.7.1 Die Control Ports 1 und 2

Auf der rechten Seite, links neben dem Netzschalter, befinden sich zwei Anschlußstecker, die mit je 9 Kontakten ausgestattet sind und die Bezeichnung Control Port 1 und 2 tragen. Die Kontakte sind in zwei Reihen übereinander angeordnet und werden wie folgt bezeichnet:

oben: 1 2 3 4 5
unten: 6 7 8 9

Hauptsächlich dienen diese Stecker zum Anschließen von Joysticks. Aber auch Paddles (Drehpotentiometer) oder eine Light Pen-Funktion (Lichtgriffel) können hier angeschlossen werden. Die Belegung der Kontakte sieht folgendermaßen aus:

CONTROL PORT 1

1 = JOYA0 → PRA(0) AM CIA 1
2 = JOYA1 → PRA(1) AM CIA 1
3 = JOYA2 → PRA(2) AM CIA 1
4 = JOYA3 → PRA(3) AM CIA 1
5 = POTAY → REGISTER 25 IM SID
6 = BUTTON A → PRA(4) AM CIA 1
ODER
6 = LIGHTPEN → LICHTGRIFFELKNOPF
7 = + 5V
8 = MASSE
9 = POTAX → REGISTER 26 IM SID

CONTROL PORT 2

1 = JOYB0 → PRB(0) AM CIA 1
2 = JOYB1 → PRB(1) AM CIA 1
3 = JOYB2 → PRB(2) AM CIA 1
4 = JOYB3 → PRB(3) AM CIA 1
5 = POTBY → REGISTER 25 IM SID
6 = BUTTON B → PRB(4) AM CIA 1
7 = + 5V
8 = MASSE
9 = POTBX → REGISTER 26 IM SID

Anhand einer Joystickabfrage am Control Port 2 soll demonstriert werden, wie diese programmtechnisch zu behandeln sind.

```
START    LDA  #$E0      ; $E0 = %1110 0000 Bit 0 – 4 auf Null setzen
         STA  $DD03     ; PRB(0–4) als Eingänge durch Datenricht.Reg.
JOYST    LDA  $DD01     ; PRB (0–4) im CIA 1 abfragen
         CMP  #$01      ; Ist Bit 0 gesetzt (Joystickhebel nach oben)
         BEQ  ROUTUP    ; wenn ja, verzweige nach ROUTUP
         CMP  #$02      ; Ist Bit 1 gesetzt (Joystickhebel nach unten)
         BEQ  ROUDOWN   ; wenn ja, verzweige nach ROUDOWN
         CMP  #$04      ; Ist Bit 2 gesetzt (Joystickhebel nach links)
         BEQ  ROULEFT   ; Wenn ja, verzweige nach ROULEFT
         CMP  #$08      ; Ist Bit 3 gesetzt (Joystickhebel nach rechts)
         BEQ  ROURGHT   ; Wenn ja, verzweige nach ROURGHT
         CMP  #$10      ; Ist Bit 4 gesetzt (Feuerknopf)
         BNE  JOYST     ; Wenn nicht, Joystick erneut abfragen
         JMP  BUTTON    ; Sprung zur Knopfdruckroutine BUTTON
ROUTUP   JMP  UP        ; Sprung zur Routine OBEN
ROUDOWN  JMP  DOWN      ; Sprung zur Routine UNTEN
ROULEFT  JMP  LEFT      ; Sprung zur Routine LINKS
ROURGHT  JMP  RIGHT     ; Sprung zur Routine RECHTS
END      LDA  #$FF      ; Sämtliche Bits auf 1 setzen um PRB (0–7)
         STA  $DD03     ; als Ausgänge zu schalten (Tastaturabfrage)
         RTS            ; Rückkehr aus der Joystickabfrage
```

Dieses Maschinenprogramm fragt den Joystick, in welche Richtung der Hebel bewegt oder, ob der sogenannte Feuerknopf gedrückt wurde. Falls der Hebel oder Knopf bewegt wurde, wird im Ausgaberegister das entsprechende Bit gesetzt. Die CMP-Befehle stellen nun fest, welche Bewegung ausgeführt wurde und das Programm verzweigt zum jeweiligen Sprungbefehl JMP. Dort erfolgt dann der unbedingte Sprung zur jeweiligen Routine, die dann die erforderliche Manipulation vornimmt. Zum Beispiel könnten diese Routinen ein Sprite in die jeweilige Richtung bewegen. Vor der Joystickabfrage werden die ersten 5 Bits im Datenrichtungsregister gelöscht und dadurch die entsprechenden Ports im Ausgaberegister als Eingänge geschaltet. Dadurch wird der Zugriff der Ports auf die Tastaturmatrix abgeschaltet. Um nach Rückkehr oder Beendigung der Joystickabfrage die Tastatur wieder benutzen zu können, müssen sämtliche Ports auf Ausgang geschaltet werden. Das geschieht im obigen Programm mit der Assemblersequenz END. Zu erwähnen wäre

noch, daß Zwischenwerte der Hebelbewegung, wenn z.B. der Hebel nach rechts oben gedrückt wird, von der Joystickabfrage nicht berücksichtigt werden, da ja die CMP-Befehle nur immer 1 Bit abfragen.

2.6.7.2 Der USERPORT

Blickt man hinten auf die Rückseite des VC-64, so sieht man ganz rechts einen Anschlußstecker mit 24 Kontakten, der die Bezeichnung USERPORT trägt. Der Stecker ist zu 2 Reihen mit je 12 Kontakten ausgestattet, die übereinander angeordnet sind. Die Kontakte tragen folgende Bezeichnung:

OBEN: 1 2 3 4 5 6 7 8 9 10 11 12
UNTEN: A B C D E F H J K L M N

Wie der Name (Anwender Port) sagt, wurde der User Port geschaffen, um anwenderspezifische Peripheriegeräte anschließen zu können. Beispielsweise könnte man damit eine Lichtorgel betreiben oder eine Parallelschnittstelle ermöglichen. Die Belegung der Kontakte sieht folgendermaßen aus.:

1 = MASSE	A = MASSE
2 = +5V	B = –FLAG (Eingang ICR im CIA 2)
3 = –RESET (CPU)	C = PB0 (PRB(0) im CIA 2)
4 = CNT (TIMER A und B im CIA 1)	D = PB1 (PRB(1) im CIA 2)
5 = SP (Schieberegister im CIA 1)	E = PB2 (PRB(2) im CIA 2)
6 = CNT (TIMER A und B im CIA 2)	F = PB3 (PRB(3) im CIA 2)
7 = SP (Schieberegister im CIA 2)	H = PB4 (PRB(4) im CIA 2)
8 = –PC (Handshake im CIA 2)	J = PB5 (PRB(5) im CIA 2)
9 = ATN-Ausgang (PRA(3) im CIA 2)	K = PB6 (PRB(6) im CIA 2)
10 = 9 VAC (Wechselspannung)	L = PB7 (PRB(7) im CIA 2)
11 = 9 VAC (Gegenpol)	M = PA2 (PRA(2) im CIA 2)
12 = MASSE	N = MASSE

Sie können jetzt vielleicht erahnen, was man mit dieser Schnittstelle alles anfangen kann. Eine weitere Anwendung wäre der Datentransfer in beiden Richtungen mit den CBM-Rechnern, die ja ebenfalls mit dem User Port ausgestattet sind. Außer entsprechenden Leitungen wird für diesen Zweck hardwaremäßig weiter nichts benötigt.

2.6.7.3 Der Serielle Bus

Auf der Rückseite des VC-64 befindet sich eine Rundbuchse mit 6 Kontakten, die wie folgt angeordnet sind:

```
5   1
  6
4   2
  3
```

An diesen Stecker werden die OEM-Produkte wie Floppy Laufwerk oder Drucker angeschlossen. Die Belegung der Kontakte sieht folgendermaßen aus:

1 = SRQ (Signal FLAG am ICR des CIA 1)
2 = MASSE
3 = ATN (PA3 am PRA(3) des CIA 2)
4 = CLK (PA4 OUT und PA6 IN beim CIA 2)
5 = DATA (PA5 OUT und PA7 IN beim CIA 2)
6 = IFC (Resetleitung zur CPU)

Der Serielle Datenbus wird ähnlich gehandhabt, wie der IEC-Bus der CBM-Geräte. Nur werden hier die Daten bitweise seriell übertragen. Daher ist die Bandrate ca. 5 mal langsamer als beim 8-Bit-Paralleltransfer der größeren Brüder. Die Datenübertragung findet nun nicht so einfach statt, daß man mit LDA oder STA darauf zugreift, sondern findet im sogenannten Handshake- oder Quittierungsverfahren statt. Die Quittierungslogik im Handshakeverfahren benötigt immer 3 Geräte:

a) Den TALKER (Sprecher bzw. Sender)
b) Den LISTENER (Zuhörer bzw. Empfänger)
c) Den CONTROLLER (Manager – Steuerung)

Der TALKER ist das Gerät, das die Daten über die DATA-Leitung ausgibt. Es kann immer nur ein Gerät zur gleichen Zeit senden.

Der LISTENER ist das Gerät, das die Daten über die DATA-Leitung empfängt. Im Gegensatz zum TALKER können jedoch mehrere Geräte als LISTENER zur selben Zeit am Verfahren teilnehmen.

Der CONTROLLER ist der „Manager" des Ganzen, der die Übertragung steuert. Auch hier gilt: Es darf nur ein CONTROLLER zur selben Zeit angeschlossen sein. In unserem Fall ist es natürlich der VC-64.

Jedes Gerät, das im Quittierungsverfahren teilnimmt, erhält eine Adresse (im Basic-Befehl OPEN ist eine Nummer < 255). Dabei hat jedes Gerät, das senden und empfangen kann, zwei Nummern (eine als LISTENER und eine als TALKER). In diesem Sinn kann man beim VC-64 auch die Tastatur und den Bildschirm hinzurechnen.

Ähnlich der IEC-Bus Konvention geht die Datenübertragung allgemein so vor sich: Vor der Ein- und Ausgabe werden die beteiligten Geräte vom CONTROLLER zum TALKER oder LISTENER erklärt, je nachdem, welches Gerät die Daten senden oder empfangen soll. Dann teilt der TALKER über die Handshake-Leitungen mit, daß er zum Senden bereit ist und die anliegenden Daten gültig sind. Das bezeichnet man mit TALKEN. Die LISTENER quittieren, daß sie bereit sind, zum empfangen. Das nennt man dann LISTEN. Erst jetzt erfolgt der vorgesehene Datentransfer. Nachdem das letzte Byt übertragen worden ist, melden sich die Geräte wieder ab, was mit UNTALKEN und UNLISTEN bezeichnet wird.

Durch dieses Verfahren ist es möglich, daß schnelle Geräte mit langsameren in Verbindung treten können. Die gesamte Datensteuerung des Seriellen Bus findet auf 5 Leitungen statt, die wie folgt bezeichnet werden:

SRQ = SERVICE REQUEST (Bedienungsanforderung)

Durch dieses Signal meldet ein Gerät dem CONTROLLER, daß es bereit ist zur Datenübertragung. Es fordert also Bedienung an. Diese Leitung wird vom VC-64 Betriebssystem nicht verwendet.

ATN = ATTENTION (Achtungssignal)

ATN ordnet die Geräte als LISTENER oder TALKER zu. Der CONTROLLER setzt dieses Signal auf 0, während er Befehle über DATA ausgibt. Ist ATN im 0-Zustand, befinden sich nur Steuermitteilungen am Bus. Steht ATN auf 1, können vorher adressierte Geräte Daten übermitteln. Der CONTROLLER teilt also den Geräten mit, daß sie aufpassen sollen.

CLK = CLOCK (Taktleitung)

CLK kontrolliert praktisch das Handshakeverfahren. Da die verschiedenen Rückmeldungen nicht bit-, sondern byteweise erfolgen, gibt CLK bekannt, wann das gesamte Byte gesendet bzw. empfangen werden soll.

DATA = Datenleitung

Über diese Leitung werden sämtliche Datenbytes bitweise übertragen. DATA übernimmt aber auch noch verschiedene Rückmeldungen, um so am Handshakeverfahren teilnehmen zu können.

IFC = INTERFACE CLEAR (System Reset)

IFC initialisiert intern alle Geräte, die am Bus hängen und entspricht dem RESET-Signal der CPU.

Mit Hilfe dieser Leitungen kann nun die Busübertragung auf folgende Weise spezifiziert werden:

Der CONTROLLER schickt ein LOW-Signal (Bus wird in negativer Logik betrieben) auf der ATN-Leitung zum Bus. Dadurch werden die LISTENER- über DATA und die TALKER-Adressen über CLK allen angeschlossenen Geräten mitgeteilt. Jedes Gerät vergleicht nun seine Adresse (= Gerätenummer) mit der mitgeteilten Adresse und wird so zum TALKER oder LISTENER erklärt. Dabei werden die Signale CLK und DATA auf 0 gehalten und signalisieren so, daß sie für das Busverfahren bereit sind. Die Datenübertragung erfolgt jetzt ohne Eingreifen des Controllers so, daß der TALKER die Daten auf den Bus legt, während er CLK auf 1 setzt. Er gibt dadurch bekannt, daß er bereit ist, Daten auszugeben. Sobald der LISTENER bereit ist, Daten zu empfangen, legt er DATA auf 1 und die bitweise Übertragung kann beginnen. Normalerweise legt der TALKER beim Senden innerhalb von 200 Mikrosekunden (Ms) CLK auf 0. Wird diese Zeit aber überschritten, so teilt der TALKER dem LISTENER mit, daß nun das letzte Byte folgt. In diesem Fall setzt der LISTENER mindestens 60 µs DATA auf 0 und gleich wieder auf 1 und bestätigt somit das Endesignal. Innerhalb der nächsten 60 µs muß der TALKER die CLK-Leitung wieder auf 0 ziehen. Angefangen mit Bit 0 werden jetzt die einzelnen Bits eines Bytes über DATA transferiert. Dabei wird bei jedem Bit CLK auf 1 gesetzt, um die Gültigkeitsmeldung anzuzeigen. Nach dem Senden von Bit 7 hat der Talker CLK auf 0 und DATA auf 1 gelegt.

Der LISTENER bestätigt den Empfang des gesamten Bytes, indem er DATA auf 0 setzt. Damit liegen beide Leitungen CLK und DATA auf 0 und geben das Übertragungsende an. War es das letzte Byte des Datentransfers, dann werden beide Leitungen auf 1 gezogen und dadurch freigegeben. Es fragt sich jetzt nur noch, wie deren Befehle vom CONTROLLER gesendet werden? Dazu benutzt der CONTROLLER die ATN-Leitung und setzt sie auf 0. Solange ATN auf 0 bleibt, können Befehle vom CONTROLLER über

DATA gesendet werden. Dabei geschieht das gleiche Handshakeverfahren wie beim Datenverkehr, nur, daß diesmal der CONTROLLER Vorrang hat.

Für die Funktionen „Gerät anmelden", „Byte vom Gerät zum Rechner" oder „Byte vom Rechner zum Gerät" und „Gerät abmelden", stehen jeweils eigene Maschinenroutinen des Betriebssystems zur Verfügung. Der Aufruf dieser Routinen sollte über den sogenannten Sprungverteiler laufen. Das sind Vektoradressen hinter JMP-Befehlen am Ende des ROM-Bereiches, die unter anderem auch auf die Bus-Routinen zeigen. Da die Adressen des Sprungverteilers bei allen Betriebssystemen der Commodore-Rechner gleich geblieben sind, sollte man diese auch benutzen. Die eigentlichen Routinen haben nämlich bei den verschiedenen Betriebssystemen andere Startadressen.

Anhand eines kleinen Assemblerprogramms wollen wir nun demonstrieren, wie man auf ein Zeichen, das auf Diskette abgespeichert ist, zugreift.

```
        LDA   #NAMLEN   ; Filenamenlänge
        LDX   #NAMADL   ; ADL wo Filename steht
        LDY   #NAMADH   ; ADH wo Filename steht
        JSR   SETNAM    ; Filenamenvorbereitung
        LDA   #FILNUM   ; Logische Filenummer
        LDX   #DEVNUM   ; Gerätenummer
        LDY   #SECNUM   ; Sekundäradresse
        JSR   SETLFS    ; Fileparametervorbereitung
        JSR   OPEN      ; File öffnen
        LDA   #$00      ; Statusmeldung auf 0
        STA   STATUS    ; in der Zeropage zurücksetzen
        JSR   TALK      ; Peripheriegerät wird zum TALKER erklärt
        LDA   SECADR    ; Sekundäradresse
        JSR   SECOND    ; auf den Bus legen
        JSR   ACPTR     ; Erstes Byte holen
        STA   MEM1      ; und abspeichern
        JSR   ACPTR     ; Zweites Byte holen
        STA   MEM2      ; und abspeichern
        JSR   UNTLK     ; TALKER abmelden
        LDA   STATUS    ; Status holen
        BNE   ERRMSG    ; wenn größer 0, dann Fehlermeldung
        RTS             ; Ende
ERRMSG  JSR   SETMSG    ; Ausgabe der Fehlermeldung
```

Zum Abschluß sei noch erwähnt, daß sämtliche JSR im obigen Programm auf die Hauptsprungliste zurückgreifen.

2.6.7.4 Der Modul Steckplatz (Cartridge Eingang)

An der Rückseite des VC-64 ist ganz links ein 44-poliger Anschlußstecker. Die Kontakte sind wie folgt bezeichnet:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
A	B	C	D	E	F	H	J	K	L	M	N	P	R	S	T	U	V	W	X	Y	Z

In diesen Steckplatz, der auch „Cartridge Stecker" benannt wird, lassen sich Module einstecken, die zum Beispiel für Spiele oder Tools geeignet sind. Die Belegung der Kontakte lautet folgendermaßen:

1 =	MASSE	A =	MASSE
2 =	+5 V	B =	–ROMH (OUT ADR.RAUM-CONTR.)
3 =	+5 V	C =	–RESET (RESET VON CPU)
4 =	–IRQ (IRQ VON CPU)	D =	–NMI (NMI VON CPU)
5 =	CR/–W (R/W VON CPU)	E =	PHI2 (SYSTEMTAKT)
6 =	DOT CLOC (FÜR VIC-TAKT)	F =	CA15 (AM ADRESSBUS)
7 =	–I/O1 (I/O FÜR $DE00–$DEFF)	H =	CA14 (AM ADRESSBUS)
8 =	–GAME (IN ADRESSRAUMCONTROLLER)	J =	CA13 (AM ADRESSBUS)
9 =	–EXROM (IN ADRESSRAUMCONTROLLER)	K =	CA12 (AM ADRESSBUS)
10 =	–I/O2 (I/O FÜR $DF00–$DFFF)	L =	CA11 (AM ADRESSBUS)
11 =	–ROML (OUT ADRESSRAUMCONTROLLER)	M =	CA10 (AM ADRESSBUS)
12 =	BA (LESESIGNAL VOM VIC)	N =	CA09 (AM ADRESSBUS)
13 =	–DMA (IN – FÜR EXTERNEN ZUGRIFF)	P =	CA08 (AM ADRESSBUS)
14 =	CD7 (AM DATENBUS)	R =	CA07 (AM ADRESSBUS)
15 =	CD6 (AM DATENBUS)	S =	CA06 (AM ADRESSBUS)
16 =	CD5 (AM DATENBUS)	T =	CA05 (AM ADRESSBUS)
17 =	CD4 (AM DATENBUS)	U =	CA04 (AM ADRESSBUS)
18 =	CD3 (AM DATENBUS)	V =	CA03 (AM ADRESSBUS)
19 =	CD2 (AM DATENBUS)	W =	CA02 (AM ADRESSBUS)
20 =	CD1 (AM DATENBUS)	X =	CA01 (AM ADRESSBUS)
21 =	CD0 (AM DATENBUS)	Y =	CA00 (AM ADRESSBUS)
22 =	MASSE	Z =	MASSE

Sobald ein Modul eingesteckt wird, wirken die Signale GAME und EXROM auf den Adreßraum Controller ein. Je nachdem, welchen Pegel diese Signale besitzen, wird die interne Speicherkonfiguration umgeändert, so daß dem Speicher des Moduls der ROM-Adreßraum ab $8000 zugewiesen wird. Wenn nun die RESET-Routine des Kernal-ROMs, die ja auf Modul prüft, auf den Code „CBM80" im Adreßbereich ab $FD00 trifft, so verzweigt die RESET-Routine nach $8000. Dort muß natürlich entweder eine eigene Initialisierungsroutine stehen oder von dort die alte Einschaltroutine aufgerufen werden.

Der Vorteil des ganzen ist nun der, daß Programm-Module, die am Cartridge-Eingang liegen, sofort nach dem Einschalten zur Verfügung stehen. Desweiteren lassen sich Programme ohne Modul im normalen RAM-Bereich entwickeln und austesten, um dann später als Modul-ROM-Version in Serie zu gehen.

2.6.8 Die Graphik des VC-64

In diesem Abschnitt möchte ich nur ganz kurz auf die Graphik des VC-64 eingehen, um nur einige der vielfältigen Möglichkeiten aufzuzeigen. Auf eine eingehende Betrachtung wurde in diesem Rahmen verzichtet, da es dazu schon sehr gute Literatur gibt (siehe Literaturverzeichnis). Außerdem läuft die Programmierung über die VIC-Register, die eingehend behandelt wurden.

Wie schon erwähnt, verfügt der VC-64 über mehrere Graphik-Modis. Dazu zählen folgende drei verschiedene grundsätzliche Arten:

Die Zeichengenerator Graphik
Die Hochauflösende Graphik
Die Sprites

Dazu können diese einzelnen Arten teilweise noch im Multi Color Mode (Mehrfarbenmodus) oder Extended Color Mode (Erweiterter Farbenmodus) betrieben werden.

Die Zeichengeneratorgraphik

Kurz nach dem Einschalten befindet sich der VC-64 im normalen Zeichen-Modus. Dort greift er auf den Video-RAM (ab 1024) zu. Das Byte, das er an der entsprechenden Stelle vorfindet, benützt er gleichzeitig als Vektor in das Zeichengenerator-ROM. Zugleich muß er aber auch noch vom Farb-RAM die Information für die Farbe des Zeichens hervorholen. Erst dann erscheint das Zeichen auf dem Bildschirm. In einem Byte des Farb-RAMs sind 4 Bits für die Farbe des eigentlichen Zeichens zuständig, während die restlichen 4 Bits den Hintergrund des Zeichens zu verantworten haben. Daraus erklärt sich die Tatsache, daß die Hintergrundfarbe für alle Zeichen dieselbe ist. Übrigens besteht jedes Zeichen aus einer 8*8-Matrix.

Es gibt für die Zeichendarstellung noch die Möglichkeit einer 4*8-Matrix, wobei dann ein Zeichen gleichzeitig aus einer 4-Farbenmischung zusammengesetzt ist. Das nennt sich dann Multi-Color-Charakter-Mode. Dieser Mehrfarbenzeichenmodus wird durch Setzen des Bit 4 im VIC-Register 22 eingeschaltet. Das Bit 3 (höchstwertigstes) der 4 Bits, die für das Zeichen selbst verantwortlich sind, besitzt jetzt einen Sonderstatus. Ist dieses Bit auf 1 gesetzt, so setzt sich in der 4*8-Matrix die Farbe des Zeichens aus 2 Punkten zusammen. Dadurch verschlechtert sich natürlich die Auflösung des Zeichens. Zwei Bits des Zeichenmusters stellen jetzt dafür aber 4 Farben dar. Sollte das Bit 3 auf Null gesetzt sein, springt die normale 8*8-Matrix ein, und die restlichen Bits 0,1 und 2 können jetzt nur 8 Farben darstellen. Daher ist es unumgänglich, im Multi-Color-Charakter-Mode das Bit 3 im Farb-RAM auf 1 zu setzen.

Zusätzlich im Zeichenmodus gibt es den sogenannten Extended-Color-Modus, der nur für die Zeichenausgabe verwendet wird. Dieser wird eingeschaltet, indem das Bit 6 im Register 17 auf 1 gesetzt wird. Dann wird die Farbe des Zeichens selbst, das ja die gesetzten Bits sind, wie im Normalmodus aus dem Farb-RAM geholt. Die nicht gesetzten Bit, die die Farbe des Hintergrundes bestimmen, holen sich dabei die Farbe aus dem Video-

RAM, wobei dort Bit 6 und 7 bestimmen, welches Hintergrundfarbregister im VIC die Farbe bestimmt.

BIT 6 7	NUMMER DES VIC-REGISTERS
0 0	33 (Hintergrundfarbregister 0)
0 1	34 (Hintergrundfarbregister 1)
1 0	35 (Hintergrundfarbregister 2)
1 1	36 (Hintergrundfarbregister 3)

Allerdings verbleiben jetzt im Video-RAM nur noch 6 Bits zur Zeichendarstellung, so daß maximal 64 verschiedene Zeichen zur gleichen Zeit erscheinen können.

Die Hochauflösende Graphik

Der Graphik-Modus wird oft auch mit HIRES-Graphik bezeichnet, was die Abkürzung für High-Resolution-Graphik bedeutet. In dieser Betriebsart besitzt der Bildschirm eine Auflösung von 320*200 Punkten und wird durch Setzen des Bit 5 im VIC-Register 17 eingeschaltet. Irgendwo muß nun die Information abgelegt werden, ob nun ein Punkt von insgesamt 64000 Punkten gesetzt ist oder nicht. Dazu wird ein RAM-Speicherbereich gebraucht, der 8 K groß ist. Jedes Bit (8 * 8000 Bit) davon liefert diese Information. Der Adressierungsbereich dieses Graphikspeichers wird durch die Bits 1–7 im VIC-Register eingestellt. Die Farbinformation kommt jetzt aber nicht mehr aus dem Farb-RAM, sondern aus dem Video-RAM. Jedes Byte des 1K großen Video-RAM ist für eine 8*8 Punktmatrix zuständig. Dabei liefern die unteren 4 Bits die Farbe für die nicht gesetzten und die oberen 4 Bits die Farbe für die gesetzten Bits des Graphikspeichers.

Im Graphikbetrieb kann man auch noch in den Multi-Color-Modus umschalten, indem die Bits 5, 6 im VIC-Register 17 und Bit 4 im Register 22 auf 1 gesetzt werden. Da jetzt 2 Bits im Graphikspeicher für einen Punkt zuständig sind, beträgt die Auflösung natürlich nur noch 160*200. Dafür lassen sich jetzt in der 8*8-Matrix 4 Farben bilden, wobei diese Information folgendermaßen gewonnen wird.

Graphikspeicherbits	Farbquelle
00	VIC-Register 33
01	Video-RAM niederwertige 4 Bits
10	Video-RAM höherwertige 4 Bits
11	Farb-RAM

Die Sprites

Sprites sind ebenfalls Graphiken, aber mit einer Auflösung von 24*21 Punkten. Die Besonderheit daran ist, daß sie sich bewegen lassen, indem in den entsprechenden Koordi-

natenregistern des VIC die X- und Y-Werte im Rahmen eines Programmes laufend geändert werden. Zusätzlich kann man die Sprites um das vierfache vergrößern, wenn die dafür verantwortlichen Register ein Sprite in X- und Y-Richtung verdoppeln. Insgesamt benötigt ein Sprite einen eigenen Speicherbereich von 63 Bytes. Das errechnet sich so:

```
 24 *  21 = 504 Punkte
504 /   8 Bit = 63 Bytes.
```

Das Setzen oder Löschen der einzelnen Punkte des Sprites erfolgt so, daß jeweils 24 Bits (3 Bytes) für eine Zeile (X-Richtung) zuständig sind. Die Farbinformation der gelöschten Bits kommt dabei aus dem Hintergrundfarbenregister 0 (VIC-Register 33), so daß diese praktisch unsichtbar sind. Die Farbe der gesetzten Bits stammt aus dem jeweiligen Sprite-Color-Register des VIC.

Den Speicherbereich von 63 Bytes kann man in etwa mit dem Graphikspeicher vergleichen, nur, daß dieser Spritespeicherbereich woanders liegt. Dazu dienen die letzten 8 Bytes des Video-RAMs (2040–2047). Der Inhalt dieser Bytes wird mit 64 multipliziert und ergibt damit den Adreßbereich der Spritespeicherung, der relativ zum Adressierungsbereich des VIC angeordnet ist. Will man die Sprites in einem höheren RAM-Bereich abspeichern, so muß zuerst der Adressierungsbereich des VIC verschoben werden, was mit Datenrichtungsregister und Ausgaberegister A im CIA 2 zu geschehen hat. Das ganze ist aber von Anfängern mit äußerster Vorsicht durchzuführen.

Neben dem normalen Spritebetrieb wird durch Setzen der Bits 0–7 im VIC-Register 28 (Sprite Multi Color Register) der Mehrfarbenmodus der Sprites 0–7 eingeschaltet. Dadurch schrumpft die Auflösung in X-Richtung und der Graphikmodus um die Hälfte (12*21). 2 Bits stellen jetzt einen Punkt dar und bestimmen die Farbe auf folgende Weise:

Spritegraphik Speicherbits	Farbquelle
00	Hintergrundfarbregister 0
01	Mehrfarbenspriteregister 0
10	Mehrfarbenspriteregister 1
11	Spritefarbenregister

Über die Programmierung der graphischen Funktionen sei nur soviel erwähnt, daß an sich nur die einzelnen Bits gesetzt oder gelöscht werden. Um zu sehen, wie das durchgeführt wird, sollten Sie sich vielleicht noch einmal im Kapitel 1 die Maschinenbefehle AND, ORA und EOR ansehen.

Wir sind damit am Ende des Abschnitts über die Interfacetechnik. Anschließend folgt nun der Bericht über die Systemprogramme in den ROMs.

2.7 Der ROM-Bereich

Wir wollen uns jetzt den gesamten ROM-Bereich ansehen. Gleich am Anfang stehen zwei Vektoren, die auf Reset ohne Modultest und auf die NMI-Routine deuten. Danach befindet sich eine ASC64-Tabelle mit dem Text CBMBASIC und dahinter kommt die Adressentabelle, die auf die Basicroutinen zeigt.

BASIC 2 hex	BASIC 2 dez	Inhalt hex	Vektoradr. dez	Bedeutung
A000	40960	94 E3	58260	zur Teilrout. Reset
A002	40962	7B E3	58235	zur NMI-Routine

BASIC 2 hex	BASIC 2 dez	Inhalt dez	ASC64-Code
A004	40964	67 66 77 66 65 83 73 67	CBMBASIC

BASIC 2 hex	BASIC 2 dez	Inhalt hex	Startadr. - 1 (dez)	Befehl
A00C	40972	30 A8	43056	END
A00E	40974	41 A7	42817	FOR
A010	40976	1D AD	44317	NEXT
A012	40978	F7 A8	43255	DATA
A014	40980	A4 AB	43940	INPUT#
A016	40982	BE AB	43966	INPUT
A018	40984	80 B0	45184	DIM
A01A	40986	05 AC	44037	READ
A01C	40988	A4 A9	43428	LET
A01E	40990	9F A8	43167	GOTO
A020	40992	70 A8	43120	RUN
A022	40994	27 A9	43303	IF
A024	40996	1C A8	43036	RESTORE
A026	40998	82 A8	43138	GOSUB
A028	41000	D1 A8	43217	RETURN
A02A	41002	3A A9	43322	REM
A02C	41004	2E A8	43054	STOP
A02E	41006	4A A9	43338	ON
A030	41008	2C B8	47148	WAIT
A032	41010	67 E1	57703	LOAD
A034	41012	55 E1	57685	SAVE
A036	41014	64 E1	57700	VERFIY
A038	41016	B2 B3	46002	DEF
A03A	41018	23 B8	47139	POKE
A03C	41020	7F AA	43647	PRINT#
A03E	41022	9F AA	43679	PRINT
A040	41024	56 A8	43094	CONT
A042	41026	9B A6	42651	LIST
A044	41028	5D A6	42589	CLR
A046	41030	85 AA	43653	CMD
A048	41032	29 E1	57641	SYS
A04A	41034	BD E1	57789	OPEN
A04C	41036	C6 E1	57798	CLOSE
A04E	41038	7A AB	43898	GET
A050	41040	41 A6	42561	NEW

Die Startadressen der Basic-Routinen werden in der Interpreterschleife auf den Stapel gelegt und durch RTS in der CHRGET-Routine aufgerufen. Durch das RTS springt der Interpreter auf die Startadresse + 1. Die Routinen beginnen also erst bei der um 1 erhöhten Adresse. So beginnt z.B. die Routine „LOAD" nicht bei 57703, sondern bei 57704.

ADRESSEN DER BASIC-FUNKTIONEN

BASIC 2 hex	dez	Inhalt hex	Startadr. dez	Statement
A052	41042	39 BC	48185	SGN
A054	41044	CC BC	48332	INT
A056	41046	58 BC	48216	ABS
A058	41048	10 03	784	USR
A05A	41050	7D B3	45949	FRE
A05C	41052	9E B3	45982	POS
A05E	41054	71 BF	49009	SQR
A060	41056	97 E0	57495	RND
A062	41058	EA B9	47594	LOG
A064	41060	ED BF	49133	EXP
A066	41062	64 E2	57956	COS
A068	41064	6B E2	57963	SIN
A06A	41066	B4 E2	58036	TAN
A06C	41068	0E E3	58126	ATN
A06E	41070	0D B8	47117	PEEK
A070	41072	7C B7	46972	LEN
A072	41074	65 B4	46181	STR$
A074	41076	AD B7	47021	VAL
A076	41078	8B B7	46987	ASC
A078	41080	EC B6	46828	CHR$
A07A	41082	00 B7	46848	LEFT$
A07C	41084	2C B7	46892	RIGHT$
A07E	41086	37 B7	46903	MID$

Diese Basic-Adressen werden in der Ausdrucksauswertungsroutine in die Adressen 85, 86 übertragen. In 84 steht der Code 76 ($4C = JMP). Der jeweilige Aufruf erfolgt dann ebenfalls wieder in der Ausdrucksauswertung durch JSR 84.

TABELLE DER OPERATOREN

TABELL.ADR hex	dez	PF ADR-1 hex	PF ADR-1 dez	OPER.
A080	41088	79 69 B8	121 47209	+
A083	41091	79 52 B8	121 47186	-
A086	41094	7B 2A BA	123 47658	*
A089	41097	7B 11 BB	123 47889	/
A08C	41100	7F 7A BF	127 49018	ß
A08F	41103	50 E8 AF	80 45032	AND
A092	41106	46 E5 AF	70 45029	OR
A095	41109	7D B3 BF	125 49075	"+/-"
A098	41112	5A D3 AE	90 44755	NOT
A09B	41115	64 15 B0	100 45077	")/("

In dieser Tabelle stehen die Vektoren für die Auswertung von Operationen mit den Operatoren. Die Vektoradressen werden durch eine Routine auf den Stapel gelegt und durch RTS aufgerufen. Somit beginnt die entsprechende Funktionsroutine an die nächst

höhere Adresse. Vor jeder Operatoradresse steht jeweils ein Bytewert. Dieser legt die Priorität der Operatoren untereinander fest und wird damit als Prioritätsflag (PF) behandelt.

TABELLE DER BASIC-STATEMENTS

Die Statementtabelle beginnt beim VC-64 an der Adresse 41118 ($A09E). Jeder Buchstabe eines Befehlswortes ist im ASC64-Code abgelegt. Der letzte Buchstabe wird mit einem Additionswert von 128 dargestellt. Das heißt, daß beim letzten Buchstaben eines Wortes das Bit 7 gesetzt ist. Die Worttabelle ist zuständig für die Kodierung und für LIST der Basic-Statements. Durch das Setzen des höchstwertigsten Bits unterscheidet der Interpreter die Befehlsworte untereinander. Das letzte Byte der Tabelle besteht aus dem Wert Null und kennzeichnet so das Ende der Worttabelle.

Ausschnitt aus der Worttabelle:

BASIC 2 hex	dez	ASCBM-Codes dezimal	Statement
A09E	41118	69 78 196	END
A0A1	41121	70 79 210	FOR
		:	
A197	41367	77 73 68 164	MID$
A19B	41371	71 207 0	GO

Wir haben hier nicht alle Adressen und deren Statements aufgeführt, da das zu weit führen würde. Falls Interesse besteht, festzustellen, welcher Buchstabe zu welcher Adresse gehört, kann man das folgende Basic-Programm auf den Bildschirm listen.

```
10  I = 0 : X = 41118
20  A$ = CHR$ (PEEK (X+I) )
30  IF ASC(A$)= 0 THEN END
40  PRINT X+I "="+A$; : I = I+1
50  IF ASC(A$) > 128 THEN PRINT
60  GOTO20
```

Hinter der Befehlsworttabelle beginnt die Worttabelle der Fehlermeldungen, die ebenso wie die Statements abgespeichert sind.

WORTTABELLE DER FEHLERMELDUNGEN

HEX	DEZ	INHALT	MELDUNG
$A19E	41374	84 79 79 32	too->
$A1A2	41378	77 65 78 89 32	many->
$A1A7	41383	70 73 76 69 211	fileS
		:	
		:	
$A324	41764	76 79 65 196	loaD

ADRESSENTABELLE DER FEHLERMELDUNGEN

ROM-ADRESSE hex	ROM-ADRESSE dez	INHALT hex	ADRESSE dez	FEHLERMELDUNG
A328	41768	9E A1	41374	TOO MANY FILES
A32A	41770	AC A1	41388	FILE OPEN
A32C	41772	B5 A1	41397	FILE NOT OPEN
A32E	41774	C2 A1	41410	FILE NOT FOUND
		:	:	
A360	41824	24 A3	41764	LOAD
A362	41826	83 A3	41859	BREAK

INTERPRETERMELDUNGEN

ROM-ADRESSE hex	ROM-ADRESSE dez	INHALT ASC64-Code
A364	41828	chr$(13)+"ok"+chr$(13)+chr$(0)
A369	41833	chr$(32)+chr$(32)+"error"+chr$(0)
A371	41841	chr$(32)+"in"+chr$(32)+chr$(0)
A376	41846	chr$(13)+chr$(10)+"ready."+chr$(13)+chr$(10)+chr$(0)
A381	41857	chr$(13)+chr$(10)+"break"+chr$(0)+chr$(160)

Ab der Adresse 41866 ($A38A) beginnen die Routinen des Basic-Interpreters.

STARTADRESSEN BASIC VC-64 hex	STARTADRESSEN BASIC VC-64 dez	BESCHREIBUNG
A38A	41866	Stapel-Testroutine für "FOR" und "GOSUB". Der Aufruf erfolgt durch "NEXT" oder "RETURN".
A3B8	41912	Blockverschieberoutine: Diese Routine dient zum Einbau von Programmzeilen und zum Verschieben von Variablenfeldern. Diese Routine kann ganz allgemein zur Verschiebung eines Speicherbereichs benützt werden. Allerdings benötigt diese Routine folgende ZP-Adressen: 95/96 = ADL/ADH alter Blockanfang 90/91 = ADL/ADH Altes Blockende + 1 88/89 = ADL/ADH neues Blockende + 1 in die die entsprechenden Speicherbereiche eingesetzt werden müssen. Am Anfang der Routine steht ein JSR zu einer weiteren Routine, die auf freien Platz prüft. Da man das bei eigenem Programm nicht benötigt, erfolgt der Einsprung der Verschieberoutine bei 41919.

STARTADRESSEN BASIC VC-64 hex dez	BESCHREIBUNG
A3FB 41979	Testroutine für Platz am Stapel: Wenn der Test negativ verläuft wird die Fehlermeldung "OUT OF MEMORY ERROR" ausgegeben.
A408 41992	Testroutine für Platz im Speicher: Diese Routine überprüft die Zeigeradresse auf den Stringbereich, ob dieser mit anderen Zeigern kollidiert. Falls ja, erfolgt ein JSR zur GARBAGE COLLECT. Wenn auch das nicht hilft, kommt "OUT OF MEMORY ERROR".
A437 42039	Routine zur Ausgabe von Fehlermeldungen.
A474 42100	Ab hier Ausdruck von READY. Danach befindet sich der Rechner im Direktmodus (Basic-Warmstart).
A49C 42140	Routine zum Löschen und einfügen von Programmzeilen.
A4A9 42153	Ab hier Zeile löschen.
A4ED 42221	Ab hier Zeile einfügen.
A533 42291	Routine zur Berechnung der Linkzeiger im Basicprogramm
A560 42336	Routine zur Übernahme von Zeichen aus dem Programmtext in den Eingabepuffer: Basic 2 wartet zuerst ein Zeichen ab und schreibt es dann in Eingabepuffer.
A579 42361	Routine zur Umwandlung des Basic-Textes im Eingabepuffer in komprimierten Interpreter-Code.
A613 42515	Routine zur Suche der Programmzeilen. Für Eingaben ist die Zeilennummer im Adreßformat in 20,21 abgelegt. Für die Ausgabe wird der Zeiger in 95,96 benötigt, der entweder die Startadresse des Programms oder Adresse der nächsten Zeile enthält.
A642 42562	Basic-Routine NEW
A65E 42590	Basic-Routine CLR
A68E 42638	Routine initialisiert den Programmzeiger auf die Startadresse des Programms. D.h., der Programmzeiger 122,123 in der CHRGET-Routine wird mit dem Wert von 43,44 geladen.

STARTADRESSEN BASIC VC-64 hex dez	BESCHREIBUNG
A69C 42652	Basic-Routine LIST: Dabei werden sämtliche Parameter des komprimierten Basic-Textes getestet.
A742 42818	Basic-Routine FOR
A7AE 42926	Routine für Basic-Warmstart. Dabei handelt es sich um die sogenannte Interpreter-schleife die den Aufruf der einzelnen Statements veranlaßt. Es wird geprüft, ob die Stoptaste gedrückt wurde, ob sich der Rechner im Direktmodus befindet und die Zeichen zwischen den Befehlen.
A81D 43037	Basic-Routine RESTORE
A82F 43055	Basic-Routine STOP: Der Einsprung für END erfolgt in Basic 2 bei 43057
A857 43095	Basic-Routine CONT
A871 43121	Basic-Routine RUN
A883 43139	Basic-Routine GOSUB: Dabei wird auch der Stack auf freien Platz überprüft. Wenn kein Platz frei ist, wird die Fehlermeldung OUT OF MEMORY ERROR ausgegeben.
A8A0 43168	Basic-Routine GOTO: Falls die Zeile in die gesprungen werden soll, nicht vorhanden ist, erfolgt die Fehlermeldung UNDEF'D STATEMENT.
A8D2 43218	Basic-Routine RETURN
A8F8 43256	Basic-Routine DATA
A906 43270	Routine zur Bestimmung des Zählers beim Programmzeiger bis zum nächsten Trennzeichen
A928 43304	Basic-Routine IF
A93B 43323	Basic-Routine REM
A94B 43339	Basic-Routine ON
A96B 43371	Routine zur Umwandlung einer Zeilennummer im Stringformat in das Adreßformat.
A9A5 43429	Basic-Routine LET: Hier wird einer Variablen ein Wert zugewiesen. Z.B. A$ = B$ Diese Routine ist so ausgelegt, daß das Statement LET nicht unbedingt benötigt wird.

STARTADRESSEN BASIC VC-64 hex dez	BESCHREIBUNG
AA80 43648	Basic-Routine PRINT#
AA86 43654	Basic-Routine CMD
AAA0 43680	Basic-Routine PRINT: In dieser Routine werden auch die möglichen Parameter für PRINT (z.B. SPC, TAB, ; usw.) behandelt.
AB1E 43806	Routine druckt einen String.
AB3B 43835	Routine druckt ein Zeichen (aber auch Einsprünge für SPACE und CRSR).
AB4D 43853	Routine gibt entsprechende Fehlermeldung bei der Verwendung von Eingabestatements (INPUT, GET und READ) aus.
AB7B 43899	Basic-Routine GET
ABA5 43941	Basic-Routine INPUT#
ABBF 43967	Basic-Routine INPUT
AC06 44038	Basic-Routine READ
ACFC 44284	Worttabelle der Fehlermeldungen bei INPUT: ?EXTRA IGNORED und ?REDO FROM START
AD1D 44317	Basic-Routine NEXT
AD8A 44426	FRMEVL (= formula evaluater): Formelauswertungsroutine. Diese Routine ist mit fast 1 K ziemlich umfangreich und mit einigen Raffinessen ausgestattet. Deshalb werden hier noch einige Unteradressen angegeben
AD8D 44429	Test auf Numerische- oder Stringvariable
ADB4 44468	Operatormaske in 77 initialisieren
AE20 44576	Adr.Operatorenroutin. u. Operanden auf Stack
AE83 44675	Auswertung des nächsten Elements im Ausdruck
AE9A 44698	Test auf Zeichen PI (Code 255)
AE9E 44702	Überträgt PI-Konstante nach FAC#1
AEA8 44712	130 73 15 218 161 = 3.14159265 = PI
AEAD 44717	Folgende Zahl nach FAC#1
AED4 44756	Routine NOT
AEF1 44785	Test auf "(" -> Klammerauswertung
AEFA 44794	Einsprung wenn Klammer
AEFD 44797	Einsprung wenn Komma
AF08 44808	Bei Fehler die Meldung SYNTAX ERROR
AF14 44820	Variablentest innerhalb Basictext

STARTADRESSEN BASIC VC-64 hex	dez	BESCHREIBUNG
AF28	44840	Variablensuche und Test auf Stringformat
AF40	44864	Test auf TI$ und Übertrag in String
AF5D	44893	Test auf Integer
AF73	44915	Test auf TI und Übertrag nach FAC#1
AF92	44946	Test auf ST und Übertrag nach FAC#1
AFA0	44960	numerische Variable nach FAC#1 übertragen
AFB1	44977	Routinen LEFT$, RIGHT$ und MID$
AFE6	45030	Routinen OR und AND (FAC#1/2-Vergleich)
B02E	45102	Stringvergleich
B07E	45182	Routine zur Verwaltung der Variablen: Bei 45185 (Basic VC-64) ist am Anfang noch die Basic-Routine DIM untergebracht.
B37D	45949	Basic-Routine FRE
B391	45969	Routine INTFLP:Umwandlung Integer/Fließkomma
B39E	45982	Basic-Routine POS
B3A6	45990	Basic-Routine DEF
B3F4	46068	Basic-Routine FN
B465	46181	Basic-Routine STR$
B475	46197	Routine zur Verwaltung der Strings: In diesem Programmbereich werden die Strings so manipuliert, wie sie dann im oberen RAM-bereich während der Basicprogrammausführung behandelt werden. Es werden sämtliche Zeiger entsprechend gesetzt und verwaltet.
B526	46374	Rotuine GARBAGE COLLECT: Diese Routine gehört noch zur Stringverwaltung und ist für die Beseitigung von Stringleichen zuständig (Stringmüllbeseitigung).
B63D	46653	Basic-Routine zum Zusammensetzen von Strings durch "+".
B6A3	46755	Weitere Routine zur Stringverwaltung: Prüft ob Platz am unteren Ende ist und setzt die Deskriptoren richtig.
B6EC	46828	Basic-Routine CHR$
B700	46848	Basic-Routine LEFT$
B72C	46892	Basic-Routine RIGHT$

STARTADRESSEN BASIC VC-64 hex	dez	BESCHREIBUNG
B737	46903	Basic-Routine MID$
B77C	46972	Basic-Routine LEN
B78B	46987	Basic-Routine ASC
B79B	47003	Routine GETBYT: Diese Routine holt das nächste Byte aus dem komprimierten Basictext und speichert es im Indexregister X.
B7AD	47021	Basic-Routine VAL
B7EB	47083	Aufruftabelle zur Auswertung der Parameter für POKE und WAIT: JSR FRMEVL -> Formelauswertung JSR GETADR -> FAC#1-Wert nach 20,21 JSR CHKCOM -> Prüfung auf Komma JMP GETBYT -> Byte nach X
B7F7	47095	Routine GETADR: Diese Routine holt den Inhalt von FAC#1, wandelt ihn in eine Adresse um und speichert diese in 20,21 als Zwischenspeicher ab.
B80D	47117	Basic-Routine PEEK
B824	47140	Basic-Routine POKE
B82D	47149	Basic-Routine WAIT
B849	47177	arithmetische Routine FAC#1 + .5: Diese Routine addiert zum Inhalt des FAC#1 .5. Sie wird immer aufgerufen, wenn gerundet wird.
B853	47187	Arithmetische Routine FAC#2 minus FAC#1
B86A	47210	Arithmetische Routine FAC#1 plus FAC#2
B983	47491	Routine zum Rechtsverschieben eines Registers. Sie wird von verschiedenen arithmetischen Routinen verwendet, z.B. bei der Multiplikation.

STARTADRESSEN BASIC VC-64 hex dez	BESCHREIBUNG
B9BC 47548	Beginn der arithmetischen Konstanten als Tabelle in Fließkommaformat (für LOG): $81 00 00 00 00 = 1 $03 = 3 = Polynomgrad $7F 5E 56 CB 79 = .434255942 = 1.Koeffizient $80 13 9B 0B 64 = .576584541 = 2.Koeffizient $80 76 38 93 16 = .961800759 = 3.Koeffizient $82 38 AA 3B 20 = 2.88539007 = 4.Koeffizient $80 35 04 F3 34 = .707106781 = Wurzel aus .5 $81 35 04 F3 34 = 1.41421356 = Wurzel aus 2 $80 80 00 00 00 = -.5 $80 31 72 17 F8 = .693147181 = natürlicher Logarithmus aus 2
B9EA 47594	Basic-Routine LOG
BA2B 47659	arithmetische Routine FAC#1 * FAC#2
BAE2 47842	arithmetische Routine FAC#1 * 10
BAFE 47870	arithmetische Routine FAC#1 / 10
BB12 47890	arithmetische Routine FAC#2 / FAC#1
BBA2 48034	Routine zur Übertragung einer Konstanten in Fließkomma Akkumulator #1 (FAC#1)
BBC7 48071	Routine zur Übertragung von FAC#1 in den Speicherbereich, in dem normalerweise die FOR...NEXT-Variablen abgespeichert sind.
BBFC 48124	Routine zur Übertragung FAC#2 nach FAC#1
BC0C 48140	Routine zur Übertragung FAC#1 nach FAC#2
BC1B 48155	Routine zum Runden von FAC#1
BC2B 48171	Routine zum Suchen des Vorzeichens von FAC#1 Ergebnis: Akkuinhalt = $00 -> Wert ist 0 Akkuinhalt = $01 -> Wert ist + Akkuinhalt = $FF -> Wert ist -
BC39 48185	Basic-Routine SGN
BC58 48216	Basic-Routine ABS
BC5B 48219	Routine zum Vergleich zwischen einer Konstanten (Adreßwert im Akku und Y-Register) mit dem Wert im FAC#1.

STARTADRESSEN
BASIC VC-64
hex dez

BESCHREIBUNG

BC9B 48283

Routine zur Umwandlung einer Fließkommazahl ins Integerformat.

BCCC 48332

Basic-Routine INT

BCF3 48371

Routine zur Umwandlung eines Zahlenstrings in eine Fließkommazahl im FAC#1.

BDB3 48563

Konstantentabelle:
99 999 999.9
999 999 999.75
1 000 000 000 = 1E+09

BDC2 48578

Routine zur Ausgabe von "IN" und Zeilennummer bei Fehlermeldungen in Basic.

BDCD 48589

Routine zur Umwandlung einer Fließkommazahl im FAC#1 ins Stringformat

BF11 48913

Kostantentabelle für Stringumwandlungen
$80 00 00 00 00 = .5 = f. Rundung und Wurzel
$FA 0A 1F 00 = -100000000 Dezimale Stellen-
$00 98 96 80 = 10000000 werte im Ineger-
$FF F0 BD C0 = -1000000 format zur Berech-
$00 01 86 A0 = 100000 nung der Ziffern
$FF FF D8 F0 = -10000 in der Dezimaldar-
$00 00 03 E8 = 1000 stellung
$FF FF FF 9C = -100
$00 00 00 0A = 10
$FF FF FF FF = -1

BF3A 48954

Konstantentabelle zur Berechnung der Ziffern von TI$ mit entsprechenden Stellenwerten im Integerformat
$FF DF 0A 80 = -2 160 001
$00 03 4B C0 = 216 000 (= 1 Stunde)
$FF FF 73 60 = -36 000
$00 00 0E 10 = 3 600 (= 1 Minute)
$FF FF FD A8 = -600
$00 00 00 3C = 60 (= 1 Sekunde)

BF71 49009

Basic-Routine SQR

BF78 49016

Arithmetische Routine FAC#2 hoch FAC#1

STARTADRESSEN BASIC VC-64 hex dez	BESCHREIBUNG
BFBF 49087	Konstantentabelle im Fließkommaformat für die Berechnung der Basic-Routine EXP $81 38 AA 3B 29 = 1.44269504 = 1/LN2 $07 = Polynomgrad $71 34 58 3E 56 = 2.14987637E-5 = 1.Koeffiz. $74 16 7E B3 1B = 1.43523140E-4 = 2.Koeffiz. $77 2F EE E3 85 = 1.34226348E-3 = 3.Koeffiz. $7A 1D 84 1C 2A = 9.61401701E-3 = 4.Koeffiz. $7C 63 59 58 0A = 5.55051269E-2 = 5.Koeffiz. $7E 75 FD E7 C6 = 2.40226385E-1 = 6.Koeffiz. $80 31 72 18 10 = 0.693147186 = LN2 = 7.Koef $81 00 00 00 00 = 1 = 8.Koeffizient
BFED 49133	Basic-Routine EXP
E043 57411	Routine zur Auswertung der Polynome. Der Polynomgrad wird dabei als Zähler verwendet.
E08D 57485	2 Konstanten für die Berechnung von Zufallszahlen. $98 35 44 7A 00 = 11 879 546.4 = RND-Multiplikationskonstante $68 28 B1 46 00 = 3.927 677 78 E-8 = RND-Additionskonstante
E097 57495	Basic-Routine RND
E0F9 57593	Fehlerausgabe bei Peripheriebefehlen
E12A 57642	Basic-Routine SYS
E156 57686	Basic-Routine SAVE
E165 57701	Basic-Routine VERIFY
E168 57704	Basic-Routine LOAD
E1BE 57790	Basic-Routine OPEN
E1C7 57799	Basic-Routine CLOSE
E1D4 57812	Routine zur Übergabe der LOAD- und SAVE-Parameter
E219 57881	Routine zur Übergabe der OPEN-Parameter
E264 57956	Basic-Routine COS
E26B 57963	Basic-Routine SIN

STARTADRESSEN BASIC VC-64 hex dez	BESCHREIBUNG
E2B4 58036	Basic-Routine TAN
E2E0 58080	Konstantentabelle im Fließkommaformat zur Berechnung von COS und SIN $81 49 0F DA A2 = 1.57079633 = PI/2 $83 49 0F DA A2 = 6.28318531 = PI*2 $7F 00 00 00 00 = 0.25 $05 = Polynomgrad $84 E6 1A 2D 1B = -14.3813907 = 1.Koeffiz. $86 28 07 FB F8 = 42.0077971 = 2.Koeffiz. $87 99 68 89 01 = -76.7041703 = 3.Koeffiz. $87 23 35 DF E1 = 81.6052237 = 4.Koeffiz. $86 A5 5D E7 28 = -41.3417021 = 5.Koeffiz. $83 49 0F DA A2 = 6.28318531 = 6.Koeff.=PI*2
E30E 58126	Basic-Routine ATN
E33E 58174	Konstantentabelle im Fließkommaformat für die Berechnung der ATN-Funktion $0B = Polynomgrad $76 B3 83 BD D3 = -6.84793912E-4/ 1.Koeffiz. $79 1E F4 A6 F5 = 4.85094216E-3/ 2.Koeffiz. $7B 83 FC B0 10 = -0.0161117018 / 3.Koeffiz. $7C 0C 1F 67 CA = 0.034209638 / 4.Koeffiz. $7C DE 53 CB C1 = -0.0542791328 / 5.Koeffiz. $7D 14 64 70 4C = 0.0724571965 / 6.Koeffiz. $7D B7 EA 51 7A = -0.0898023954 / 7.Koeffiz. $7D 63 30 88 7E = 0.110932413 / 8.Koeffiz. $7E 92 44 99 3A = -0.142839808 / 9.Koeffiz. $7E 4C CC 91 C7 = 0.19999912 /10.Koeffiz. $7F AA AA AA 13 = -0.333333316 /11.Koeffiz. $81 00 00 00 00 = 1 /12.Koeffiz.
Ab hier beginnt nun das eigentliche Betriebssystem des VC-64	
E37B 58235	Basic-Warmstartroutine nach NMI-Auslösung Der Rechner befindet sich danch im READY-Modus.
E394 58260	Basic-Kaltstartroutine durch Einschaltreset
E3A2 58274	CHRGET-ROM-Routine wird durch Einschaltreset in die Zeropage übertragen.
E3BA 58289	Initialisierungswert für RND $80 4F C7 52 58 = .811636157

STARTADRESSEN BASIC VC-64 hex dez	BESCHREIBUNG
E3BF 58303	Fortsetzung der Kaltstartroutine.
E447 58439	Tabelle der RAM-Vektoren 768 - 779
E453 58451	Routine überträgt die Werte der Tabelle ab 58439 nach 768 und folgende.
E45F 58463	ASC64-Tabelle der Systemmeldungen: "BASIC BYTES FREE" "COMMODORE 64 BASIC V2" "64 K RAM SYSTEM"
E4AD 58541	Basic-Routine zum Setzen des Ausgabegerätes
E4DA 58586	Routine legt Hintergrundfarbe fest.
E4E0 58592	Routine testet Commodore-Taste.
E4EC 58604	Konstanttabelle für Timer im CIA 1 (wird von RS232-Anschluß für Baud Rate) benötigt.
E500 58624	Routine holt Basisadresse vom CIA in X- und Y-Register des Prozessors.
E505 58629	Routine holt Anzahl Zeilen (25) ins Y-Register und die Spaltenanzahl (40) ins X-Register.
E50A 58634	Routine zum Setzen oder holen der Cursorposition.
E518 58648	Bildschirm-Reset
E5A0 58784	Videocontroller-Reset
E5B4 58804	Routine zum Abbau des Tastaturpuffers. Das erste Zeichen in 631 wird in Y-Register übernommen und die restlichen Zeichen um ein Byte nach unten geschoben.
E5CA 58826	Routine wartet auf Tastendruck, schiebt den Cursor bei Tastendruck weiter und wird durch die RETURN-Taste beendet. In dieser Routine wird aber auch die Ausführung von SHIFT/RUN veranlaßt.
E602 58882	Routine führt Funktion RETURN-Taste aus.

STARTADRESSEN BASIC VC-64 hex	dez	BESCHREIBUNG
E632	58930	Diese Routine ist zuständig für die Übernahme eines Zeichens vom Bildschirm oder der Tastatur. D.h., wird eine Taste gedrückt, dann wird das jeweilige Zeichen auf dem Schirm ausgegeben und dann erst übernommen.
E684	59012	Routine schaltet "Gänsefußmodus" um. Flag in 212 ist 0 oder 1.
E691	59025	Druckt Zeichen auf Schirm und bewegt den Cursor weiter (Abschluß der Druckausgabe).
E6B6	59062	Routine markiert Zeile als Einfachzeile
E701	59137	Routine ist zuständig für Rückwärtsschritt in die vorhergehende Zeile, wenn der Cursor die Anfangsspalte erreicht hat.
E716	59158	Beginn der Routine zur Ausgabe eines Zeichens auf dem Bildschirm. Sie enthält die Einsprungadressen folgender Steuerzeichen.
		Ungeshiftete Zeichen:
E72A	59178	Carriage Return, CHR$(13)
E731	59185	Normale ASC64-Codes, CHR$(32)-(127)
E74C	59212	Delete, CHR$(20)
E785	59269	Revers chr$(18)
E78B	59275	Home, CHR$(19)
E792	59282	Cursor nach rechts, CHR$(29)
E7AD	59309	Cursor nach unten, CHR$(17)
E7D4	59348	Geshiftete Zeichen:
E7D6	59350	Pi CHR$(255)
E7E3	59363	Shift-Return, CHR$(141)
E7EE	59374	Insert, CHR$(148)
E832	59442	Cursor nach oben, CHR$(145)
E84C	59468	Reverse off, CHR$(146)
E854	59476	Cursor nach links, CHR$(157)
E86A	59498	Clear, CHR$(147)
E8CB	59595	Routine vergleicht Farbtabelle und legt den Farbencode fest.
E8DA	59610	Tabelle der 16 Farbencodes
E8EA	59626	Routine SCROLL mit Abfrage der Control-Taste
E965	59749	Routine zum Einfügen einer Fortsetzungszeile in Basic.

STARTADRESSEN BASIC VC-64 hex dez	BESCHREIBUNG
E9C8 59848	Hilfroutinene für SCROLL
EA31 59953	Interruptroutine IRQ. Diese Routine wird in der Sekunde 60 mal aufgerufen, sofern kein Programm abgearbeitet wird. Die Steuerung erfolgt durch Bildwechselimpuls vom VIC. **Zudem wird am Anfang** gleich zu der Routine gesprungen, die die Uhr weiterschaltet und die Stoptaste ab- abfrägt.
EA61 60001	Abfrage der Rekordertasten
EA7E 60030	Abschluß der Interruptroutine. Akku, Y- und X-Register werden wieder hergestellt und der Rücksprung durch RTI veranlaßt.
EA87 60039	Ab hier beginnt nun die Tastaturabfrage. Der Der Dekoderausgang der Tastaturmatrix 56320 gibt ein Bitmuster aus. Entsprechend wird dann aus einer der ASC64-Tabellen der jeweilige Code geholt.
EB48 60232	Routine testet auf die Tasten SHIFT, CONTROL und C=
EB79 60281	$EB81 = 60289 Zeiger auf Tastaturtabelle ungeshiftete Codes
EB7B 60283	$EBC2 = 60354 Zeiger auf Tastaturtabelle geshiftete Codes
EB7D 60285	$EC03 = 60419 Zeiger auf Tastaturtabelle mit C=-gedrückte Codes
EB7F 60287	$EC78 = 60536 Zeiger auf Tastaturtabelle mit CONTROL-gedrückte Codes
EB81 60289	3 Tastaturtabellen mit je 64 Codes für die Auswahl mit nicht geshiftet, geshiftet und mit Commodoretaste
EC44 60484	Schaltet nach SHIFT+C= jeweils in den Text- oder Graphikmodus um oder sperrt bzw. gibt SHIFT+C= frei.
EC78 60536	Tastaturtabelle mit 64 Codes für die Auswahl mit der CONTROL-Taste

STARTADRESSEN

BASIC VC-64 hex	dez	BESCHREIBUNG
ECB9	60601	Tabelle der Konstanten für den Videocontroller VIC
ECE7	60647	Worttabelle LOAD + RUN nach Drücken der Tasten SHIFT/RUN
ECF0	60656	Tabelle der niederwertigen Adreßbytes für die Anfangsspalten jeder Bildschirmzeilen
		Ab hier beginnen nun die Routinen für die Behandlung der Bussysteme
ED09	60681	Routine zum Senden von TALK und LISTEN
ED09	60681	Einsprungadresse für TALK
ED0C	60684	Einsprungadresse für LISTEN
ED40	60736	Routine sendet ein Zeichen zum Bus.
EDAD	60845	Routine setzt Status auf DEVICE NOT PRESENT oder Zeitüberschreitung
EDB9	60857	Routine bringt Sekundäradresse in Bytepuffer (149) zur Ausgabe eines Zeichens und setzt die ATN-Leitung auf High um den Kontroll modus zu beenden.
EDC7	60871	Routine übergibt Sekundäradresse nach TALK
EDDD	60893	Routine frägt Ausgabeflag in 148 ab. Je nach dem welchen Inhalt dieses Flag hat, wird ein Byte ausgegeben oder nur in den Bytepuffer in 149 gebracht.
EDEF	60911	Routine sendet UNTALK
EDFE	60926	Routine sendet UNLISTEN
EE13	60947	Routine holt ein Zeichen vom Bus.
EE85	61061	Hilfsroutinen für die Busbehandlung. (Takt ein/aus, Bit ausgeben usw.)
EEBB	61115	Beginn der Ausgaberoutine für RS232-Schnittstelle
EF4A	61258	Routine berechnet die Anzahl der RS232-Ausgabe-Bits.
EFE1	61409	Routine gibt auf RS232-Bus aus.

STARTADRESSEN BASIC VC-64 hex	dez	BESCHREIBUNG
F014	61460	Routine übergibt RS232-Daten an Puffer
F04D	61517	Routine legt RS232 Eingabekanal fest.
F086	61574	Routine holt ein RS232-Byte vom Puffer.
F0A4	61604	Routine wartet auf RS232-Bit.
F0BD	61629	Worttabelle der Systemmeldungen: I/O ERROR # / SEARCHING / FOR PRESS PLAY ON TAPE PRESS RECORD & PLAY ON TAPE LOADING / SAVING / VERIFYING / FOUND / OK
F12B	61739	Routine gibt Systemmeldungen aus.
F13E	61758	Routine GETIN: Das ist der Eintritt aus der Sprungtabelle für GET. Es wird ein Zeichen vom Eingabekanal des RS232 übernommen, sofern es sich um RS232 handelt. Ansonsten Sprung zur Routine BASIN.
F157	61783	Routine BASIN: Hier wird ein Zeichen entweder von der Tastatur, Bildschirm, seriellen Bus oder Band geholt.
F1CA	61898	Routine BSOUT: Ein Zeichen wird ausgegeben.
F208	61690	Routine gibt auf RS232 aus.
F20E	61966	Routine CHKIN: Eingabegerät festlegen.
F250	62032	Routine CHKOUT: Ausgabegerät festlegen.
F291	62097	Routine CLOSE: Das ist der Eintritt aus der Sprungtabelle für CLOSE.
F2AB	62123	Routine schließt RS232-Kanal (Äquivalent zu CLOSE).
F2C8	62152	Routine schließt Bandkanal.
F30F	62223	Routine sucht logische Filenummer und vergleicht mit Eintrag in Tabelle.
F31F	62239	Routine legt logische, Primär- und Sekundär-Adresse fest.

STARTADRESSEN

BASIC VC-64 hex	dez	BESCHREIBUNG
F32F	62255	Routine CLALL: Es werden sämtliche Ein- und Ausgabekanäle geschlossen.
F333	62295	Routine CLRCH (clear channel) schließt alle geöffneten Kanäle (Unterroutine von CLALL)
F34A	62282	Routine OPEN: Das ist der Eintritt aus der Sprungtabelle für OPEN.
F3D5	62421	Routine öffnet auf seriellen Bus logisches File.
F409	62473	Routine Open für RS232-Bus.
F49E	62622	Routine LOAD: Das ist der Eintritt aus der Sprungtabelle für LOAD.
F4B8	62648	Routine Load für RS232-Bus.
F5AF	62895	Routine gibt Meldung "SEARCHING" aus.
F5D2	62930	Routine gibt Meldung "LOADING" oder "VERIFYING" aus.
F5DD	62941	Routine SAVE: Das ist der Eintritt aus der Sprungtabelle für SAVE.
F5FA	62970	Routine Save für RS232-Bus.
F68F	63119	Routine gibt Meldung "SAVING" aus.
F69B	63131	Routine UDTIM: Diese Routine stellt die Uhr weiter und frägt die Stoptaste ab.
F6FB	63227	Routine gibt Meldungen des Betriebssystems zu den I/O-Routinen aus. Bei Auftreten eines Fehlers erscheint "I/O ERROR #"
Ab hier beginnen die Routinen zur Bedienung des Kassettenrekorders		
F72C	63276	Routine sucht File-Header (Dateikopf)
F76A	63338	Routine schreibt File-Header
F7D7	63447	Hilfsroutine holt Start- und Endadresse aus File-Header.
F7EA	63466	Routine such Namen des Fileheaders

STARTADRESSEN BASIC VC-64 hex	dez	BESCHREIBUNG
F80D	63501	Hilfsroutine erhöht Pufferzeiger
F817	63511	Routine frägt PLAY-Taste ab.
F841	63553	Routine liest Block vom Band in den Puffer.
F86B	63595	Routine schreibt vom Puffer Block auf Band.
F8BE	63678	Routine testet Stoptaste und IRQ-Vektor.
F8E2	63714	Routine setzt CIA-Timer #1 auf neuen Wert und synchronisiert Timer #2 mit 0.
F92C	63788	Routine liest Bits vom Band: Interruptaufruf wenn Tabellenoffset = 14.
FA60	64096	Routine speichert vom Band angenommene Bytes für Fehlertest.
FB97	64407	Routine setzt Bitzähler für serielle Ausgabe fest
FBA6	64422	Routine speichert ein Bit auf Band
FBCD	64461	Interruptroutine beim Bandschreiben für Rekordermotor
FCE2	64738	Routine RESET: Hier liegt der Eintrittspunkt der Einschaltroutine, die durch den Interruptvektor des Prozessors gegeben ist. Am Anfang dieser Routine wird gleich geprüft, ob ein Modul am Rechner steckt. Wenn ja, dann folgt der Sprung zur Moduladresse. Wenn nicht dann wird der IRQ-Interrupt vorbereitet, der Basic-RAM-Bereich initialisiert, sämtliche Softwarevektoren gesetzt und der Bildschirm in Ausgangsposition gebracht. Anschließend springt der Rechner über den indirekten Zeiger in $A000 (40960) in eine Routine, die die Einschaltmeldung ausgibt und verzweigt dann zur READY-Routine.
FD02	64770	Subroutine vergleicht Modulkennung "CBM80" mit eigener Tabelle.
FD10	64784	ASC64-Tabelle CBM80 (Modulkennung)
FD15	64789	Routine holt verschiedene-Vektoren aus der nachfolgenden Tabelle und setzt sie in Page 3 ab.

STARTADRESSEN BASIC VC-64 hex dez	BESCHREIBUNG
FD30 64816	Vektorentabelle für Interrupt und Ein/Ausgabe.
FD50 64848	Subroutine initialisiert Arbeitsspeicher.
FD9B 64923	Tabelle der IRQ-Vektoren
FDA3 64931	Routine Interruptvorbereitung: Es werden die Register der CIA-Bausteine und der Videocontroller initialisiert.
FDF9 65017	Verschiedene Hilfsroutinen für I/O-Behandlung. Unter anderem befinden sich hier auch die Eintrittspunkte aus der Hauptsprungliste für Status initialisieren, Flag für Zeitüberschreitung setzen.
FE25 65061	Routine MEMTOP (Lesen/Setzen der Speicherendadresse): Hier ist der Eintrittspunkt aus der Hauptsprungliste.
FE34 65076	Routine MEMBOT (Lesen/Setzen der Speicheranfangsadresse): Hier ist der Eintrittspunkt der Hauptsprungliste.
FE43 65091	Routine NMI: Hier liegt der Eintrittspunkt der NMI-Routine, die durch den Interruptvektor des Prozessors gegeben wird. Es wird wiederum auf Modulkennung geprüft und die Standardvektoren gesetzt. Außerdem erfolgt ein Test auf RS232-Anschluß. Wenn kein RS232 Betrieb vorliegt verzweigt das Programm zum READY-Modus. Ansonsten wird in die NMI-Routine gesprungen, die für den RS232-Betrieb zuständig ist.
FE72 65138	Routine NMI-RS232: Ist für den RS232-Betrieb verantwortlich.
FEC2 65218	Konstantentabelle für RS232 Übertragungsraten (Baudrate, jeweils 2 Byte) 50 Baud 75 Baud 110 Baud 134.5 Baud 150 Baud 300 Baud 600 Baud 1200 Baud 1800 Baud 2400 Baud

STARTADRESSEN BASIC VC-64 hex	dez	BESCHREIBUNG
FED6	65238	Subroutine NMI-RS232 für Eingabe
FF07	65287	Subroutine NMI-RS232 für Ausgabe
FF43	65347	Interrupteinsprung aus der Bandroutine
FF48	65352	Routine IRQ: Hier liegt der Eintrittspunkt der IRQ-Routine, der durch den Interruptvektor des Prozessors gegeben ist. Nach der Rettung der Register wird das BRK-Flag abgefragt und je nach Zustand über die Vektoren in Page 3 verzweigt. Diese sind allerdings nach dem Einschalten so initialisiert, daß in beiden Fällen in den READY-Modus gesprungen wird.
FF5B	65371	Routine VIC-Reset
FF6E	65390	Routine setzt Timer im CIA für Interrupt

Ab hier beginnt die Hauptsprungliste des VC-64. Maschinenprogramme sollten, sofern sie mit Systemroutinen zusammenarbeiten, über diese Sprungtabelle miteinander verknüpft sein.

EINSPRUNG-ADRESSE HEX	DEZ	SPRUNGADRESSE NACH JMP HEX	DEZ		BESCHREIBUNG
FF81	65409	FF5B	65371	JMP VICRST	Video Reset
FF84	65412	FDA3	64931	JMP IOINIT	CIA initialisieren + VICRST
FF87	65415	FD50	64848	JMP CHKRAM	RAM initialisieren u. testen
FF8A	65418	FD15	64789	JMP RESTOR	I/O-Vektoren voreinstellen
FF8D	65421	FD1A	64794	JMP VECTOR	I/O-Vektoren initialisieren
FF90	65424	FE18	65048	JMP SETMSG	Subroutine für Statusmeldung
FF93	65427	EDB9	60857	JMP SECOND	Ausgabe der Sekundäradresse
FF96	65430	EDC7	60871	JMP TKSA	Sek.Adr.nach TALK-Befehl senden
FF99	65433	FE25	65061	JMP MEMTOP	Lesen/Setzen der Speicherendadr.
FF9C	65436	FE34	65076	JMP MEMBOT	Lesen/Setz.d.Speicheranfangsadr.
FF9F	65439	EA87	60039	JMP SCNKEY	Tastatur abfragen
FFA2	65442	FE21	65057	JMP SETTMO	Zeitüberschreitungsflag setzen
FFA5	65445	EE13	60947	JMP ACPTR	Byteübergabe **vom** seriellen Bus
FFA8	65448	EDDD	60893	JMP CIOUT	Byteübergabe **zum** seriellen Bus
FFAB	65451	EDEF	60911	JMP UNTLK	UNTALK zum seriellen Bus
FFAE	65454	EDFE	60926	JMP UNLSN	UNLISTEN zum seriellen Bus
FFB1	65457	ED0C	60684	JMP LISTEN	LISTEN zum serellen Bus
FFB4	65460	ED09	60681	JMP TALK	TALK zum seriellen Bus
FFB7	65463	FE07	65031	JMP READST	Statusbyte ST lesen

EINSPRUNG-ADRESSE HEX	DEZ	SPRUNGADRESSE NACH JMP HEX	DEZ		BESCHREIBUNG
FFBA	65466	FE00	65024	JMP SETLFS	Log.-,Prim.- u. Sek.-Nr. setzen
FFBD	65469	FDF9	65017	JMP SETNAM	Filenamen setzen
FFC0	65472	031A	794	JMP (OPEN)	OPEN (logisches File öffnen)
FFC3	65475	031C	796	JMP (CLOSE)	CLOSE (logisches File schließen)
FFC6	65478	031E	798	JMP (CHKIN)	Eingabegerät festlegen
FFC9	65481	0320	800	JMP (CKOUT)	Ausgabegerät festlegen
FFCC	65484	0322	802	JMP (CLRCH)	Ein-/Ausgabekanal schließen
FFCF	65487	0324	804	JMP (BASIN)	Byteeingabe über Kanal
FFD2	65490	0326	806	JMP (BSOUT)	Byteausgabe über Kanal
FFD5	65493	F49E	62622	JMP LOAD	LOAD (Arbeitsspeicher laden)
FFD8	65496	F5DD	62941	JMP SAVE	SAVE (RAM abspeichern)
FFDB	65499	F6E4	63204	JMP SETTIM	Systemuhr festlegen
FFDE	65502	F6DD	63197	JMP RDTIM	Systemuhr lesen
FFE1	65505	0328	808	JMP (STOP)	Stoptaste abfragen
FFE4	65508	032A	810	JMP (GETIN)	GET (Zeichen holen)
FFE7	65511	032C	812	JMP (CLALL)	Alle Files schließen
FFEA	65514	F69B	63131	JMP (UDTIM)	Systemuhr erhöhen
FFED	65517	E505	58629	JMP SCREEN	Zeile/Spalte in Y/X-Register
FFF0	65520	E50A	58634	JMP PLOT	Lesen/Setzen der Cursorposition
FFF3	65523	E500	58624	JMP IOBASE	I/O-Basisadresse holen
FFF6	65526	ASCII-Code		RRBY	?

3 Programmbeispiele

In diesem Abschnitt werden wir nun in die Programmierung einsteigen und dabei einige Programmbeispiele kennenlernen. Dazu sind aber ein paar grundsätzliche Begriffe zu erläutern.

3.1 Begriffe und Symbole

Wir haben bisher die Maschinensprache unter dem Gesichtspunkt ihrer Arbeitsweise betrachtet. Dazu wurden auch verschiedene Hilfen beschrieben. Auch haben wir bei einigen Programmbeispielen schon die Programmadressen und Operanden symbolisch ausgedrückt. Das hat den Vorteil, daß man Programme unabhängig vom Rechnertyp ganz allgemein entwickeln bzw. beschreiben kann. Wir wollen deshalb an dieser Stelle einige Begriffe erläutern, die für einen Assemblerprogrammierer unabdingbare Voraussetzung sind, um fremde Maschinenprogramme lesen zu können und um selber viel effektvoller Programme zu schreiben. Dazu betrachten wir einmal die verschiedenen Ebenen der Programmiersprachen.

Auf der untersten Ebene findet man das sogenannte Mikroprogramm. Dies befindet sich im Prozessor selbst und stellt praktisch die logische Struktur für die Maschinensprache dar. Das Mikroprogramm besteht aus einer Anordnung digitaler Schaltungen, die auf Signale so reagieren, wie es die Boolsche Mathematik vorschreibt. Die Struktur ist dabei von den Prozessorherstellern so aufgebaut, daß damit das sogenannte Makroprogramm generiert wird. Für den Prozessor bedeutet ein Makro das Resultat einer Signalkette, die die Anordnung logischer Schaltungen dahingehend veranlaßt, daß der Signalaustritt das gewünschte Ergebnis erzeugt. In unserem Fall heißt das, der Prozessor wird gezwungen, einen Makrobefehl zu vollziehen. Ein Makrobefehl ist damit der Maschinencode eines Prozessorbefehls, der sogenannte Operationscode.

Somit kommen wir zur nächsten Ebene. Die Zusammenfassung der Makrobefehle ist der Befehlssatz des Mikroprozessors auf Anwenderebene, wenn in Maschinensprache programmiert werden soll. Da hier wiederum die unterste Struktur aus 8 Zuständen (Bits) besteht und der Zugriff nur indirekt über die hexadezimale Schreibweise erfolgt, wendet der Programmierer die nächsthöhere Ebene an. Darunter versteht man nun die Programmiersprache ASSEMBLER, in der der Maschinencode symbolisch dargestellt wird. Die Symbolik erleichtert nun dem Programmierer erheblich das Umsetzen einer Aufgabe in die Maschinenform.

Als weitere Vereinfachung für die Programmierung der Computer steht in der nächsten Ebene eine sogenannte höhere Programmiersprache zur Verfügung. In diesen höheren Programmiersprachen kann dem Computer direkt wörtlich mitgeteilt werden, was er durchführen soll. Die höhere Sprache, wie z.B. BASIC, besteht ebenso aus einem Befehlssatz. Jeder Basic-Befehl entspricht einer gezielten Anordnung von Maschinen-

codes. Man könnte auch sagen, daß ein Basic-Befehl einen Makrobefehl mehrerer zusammengehöriger Maschinenbefehle darstellt.

Zwischen dem Maschinencode und der höheren Programmiersprache steht der Assembler als Zwischencode und ist ebenfalls wiederum ein Programm, das symbolisch dargestellte Befehle in den Maschinencode umsetzt. Je komfortabler ein Assembler ist, desto leichter fällt das Programmieren. Man kann sogar sagen, daß ein guter Assembler fast den Komfort erreichen kann, wie z.B. ein Basic-Interpreter.

a) Begriff: MAKRO
Wir haben jetzt den Begriff Makro kennengelernt. Ein Makro-Befehl ist ganz allgemein ein Großbefehl, der einen definierten Satz von Mikrobefehlen zusammenfaßt. Wenn man eine Sprachenebene mit einem Befehlssatz von Mikrobefehlen darstellt, dann ist ein Makrobefehl ein Mikrobefehl der nächsthöheren Sprachenebene.

b) Begriff: SOURCE
Source heißt übersetzt Quelle. Dieser Begriff taucht oft im Zusammenhang mit Assemblerprogrammen auf. Die Entwicklung von Maschinenprogrammen erfolgt in symbolischer Schreibweise, mit Mnemoniks, Labels und Symbolen. Das so entwickelte Programm, das in dieser Form noch nicht lauffähig ist, nennt man SOURCE-Code oder Quellprogramm. Es muß erst durch das Assemblerprogramm in den reinen Maschinencode übersetzt werden.

c) Begriff: OBJEKT
Das übersetzte Programm, das nun auch lauffähig ist und nur aus den Befehlen einer Sprachenebene besteht, nennt man OBJEKT-Code. Im Fall der Maschinensprache heißt das, daß das aus dem Assembler entwickelte Maschinenprogramm das eigentliche Objekt ist.

d) Begriff: LABEL
Ein Label ist ein Kennzeichen oder symbolischer Name für den Anfang einer Befehlsgruppe. Normalerweise kennzeichnet man den Beginn eines Programmteils mit einer Zahl. In Basic ist es eine Zeilennummer, während im Maschinencode die Programmadresse diese Festlegung trifft. Bei der Entwicklung z.B. in Assemblerschreibweise, wird man statt Adressen eben Labels benutzen. Aus diesem Grunde wollen wir uns noch ein kleines Beispiel dazu anschauen, das einen Speicherbereich löscht.

```
DELETE   LDX  #LEN
         LDA  # 0
ZERO     STA    BASIS,X
         DEX
         BNE  ZERO
         RTS
```

DELETE und ZERO sind Labels und stehen als Namen stellvertretend für Adressen. Das hat den Vorteil, daß man ein Programm schreiben kann und sich dabei nicht darum kümmern muß, in welchem Speicherbereich später dann der Objektcode liegen soll. Die Labels werden aber auch benutzt, um Speicherplätze symbolisch darzustellen. Deshalb findet man sie auch hinter den Assemblerbefehlen, wie im Beispiel dargelegt.

e) Begriff: EDITOR
Ein Editorprogramm stellt die Verbindung zwischen Computer und Anwender her. Sobald eine Taste gedrückt wird, löst diese eine bestimmte Funktion aus. Wird z.B. die Taste „A" gedrückt, erscheint auf dem Bildschirm das A und der Cursor rückt um eine Stelle weiter. Unter Einbeziehung der Steuertasten können nun am Computer ganze Texte, Befehle, Graphik usw. eingegeben werden. Das Ganze nennt man dann Edition. Es gibt dabei zwei Möglichkeiten. Entweder die Edition wird über Befehle geregelt, oder erfolgt durch Cursorsteuerung ‚interaktiv' über die Tastatur. Der Editor ist nun das Teilprogramm im Betriebssystem, das die Eingaben des Anwenders editiert.

f) Begriff: ASSEMBLER
Wir wissen bisher, daß ein Assembler ein Programm ist, das einen symbolischen Programmcode in den Maschinencode umsetzt. Der Assembler macht also aus einem Quellprogramm einen Objektcode. Die einfachste Assemblerversion ist ein erweiterter Monitor, wie es z.B. MIKROMON darstellt. Hierbei wird der Assembler-Code nach jeder Eingabe einer Zeile unmittelbar assembliert. Daher nennt man diese Methode „ON LINE-Assembler" oder auch „LINE BY LINE-Assembler". In gewissen Grenzen kann man damit schon einige Programmierarbeiten verrichten. Werden jedoch größere Maschinenprogramme entwickelt, dann ist auch die Assemblierung im ON-LINE-Verfahren nicht besonders geeignet. In diesem Fall benötigt man einen komfortableren Assembler, wie es z.B. der MAE darstellt. Ein solches Assemblerprogramm verfügt meist über einen Texteditor, der es gestattet, das gesamte Programm als Quellcode einzugeben. Der eigentliche Assembler benötigt dann mehrere Läufe, um den Source-Code in den Objektcode umzusetzen. Es werden da z.B. sämtliche Label adressiert und eine Referenzliste ausgegeben. Bei einem weiteren Lauf werden z.B. Makros eingebaut usw. Das Ablegen in den endgültigen Objektbereich erfolgt dann im dritten Lauf. Man bezeichnet einen Assemblerlauf als „PASS". Daher gibt es auch verschiedene Assemblerversionen, die man mit „2-PASS-Assembler" oder „3-PASS-Assembler" bezeichnet. Wenn man bereits mit einem derartigen Programm gearbeitet hat, wird man Maschinenprogramme kaum mehr anders entwickeln.

In Basic wird ein Programm durch Zeilennummern editiert. Bei einem Assemblerprogramm wird jede Zeile im Quelltext ebenfalls durchnumeriert. Das ist zwar nicht unbedingt erforderlich (es gibt auch Assembler, die keine Zeilennummern bieten), aber es erleichtert den Zugriff auf bestimmte Programmzeilen mit Hilfe des Texteditors, der meist zeilenorientiert ist.

Die vorliegenden Betrachtungen waren notwendig, da wir uns in diesem Kapitel mit Programmbeispielen beschäftigen, die meist in symbolischer Schreibweise disassembliert sind.

3.2 Bekannte Programmiermethoden

An dieser Stelle werden wir nun ein paar wichtige Programmiermethoden einführen. Diese sollte der Assemblerprogrammierer seinem Standardwissen einverleiben.

a) Inkrementierung zweier Bytes

```
INC  LOWBYTE    ; (ADL)
BNE  +3         ; (+2 wenn ZP)
INC  HIGHBYTE   ; (ADH)
```

Das HIGHBYTE wird nur inkrementiert, wenn das LOWBYTE den Wert $FF überschreitet und somit auf Null geht. Das hat den Vorteil, daß man eine Absolutadresse als Zähler betreiben kann.

b) Dekrementierung zweier Bytes

```
LDA LOWBYTE
BNE +3 oder +2
DEC HIGHBYTE
DEC LOWBYTE
```

Das HIGHBYTE wird jedesmal dekrementiert, wenn das LOWBYTE den Wert Null erreicht hat. Unmittelbar darauf wird das LOWBYTE auf $FF dekrementiert. Will man einen Aussprung erreichen, wenn beide Bytes den Wert Null haben, dann wird DEC HIGHBYTE durch

```
LDA   HIGHBYTE
BEQ   EXIT      ; Aussprung
DEC   HIGHBYTE
```

ersetzt.

c) Addition zweier Byte-Paare

```
CLC
LDA   AL1
ADC   AL2
STA   AL2
LDA   AH1
ADC   AH2
STA   AH2
```

d) Subtraktion zweier Byte-Paare.

```
SEC
LDA   AL1
SBC   AL2
STA   AL2
LDA   AH1
SBC   AH2
STA   AH2
```

In der vorliegenden Addition und Subtraktion steht das Ergebnis jeweils in AL2 und AH2. Mit

```
PRINT PEEK(AL2)+256*PEEK(AH2)
```

könnte man nun das Ergebnis ausgeben lassen.

e) Die Multiplikation zweiter Bytes

Multiplikation und Division sind Funktionen, die dem 6502 als Befehl nicht zur Verfügung stehen. Das nachfolgende Beispiel führt die Multiplikation mit Zeropage-Adressen durch.

```
$033C  18        CLC
$033D  A9 00     LDA  #$00
$033F  A2 08     LDX  #$08
$0341  6A        ROR
$0342  66 4E     ROR  $4E
$0344  90 03     BCC  $0349
$0346  18        CLC
$0347  65 4F     ADC  $4F
$0349  CA        DEX
$034A  10 F5     BPL  $0341
$034C  85 4F     STA  $4F
$028C  60        RTS
```

Die Wirkung des Programms können Sie mit folgendem Basic-Programm betrachten

```
10      INPUTX,Y:POKE78,X:POKE79,Y:SYS828:PRINTPEEK(78)+256*PEEK(79)
```

Es werden 2 Bytewerte multipliziert und um einiges schneller als es der Basic-Interpreter tut. Da die Routine ca. 165 Taktzyklen benötigt, können um die 6000 Multiplikationen pro Sekunde durchgeführt werden.

f) Division zweier Bytes durch ein Byte

Eine 16-Bit-Zahl wird durch eine 8-Bit-Zahl dividiert. Der Quotient besteht aus dem Ergebnis und dem Restwert. Wird der Divisor gegenüber dem Dividenden zu klein gehalten (z.B. 65300/2), dann ist der Quotient größer als 255 und kann nicht mehr in dieser Form dargestellt werden. Darauf sollte man bei der Eingabe achten.

Für die Veranschaulichung der nachfolgenden Maschinenroutine geben Sie bitte folgendes Basic-Programm ein:

```
10  PRINTCHR$(147):INPUT"DIVIDEND:";X:INPUT"DIVISOR :";Y
20  POKE78,X–INT(X/256)*256 : POKE79,X/256 : POKE84,Y
30  SYS 828
40  PRINT"QUOTIENT ="PEEK(78)
50  PRINT"RESTWERT="PEEK(79)
```

Die Maschinenroutine sieht folgendermaßen aus:

```
$033C  18        CLC
$033D  A2 08     LDX  #$08
$033F  A5 4F     LDA  $4F
$0341  26 4E     ROL  $4E
$0343  2A        ROL
$0344  B0 04     BCS  $034A
$0346  C5 54     CMP  $54
$0348  90 03     BCC  $034D
$034A  E5 54     SBC  $54
$034C  38        SEC
$034D  CA        DEX
$034E  D0 F1     BNE  $0341
$0350  26 4E     ROL  $4E
$0352  85 4F     STA  $4F
$0354  60
```

Auch diese Routine ist sehr schnell. Es können bis zu 5400 Divisionen pro Sekunde ablaufen. Ein Programmlauf kann z.B. so aussehen:

```
DIVIDEND:  ? 1020
DIVISOR  :  ? 250
QUOTIENT = 4
RESTWERT = 20
```

g) Vergleich zweier Byte-Paare

Der Trick im nächsten Fall besteht darin, daß der Befehl „CMP" vermieden wird und stattdessen der Befehl „SBC" zur Anwendung kommt. Das hat den Vorteil, daß keinerlei Beeinflussung der Flags stattfindet, wie es bei CMP der Fall ist. Dadurch wird eine Routine ermöglicht, die auf Gleichheit als auch auf kleiner und gleicher Bedingung mit der Hilfe der Carry-Flag testet.

```
SEC
LDA   AL1
SBC   AL2
STA   ZWISCHENSPEICHER
LDA   AH1
SBC   AH2
ORA   ZWISCHENSPEICHER
```

Ist der Inhalt der ersten Adresse (AL1, AH1) identisch mit dem Inhalt der zweiten Adresse (AL2, AH2), dann wird die Zero-Flag gesetzt. War der Inhalt der ersten Adresse kleiner als der der zweiten, dann ist Carry-Flag gelöscht, während umgekehrt (1. Adresse > 2. Adresse) das C-Bit gesetzt wird. Somit kann man anschließend mit BEQ, BCC und BCS auf „=", „<" und „>" sehr leicht prüfen.

Eine ähnliche Routine benützt das Betriebssystem, um Zeichen zu holen und dabei abzufragen, ob diese numerisch oder alphanumerisch sind, die sogenannte CHRGET-Routine. Wir wollen uns diese einmal näher betrachten, da man daraus auch einiges lernen kann.

Die CHRGET-Routine

Im ROM-Bereich liegt eine Routine, die eine zentrale Bedeutung für den Basic-Modus darstellt. Diese Routine, die man mit dem symbolischen Namen ‚CHRGET' (übersetzt = hole Zeichen) kennzeichnet, wird von der Einschalt-Reset-Routine in die Zeropage ab der Adresse $73 kopiert. Warum die CHRGET-Routine in den Arbeitsspeicherbereich umkopiert wird, hängt damit zusammen, daß zwei Bytes in dieser Routine laufend verändert werden, was ja im ROM-Bereich nicht geschehen kann. Die Routine hat nun folgendes Aussehen.

```
0073  E6 7A    INC  $7A
0075  D0 02    BNE  $0079
0077  E6 7B    INC  $7B
0079  AD .. ..  LDA  TEXTZEIGER
007C  C9 3A    CMP  #$3A
007E  B0 0A    BCS  $008A
0080  C9 20    CMP  #$20
0082  F0 EF    BEQ  $0073
0084  38       SEC
0085  E9 30    SBC  #$30
0087  38       SEC
0088  E9 D0    SBC  #$D0
008A  60       RTS
```

Die Punkte hinter dem OP-Code ‚AD' sollen die laufende Veränderung dieser 2 Bytes andeuten. Die beiden Bytes stellen einen Zeiger dar, der auf eine Adresse im Basic-Eingabepuffer zeigt, wobei gerade der Basictext unter Bearbeitung liegt. Zur Untersuchung des Basictextes wird eben die CHRGET-Routine aufgerufen.

Zuerst wird durch die Befehle ‚INC' der Textzeiger um 1 erhöht und zeigt dadurch auf das nächste zu holende Zeichen. Das Zeichen wird nun in den Akku geladen und untersucht. Der Befehl ‚CMP #$3A' vergleicht den Akkuinhalt mit dem Wert $3A. ‚BCS' fragt dann ab, ob das CARRY-Flag gesetzt ist. Das bedeutet nichts anderes, als daß zu $0087 (RTS) verzweigt wird, wenn der Akkuinhalt größer oder gleich $3A ist. Oder anders ausgedrückt, der Aussprung aus dieser Routine erfolgt, wenn der Akkuwert größer als $39 ist und $39 (57) ist gerade das Ende des Ziffernbereiches im ASCII-Code.

Falls der Akkuinhalt kleiner als $3A sein sollte, setzt sich der Programmlauf mit dem zweiten CMP-Befehl fort. Dieser prüft, ob es sich bei dem geholten Zeichen um ein Leerzeichen (ASCII-Code = $20) handelt. Wenn ja, dann erfolgt ein Sprung zum Anfang der Routine und das nächste Zeichen wird geholt. Ansonsten erfolgt eine doppelte Subtraktion mit den Werten $30 und $D0.

Das heißt, war der Akkuwert kleiner als $30 und somit unterhalb der ASCII-Ziffern, so wird das Carry-Bit durch die erste Subtraktion gelöscht und gleich wieder durch ‚SEC' gesetzt. Liegt der Akkuinhalt jedoch im Bereich zwischen $30 und $39 (ASCII-Ziffern 0 – 9), dann wird die Carry-Flag erst durch die zweite Subtraktion gelöscht. Das Interessante an der doppelten Subtraktion ist die Tatsache, daß im Grunde genommen nämlich nichts subtrahiert wird. $30 und $D0 ergeben zusammen $100, so daß nach den Subtraktionen vom Akku der Wert Null abgezogen wurde. Nur die Carry-Flag wurde entsprechend beeinflußt.

Der Nutzen der sich daraus ergibt, ist der, daß mit der CHRGET-Routine geprüft wird, ob das geholte Zeichen numerisch ist oder nicht, und beim CBM-Betriebssystem wird dadurch zwischen Direktmodus und Basicmodus unterschieden.

Man kann nun diese Routine dazu benutzen, um Befehlserweiterungen unterzubringen. Das wird auch von den meisten Toolkits und Hilfsprogrammen verwendet.

h) 2er-Komplementbehandlung

In der Regel wird das 2er-Komplement so gebildet, indem man die Bits negiert und dann 1 addiert. Eine 8-Bit-Zahl kann man somit sehr leicht ins 2er-Komplement überführen.

```
LDA   ZAHL
EOR   #$FF    ; negiert jedes Bit
CLC
ADC   #$01    ; oder SEC / ADC #$00
```

Führt man diese Operation zweimal durch, erhält man wieder den Ausgangswert. Es ist nämlich nicht möglich, den Ausgangswert durch Subtraktion mit 1 und anschließender Komplementierung zurückzugewinnen. Bei einer 16-Bit-Zahl werden beide Bytes mit „EOR #$FF" verknüpft und dann durch 2-Byte-Addition mit #$01 zusammengefaßt.

i) Retten und Wiederherstellen der Zero-Page

Wenn man längere Maschinenprogramme schreibt, so wird man sich auch der Zero-Page bedienen. Da aber der Interpreter und das Betriebssystem ebenfalls darauf zugreifen, kommt es sehr wahrscheinlich zum Konflikt, wenn vom Maschinenprogramm zu Basic

zurückgekehrt wird bzw. das Maschinenprogramm mit Basic zusammenarbeitet. Deshalb soll vor dem eigentlichen Maschinenprogramm folgende Rettungsroutine stehen:

```
SAVE-ZP  LDX  #$00
LOOP     LDA  $00,X
         STA  MEMORY,X
         INX
         BNE  LOOP
```

Nach Ende des Maschinenprogramms, kurz vor der Rückkehr, muß dann die Wiederherstellung der Zero-Page erfolgen:

```
RESTORE-ZP  LDX  #$00
LOOP        LDA  MEMORY,X
            STA  $00,X
            INX
            BNE  LOOP
```

Mit dieser Methode können aber auch andere Pages verschoben werden. Nachfolgendes Programm verschiebt z.B. den Stapel nach $7100.

```
         LDX  #$00
LOOP     LDA  $0100,X
         STA  $7100,X
         INX
         BNE  LOOP
```

j) Stringübernahme

Um eine Zeichenkette aus einer Tabelle zu holen und auf dem Bildschirm auszugeben, bedient man sich meist der nachfolgenden Routine. Die Zeichenkette wird durch den Bytewert „0" abgeschlossen. Zudem muß die Startadresse der Strings bekannt sein.

```
         LDX  #$00
LOOP     LDA  STRING,X    ; STRING = Startadresse der Zeichenkette
         BEQ  RETURN
         JSR  PRINT       ; Standardausgabe
         INX
         BNE  LOOP
RETURN   ...
```

k) Verschiebung eines größeren Speicherbereiches

Um einen größeren Speicherbereich zu verschieben, benutzt man die nachindizierte Adressierung. Zuvor müssen jedoch ein paar Deklarationen getroffen werden. Die Startadresse des Ausgangs- und des Zielbereiches sowie die Länge des Ausgangsbereiches werden in der Zeropage abgespeichert. Die Länge kann man ermitteln durch Subtraktion der Anfangs- und Endadresse des Ausgangsbereiches. Das Ergebnis ist eine 16-Bit-Zahl, die man in Blöcke (höherwertigen Teil, niederwertigen Teil, Rest) einteilt und ebenfalls in der Zeropage ablegt.

```
Deklarationen:  BASADL  = Basisadresse low
                BASADH  = Basisadresse high
                ZIELAL  = Zieladresse low
                ZIELAH  = Zieladresse high
                LENADL  = Bereichslänge (Anzahl der Blöcke)
                LENADH  = Bereichslänge (Rest)
                Z1 – Z6 = 6 Zeropageadressen

Programm:       LDA  #BASADL
                STA  Z1
                LDA  #BASADH
                STA  Z2
                LDA  #ZIELAL
                STA  Z3
                LDA  #ZIELAH
                STA  Z4
                LDA  #LENADL
                STA  Z5
                LDA  #LENADH
                STA  Z6
```

Mit diesem Programmteil werden die Ausgangswerte in die Zeropage übertragen. Das kann natürlich auch auf andere Weise geschehen. Der Endeffekt sollte jedoch immer der gleiche sein, nämlich die Abspeicherung der Ausgangswerte in die Zeropage.

Verschiebeprogramm:

```
UEBERTRG  LDX  Z6        ; Blockanzahl
          LDY  #$00      ; 256 Bytes/Block
RELOC     LDA  (Z1),Y    ; Byte holen
          STA  (Z3),Y    ; und übertragen
          DEY            ; Bytezähler–1
          BNE  RELOC     ; wenn <> 0, dann weiter übertragen
          INC  Z2        ; BASADH+1
          INC  Z4        ; ZIELAH+1
          DEX            ; Blockzähler–1
          BMI  RETURN    ; alles übertragen ? / ja –>fertig
          BNE  RELOC     ; alle Blöcke ? /nein –> nächst. Block
          LDY  Z5        ; ja, alle Blöcke, LENADL laden
          BNE  RELOC     ; Rest übertragen, wenn LENADL <> 0
RETURN    ...            ;
```

Da dieses Programm doch schon ein wenig komplizierter ist, einige Erklärungen dazu.

Ein Block besteht aus 256 Bytes, so wie eine Page. Wie bereits am Anfang erwähnt, besteht der zu verschiebende Bereich aus einem niederwertigen und einem höherwertigen Teil. Der höherwertige Teil ist eben die Blockanzahl. Deshalb wird am Anfang der Routine die Blockanzahl in das X-Register geladen. Das Y-Register bekommt den Wert „0". Das heißt, wenn Y dekrementiert wird und die Herabzählung von 256 bis 1 erfolgt. Solange Y nicht gleich Null ist, wird Byte für Byte eines Blockes übertragen. Danach wird

die Basis- und Zieladresse jeweils inkrementiert und steht somit für die Übertragung des nächsten Blocks bereit. Erst dann wird der Blockzähler (X-Register) dekrementiert. Wenn alle Blöcke verschoben wurden, enthält das X-Register den Wert Null. Da jetzt die Negativ-Flag noch nicht gesetzt ist, wird der Befehl „BMI" übergangen. Das geschieht auch, wenn noch nicht alle Blöcke übertragen wurden und somit der nächste Befehl (BNE) einen Sprung zum Schleifen veranlaßt. Ansonsten wird auch dieser Befehl übergangen. „LDY" lädt nun den niederwertigen Teil des zu übertragenden Bereiches (Rest). Falls dieser gleich Null ist, ist die Verschiebung beendet. Sonst wird zum Schleifenanfang gesprungen und die restlichen Bytes übertragen. Da jetzt noch einmal der Wert im X-Register dekrementiert (00–01=FF), wird und dadurch die Negative-Flag gesetzt, verzweigt das Programm zum Ende hin.

Obwohl diese Routine zuerst recht „umfangreich" aussieht, „schluckt" sie doch nur 24 Bytes. Man kann sie recht gut anwenden, wenn man z.B. Bewegungen (Animation) auf dem Bildschirm darstellen will. Die Einzelfiguren bzw. Graphiken werden in Tabellen abgelegt und deren Anfangsadressen und Längen in die Zeropage gespeichert. Die Zieladresse ist in diesem Fall dann die Anfangsadresse des Bildschirms ($0400). Mit „JSR UEBERTRAG" erfolgt jeweils die Verschiebung der Graphiktabelle auf den Bildschirm.

Wir haben nun einige Programmtricks kennengelernt. Der Assemblerneuling könnte nun beginnen, eigene Routinen zu schreiben. Normalerweise hat man am Anfang seiner Programmiertätigkeit nur Papier und Bleistift zur Verfügung. Es dauert nicht lange und man wird sich nach einem Assembler oder erweiterten Maschinensprache-Monitor umschauen, da die Umsetzung von DATA's mit POKE sehr aufwendig ist.

Der nächste Schritt ist die Anschaffung eines Programms, das die VC-64-Befehle um einige Befehle erweitert, so daß man zumindest assemblieren und disassemblieren kann. Hat man sich nun zu dem Entschluß durchgerungen, ein erweitertes Monitorprogramm einzukaufen, dann erhebt sich die Frage, ob das Programm auf Diskette (bzw. Kassette) oder in einem EPROM stehen soll. Allgemein wird zur EPROM-Version tendiert, da dann dieses Programm jederzeit im Rechner zur Verfügung steht. Übrigens nennt man solche Programmierhilfen „TOOL KIT", was übersetzt soviel wie Hilfswerkzeug bedeutet.

3.3 Gemischte Programme

Damit Sie sehen, daß man nicht unbedingt nur auf Maschinenebene arbeiten muß, werden in diesem Abschnitt mehrere Programme vorgestellt, die teilweise nur in Basic arbeiten und fast die gleiche Wirkung erzielen, wie entsprechende Maschinenprogramme. Allerdings ist es dabei für den Anwender wichtig, wenn er ein wenig Erfahrung über die verschiedenen Speicherstellen im RAM und ROM besitzt, da man oft mit PEEK und POKE zugreifen muß.

3.3.1 Maschinencode-Umwandlung in DATA's

Oft ist es so, daß man ein Maschinenprogramm zusammen mit einem Basicprogramm abgespeichert haben will. Dabei bietet sich folgende Möglichkeit:

Die zweiziffrigen hexadezimalen Maschinencodes werden in Dezimalzahlen umgewandelt. Danach schreibt man sie in DATA-Zeilen und mit

```
FOR I = ANFANG TO ENDE : READ X : POKE I , X : NEXT
```

gelangen die Zahlen in den entsprechenden Speicherbereich. Da jedoch das manuelle Umwandeln und Einschreiben in DATA's ziemlich umständlich ist, wurde folgendes Basic-Programm geschrieben:

```
1 INPUT „ANFANGSADRESSE“ ; A
2 INPUT „ENDADRESSE“ ; E
3 INPUT „DATA'S AB ZEILENNUMMER“ ; Z
4 PRINT CHR$ (147) CHR$ (17) CHR$ (17) Z „DATA“ ; : IFA > ETHENEND
5 FOR A=A TOA+15+(E < A+15 ) * (A–E+15)
6 PRINT MID$ (STR$( PEEK (A) ) , 2) „ , “ ; : NEXT
7 PRINT CHR$ (157)+ “ “ : PRINT“ A = „A“ : E = „E“ : Z = „Z + 10“ : GOTO4“ +
                                                              CHR$ (19);
8 POKE 631,13 : POKE 632,13 : POKE 198,2 : END
```

Dieses Programm wandelt nun um und schreibt gleichzeitig die Basic-Zeilennummer und den Befehl DATA dazu. Es müssen nur die Anfangs- und Endadresse des Maschinenprogramms dezimal angegeben werden, sowie die Zeilennummer, von der ab die DATA's stehen sollen. Natürlich muß diese größer als 8 sein, sonst zerstört sich das Programm selbst. Nach Programmausführung erscheint am Bildschirm nach LIST das Umwandlungsprogramm mit den Zeilen 1 – 8 und die entsprechenden DATA-Zeilen.

3.3.2 AUTONUMBER

Das nächste Programm wurde ebenfalls in Basic geschrieben und erscheint auf Anhieb erstaunlich kurz. Es wird als erstes Programm in den Speicher eingegeben bzw. eingeladen und mit RUN gestartet. Nun fragt das Programm ab, bei welcher Zeilennummer begonnen werden kann und in welche Zeilendifferenz (meist 10) programmiert werden soll. Danach können Sie nun mit dem Programm beginnen. Das Programm gibt Ihnen nun jedesmal, wenn Sie eine Basic-Zeile mit RETURN abschließen, automatisch die nächste Zeilennummer vor. Es dürfte klar sein, daß das AUTONUMBER-Programm mit hohen Zeilennummern versehen wird.

```
60000  INPUT„STARTZEILE“; S
60001  INPUT„ZEILENWEITE“; I
60002  U$=CHR$ (147) + CHR$ (17) + CHR$ (17) + CHR$ (17)
60003  PRINT U$; S; : POKE204,0
60004  GET A$ ; : IFA$= “ “THEN 60004
60005  PRINT A$ ; : IF ASC (A$ ) <> 13 THEN 60004
60006  P=PEEK (1145+LEN ( STR$ (S) ) )
60007  IF P=32 OR P=160 THEN 60003
60008  PRINT“S=„S+I“ : I = „I“ : GOTO 60002“+CHR$ (19)
60009  POKE 631,13 : POKE 632,13 : POKE 198,2 : END
```

3.3.3 OLD

Wie man von Basic her weiß, wird durch NEW das gesamte Basic-Programm gelöscht. In Wirklichkeit jedoch werden nur die Speicherstellen 2049 und 2050 auf Null gesetzt und der Programmendzeiger auf den Programmanfang umgestellt. Der eigentliche komprimierte Programmtext jedoch bleibt erhalten. Deshalb ist es möglich, nach einem ausgeführten NEW das Basic-Programm wieder zurückzuholen. Das nachfolgende Programm, das ich in Assembler geschrieben habe, macht praktisch ein NEW rückgängig, was manchmal bestimmt wertvoll ist. Das Programm kann in fast jedem Speicherbereich liegen, da es völlig frei verschiebbar ist.

```
LDY  #4
INY
LDA  2048,Y
BNE  -6
INY
STY  2049
STY  95
LDA  #4
STA  2050
STA  96
LDY  #1
LDA  (95),Y
BEQ  11
TAX
DEY
LDA  (95),Y
STA  95
STX  96
LDA  #0
BEQ  -18
CLC
LDA  95
ADC  #2
STA  45
STA  47
STA  49
LDA  96
ADC  #0
STA  46
STA  48
STA  50
RTS
```

Ich habe dieses Programm dezimal disassembliert, damit Sie sehen, welche Speicherstellen und Zeiger beeinflußt werden. Es wäre nun eine ganz gute Übung für Sie, das disassemblierte Programm in Dezimalcodes umzuwandeln und über DATA-Statements in einem entsprechenden Speicherbereich unterzubringen. Empfohlen wird der Bereich ab $ C000, der vom Basic-Interpreter nicht benutzt wird. Haben Sie es dort abgespeichert,

so können Sie im Direktmodus zu jeder Zeit das Basic-Programm mit SYS 49152 zurückholen, was Sie mit LIST sofort kontrollieren können.

3.3.4 HEX/DEZ-Umwandlung

Das nächste Programm habe ich wieder in Basic geschrieben, um zu demonstrieren, wie man auch in Basic ein Programm schnell machen kann. Das Programm wandelt hexadezimale Zahlen in dezimale um und umgekehrt. Die eigentlichen Umwandlungsroutinen könnte man in 2 Basic-Zeilen unterbringen, hier aber zwecks der Übersichtlichkeit etwas gestreckt sind.

```
 10 PRINT CHR$(147) : CLR
 20 PRINT" (H) EX / (D) EZ – UMWANDLUNG"
 30 GET A$ : IF A$ = " " THEN 30
 40 IF A$ = „D" THEN 170
 50 IF A$ <> „H"THEN 30
 60 PRINT CHR$ (147) „HEX –> DEZ" : PRINT
 70 INPUT „HEXZAHL (VIERSTELLIG)" ; H$ : X$ = H$
 80 D = 0 : FOR I = 1 TO 4 : D% = ASC (H$)
 90 D% = D% – 48 + (D% > 64) * 7
100 H$ =MID$(H$ ,2) : D = 16*D+D% : NEXT
110 PRINT CHR$ (147) "$ "X$ " = "D
120 PRINT : PRINT"NOCH EINE UMWANDLUNG J/N ?"
130 GET A$ : IF A$ = " " THEN 130
140 IF A$ = "N" THEN PRINT"ENDE" : END
150 IF A$ <> "J" THEN 130
160 GOTO 10
170 PRINT CHR$ (147) "DEZ –> HEX" : PRINT
180 INPUT"DEZ–ZAHL ( <= 65535)" ; D : X = D : X$= "$"
190 D = D/4096 : FOR I=1 TO 4 : D% = D
200 H$ = CHR$ (48+D%– (D% > 9) * 7) : X$+ H$
210 D = 16* (D–D%) : NEXT
220 PRINT CHR$ (147) X" = "X$ : GOTO120
```

Eigentlich wären wir damit am Ende des Buches angelangt. Wenn Sie sich durch die einzelnen Kapitel durchgearbeitet haben, sollten Sie die entsprechende Einführung in die Maschinensprache der VC-Serie besitzen, um auf Maschinenebene hinunterzusteigen. Das bedeutet nun nicht wörtlich, daß es sich um einen Abstieg handelt. Vielmehr ist es so, daß mit Assemblerprogrammen ein gewisser Komfort und besondere Schnelligkeit erreicht wird. Sie werden am Anfang jedoch kein Assemblerspezialist sein. Sie werden, wie jeder Einsteiger, am Anfang recht kräftig experimentieren und wahrscheinlich die Entdeckung machen, daß man bei den meisten Problemen doch lieber bei Basic bleibt. So ist es denkbar, daß sich eine Prozessorfamilie radikal ändert und damit alte Maschi-

nenprogramme unbrauchbar werden, während Basic-Programme ohne Probleme laufen. Es gibt natürlich Aufgabenstellungen, die man nur in Maschinensprache lösen kann. So sollte z.B. ein Textverarbeitungsprogramm in Maschinensprache geschrieben sein, um dadurch die Geschwindigkeit kommerzieller Anwendung zu erreichen.

Zu guter letzt sei noch gesagt, daß dieses Werk keinen Anspruch auf Vollständigkeit erhebt, doch wird es dem Assemblerneuling eine wertvolle Hilfe sein.

4 Anhang

4.1 Der Befehlssatz der CPU 6502 alphabetisch geordnet (ohne Adressierungsmethoden)

Mnemonik	Englische Bedeutung	Deutsche Bezeichnung
ADC	Add memory to accumulator with carry	Addiere Inhalt des Speichers zum Akkuinhalt mit Übertrag
AND	„AND" memory with accumulator	„UND"-Verknüpfung Speicher mit Akku
ASL	Shift left one bit (memory or accu)	Verschiebung um 1 Bit nach links (Speicher oder Akku)
BCC	Branch on carry clear	Verzweigung bei gelöschter Carry Flag
BCS	Branch on carry set	Verzweigung bei gesetztem Carry
BEQ	Branch on result zero	Verzweigung bei Null-Ergebnis
BIT	Test bits in memory with accumulator	Bit-Test im Speicher mit Akkumulator
BMI	Branch on result minus	Verzweigung bei negativem Ergebnis
BNE	Branch on result not zero	Verzweigung bei Ergebnis nicht Null
BPL	Branch on result plus	Verzweigung bei positivem Ergebnis
BRK	Force break	Unterbrechungspunkt anlegen
BVC	Branch on overflow clear	Verzweigung bei gelöschter Overflowflag
BVS	Branch on overflow set	Verzweigung bei gesetzter Overflowflag
CLC	clear carry flag	Carry Flag wird gelöscht
CLD	clear decimal mode	Dezimalzustand wird gelöscht
CLI	clear interrupt disable bit	Interrupt Disable Flag wird gelöscht

Mnemonik	Englische Bedeutung	Deutsche Bezeichnung
CLV	clear overflow flag	Overflow Flag wird gelöscht
CMP	compare memory and accumulator	Vergleiche Speicher und Akkumulator
CPX	compare memory and index X	Vergleiche Speicher und Index X
CPY	compare memory and index Y	Vergleiche Speicher und Index Y
DEC	decrement memory by one	Speicherinhalt um 1 erniedrigen
DEX	decrement index X by one	Index X um 1 erniedrigen
DEY	decrement index Y by one	Index Y um 1 erniedrigen
EOR	„Eclusive-OR" memory with accumulator	„Exklusiv-ODER"-Verknüpfung Speicher mit Akkumulator
INC	increment memory by one	Speicherinhalt um 1 erhöhen
INX	increment index X by one	Index X um 1 erhöhen
INY	increment index Y by one	Index Y um 1 erhöhen
JMP	jump to new location	Sprung zu neuer Adresse
JSR	jump to new location saving return address	Sprung zum Unterprogramm
LDA	load accumulator with memory	Lade Akku mit Speicherinhalt
LDX	load index X with memory	Lade Index X mit Speicherinhalt
LDY	load index Y with memory	Lade Index Y mit Speicherinhalt

Mnemonik	Englische Bedeutung	Deutsche Bezeichnung
LSR	shift right one bit (memory or accu)	Verschiebung um ein Bit nach rechts (Speicher oder Akku)
NOP	no operation	Keine Operation
ORA	„OR" memory with accu	„ODER"-Verknüpfung Speicher mit Akkumulator
PHA	push accu on stack	Lege Akku-Daten auf Stapel
PHP	push processor status on stack	Lege Status-Registerdaten auf Stapel
PLA	pull accumulator from stack	Hole Akku-Daten vom Stapel
PLP	pull processor status from stack	Hole Status Registerdaten vom Stapel
ROL	rotate one bit left (memory or accu)	Rotation nach links (Speicher oder Akku)
ROR	rotate one bit right (memory or accu)	Rotation nach rechts (Speicher oder Akku)
RTI	return from interrupt	Rückkehr vom Interrupt
RTS	return from subroutine	Rückkehr vom Unterprogramm
SBC	subtract memory from accumulator with borrow	Subtrahiere Speicherinhalt vom Akku mit negiertem Übertrag (borrow)
SEC	set carry flag	Setze Carry Flag
SED	set decimal mode	Setze Dezimalzustand
SEI	set interrupt disable status	Setze Interrupt Disable Zustand
STA	store accumulator in memory	Speichere Akkudaten im Speicher

Mnemonik	Englische Bedeutung	Deutsche Bezeichnung
STX	store index X in memory	Speichere Index X im Speicher
STY	store index Y in memory	Speichere Index Y im Speicher
TAX	transfer accu to index X	Bringe Akkudaten nach Index X
TAY	transfer accu to index Y	Bringe Akkudaten nach Index Y
TSX	transfer stack pointer to index X	Bringe Stapeldaten nach Index X
TXA	transfer index X to accu	Bringe Index X-Daten nach Akku
TXS	transfer index X to stack pointer	Bringe Index X-Daten nach Stapel
TYA	transfer index Y to accu	Bringe Index Y-Daten nach Akku

4.2 Operationscode der CPU-Befehle

Mnemoniks in alphabetischer Reihenfolge

Mnemonik	Immediate	Zero Page	Zero Page,X	Zero Page,Y	Absolut	Absolut,X	Absolut,Y	Relativ	Akku	Impliziert	(IND),Y	(IND,)X	Indirekt
ADC	69	65	75	–	6D	7D	79	–	–	–	71	61	–
AND	29	25	35	–	2D	3D	39	–	–	–	31	21	–
ASL	–	06	16	–	0E	1E	–	–	0A	–	–	–	–
BCC	–	–	–	–	–	–	–	90	–	–	–	–	–
BCS	–	–	–	–	–	–	–	B0	–	–	–	–	–
BEQ	–	–	–	–	–	–	–	F0	–	–	–	–	–
BIT	–	24	–	–	2C	–	–	–	–	–	–	–	–
BMI	–	–	–	–	–	–	–	30	–	–	–	–	–
BNE	–	–	–	–	–	–	–	D0	–	–	–	–	–
BPL	–	–	–	–	–	–	–	10	–	–	–	–	–
BRK	–	–	–	–	–	–	–	–	–	00	–	–	–
BVC	–	–	–	–	–	–	–	50	–	–	–	–	–
BVS	–	–	–	–	–	–	–	70	–	–	–	–	–
CLC	–	–	–	–	–	–	–	–	–	18	–	–	–
CLD	–	–	–	–	–	–	–	–	–	D8	–	–	–
CLI	–	–	–	–	–	–	–	–	–	58	–	–	–
CLV	–	–	–	–	–	–	–	–	–	B8	–	–	–
CMP	C9	C5	D5	–	CD	DD	D9	–	–	–	D1	C1	–
CPX	E0	E4	–	–	EC	–	–	–	–	–	–	–	–
CPY	C0	C4	–	–	CC	–	–	–	–	–	–	–	–
DEC	–	C6	D6	–	CE	DE	–	–	–	–	–	–	–
DEX	–	–	–	–	–	–	–	–	–	CA	–	–	–
DEY	–	–	–	–	–	–	–	–	–	88	–	–	–
EOR	49	45	55	–	4D	5D	59	–	–	–	51	41	–
INC	–	E6	F6	–	EE	FE	–	–	–	–	–	–	–
INX	–	–	–	–	–	–	–	–	–	E8	–	–	–
INY	–	–	–	–	–	–	–	–	–	C8	–	–	–
JMP	–	–	–	–	4C	–	–	–	–	–	–	–	6C
JSR	–	–	–	–	20	–	–	–	–	–	–	–	–
LDA	A9	A5	B5	–	AD	BD	B9	–	–	–	B1	A1	–
LDX	A2	A6	–	B6	AE	–	BE	–	–	–	–	–	–
LDY	A0	A4	B4	–	AC	BC	–	–	–	–	–	–	–
LSR	–	46	56	–	4E	5E	–	–	4A	–	–	–	–
NOP	–	–	–	–	–	–	–	–	–	EA	–	–	–
ORA	09	05	15	–	0D	1D	19	–	–	–	11	01	–
PHA	–	–	–	–	–	–	–	–	–	48	–	–	–
PHP	–	–	–	–	–	–	–	–	–	08	–	–	–
PLA	–	–	–	–	–	–	–	–	–	68	–	–	–
PLP	–	–	–	–	–	–	–	–	–	28	–	–	–

Mnemonik	Immediate	Zero Page	Zero Page,X	Zero Page,Y	Absolut	Absolut,X	Absolut,Y	Relativ	Akku	Impliziert	(IND),Y	(IND,)X	Indirekt
ROL	–	26	36	–	2E	3E	–	–	2A	–	–	–	–
ROR	–	66	76	–	6E	7E	–	–	6A	–	–	–	–
RTI	–	–	–	–	–	–	–	–	–	40	–	–	–
RTS	–	–	–	–	–	–	–	–	–	60	–	–	–
SBC	E9	E5	F5	–	ED	FD	F9	–	–	–	F1	E1	–
SEC	–	–	–	–	–	–	–	–	–	38	–	–	–
SED	–	–	–	–	–	–	–	–	–	F8	–	–	–
SEI	–	–	–	–	–	–	–	–	–	78	–	–	–
STA	–	85	95	–	8D	9D	99	–	–	–	91	81	–
STX	–	86	–	96	8E	–	–	–	–	–	–	–	–
STY	–	84	94	–	8C	–	–	–	–	–	–	–	–
TAX	–	–	–	–	–	–	–	–	–	AA	–	–	–
TAY	–	–	–	–	–	–	–	–	–	A8	–	–	–
TYA	–	–	–	–	–	–	–	–	–	98	–	–	–
TSX	–	–	–	–	–	–	–	–	–	BA	–	–	–
TXA	–	–	–	–	–	–	–	–	–	8A	–	–	–
TXS	–	–	–	–	–	–	–	–	–	9A	–	–	–

4.3 Zusätzliche OP-Codes und ein CPU-Fehler

Hexadezimaler Code	Ausführung
A7 xx	Akku und X-Register werden mit dem Inhalt der Zeropage-Adresse xx geladen.
87 xx	Das Ergebnis einer UND-Funktion zwischen Akku und X-Register wird in der Zeropage-Adresse xx abgespeichert.
97 xx	Das Ergebnis einer UND-Funktion zwischen Akku und X-Register wird in der indizierten Zeropage-Adresse xx+Y abgespeichert.
9E xxxx	Das Ergebnis einer UND-Funktion zwischen dem X-Register und der Konstanten 02 wird in der Adresse xxxx abgespeichert.
9F xxxx	Das Ergebnis einer UND-Funktion zwischen Akku und X-Register und der Konstanten 02 wird an der Adresse xxxx abgespeichert.
E7 xx	Der Inhalt der Zeropage-Adresse xx wird um 1 inkrementiert und das Ergebnis dann vom Akkuinhalt subtrahiert.
C7 xx	Der Inhalt der Zeropage-Adresse xx wird um 1 dekrementiert und dann mit dem Akkuinhalt verglichen (wie bei CMP-Befehlen).

Diese zusätzlichen Maschinen-Befehle erweitern natürlich den Befehlssatz des 6502. Es dürfte aber kaum ein Bedarf dieser Befehle geben, außer vielleicht den ersten Befehl (A7 xx).

Wenig bekannt dürfte der CPU-Fehler (Maskenfehler des Chips) sein. Wenn dem Befehl $6C (indirekter Sprung) als ADL-Wert ein $FF folgt, so wird die Sprungadresse nicht aus der ADL und ADH-Adresse geholt, sondern aus ADL und der um 1 dekrementierten ADH-Adresse (ADH−1).
Beispiel:

Adresse	Inhalt
0300	00
03FF	AB
0400	50

Wird nun der indirekte Sprung 6C FF 03 ausgeführt, dann nimmt der Programmzähler nicht die Adresse 50AB auf, sondern:

00AB

4.4 Hexadezimal – Dezimal – Umwandlungstabelle

Hex	ADL dez	ADH dez	hex	ADL dez	ADH dez	hex	ADL dez	ADH dez	hex	ADL dez	ADH dez
$00	0	0	$40	64	16384	$80	128	32768	$C0	192	49152
$01	1	256	$41	65	16640	$81	129	33024	$C1	193	49408
$02	2	512	$42	66	16896	$82	130	33280	$C2	194	49664
$03	3	768	$43	67	17152	$83	131	33536	$C3	195	49920
$04	4	1024	$44	68	17408	$84	132	33792	$C4	196	50176
$05	5	1280	$45	69	17664	$85	133	34048	$C5	197	50432
$06	6	1536	$46	70	17920	$86	134	34304	$C6	198	50688
$07	7	1792	$47	71	18176	$87	135	34560	$C7	199	50944
$08	8	2048	$48	72	18432	$88	136	34816	$C8	200	51200
$09	9	2304	$49	73	18688	$89	137	35072	$C9	201	51456
$0A	10	2560	$4A	74	18944	$8A	138	35328	$CA	202	51712
$0B	11	2816	$4B	75	19200	$8B	139	35584	$CB	203	51968
$0C	12	3072	$4C	76	19456	$8C	140	35840	$CC	204	52224
$0D	13	3328	$4D	77	19712	$8D	141	36096	$CD	205	52480
$0E	14	3584	$4E	78	19968	$8E	142	36352	$CE	206	52736
$0F	15	3840	$4F	79	20224	$8F	143	36608	$CF	207	52992
$10	16	4096	$50	80	20480	$90	144	36864	$D0	208	53248
$11	17	4352	$51	81	20736	$91	145	37120	$D1	209	53504
$12	18	4608	$52	82	20992	$92	146	37376	$D2	210	53760
$13	19	4864	$53	83	21248	$93	147	37632	$D3	211	54016
$14	20	5120	$54	84	21504	$94	148	37888	$D4	212	54272
$15	21	5376	$55	85	21760	$95	149	38144	$D5	213	54528
$16	22	5632	$56	86	22016	$96	150	38400	$D6	214	54784
$17	23	5888	$57	87	22272	$97	151	38656	$D7	215	55040
$18	24	6144	$58	88	22528	$98	152	38912	$D8	216	55296
$19	25	6400	$59	89	22784	$99	153	39168	$D9	217	55552
$1A	26	6656	$5A	90	23040	$9A	154	39424	$DA	218	55808
$1B	27	6912	$5B	91	23296	$9B	155	39680	$DB	219	56064
$1C	28	7168	$5C	92	23552	$9C	156	39936	$DC	220	56320
$1D	29	7424	$5D	93	23808	$9D	157	40192	$DD	221	56576
$1E	30	7680	$5E	94	24064	$9E	158	40448	$DE	222	56832
$1F	31	7936	$5F	95	24320	$9F	159	40704	$DF	223	57088
$20	32	8192	$60	96	24576	$A0	160	40960	$E0	224	57344
$21	33	8448	$61	97	24832	$A1	161	41216	$E1	225	57600
$22	34	8704	$62	98	25088	$A2	162	41472	$E2	226	57856
$23	35	8960	$63	99	25344	$A3	163	41728	$E3	227	58112
$24	36	9216	$64	100	25600	$A4	164	41984	$E4	228	58368
$25	37	9472	$65	101	25856	$A5	165	42240	$E5	229	58624
$26	38	9728	$66	102	26112	$A6	166	42496	$E6	230	58880
$27	39	9984	$67	103	26368	$A7	167	42752	$E7	231	59136
$28	40	10240	$68	104	26624	$A8	168	43008	$E8	232	59392
$29	41	10496	$69	105	26880	$A9	169	43264	$E9	233	59648
$2A	42	10752	$6A	106	27136	$AA	170	43520	$EA	234	59904
$2B	43	11008	$6B	107	27392	$AB	171	43776	$EB	235	60160
$2C	44	11264	$6C	108	27648	$AC	172	44032	$EC	236	60416
$2D	45	11520	$6D	109	27904	$AD	173	44288	$ED	237	60672
$2E	46	11776	$6E	110	28160	$AE	174	44544	$EE	238	60928
$2F	47	12032	$6F	111	28416	$AF	175	44800	$EF	239	61184
$30	48	12288	$70	112	28672	$B0	176	45056	$F0	240	61440
$31	49	12544	$71	113	28928	$B1	177	45312	$F1	241	61696
$32	50	12800	$72	114	29184	$B2	178	45568	$F2	242	61952
$33	51	13056	$73	115	29440	$B3	179	45824	$F3	243	62208
$34	52	13312	$74	116	29696	$B4	180	46080	$F4	244	62464
$35	53	13568	$75	117	29952	$B5	181	46336	$F5	245	62720
$36	54	13824	$76	118	30208	$B6	182	46592	$F6	246	62976
$37	55	14080	$77	119	30464	$B7	183	46848	$F7	247	63232
$38	56	14336	$78	120	30720	$B8	184	47104	$F8	248	63488
$39	57	14592	$79	121	30976	$B9	185	47360	$F9	249	63744
$3A	58	14848	$7A	122	31232	$BA	186	47616	$FA	250	64000
$3B	59	15104	$7B	123	31488	$BB	187	47872	$FB	251	64256
$3C	60	15360	$7C	124	31744	$BC	188	48128	$FC	252	64512
$3D	61	15616	$7D	125	32000	$BD	189	48384	$FD	253	64768
$3E	62	15872	$7E	126	32256	$BE	190	48640	$FE	254	65024
$3F	63	16128	$7F	127	32512	$BF	191	48896	$FF	255	65280

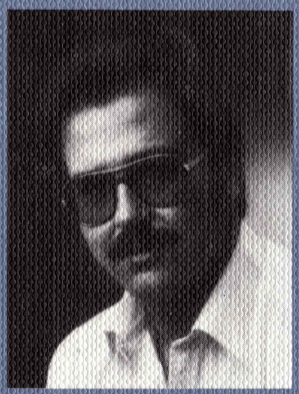
Literaturverzeichnis (Quellennachweis)

Franz Wunderlich
Erfolgreicher mit CBM arbeiten
Franzis-Verlag GmbH, München

Angerhausen, Brückmann, Englisch, Gerits
64 intern
DATA BECKER GmbH, Düsseldorf

Sachverzeichnis

Wunderlich
Erfolgreicher mit dem VC 64 arbeiten



Dipl.-Ing.
Franz Wunderlich

Wer einen VC 64 sein eigen nennt, bekommt mit diesem Buch eine Unterweisung, die er sich eigentlich von Anfang an gewünscht hat. Ein äußerst präziser und differenzierter Lehrgang zeigt ihm das Programmieren in der Maschinensprache. Die Eigenart seiner Maschine wird ihm dabei in allen Dimensionen deutlich, so daß das Schreiben eines guten Programmes in der Maschinensprache von vornherein erfolgreich ist.
Es sei hier stichwortartig auf ein paar Einzelheiten hingewiesen: Grundsätzliches zur Maschinensprache der 65xxSerie. Basic-Interpreter und Betriebssystem auf der Maschinenebene betrachtet. Liste der ROM-Routinen mit hex.- und dezimalen Adressen. Tabellen zur Umrechnung bei der Maschinenprogrammierung.
Dieses Buch ist ein Positivum für alle VC 64 Anwender, auch für die Anfänger unter ihnen, wenn sie sich ernsthaft der Maschinensprache bedienen wollen. Dem Routinier dient es immer wieder als zuverlässiges Nachschlagewerk.

ISBN 3-7723-7781-5

Franzis'